LNS™ Plug-In Programmer's Guide

Version 3.1
Revision 2

ECHELON

Corporation

078-0178-02E

Echelon, LON, LONWORKS, LonBuilder, LonManager, LonTalk, LonUsers, Neuron, 3120, 3150, the LonUsers logo, the Echelon logo, and the LONMARK logo are registered trademarks of Echelon Corporation. LNS, LonMaker, LonSupport, and NodeBuilder are trademarks of Echelon Corporation.

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

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

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

ECHELON MAKES NO REPRESENTATION, WARRANTY, OR CONDITION OF ANY KIND, EXPRESS, IMPLIED, STATUTORY, OR OTHERWISE OR IN ANY COMMUNICATION WITH YOU, INCLUDING, BUT NOT LIMITED TO, ANY IMPLIED WARRANTIES OF MERCHANTABILITY, SATISFACTORY QUALITY, FITNESS FOR ANY PARTICULAR PURPOSE, NONINFRINGEMENT, AND THEIR EQUIVALENTS.

Document No. 39820

Printed in the United States of America.

Echelon Corporation
www.echelon.com

Contents

Preface

This document describes how to write plug-ins. The most common type of plug-ins are device plug-ins, which are used to simplify and automate the installation of your devices by network integrators. Therefore, the focus of this manual is on how to write device plug-ins using the LNS Device Plug-in Wizard, an add-in for Microsoft® Visual Basic® 6. Most of the information in this manual is applicable to writing plug-ins regardless of which development environment you choose to use.

Purpose

This document describes how to write plug-ins using Visual Basic 6. After reading this document, you should understand not only the basic mechanics of how plug-ins work, but how to write a plug-in as well.

Audience

This document assumes that the reader has a basic understanding of both the LNS Object Server and Visual Basic.

Content

This document has four chapters as follows:

- Chapter 1, *Introduction*, provides an introduction to plug-ins and how they operate.
- Chapter 2, *Generating a Plug-in with the LNS Device Plug-in Wizard*, describes how to use the LNS Device Plug-in Wizard to generate customized device plug-ins.
- Chapter 3, *Modifying Code Generated by the LNS Device Plug-in Wizard*, describes how to customize the code generated by the plug-in wizard to provide a custom look-and-feel for your plug-in, and to further customize the plug-in for your device.
- Chapter 4, *How Plug-Ins Work — The Big Picture*, provides an overview of the life cycle of a plug-in. It describes how plug-ins make themselves known to directors, how plug-ins let directors know what they can do, and how directors and plug-ins interact.
- Chapter 5, *How Plug-Ins Work — The Details*, provides more detail on each of the tasks introduced in Chapter 4. For each task it describes why the task is done, how it is done when you are writing a plug-in from scratch (i.e., when not using the LNS Device Plug-in Wizard), and in some instances shows an example of how it is done.

Related Manuals

The following manuals provide supplemental information to the material in this guide:

LNS for Windows Programmer's Guide (Echelon)	Describes how to develop LNS host applications for Windows XP, Windows 2000, Windows NT 4, Windows Me, and Windows 98 hosts using the LNS Object Server ActiveX Control.
Visual Basic User's Guide (Microsoft)	Describes how to develop applications using Visual Basic.
LonMaker™ User's Guide (Echelon)	Describes how to use the LonMaker Integration Tool to design, commission, modify, and maintain LONWORKS networks.
NodeBuilder™ 3 User's Guide (Echelon)	Describes how to use the LNS Device Plug-In Wizard for creation of device plug-in software as part of a LONWORKS device development project.

Other Related Material

This document refers to and describes the LNS Plug-in API that all plug-ins use. This document also describes how to use the LNS Device Plug-in Wizard, a tool that simplifies the development of device plug-ins using Visual Basic 6. The wizard produces Visual

Basic source code that implements the basic framework of a device plug-in, and is included with the NodeBuilder Development Tool and the LNS Application Developer's Kits for Windows. Different wizards are included with the NodeBuilder tool and the LNS developer's kit. This document describes the wizard included with the NodeBuilder tool.

Because plug-ins work in concert with directors, a director is needed in order to fully test and debug the plug-ins that you write. While any director can be used for this purpose, this guide describes the use of the LonMaker tool as the director. See the *LonMaker User's Guide* for more detailed information on the use of the LonMaker tool.

Key Terms And Concepts

The following important terms and concepts are used in this document. These definitions might seem unfamiliar at first glance. However, as you read this document and come across these terms in the various chapters, if you refer back to this list, you should find that the you can understand the definitions when the terms are used in context.

Action
A command/object class pair implemented by a plug-in. A plug-in is defined by the actions that it can perform (i.e., by the set of commands that it provides and by the class of objects that those commands operate on). For example, a plug-in might implement two actions, a test command of `AppDevice` class objects and a test command of `Router` class objects.

ActiveX
A Windows standard for component-based software. ActiveX defines a hierarchy of components, objects, and interfaces. ActiveX components are made up of one or more objects, where each object encapsulates functionality and data. ActiveX objects expose their functionality and data to other components through one or more interfaces.

ActiveX Automation Server
A software component that exposes one or more programmable objects to other software components that are called ActiveX automation clients. The definition for a programmable object is called an ActiveX class.

ActiveX Class
The definition for a programmable ActiveX object.

Class
See Object Class.

Class ID
A number that defines a particular LNS object class. Each action implemented by a plug-in applies to a specific class of LNS object. For example, a plug-in might implement two actions, a test command of `AppDevice` objects and a test command of `Router` objects.

Command
Operations that a plug-in can perform on an object. Each action implemented by a plug-in performs a specific command on a specific class of objects. For example, a plug-in might implement two actions, a test command of `AppDevice` objects and a test command of `Router` objects.

Command ID
A number that defines a particular plug-in command.

ComponentApp
The type of LNS object used to represent plug-ins and their

Object

actions in the LNS Object Server. There is one ComponentApp object for each action that a plug-in implements and one ComponentApp object for the plug-in itself. For example, a plug-in might implement two actions, a test command of AppDevice objects and a test command of Router objects. Each of these actions would be represented by a ComponentApp object (one with a command ID/object class pair of [LcaCommandTest (33), LcaClassIdAppDevice (7)] and the other with command ID/object class pair of [LcaCommandTest (33), LcaClassIdRouter (9)]. For the plug-in itself, the command ID of the ComponentApp object is always LcaCommandRegister (50).

Each ComponentApp object contains the following properties:

ClassId The class ID of ComponentApp objects. For ComponentApp objects this value is lcaClassIdComponentApp (30).

CommandId The command ID implemented by this ComponentApp, for example LcaCommandTest (33). If this ComponentApp object represents the plug-in itself, the CommandId is always LcaCommandRegister (50).

ComponentClassId The class ID of the LNS object to which this command applies. For example, LcaClassIdAppDevice (7).

DefaultAppFlag A flag that indicates if this command is the default action for objects of this type. Directors can use this flag to launch component applications as a default action, such as when a users double-clicks an icon representing the object. There can only be one ComponentApp object per ComponentApps collection with this flag set.

Description A description of this command. Typically directors display this information in a tooltip or status bar.

ManufacturerId The manufacturer ID of the company that wrote the plug-in that implements the command.

Name The name of this command. Typically directors display this information in a context-sensitive menu.

Parent The ComponentApps collection object to which this ComponentApp object belongs.

RegisteredServer The name of the ActiveX class (i.e., the plug-in) that implements this command.

VersionNumber The version number of the plug-in that implements this command.

Director
A special kind of LNS application that makes use of the LNS Plug-in API. Directors have the ability to start plug-ins.

GUID
A 128-bit Globally Unique IDentifier. GUIDs are used to uniquely identify entries in the Windows registry. Visual Basic automatically generates GUIDs that identify your type library, each public class, and each interface in your application. In Windows documentation, the GUIDs for public classes are often referred to as class IDs (CLSID); the GUIDs for interfaces are often referred to as interface IDs (IID). Windows class IDs and Windows interface IDs are the keys to version compatibility for components authored using Visual Basic; see *Design Rules For Future Revisions Of Your Plug-ins* in Chapter 3,

Generating a Plug-in with the LNS Device Plug-in Wizard, for details. Note that the term *Windows class ID* does NOT mean the same thing as the term *class ID* used in this document.

LNS Object
The items managed by LNS. LNS treats each network as a collection of objects. These objects include application devices, routers, connections, functional blocks, network variables, and the system.

LNS Plug-in API
The items managed by LNS. LNS treats each network as a collection of objects. These objects include application devices, routers, connections, functional blocks, configuration properties, network variables, and the system.

Object
See LNS Object.

Object Class
The category of an LNS object. Each object managed by LNS (such as application devices and routers) is in a particular class, identified by ID (such as `LcaClassIdAppDevice` [7] and `LcaClassIdRouter` [9]).

Object Name
A string passed from a director to a plug-in that specifies the location of the target object in the LNS object hierarchy. The name includes any qualification required to find the appropriate object in the hierarchy.

Plug-in
A special kind of LNS application, implemented as an ActiveX automation server, that implements the LNS Plug-in API. Plug-ins provides a standard way to extend and customize the functionality of LNS applications.

Registered Server
The plug-in's ActiveX class that implements the LNS Plug-in API. In Visual Basic, this name is always the name of your project (as set in the project properties dialog box) followed by a dot and the name of the class that implements the LNS Plug-in API. Windows requires that this name contain no more than 38 characters. The project name is set automatically by the LNS Device Plug-in Wizard or manually by selecting the `<your project name> Properties` option from the `Project` menu and filling in the `Project Name` field. For example, if the project name is `MyPlugIn, and` if the class that implements the LNS Plug-in API is named `LNSPlugInAPI`, the registered

server name for this plug-in would be `MyPlugIn.LNSPlugInAPI`.

Registration

The two-step process by which a plug-in installed is imported into the LNS Object Server. In the first phase of registration, the plug-in registers a registration command for itself in the Windows registry. This step is typically performed when the plug-in is installed onto the user's machine.

In the second phase of registration, the plug-in creates `ComponentApp` objects in the LNS Object Server that represent the plug-in's functionality. This phase of registration is initiated by a director sending a registration command to the plug-in. The director knows which plug-ins are installed on the user's machine by accessing the information that was placed into the Windows registry during the first phase of the registration process.

Scope

The "breadth" of a particular plug-in or one of its actions. For example, if the scope of a command that applies to `AppDevice` objects is ObjectServer (1), the command applies to all `AppDevice` objects managed by the LNS Object Server. If the scope of the same command were System (2), then the command would apply to all `AppDevice` objects in a particular system. If the scope of the same command were DeviceTemplate (3), then the command would apply to all `AppDevice` objects that use a particular device template (i.e., all devices of a particular type). If the scope of the same command were LonMarkObject (4), then the command would apply only to a specific functional block on all `AppDevice` objects that use a particular device template.

The scope of an action is indicated by the `ComponentApps` collection that the command is in. If an action is in the `ObjectServer` object's `ComponentApps` collection, the action has object server-wide scope. If it is in a System object's `ComponentApps` collection, the action has system-wide scope. If it is in a `DeviceTemplate` object's `ComponentApps` collection, the action applies only to devices of that type. If it is in a `LonMarkObject` object's `ComponentApps` collection, the action applies only to functional blocks on devices of that type.

The Registration command is a special kind of command that must be supported by every LNS plug-in. It has either server-wide scope (1) or system-wide scope (2). The scope of the Registration command is often referred to as the *plug-in scope*.

Server

See Registered Server.

Target Object

The LNS object on which a director has asked a plug-in to operate.

Windows Registry

The Windows registry is a shared resource on the computer that contains information about how the computer runs. Among other functions, the registry provides a simple database-like mechanism that allows programs to store and exchange information. Information in the registry is stored by *key*. You can think of a key as being like a directory. Each key contains one or more *values* (just as a directory contains files) and can contain additional keys (just as a directory can contain additional directories). Each value is identified by a *name* and contains *data*. All keys have at least one value, named `(Default)`.

1

Introduction

This chapter introduces plug-ins. It describes the types of plug-ins that are used with LONWORKS networks, and describes how director application request actions from plug-ins.

Introduction

Plug-ins are a special kind of LNS application, implemented as Windows® ActiveX automation servers, that implement the LNS Plug-in API. The LNS Plug-in API is described in Chapter 4, *How Plug-Ins Work — The Details*. Plug-ins provide a standard way to extend and customize the functionality of LNS applications. For example, plug-ins allow device manufacturers to provide customized software that simplify configuration, monitoring, or control of their devices[1]. Plug-ins can also add new functionality to tools[2], such as alarming, logging, and trending.

LNS applications that can use plug-ins are called *directors*. Directors are LNS applications that make use of the LNS Plug-in API. By calling the functions of the LNS Plug-in API, directors can determine which plug-ins a user has installed on their computer and call the plug-ins at appropriate times. If it chooses to do so, the director can make access to plug-ins completely transparent to the end-user. That is, the user cannot tell when a task is built-in to the director and when it is being provided by a plug-in called by the director — to the user it looks like one seamless application.

Plug-ins provide many benefits to network integrators, who are the end-users of tools. Plug-ins make tools easier to use and make network integrators more productive. They reduce the cost of training network integrators on the use of tools. They also reduce the time and cost of installing, configuring, and maintaining systems.

Tool and device manufactures also benefit from plug-ins. For tool manufacturers, plug-ins make their tools extensible and thus more valuable. Network integrators can incrementally add features — from the tool's manufacturer or from other plug-in vendors — to adapt the tool to their needs. A tool that supports plug-ins is a tool that will become easier to use, more productive, and more powerful over time.

For device manufacturers, plug-ins allow them to make their devices easy to install, configure, and maintain — without the cost of having to develop an entire, customized tool. Devices that come with plug-ins are devices that have a competitive edge — they are lower cost to install, lower cost to service, and easier to use.

What Makes A Plug-in A Plug-in

A plug-in is a type of LNS application that is also an ActiveX automation server and that implements the LNS Plug-in API interface. The Plug-in API consists of the LNS Plug-in Commands (see Appendix A), the LNS Plug-in Properties (see Appendix B), and LNS Plug-in Exceptions (see Appendix D). A plug-in is defined by the actions that it can perform (i.e., by the set of commands that it provides and by the class of objects that each command operates on). For example, a plug-in might implement two actions, a test command of AppDevice class objects and a test command of Router class objects.

Types of Plug-ins

There are two types of plug-ins. They are *device plug-ins* and *system plug-ins*.

[1] Plug-ins that are specific to a particular type of device are sometimes called *device plug-ins*.

[2] In this guide, the term *tool* is used as shorthand for "an LNS tool that is a director application."

Device plug-ins apply to a single device or a single functional block within a device. For example, you could create a device plug-in for a device developed with the NodeBuilder tool that would allow users to monitor and control the device through a custom interface tailored to the device. You can create a device plug-in executable that operates on multiple device types. Although contained in one executable file or installation, from the point of view of the LNS Object Server, these are separate plug-ins. The single executable that is able to operate on multiple device types is a programming convenience. You can also create a separate device plug-in for each functional block in a device, but it is typical to include the support for all functional blocks of a device type in a single plug-in. The LNS Device Plug-in Wizard, described in Chapters 2 and 3, automatically creates the framework for a device plug-in.

System plug-ins apply to an entire network, or to a subsystem within that network. For example, a system plug-in could provide a custom interface that would allow you to test all devices on a network and see the results. A system plug-in could also be designed to manage all of the devices in a room, or on a floor of a building. The LNS Device Plug-in Wizard does not generate a system plug-in. You can implement a system plug-in based on the descriptions of the plug-in API in Chapters 4 and 5.

LNS Plug-in Commands

The LNS Plug-in API defines a single method and a number of properties that a plug-in must implement. The method, `SendCommand`, is used by directors to ask a plug-in to perform a command (specified by ID) on an object (specified by class ID and object name). Plug-ins can implement standard commands defined by LNS or custom commands that they define. The plug-in designer is responsible for having the plug-in implement commands in a way that makes sense (i.e. if a plug-in implements the Browse command, the plug-in designer must design the implementation of the plug-in's Browse command). The standard plug-in commands are listed in Appendix A, *Standard Plug-in Commands*.

Device plug-ins typically implement the LcaCommandConfigure command. This command is used by network tools to provide an option to network integrators to configure a device or functional block. Device plug-ins may also implement the LcaCommandBrowse command if they are used for monitoring and control in addition to device configuration.

Plug-ins can also implement custom commands. The values for custom command values are assigned by the plug-in, and may be any value greater than or equal to `LcaCommandUserStart` (10000).

LNS Plug-in Properties

All plug-ins must also implement a standard set of properties along with any custom properties required by the developer[3]. The standard properties allow the director to tell the plug-in information about the network is it operating on (such as the `NetworkName` and the `NetworkInterfaceName` properties) as well as to control the appearance of the plug-in (such as the `Left`, `Height`, and `Visible` properties). The complete list of required and optional properties is given in Appendix B. Read access must be provided to all properties. Write access may optionally be provided for most properties. Some properties however, as noted in Appendix B, must be read-only or must be read-write.

[3] The property names `CharacterEncoding` and `LanguageId` are reserved for future use.

2

Generating a Plug-in with the LNS Device Plug-in Wizard

The LNS Device Plug-in Wizard is a Microsoft Visual Basic 6 add-in that creates a customized LNS device plug-in for your device from a standard device plug-in framework . The Device Plug-in Wizard creates the customized LNS device plug-in for a device that you are developing, although you can add support for multiple device types later. You can add additional Visual Basic forms, classes, and modules as you continue to develop your device plug-in.

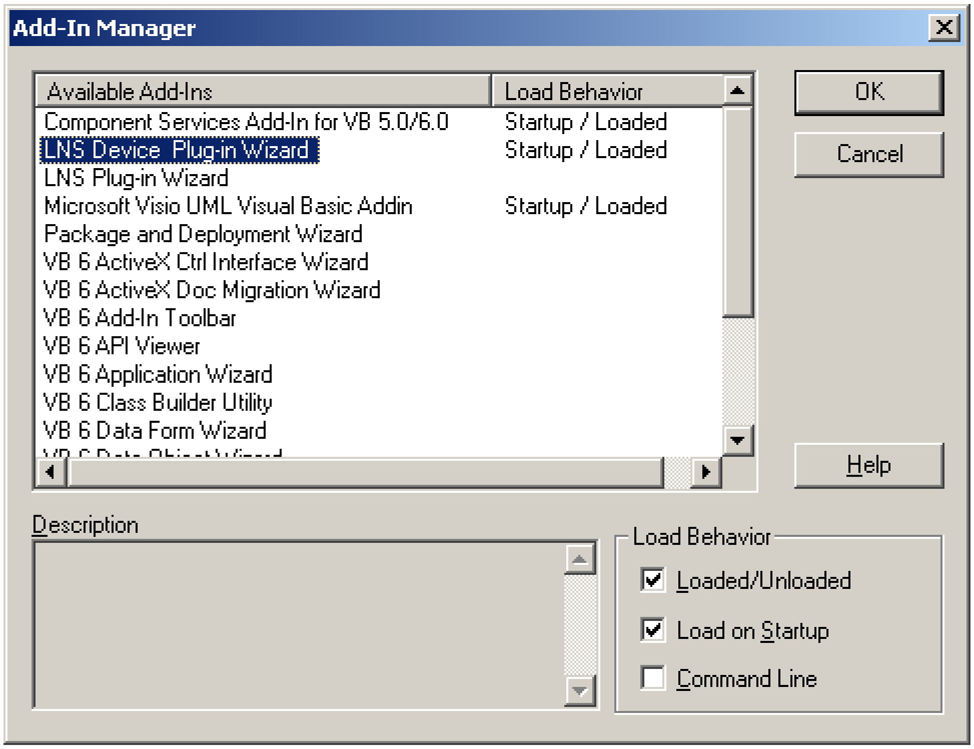
You can only use the plug-in wizard if you are developing a device plug-in with Visual Basic 6. If you are developing a device plug-in but not using Visual Basic 6, you can still use the plug-in wizard to generate an initial prototype of your plug-in. You cannot use the plug-in wizard if you are developing a system plug-in.

Installing the Plug-in Wizard

The plug-in wizard is automatically installed when you install the NodeBuilder software as long as you have installed Microsoft Visual Basic 6 (SP 5 or better) before you install the NodeBuilder software. If you have installed the NodeBuilder software before installing Visual Basic, you will need to re-install the NodeBuilder software again from the NodeBuilder installation CD.

When you install the plug-in wizard, its Visual Basic add-in and bitmap will be installed into the Visual Basic application directory (the location of which varies depending upon your operating system and configuration), and a number of other files into the default Visual Basic `Templates` directories.

The plug-in wizard installation also adds the plug-in wizard add-in to the set of active Visual Basic add-ins. If for some reason this did not happen automatically, you can add it manually by starting Visual Basic, opening the Add-Ins menu, and clicking **Add-Ins Manager**. Select **LNS Device Plug-In Wizard** and then set the Loaded/Unloaded checkbox in **Load Behavior** as shown in Figure 2-1.



Available Add-Ins	Load Behavior
Component Services Add-In for VB 5.0/6.0	Startup / Loaded
LNS Device Plug-in Wizard	Startup / Loaded
LNS Plug-in Wizard	
Microsoft Visio UML Visual Basic Addin	Startup / Loaded
Package and Deployment Wizard	
VB 6 ActiveX Ctrl Interface Wizard	
VB 6 ActiveX Doc Migration Wizard	
VB 6 Add-In Toolbar	
VB 6 API Viewer	
VB 6 Application Wizard	
VB 6 Class Builder Utility	
VB 6 Data Form Wizard	
VB 6 Data Object Wizard	

Load Behavior
☑ Loaded/Unloaded
☑ Load on Startup
☐ Command Line

Figure 2-1. The Visual Basic 6 Add-in Manager

The LNS Device Plug-in Wizard will now appear on your `Add-Ins` menu and is ready for use.

Starting the LNS Device Plug-in Wizard

You can start the LNS Device Plug-in Wizard from Visual Basic or from the NodeBuilder tool. If you are using the NodeBuilder tool, the NodeBuilder tool automatically sets the plug-in wizard start-up options. See the *Creating an LNS Device Plug-in for a NodeBuilder Device* chapter in the *NodeBuilder User's Guide* for more information about starting the LNS Device Plug-in Wizard from the NodeBuilder tool. To start the LNS Device Plug-in Wizard from Visual Basic, follow these steps:

1. Start Visual Basic and open a new project.

2. When you are asked to select a project type, choose LNS Device Plug-in.

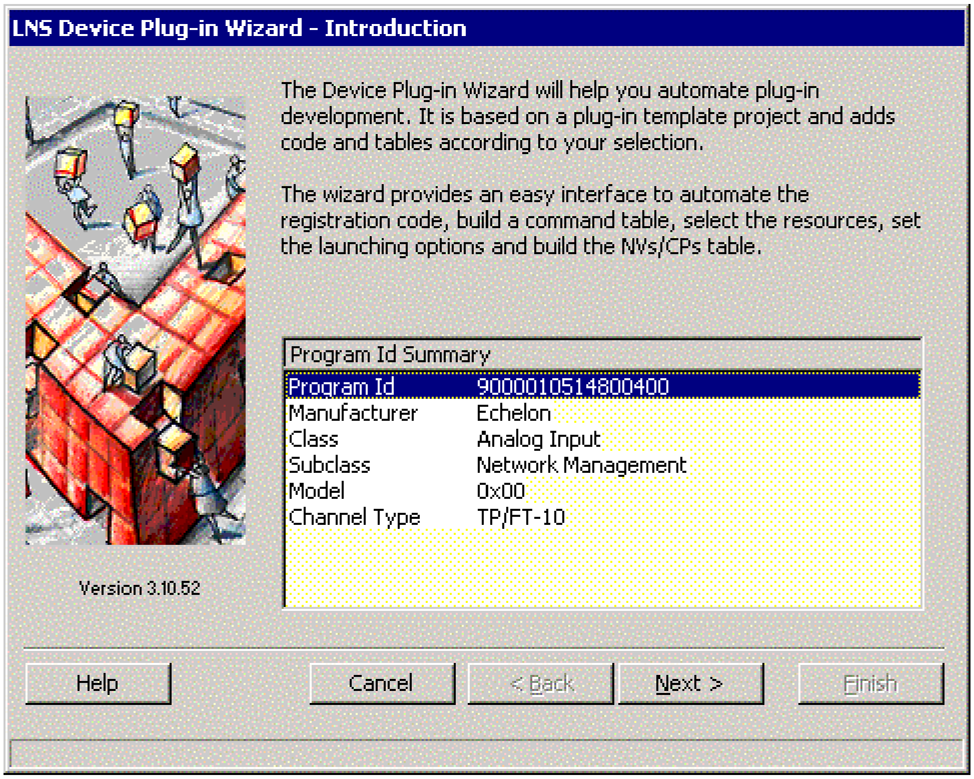
3. Select LNS Device Plug-in Wizard from the Visual Basic Add-Ins menu. If LNS Device Plug-in Wizard is not available in this menu, ensure that the LNS Device Plug-in Wizard is loaded in the Add-Ins Manager as described in *Installing the Plug-in Wizard*, earlier in this chapter. The LNS Device Plug-in Wizard appears.

Using the LNS Device Plug-in Wizard

The plug-in wizard displays a series of windows that allow you to specify the interface to your plug-in and to design an initial user interface for your plug-in. The following sections describe the LNS Device Plug-in Wizard windows.

LNS Device Plug-in Wizard: Introduction

If you start the plug-in wizard from the NodeBuilder tool, the following Plug-in Wizard Introduction window appears:



LNS Device Plug-in Wizard - Introduction

The Device Plug-in Wizard will help you automate plug-in development. It is based on a plug-in template project and adds code and tables according to your selection.

The wizard provides an easy interface to automate the registration code, build a command table, select the resources, set the launching options and build the NVs/CPs table.

Version 3.10.52

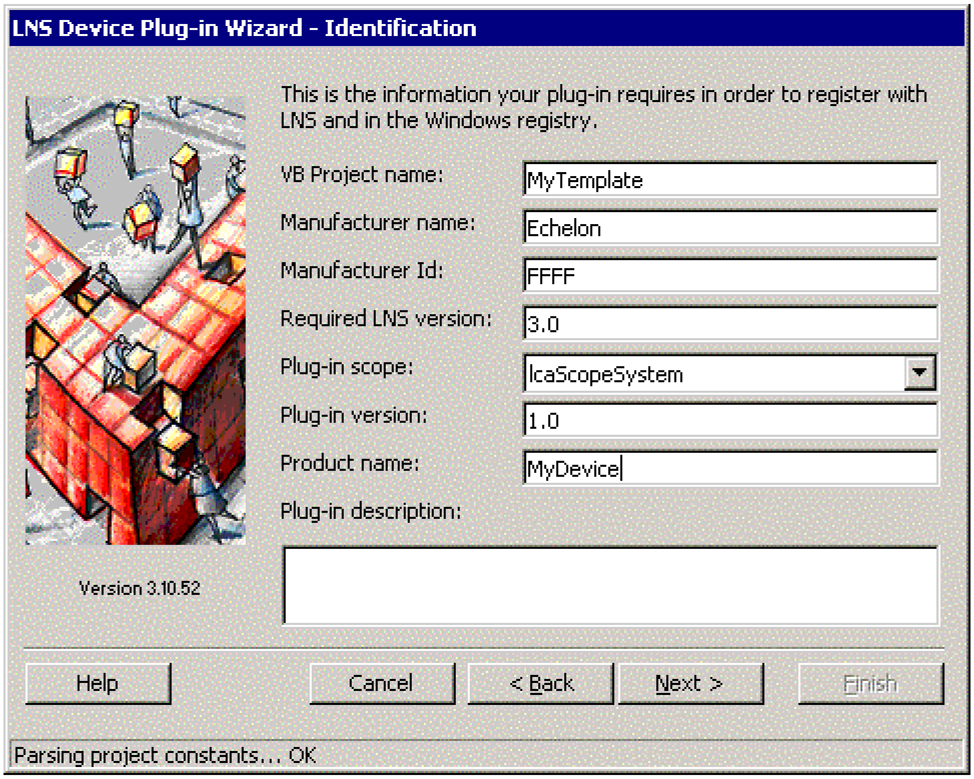
Program Id Summary

Program Id	9000010514800400
Manufacturer	Echelon
Class	Analog Input
Subclass	Network Management
Model	0x00
Channel Type	TP/FT-10

This window shows the Manufacturer, Device Class, Subclass Model Number, and Transceiver Type indicated by the program ID of the device template. If you start the plug-in wizard from the Visual Basic development environment, the Identification tab appears first, but you can click **Back** to see the Introduction window. If you have not selected a device template, this dialog will not contain summary information. Click **Next** to open the Identification window.

LNS Device Plug-in Wizard: Identification

The following window appears when you click **Next** from the Introduction window, or if you start the plug-in wizard from Visual Basic:



LNS Device Plug-in Wizard - Identification

This is the information your plug-in requires in order to register with LNS and in the Windows registry.

VB Project name: MyTemplate

Manufacturer name: Echelon

Manufacturer Id: FFFF

Required LNS version: 3.0

Plug-in scope: lcaScopeSystem

Plug-in version: 1.0

Product name: MyDevice

Plug-in description:

Version 3.10.52

Parsing project constants... OK

This window shows the identification information for the plug-in. This window contains the following fields:

VB Project Name The name to be used for the Visual Basic project. By default, this will be the same as the LNS device template name. This field may contain letters, numbers, and the underscore character. This field is enabled for a new project and disabled if you are opening an existing one

Manufacturer Name The manufacturer name indicated by the program ID. This information will be stored in the plug-in code. Do not change this value, since that will cause the registered manufacturer to disagree with the one documented in the code.

Manufacturer ID The manufacturer ID indicated by the program ID. Do not change this value, since that will cause the registered manufacturer to disagree with the one documented in the code.

Required LNS Version The minimum version of the LNS Server redistribution needed to run this plug-in. By default, the version of LNS currently on your computer will be entered here. Do not change this value unless you have verified compatibility with the new value. The minimum required version is 3.0.

Plug-in Scope

The scope of the LNS device plug-in. You can choose between **lcaScopeSystem** and **lcaScopeObjectServer**. A single instantiation of a LNS device plug-in with a scope of **lcaScopeSystem** can only affect devices in a single network. A single instantiation of a LNS device plug-in with a scope of **lcaScopeObjectServer** can affect devices on all networks on the computer.

Plug-in Version

The version of the LNS device plug-in. By default, the version is 1.0. If this is a new plug-in, this should not be changed. If this plug-in is updated after release, this field can be used to indicate the major and minor version number.

Product Name

The name of the product that will use this LNS device plug-in. This should be a unique name that identifies the LNS device plug-in.

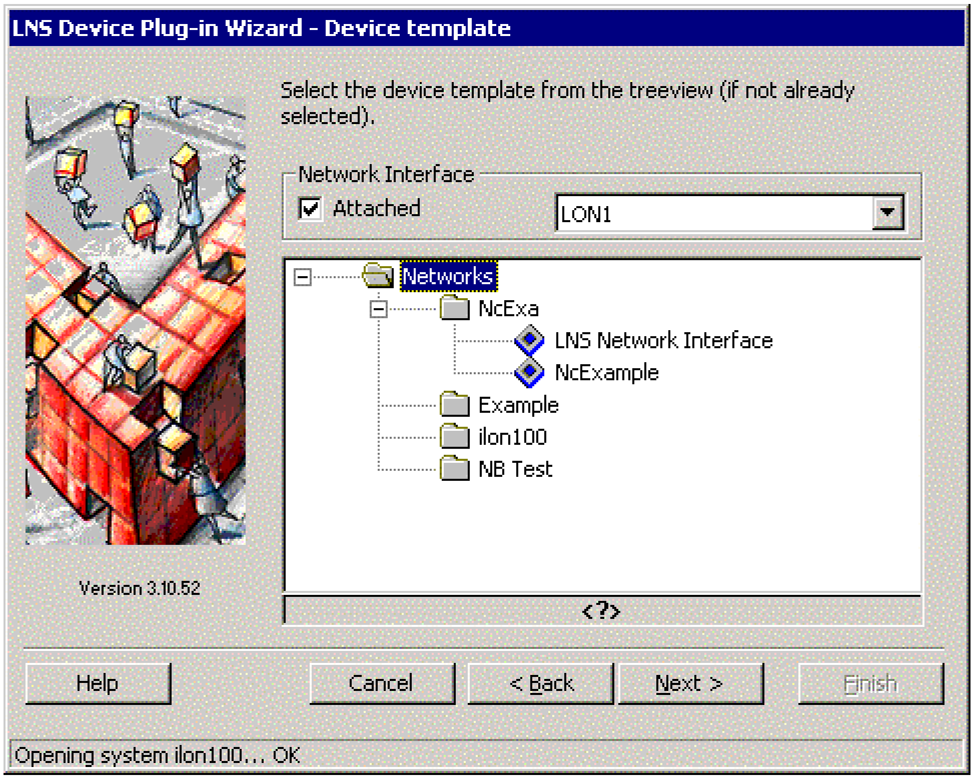
Plug-in description

A description of the purpose of this LNS device plug-in. This description may be displayed by LNS tools that support LNS plug-ins, such as the LonMaker tool.

Enter the product name and LNS device plug-in description, set the scope, and then click **Next**. If this project is not yet associated with an LNS device template, the *Device Template* dialog opens. If this project is already associated with an LNS device template, for example if started the plug-in wizard from the NodeBuilder tool, the *Command Table* dialog opens.

LNS Device Plug-in Wizard: Device Template

The following window appears when you click **Next** from the Identification window and you have not selected a device template:



Select the device template from the treeview (if not already selected).

☑ Attached LON1

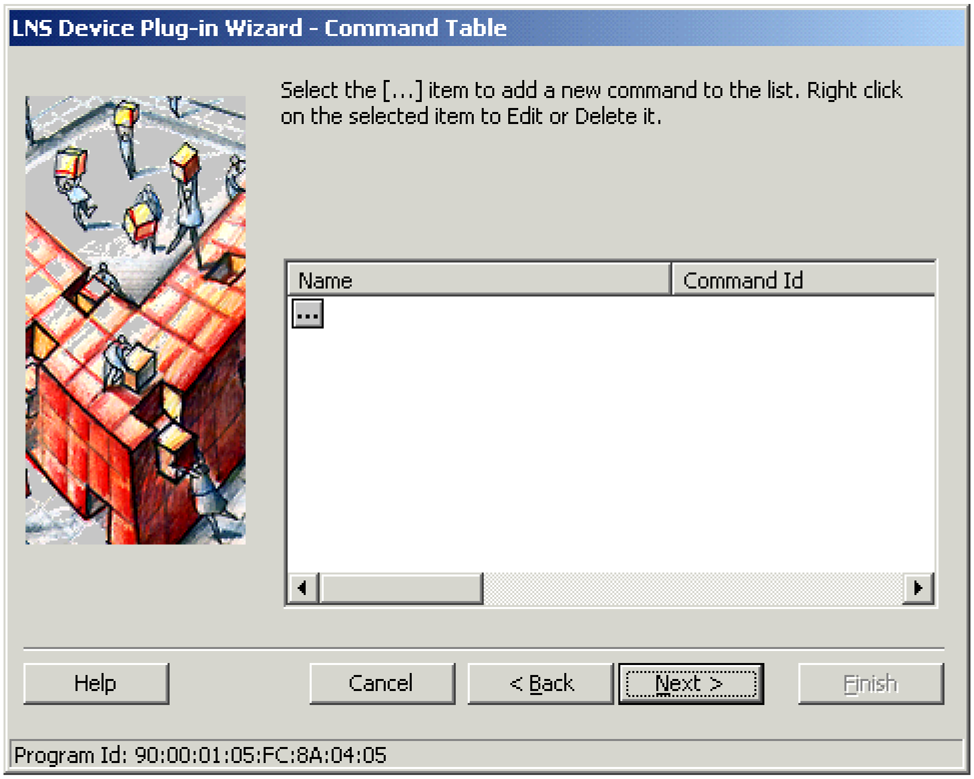
- Networks
  - NcExa
    - LNS Network Interface
    - NcExample
  - Example
  - ilon100
  - NB Test

Opening system ilon100... OK

This window allows you to browse all LNS device templates on your computer. Select the device template of the device for which you are creating a plug-in. Click **Next** to open the Command Table window.

LNS Device Plug-in Wizard: Command Table

The following window appears when you click **Next** from the Device Template window:



LNS Device Plug-in Wizard - Command Table

Select the [...] item to add a new command to the list. Right click on the selected item to Edit or Delete it.

Program Id: 90:00:01:05:FC:8A:04:05

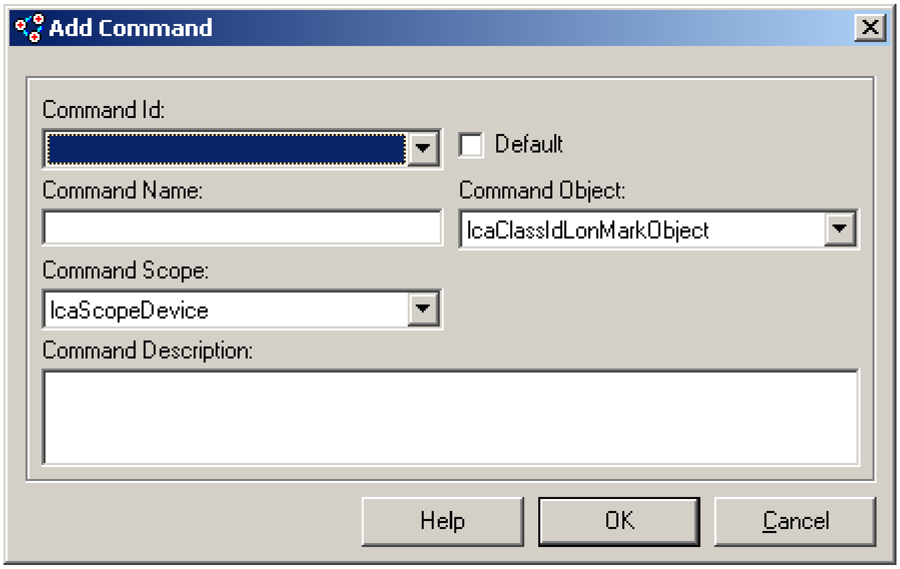
This window allows you to add LNS device plug-in actions that are invoked by LNS tools that support LNS plug-ins. To add a LNS device plug-in action, click the ... button to open the Add Command dialog. Click this button for each action to be implemented by your plug-in, and then enter command information for each command as described in the next section. You can add multiple actions. For example, if your device implements three functional blocks, you can add three actions, each specifying an lcaCommandConfigure command on each of the three functional blocks on the device.

Double-click existing commands to edit them. An Edit Command dialog appears as described in the next section.

Once you have added all commands for this LNS device plug-in, click **Next** to open the Resource Table window.

LNS Device Plug-in Wizard: Add or Edit Command

An Add Command dialog opens when you click the ... button in the Command Table window. An Edit Command dialog appears if you double-click a command in the Command Table window. These dialogs appear as shown in the following figure (the title is Edit Command for the Edit Command dialog):



Command Id:

□ Default

Command Name:

Command Object: lcaClassIdLonMarkObject

Command Scope: lcaScopeDevice

Command Description:

This dialog allows you to add a new command or modify an existing one. Enter the following:

Command ID

The command type for this command. Select **lcaCommandConfigure** if you are developing a device plug-in that allows users to configure a device or functional block. Select **lcaCommandBrowse** if you are developing a device plug-in that provides a monitoring and control interface for a device or functional block. Appendix A lists additional commands that you may implement. You can also enter a user-defined command type. You cannot select the LcaCommandRegister command. All LNS device plug-ins handle this command as described in *Registering Your Plug-in's Registration Command* in Chapter 5, How Plug-Ins Work — The Details.

Default

Specifies that this plug-in is the default handler for this action. Set this checkbox if you want director applications to run your plug-in when the user selects a device or functional block based on your selected device template and invokes a Configure or Browse command.

Command Name

The name of the new command. If Command ID specifies a standard command, the command name should incorporate the standard command name and add a unique modifier that identifies the object that the plug-in acts on. For example, a configuration plug-in command for the Sensor functional block on a MyCo device could be named “Configure MyCo Sensor,” or a configuration plug-in command for the MyCo device could be named “Configure MyCo Temperature Sensor.”

Command Object

The class of LNS object the command applies to. When this command is invoked, an object of this class will be passed to the LNS device plug-in. Object classes are listed in Appendix C, *Standard Plug-in Classes*. Most device plug-ins commands apply to functional blocks, though a device plug-in command may also apply to an entire device. Choose **lcaClassIdLonMarkObject** to indicate the command applies to a functional block. Choose **lcaClassIdAppDevice** to indicate the command applies to the entire device. See *How Directors Pass Object Names* in Chapter 4 for information about how the address of the object is passed to the plug-in.

Command Scope

The scope of this command. This indicates whether this plug-in command can be enacted on any object of the type specified in **Command Object** or only on specific ones. Select **lcaScopeDevice** to allow this command to be called by any functional blocks associated with the device type, or the device itself. Select **lcaScopeLonMarkObject** to only allow this command to be called by a specific functional block. If **lcaScopeLonMarkObject** is selected **LonMark Object** appears.

LonMark Object

Specifies the functional block on the device to which this command applies. Only appears if **lcaScopeLonMarkObject** is selected in **Command Scope.**

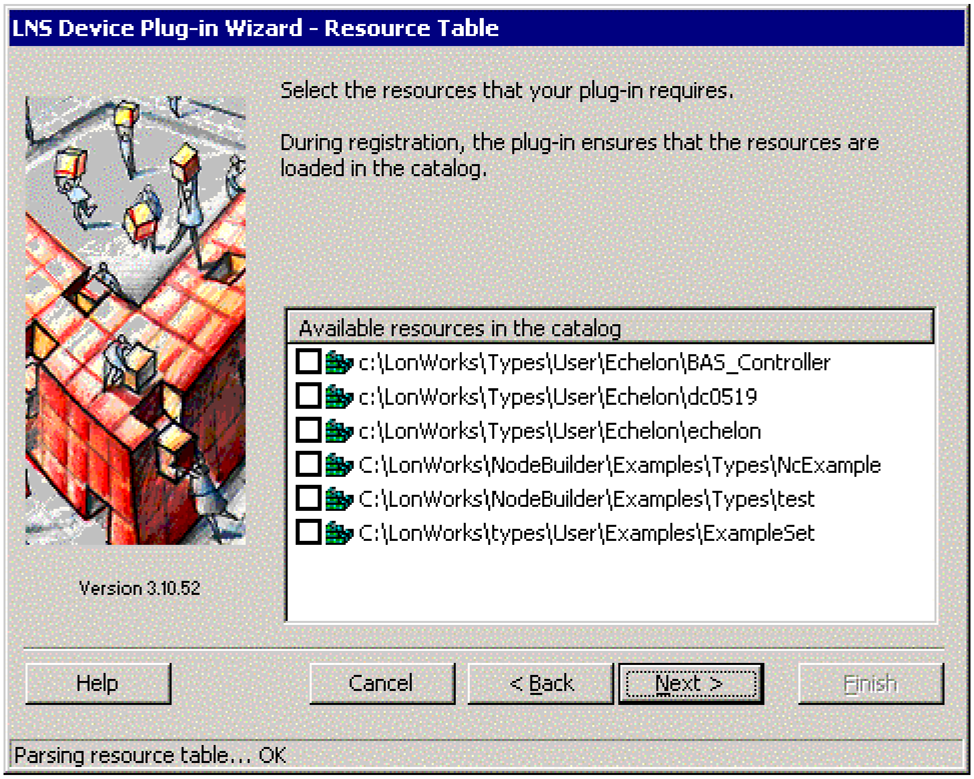
Command Description

Describes the command. This is optional, but can be used to provide additional documentation for the plug-in command.

Enter the command information and then click **OK**. The dialog closes and the command will be added to the **Command Table** window.

LNS Device Plug-in Wizard: Resource Table

The following window appears when you click **Next** from the Command Table window.



LNS Device Plug-in Wizard - Resource Table

Select the resources that your plug-in requires.

During registration, the plug-in ensures that the resources are loaded in the catalog.

Available resources in the catalog

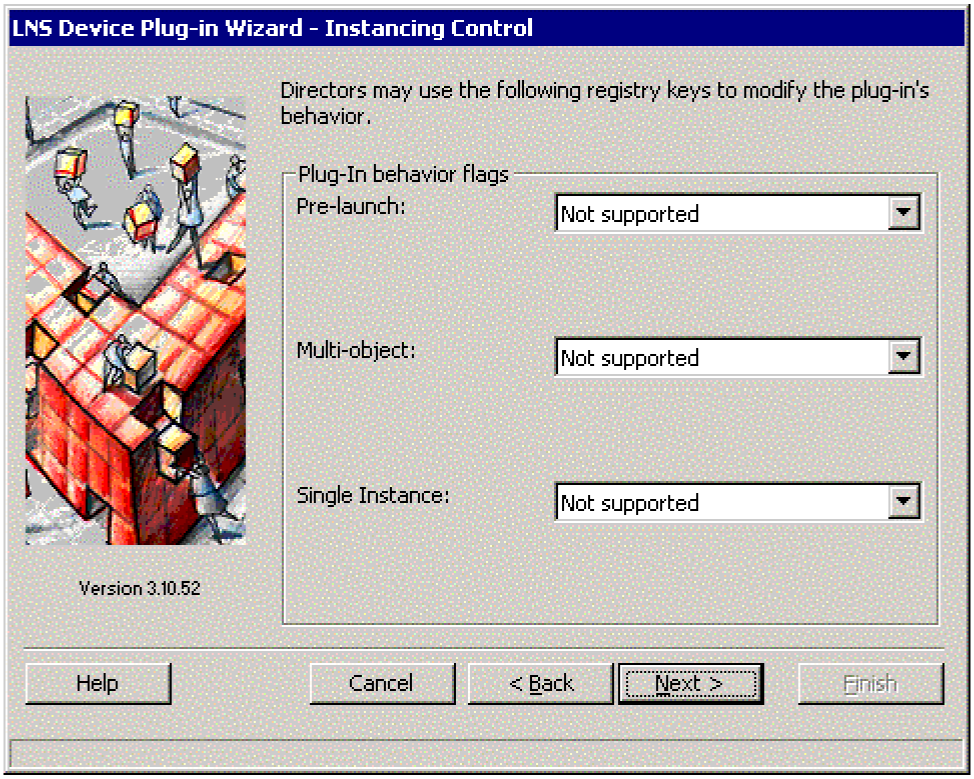
- ☐ c:\LonWorks\Types\User\Echelon\BAS_Controller
- ☐ c:\LonWorks\Types\User\Echelon\dc0519
- ☐ c:\LonWorks\Types\User\Echelon\echelon
- ☐ C:\LonWorks\NodeBuilder\Examples\Types\NcExample
- ☐ C:\LonWorks\NodeBuilder\Examples\Types\test
- ☐ C:\LonWorks\types\User\Examples\ExampleSet

Parsing resource table... OK

This window contains all resource files in the resource catalog available to the selected device template. You can select any resource files with a scope of 3, 4, 5, or 6. When the plug-in is registered with an LNS application, it will ensure that all selected resource files are available. Once you have selected all applicable resource files, click **Next** to open the Director-side Instancing Control window.

LNS Device Plug-in Wizard: Director-side Instancing Control

The following window appears when you click **Next** from the Resource Table window:


LNS Device Plug-in Wizard - Instancing Control

Directors may use the following registry keys to modify the plug-in's behavior.

Plug-In behavior flags

Pre-launch: Not supported

Multi-object: Not supported

Single Instance: Not supported

This window allows you to set options relating to optional plug-in behavior. Each of these fields can be set to **Not Supported**, **Supported**, or **Disabled**. The code generated by the plug-in wizard does not implement any of the optional behaviors, so you should typically leave all three behaviors set to **Not Supported**. If you choose to implement any of the optional behaviors, you must implement code for it. Enter the following:

Pre-launch Notifies LNS director tools that they should start the plug-in in the background as soon as the director starts. The LonMaker tool supports the pre-launching of plug-ins. This can improve the perceived startup performance of your plug-in.

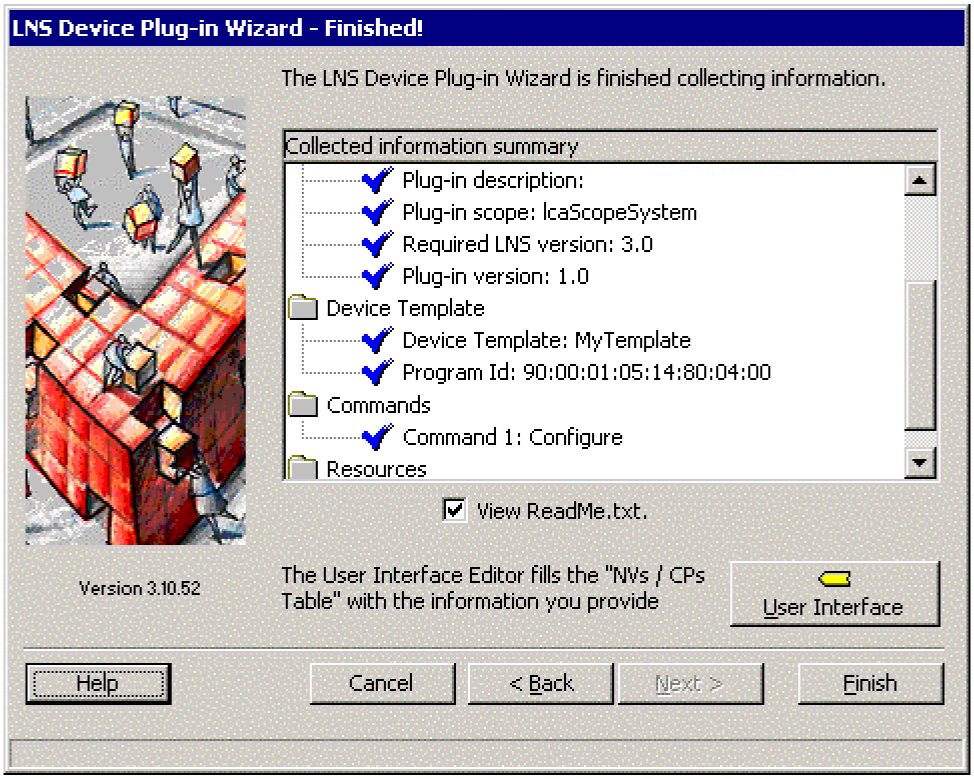
Multi-object Indicates if the plug-in accepts multiple SendCommand calls with different objects, i.e. if supported, this allows for the LNS device plug-in to be simultaneously open with multiple objects.

Single Instance Indicates if a separate instance of the plug-in will be launched for every new command applying to the same object.

Set any desired behaviors and click **Next** to open the Summary window.

LNS Device Plug-in Wizard: Summary

The following window appears when you click **Next** from the Director-side Instancing Control window:



LNS Device Plug-in Wizard - Finished!

The LNS Device Plug-in Wizard is finished collecting information.

Collected information summary

- Plug-in description:
- Plug-in scope: lcaScopeSystem
- Required LNS version: 3.0
- Plug-in version: 1.0

Device Template

- Device Template: MyTemplate
- Program Id: 90:00:01:05:14:80:04:00

Commands

- Command 1: Configure

Resources

☑ View ReadMe.txt.

Version 3.10.52

The User Interface Editor fills the "NVs / CPs Table" with the information you provide

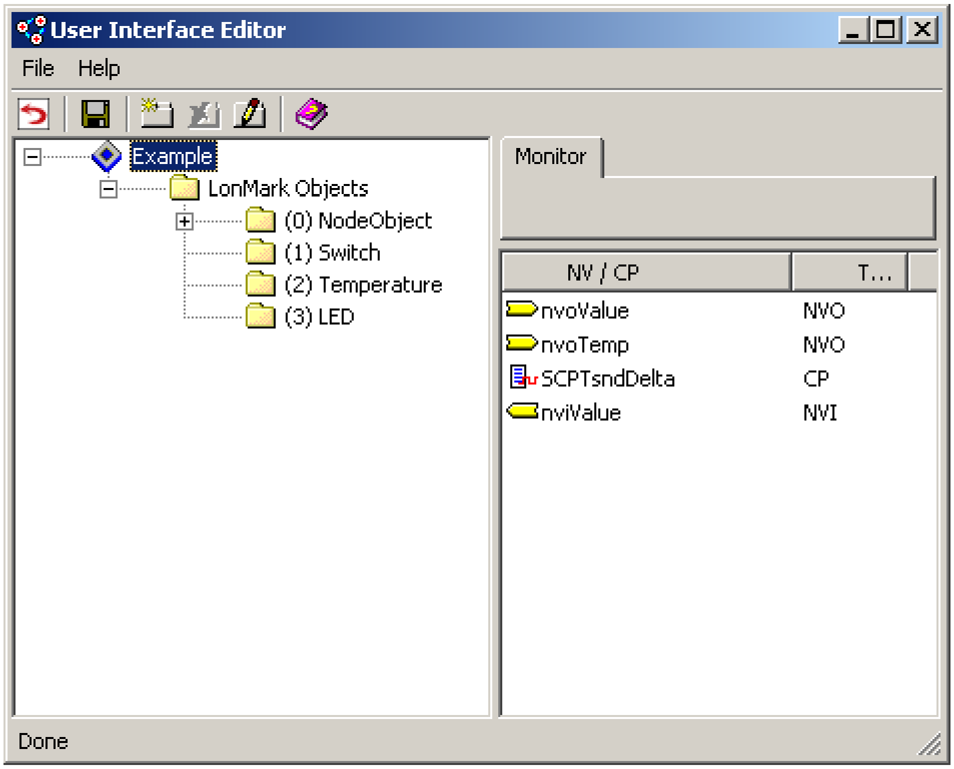
This window displays a summary of the information entered in the plug-in wizard. Review this information to ensure it is correct. Click **User Interface** to open the User Interface Editor window. You can use the User Interface window to build your initial user interface as described in the next section. Once you have finished using the User Interface Editor window, click **Finish** to exit the plug-in wizard and generate your plug-in framework.

LNS Device Plug-in Wizard: User Interface Editor

This window appears when you click **User Interface** from the Summary window.

WARNING: The user interface editor overrides existing code and controls for the `fMain` form, be careful and don't run it again if you need to preserve your existing development.

The User Interface Editor window appears as shown in the following figure. The left pane is the Application Interface pane. It displays all the functional blocks, network variables, and configuration properties in the selected device template. The right pane is the User Interface pane. It displays the network variables and configuration properties to be displayed on the tab selected above the User Interface pane.



NV / CP	T...
nvoValue	NVO
nvoTemp	NVO
SCPTsndDelta	CP
nviValue	NVI

This dialog allows you to add network variables and configuration properties to be configured, monitored, and controlled by your plug-in. The plug-in initially contains a single tab labeled **Monitor**.

To add additional tabs, click the Add Tab button ( ). If you are implementing commands to configure the functional blocks on your device, you can implement one or more tabs for each functional block.

To edit existing tabs, click the Edit Tab button ( ). For example, you can edit the default Monitor tab to change the name of the tab.

To remove existing tabs, click the Delete Tab button ( ).

When you add or edit a tab, the Tab Properties dialog appears.

To add a network variable or configuration property to a tab, follow these steps:

1. Select the desired tab above the User Interface pane.
2. Expand the interface in the Application Interface pane until the network variable or configuration property is visible.
3. Drag the network variable or configuration property from the Application Interface pane to the User Interface pane.

To clear all network variables and configuration properties from the User Interface pane, click the Reset button ( ) to clear all network variables and configuration properties.

Once you have added all the tabs for your LNS device plug-in, drag and drop the network variable and configuration property values you want available in each tab from the left pane of the **User Interface Editor**. This pane will contain all network variables and configuration properties for the device. Click the reset button (↶) to clear all network variables and configuration properties. Click the save button (💾) to generate visual basic code.

The monitoring done with an LNS Device Plug-in created using the LNS Device Plug-in Wizard is single-point monitoring, rather than monitor set monitoring (see the *LNS for Windows Programmer's Guide* for more information on the two types of monitoring). This method was chosen because each device that uses monitor set monitoring must have its own monitor set, and there is a limit of 8000 monitor sets per network. In very large networks where each device has a LNS device plug-in, this limit could be problematic. Single-point monitoring creates and removes monitor points dynamically, so there is no limit.

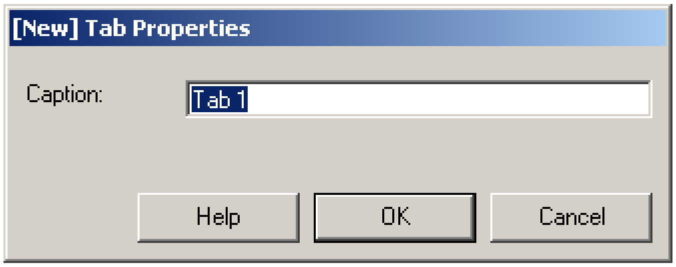
If you are creating an LNS Device Plug-in for a device that contains a configuration property that is shared between members of a functional block or network variable array (static sharing) or between different functional blocks or network variables (global sharing), you will see the configuration property in all functional blocks and network variable it is shared by. For example, a configuration property shared between all members array of functional blocks appears on all elements of that array. The configuration properties will be named *<CP name>* *<CP name>*~**1**, *<CP name>*~**2**, etc, but they all point to a single configuration property on the device. If you add more than one of these instances to the device interface, the user-interface of your Device Plug-in could get out of sync; when one is updated, the value won't be reflected in the others until the plug-in is refreshed. Therefore, whenever one instance of a shared configuration property is updated, all the other instances should be refreshed.

Once you are done with the **User Interface Editor**, click the save button, then close the editor. Note that if you have previously generated a user interface and made changes to the resulting code, these changes will be lost when the User Interface Editor is run again (this is not true of the LNS Device Plug-in Wizard in general). You will be returned to the final screen of the **LNS Device Plug-in Wizard**.

WARNING: If you have previously generated a user interface and made changes to the generated code, your changes will be lost when you click the **Save** button. This only applies to the **Save** button on the User Interface Editor window; you can run the rest of the plug-in wizard at any time without overwriting your existing code.

LNS Device Plug-in Wizard: Device Interface Editor Tab Properties

The following Tab Properties dialog appears whenever you add or edit a tab in the Device Interface Editor window:



[New] Tab Properties

Caption: Tab 1

Set **Caption** to the name that you want displayed on the selected tab.

3

Modifying Code Generated by the LNS Device Plug-in Wizard

The plug-in wizard creates a complete working plug-in application for you. This plug-in software is a fully functioning LNS device plug-in. You can customize this plug-in to tailor it to your device and further simplify device installation for your device installers and users. This chapter describes the code generated by the plug-in wizard and how you can customize it.

Introduction

The plug-in wizard generates the forms, modules, event handlers, and functions listed in Appendix E. You can modify the components shown in italics as described in this chapter. The plug-in wizard also generates a ReadMe file that lists all the forms and functions in the generated code.

The forms and modules generated by the plug-in wizard are a complete working plug-in. Even though changes are not required, you will typically modify the plug-in to customize it for your device. For example, your device may have configuration properties that have a limited set of valid values and combinations. Your plug-in can present just these valid values and combinations to the user, making it much easier for the user to set-up your device.

The only change that you have to make to the code generated by the wizard is to delete the MsgBox() function calls in the ProcessLonMarkObjectCommand() and ProcessDeviceCommand() functions within the fMain form. These MsgBox() calls identify where you can start adding your customization code.

Initializing the Device

You can add custom code to initialize your device. Your code should verify that the network is attached and the device is available before attempting to communicate with the device. The InitDevice() function in the fMain form determines the device state and collects initial values for items being monitored. You can add any required custom initialization code to this function.

Initializing Formats

You can add custom code to select the formats used to display and enter network variable and configuration property values. The ReadInitValue() function in the fMain form sets default formatting options for network variables and configuration properties, and retrieves initial values for configuration properties. The following code in this function is commented out. You can remove the comment marks to activate this code and enforce formats specified by your plug-in.

```
‘           ToDo: Uncomment the following block to enforce NV formats
'            If nv.DsFormatType <> g_NvsCpsTable(i).Format Then
'                nv.DsFormatType = g_NvsCpsTable(i).Format
'                nv.DsSaveOptions
'            End If

…

‘           ToDo: Uncomment the following block to enforce CP formats
'            If cp.FormatName <> g_NvsCpsTable(i).Format Then
'                cp.FormatName = g_NvsCpsTable(i).Format
'            End If
```

This code sets the formats to the formats specified in the g_NvsCpsTable array. These formats are written in the FillNvsCpsTable() function in the modFramework module. For example, the following statement sets the format to SNVT_tmpe_p#SI for the first entry in the g_NvsCpsTable:

```
g_NvsCpsTable(0).Format = "SNVT_temp_p#SI"
```

You can modify the assigned formats to any format name with the following syntax:

[#<*programIdTemplate*>**[**<*scope*>**].**]<*formatName*>[#<*modifier*>]"

The *programIdTemplate* and *scope* fields are an optional program ID template and scope of the resource file containing the format definition (both must be specified if either one is specified); *formatName* is the name of the format in the specified resource file, and *modifier* is an optional modifier. If the program ID template is not specified, the specified format must be defined in a resource file with a program ID template that matches the program ID of your device.

For example, "SNVT_switch" specifies a SNVT_switch format; "SNVT_temp_p#US" or "SNVT_temp_p#SI" specify a SNVT_temp_p format with a US or SI modifier

Alternatively, you can specify one of the following built-in types:

- **REAL** – The value will be formatted as a single-precision, 32-bit, IEEE 754 floating point number. This format may be used for any network variable that is 1, 2, or 4 bytes in length.
- **INT** – The value will be formatted as a signed, 32-bit integer. This format may be used for any network variable that is 1, 2, or 4 bytes in length.
- **DISCRETE** – The value will be formatted as an 8-bit value that contains either 0 or 1 for each bit. This format may be used for any network variable that is 1 byte in length.
- **BINARY** – The value will not be formatted. Byte and bitfield ordering is big endian.
- **RAW** – The value will be formatted as a text string. Each byte of the value appears as a text-formatted integer value from "0" to "255." Each byte is separate by the Raw Format Separator character, which is a tab character (0x09) by default.
- **RAW_HEX** – The value will be formatted as a text string. Each byte of the value appears as a text-formatted hex value from "0" to "ff." Each byte is separate by the Raw Format Separator character, which is a tab character (0x09) by default.

Customizing the Plug-in User Interface

You can modify the plug-in user interface by modifying the fMain form. The fMain form is where most of the wizard-defined and user-defined action takes place. For example, you can add additional user interface controls to this form. If you define monitor and control points using the User Interface Editor window, and then you can add Visual Basic or third party ActiveX controls to read and set the values in monitor and control point text boxes to provide a graphical interface to the monitor and control points.

The fMain form contains the following controls:

- 1 Tabbed Dialog Control 6.0 (SP5). This control accommodates the elements used by the monitoring and control feature provided by the plug-in.
- N text control(s): When generating a user interface with the user interface editor, a text control is added for each network variable and configuration property, grouped in tabs according to your selection in the user interface editor.
- Apply and Cancel buttons.

- An EchLog control: This is an ActiveX control named EchLog, similar to a ListBox. The EchLog control is used to record all user and system events.
- An EchStsBar control: This is an ActiveX control named EchStsBar, similar to a standard status bar. EchStsBar displays the device name, the device's state, the functional blocks, and status and alarm information for each functional block.
- Device Template Label. The device template label displays the name of the LNS device template the plug-in software applies to.

You can modify the functions listed in **Table 3-1** to customize your user interface.

Table 3-1. Typically Modified User Interface Functions

Function	Purpose
ProcessLonMarkObjectCommand() and ProcessDeviceCommand() Command Handlers	These functions are handlers that process a command that applies to a functional block (LcaLonMarkObject), or an entire device (LcaAppDevice), respectively. The default implementation for each function contains code to perform a complete resource file look-up to adjust the scope of the implemented functional blocks. It also contains pre-defined code to set the device template label and to initialize the device. The default implementation also contains a MsgBox() function call, reminding you that these handlers are subject to review and enhancement; at a minimum, you must remove the MsgBox() function calls. You can add code to enhance the plug-in functionality and to tailor the plug-in software to meet the requirements of your device and its intended user.
txtNvi_KeyPress() and txtCp_KeyPress() Event Handlers	These event handlers mark changes in the relevant text controls when using the wizard-generated user-interface. The plug-in code tries to commit pending changes when Apply is clicked. You must add similar handlers for any controls that you add to the interface and were therefore not added by the plug-in wizard. You must add code for your controls to commit pending changes from within the Apply handler.
cmdApply_Click() and cmdCancel_Click() Functions	These functions commit or cancel the changes pending in the altered text controls. You can add your Apply and Cancel methods if you add any controls that are not added by the plug-in wizard. For example, you may add controls for monitoring and control network variable fields or configuration property fields.
EnableControls() Function	This function prepares the text controls that are used for monitoring. You must add your own code if you add any controls that are not added by the plug-in wizard.

EXAMPLE:

For example, you can add a Button control to the NodeBuilder Quick-start tutorial plug-in example that causes the light on the Gizmo 4 hardware to toggle when the button is clicked. To do this, drag a Button control to the fMain form, name the new control Toggle Light, and add the following code to the fMain code:

```
'Clicking this button toggles the input to the LED functional
'block by updating the nviValue field created by the User
'Interface Editor, then clicking the Apply button.

Private Sub cmdToggleLight_Click()
    If txtNvi(0) = "100.0 1" Then
        txtNvi(0) = "0.0 0"
        cmdApply.Enabled = True
        txtNvi(0).ForeColor = DIRTYCOLOR
        m_UnsavedChanges = True
        cmdApply_Click
    Else
        txtNvi(0) = "100.0 1"
        cmdApply.Enabled = True
        txtNvi(0).ForeColor = DIRTYCOLOR
        m_UnsavedChanges = True
        cmdApply_Click
    End If

End Sub
```

This code sets the m_UnsavedChanges variable to True, the text color to DIRTYCOLOR, and enables the cmdApply button. These operations mimic the behavior of the text box when text is entered manually (see the Keypress procedure for the txtCp, txtNvo, and txtNvi textboxes). See *Updating and Output Network Variable or Configuration Property* for more details.

Changing the Start-up Tab

You can change the tab that is displayed when the plug-in starts. The SelectActiveTab() function in the fMain form sets the tab to a default page. If all network variables and configuration properties that belong to a single functional block are hosted on the same tab page, the SelectActiveTab() function automatically selects this page whenever it receives a command associated with the relevant functional block. If you have network variables and configuration properties from the same functional block on different pages, you may need to modify the SelectActiveTab() function to select the appropriate tab for each functional block.

Reading and Monitoring a Network Variable

You can use the plug-in wizard User Interface Editor window to add network variables to be monitored by your plug-in. The plug-in wizard generates a single monitor point for each input network variable that you select. Monitor sets are not used (see the *LNS for Windows Programmer's Guide* for more information on the two types of monitoring). Single-point monitoring method is used because each device that uses monitor set monitoring must have its own monitor set, and there is a limit of 8000 monitor sets per network. Device plug-ins should typically use single-point monitoring to conserve monitor sets for use by HMI applications. Single-point monitoring creates and removes monitor points dynamically, so there is no limit. You can customize your plug-in to use monitor sets, but if you do that you must provide an option for the user to delete monitor sets since a very large network could reach the 8000 monitor set limit.

The FillNvsCpsTable() function in the modFramework module builds a g_NvsCpsTable array containing an entry for each network variable and configuration property that you added to the user interface using the user interface wizard. When a network variable update is received by your plug-in, the following steps occur:

1. The LNS network operating system calls the lcaOS_OnNetworkVariableUpdate() event handler in fMain.
2. The lcaOS_OnNetworkVariableUpdate() event handler calls the NetworkVariableUpdateReceived() function with the updated network variable value and the entry in the g_NvsCpsTable array that corresponds to the updated network variable. The g_NvsCpsTable entry contains a reference to the user interface control associated with the network variable.
3. The NetworkVariableUpdateReceived() function verifies that the referenced control is a text box. If it is, it assigns the new network variable value to the text box.

If you need access to a network variable value, you can get the value from the Text Box control Text property. For example, if the name of the Text Box control is txtNvo, the txtNvo.text property contains the string value of the network variable.

You can update a custom control that you add to the plug-in with a network variable update. You can do this by creating a Change event handler for the text box control and updating your custom control from this event handler.

EXAMPLE:

The following event handler copies new values for the txtNvo control to a slider control.

```
Private Sub txtNvo_Change(Index As Integer)
    SldNvo.value = Val(txtNvo.text)
End Sub
```

Alternatively, you can manually create a monitor point for the network variable to be monitored, assign it a unique monitor tag, modify the FillNvsCpsTable() function in the modFramework module to create a new entry in the g_NvsCpsTable array with an index equal to the monitor tag - 1, and modify the NetworkVariableUpdateReceived() function to update your control.

WARNING: Do not rerun the user interface editor if you add custom entries into the g_NvsCpsTable array.

The tabData_Click() event handler in the fMain form is activated when the tab page changes. This event handler ensures that monitoring is enabled only for items on the currently visible tab page. You can modify this code if you wish to permanently monitor any network variables. Be sure to manage your network bandwidth utilization if you permanently enable monitoring on many network variables.

Reading a Configuration Property

You can use the plug-in wizard User Interface Editor window to add configuration properties to be accessed by your plug-in.

The FillNvsCpsTable() function in the modFramework module builds a g_NvsCpsTable array containing an entry for each network variable and configuration property that you added to the user interface using the user interface wizard.

If you need access to a configuration property value, you can get the value from the Text Box control Text property. For example, if the name of the Text Box control is txtCp, the txtCp.text property contains the string value of the configuration property.

EXAMPLE:

The following statement copies the value for the txtCp control to a slider control.

```
SldNvo.value = Val(txtCp.text)
```

If you are creating a LNS device plug-in for a device that uses a static configuration property in a functional block array (see *Adding a Configuration Property to the Device Interface* in the *Generating Neuron C Code Using the Code Wizard* chapter in the *NodeBuilder User's Guide*), you will see the configuration property in all functional blocks in the array. The configuration properties will be named `<CP name>` `<CP name>~1`, `<CP name>~2`, etc. These all really point to the same configuration property. If you add more than one of these to the device interface, your plug-in may get out of sync; when one is updated, the value won't be reflected in the others until the plug-in is reloaded.

Updating an Output Network Variable or Configuration Property

You can use the plug-in wizard User Interface Editor window to add network variables and configuration properties to be updated by your plug-in. To update a network variable or configuration property, follow these steps:

1. Update the value of the text box.
2. Set the Enable property of the cmdApply control to True.
3. Set the ForeColor property of the text box to DIRTYCOLOR.
4. Set the m_UnsavedChanges variable to True.
5. Call the cmdApply_Click() event handler.

EXAMPLE:

The following code updates the value of the txtNvi text box and its associated network variable:

```
txtNvi(0) = "0.0 0"
cmdApply.Enabled = True
txtNvi(0).ForeColor = DIRTYCOLOR
m_UnsavedChanges = True
cmdApply_Click
```

Changing a Network Variable Type

Your plug-in can change the type of network variables that support changeable types. To implement support for a changeable type network variable, follow these steps:

1. Create a device with a changeable type network variable as described in the *NodeBuilder User's Guide* and the *Neuron C Programmer's Guide*.
2. Add a user interface to your plug-in that allows the user to select different types for the network variable. For example, you can add a ComboBox control and code that initializes the control with the network variable types supported by your application. Alternatively, you can add two controls, one that allows the user to

select a resource file and one that allows the user to select a type from the selected resource file.

3. Add an event handler for the network variable type control. This event handler must perform the following functions:

   a Write the new type to the SCPTnvType configuration property associated with the network variable. The following example describes the fields in a SCPTnvType configuration property.

   EXAMPLE:

   The following code updates a SCPTnvType configuration property defined at index *R* of the g_NvsCpsTable array. This index may be acquired from the g_NvsCpsTable initialization within the modFramework module.

```
Dim cp As LcaConfigProperty
Dim cps As LcaConfigProperties
Dim devInterface As LcaInterface

Set devInterface = m_Device.Interface
Set cps = devInterface.ConfigProperties
Set cp = cps.ItemByHandle(g_NvsCpsTable( R ).ItemHandle)

cp.Value = "PID S:S:S:S:S:S:S:S, Scope T, Index U, V, W bytes, A=a, B=b, C=c"
```

   The formatted string used to update the SCPTnvType configuration property is based on the following symbol mapping to the SCPTnvType structure.

Symbol	Mapped Field	SCPTnvType Structure

S	type_program_ID[8]	typedef struct {
		unsigned short type_program_ID[8];
		unsigned long type_scope;
		unsigned short type_index;
		nv_type_category_t type_category;
		unsigned short type_length;
		signed long scaling_factor_a;
		signed long scaling_factor_b;
		signed long scaling_factor_c;
		} SCPTnvType;

T	type_scope	
U	type_index	
V	type_category	
W	type_length	
a	scaling_factor_a	
b	scaling_factor_b	
c	scaling_factor_c	

   The **type_program_ID** and **type_scope** values specify a program ID template and a resource scope that together uniquely identify a resource file set. The **type_index** value identifies the network variable type within that resource file set. If the **type_scope** value is 0, the **type_index** value is a SNVT index. The **type_program_ID** and **type_scope** values uniquely identify a type to the device application as well as to any network tools that wish to determine the current type, or modify the type, of the network variable to which the property applies. The device application may ignore these values if the remaining fields in the **SCPTnvType** structure provide sufficient information for the application.

   The **type_category** field is defined by the following definition from the <snvt_nvt.h> include file and requires that it equal the enumerated symbol rather than an integer value.

```
typedef enum nv_type_category_t {
   /*  0 */  NVT_CAT_INITIAL = 1,       // Initial unassigned value
   /*  1 */  NVT_CAT_SIGNED_CHAR = 1,   // Signed Char
   /*  2 */  NVT_CAT_UNSIGNED_CHAR,     // Unsigned Char
   /*  3 */  NVT_CAT_SIGNED_SHORT,      // 8-bit Signed Short
   /*  4 */  NVT_CAT_UNSIGNED_SHORT,    // 8-bit Unsigned Short
   /*  5 */  NVT_CAT_SIGNED_LONG,       // 16-bit Signed Long
   /*  6 */  NVT_CAT_UNSIGNED_LONG,     // 16-bit Unsigned Long
   /*  7 */  NVT_CAT_ENUM,              // Enumeration
   /*  8 */  NVT_CAT_ARRAY,             // Array
   /*  9 */  NVT_CAT_STRUCT,            // Structure
   /* 10 */  NVT_CAT_UNION,             // Union
   /* 11 */  NVT_CAT_BITFIELD,          // Bitfield
   /* 12 */  NVT_CAT_FLOAT,             // 32-bit Floating Point
   /* 13 */  NVT_CAT_SIGNED_QUAD,       // 32-bit Signed Quad
   /* 14 */  NVT_CAT_REFERENCE,         // Reference
   /* -1 */  NVT_CAT_NUL = 1            // Invalid Value
} nv_type_category_t;
```

This enumeration describes the type, stating whether it is a signed short, or floating-point, or structure, for example, but not providing information about structure or union fields or other similar details. The **type_length** field is necessary to provide the number of bytes of a structure or union type, though it is set for all types. To support all scalar types, test for a type_category value between NVT_CAT_SIGNED_CHAR and NVT_UNSIGNED_LONG, plus NVT_CAT_SIGNED_QUAD. To also support floating point types, also test for a **type_category** value of NVT_FLOAT.

The **scaling_factor_a**, **scaling_factor_b**, and **scaling_factor_c** fields enable the device application to convert raw fixed-point network variable values to scaled values. For example, the **SNVT_lev_cont**, type represents percentages from 0 to 100 percent, with a resolution of 0.5%, in an unsigned short. The actual data values (also called *raw* values) in the variable range from 0 to 200. The scaling factors for **SNVT_lev_cont** are a=5, b= -1, c=0. To convert a raw fixed-point value to a scaled value, use the following formula:

scaled = (*a* * (10 ** *b*) * (*raw* + *c*))

The device application may also use the scaling factors to convert scaled values back to raw values to be stored in the network variable. To convert from a scaled value to a raw fixed-point value, use the following formula:

raw = (scaled / (*a* * (10 ** *b*))) - *c*

It is entirely up to the device application to programmatically deal with varying representations of data based upon the current values in the SCPTnvType property for the network variable. This is described in *Changeable Type Network Variables* in Chapter 3, *How Devices Communicate Using Network Variables*, of the *Neuron C Programmer's Guide*.

b Update the LNS Property **LcaNetworkVariable::SNVTid** for the network variable. Set this property to the standard network variable identifier for a SNVT or 0 (zero) for a user-defined network variable type.

c If you were using non-default formatting, or want to use non-default formatting for the network variable, set the LNS Property **LcaNetworkVariable::DsFormatType** to reflect the new format.

d Refresh the display of the network variable, so that the format that is associated with the new type is used.

e If the network variable has configuration properties associated with it, and if these configuration properties inherit their types from the network variable, refresh the display of these properties. This allows the correct format to be used as required by the new type.

f If the changeable type network variable shares a SCPTnvType property with other network variables, repeat steps b through e for each network variable.

g If attached and OnNet, check the status reported on the device's **nvoStatus** network variable (see the **LcaObjectStatus** object for details). An invalid request indicates that the device does not support the desired type. In this case, revert back to the previously known, valid type, using the procedure as outlined above.

Type rejection and roll-back requires the device to be available, attached and accessible, and requires the network database to be in OnNet management mode.

Accessing a Network Variable Object

You can directly access the LNS Network Variable object for any network variable on the device associated with the plug-in. You can use this object to get or set the current value of a network variable or to control its formatting. The m_Nvs collection contains all the network variables for the plug-ins device, so you can access the Network Variable object using the ItemByIndex() function for the m_Nvs collection.

WARNING: If you update the Value property of a LNS Network Variable object directly without also updating the displayed value in your plug-in interface, the displayed and actual value will get out of synchronization. Use the technique described in *Updating an Output Network Variable* to update both the network variable and its value in the user interface.

EXAMPLE:

The following code fetches the Network Variable object for the network variable assigned to the i'th entry in the g_ NvsCpsTable array (this entry corresponds to monitor tag i+1) and then increments its value by 1.

```
Dim nv As LcaNetworkVariable

Set nv = m_Nvs.ItemByIndex(Val (g_NvsCpsTable(i).ItemHandle))
Set nv.Value = CStr(Val(nv.Value) + 1)
```

The **g_NvsCpsTable** function supports an **ItemHandle** property. This property should not be confused with any of the LNS Handle properties; it is a plug-in framework-specific handle. (For network variables, **ItemHandle** is set to the network variable index.)

The **ItemHandle** property is a string-type property. A function such as the **Val()** conversion function must be used to convert the string-encoded network variable index.

Accessing a Configuration Property Object

You can directly access the LNS Configuration Property object for any configuration property on the device associated with the plug-in. You can use this object to get or set the current value of a configuration property or to control its formatting. The m_Device object contains the LNS Device object for the plug-ins device. This object contains an Interface property that stores the device interface. This property contains a ConfigProperties collection with the configuration properties on the device.

The LNS Plug-In Wizard-generated framework uses the **GetCpByTypeIndex** routine to fetch a configuration property object. This routine, defined in fMain.bas, accepts a string identifying the configuration property by scope selector and index. This string is formatted in one of the following ways:

Device Configuration Property:

`C:<CPT scope>:<CPT index>`
e.g. `C:3:15`

Functional Block Configuration Property:

`F:<FB index>/C:<CPT scope>:<CPT index>`
e.g. `F:4/C:3:1`

Network Variable Configuration Property:

`N:<NV index>/C:<CPT scope>:<CPT index>`
e.g. `N:23/C:3:5`

The **ItemHandle** property in **g_NvsCpsTable** stores this string.

WARNING: If you update the Value property of a LNS Configuration Property object directly without also updating the displayed value in your plug-in interface, the displayed and actual value will get out of synchronization. Use the technique described in *Updating a Configuration Property* to update both the network variable and its value in the user interface.

EXAMPLE:

The following code fetches the Configuration Property object for the configuration property assigned to the i'th entry in the g_ NvsCpsTable array, and then increments its value by 1.

```
Dim cp As LcaConfigProperty

Set cp = GetCPByTypeIndex(g_NvsCpsTable(i).ItemHandle)
Set cp.Value = CStr(Val(cp.Value) + 1)
```

Using Shared Configuration Properties

If your plug-in exposes a shared configuration property in multiple places, when you update one instance of the shared configuration property, the other instances may not reflect the change immediately. You should add code that refreshed the other instances of the shared configuration property when any instance is updated.

See *Sharing a Configuration Property* in Chapter 6 of the *NodeBuilder User's Guide* for more information about sharing configuration properties

Adding Support for Configuration Property Arrays

The LNS Plug-in Wizard does not support automatic generation of user interface and code to monitor and control configuration properties that are implemented as configuration property arrays. When a configuration property array is added to one of the tabs in the Plug-in Wizard User Interface Editor (see Chapter 2), the generated code will affect only the first element of the configuration property array.

To add support for monitoring and control of configuration property arrays within the Plug-In Wizard-generated plug-in framework, follow these steps:

1. Use the Plug-In Wizard to create a user-interface. By default, this user interface will only allow you to access the first element of that array.
2. Open Mmain.bas and edit the **NvCpTableType** structure to include information about the array index or the range or array indices to be controlled from the graphical user interface.
3. Open ModFramework.bas and edit the **FillNvsCpsTable()** function to use the fields you have added to the **NvCpTableType** structure.
4. Open fMain.frm and modify the user interface to accommodate your chosen method of configuration property array access. This might require removing a Plug-In Wizard-generated control (i.e. a textbox), and adding controls that meet your requirements. For a small array, the user interface could be a small number of textbox controls, one for each element of the array. For a larger array, you could implement a spin control that allows the user to select index into the array, and a textbox control that allows the value at the selected index to be read and changed.
5. In fMain.frm, optionally edit the **OnLoad** function to initialize your newly added controls. For example, if you implement a spin control that allows the user to select an index of the configuration property array, you will need to initialize a starting index and define the index boundaries will have to be set. A separate function, **ReadInitValues()**, is used to obtain initial network variable and configuration property values.
6. In fMain.frm, edit the **ReadInitValues()** function to read data from your **NvCpTable** as required by your user interface. The LNS method **LcaConfigProperty::GetElement()** is used to obtain the value of an element of a configuration property array.
7. In fMain.frm, edit the **cmdApply_Click** event handler to update the values in your configuration property array as defined in your NvCpTable. The LNS method LcaConfigProperty::SetElement() is used to set individual elements of a configuration property array.
8. Add user interface event handlers as needed by your controls. For example, implementing a spin control that allows the user to select an element of the configuration property array requires an **OnChange** event handler to respond to the newly selected index.

Adding a Custom Property, Method, or Command

The LNSPlugInAPI class module defines the standard properties and methods that are implemented by your plug-in. These properties and methods are used by network tools to configure and call the plug-in. You can add custom properties and methods to the LNSPlugInAPI class.

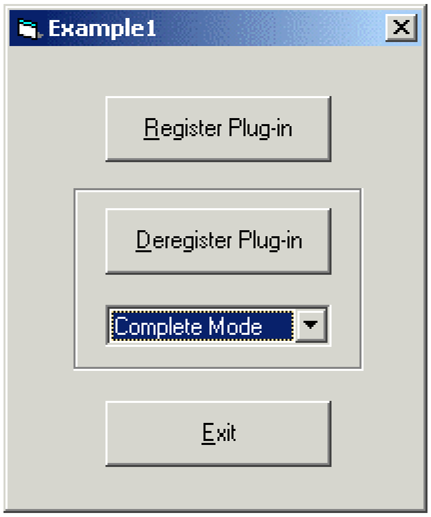
When a command is received, the SendCommand() function in the LNSPlugInAPI class calls the appropriate functions in the plug-in. The SendCommand() function looks up commands in the g_CommandTable array, which is created by the FillCommandTable() function in the modFramework module. The FillCommandTable() function is created automatically based on your inputs to the plug-in wizard. If you need to add any custom commands, use the plug-in wizard to add them.

Creating a Stand-alone Plug-in

The plug-in wizard generates a plug-in that may be called by a network tool, and that may also run standalone. By default, the standalone operation is limited to providing registration and deregistration commands.

When run stand-alone, the plug-in automatically registers itself if it is started with a command line flag of `/regplugin`, such as `MyPlugIn /regplugin`. The registration information used comes from the project and from constants entered into the plug-in wizard and contained in the `modFramework` module.

If a plug-in created using the plug-in wizard is started stand-alone without any command switches , the registration dialog appears, as shown in the following figure:



You can set the project name and constants by typing the desired values into the code, or you can use the plug-in wizard to generate the value.

You can enhance the plug-in to provide additional standalone capability. The fRegisterPlugIn form provides the standalone interface for the plug-in. You can modify this form to provide additional standalone functionality.

Note: In Visual Basic, the name of your plug-in's server class is always the name of your project (as set in the project properties dialog box) followed by a dot and the name of the class that implements the LNS Plug-in API. For example, if your project name is MyPlugIn and your class that implements the LNS Plug-in API is named LNSPlugInAPI, the name of your plug-in's server class (and thus the value for g_RegisteredServer) must be MyPlugIn.LNSPlugInAPI. The LNS Device Plug-in Wizard automatically sets the correct value for this constant. If you change the project name manually, you must either rerun the LNS Device Plug-in Wizard or manually update the value of g_RegisteredServer.

Click the **Deregister Plug-in** button to remove the plug-in's registration command from the Windows registry. When deregistering a plug-in, you can choose from three deregistration modes using the drop-down menu beneath the **Deregister Plug-in Button**:

Minimal Mode Removes the entries for this plug-in from the Windows registry and unregisters the automation server.

Complete Mode Does everything a Minimal Mode deregistration does, plus removes plug-in registration data from all network databases including the global database.

Exhaustive Mode Does everything a Complete Mode deregistration does, plus removes all device templates that this plug-in uses from all network databases unless they are in use (i.e. a device of that type exists in the network).

To re-register a plug-in that you have de-registered, you must quit the plug-in and restart it before invoking the register command.

Note: Do not try to deregister the plug-in when it is run stand-alone from the Visual Basic debugger. This will cause the deregistration to fail.

Using Utility Functions

The plug-in framework provides a number of utility functions, listed in Table 3-2. You can call these functions from your custom code.

Table 3-2 Utility Functions Provided by the Plug-in framework

Function	Purpose
GetClassIdName()	This function is defined in the MMain module. Given a class ID, this function returns the name of the class in a string.
GetCommandIdName()	This function is defined in the MMain module. Given a command ID, this function returns the name of the command in a string.
GetNumObjectNameParameterStrings()	This function is defined in the MMain module. Returns the number of substrings in the last object name passed to the plug-in by a director.
GetObjectNameParameterString()	This function is defined in the MMain module. Returns the i^{th} substring of the last object name passed to the plug-in by a director and optionally trims the class identifier, if any.
ParseWord()	This function is defined in the MMain module. Given a string and a character to search for, returns all of the characters before the search character as the result of the function (or the entire input string if the search character is not found) and returns all of the characters after the search character (or an empty string if the search character is not found) in the RemainingString parameter.

Design Rules For Future Revisions Of Your Plug-ins

If your plug-in is a new version of a plug-in that you have previously distributed, there are several rules that you must follow. Some of these rules are imposed by Windows; others are imposed by LNS.

As mentioned in Footnote 5 in Chapter 4, Windows requires that ActiveX servers register certain information about themselves in the Windows registry. One side-effect of these requirements are that, if the user does not uninstall previous versions of your plug-in and installs a new version into a different directory than the previous version, Windows may not be able to find your plug-in when directors try to launch it. This phenomenon is caused by the way in which Windows locates and launches ActiveX servers.

When a director creates a new instance of a plug-in (referenced by name), Windows searches in the registry for an ActiveX class with the specified name. If a matching name is found, Windows fetches the class's GUID from the registry. Windows then looks in the registry for a Class ID with the specified GUID. If one is found, Windows next looks at the data setting of `LocalServer32` value, which specifies the full path and file name of the executable that implements the class that the director asked to launch.

This can cause the following problem. If the user does not uninstall the old version of your plug-in, its GUID will still be in the registry when your new version is installed. If the new plug-in has the same GUID as the old plug-in, Windows will not create the proper registry entries when the new plug-in is installed (since it will think that the plug-in is already registered). If the user installs the new version into a different directory than the previous version, Windows will not have the correct path information in the `LocalServer32` value.

There are two easy ways to prevent this problem. One way is to give new versions of your plug-ins a new GUID. This is done by setting the `Project Compatibility` option (selected using the `Component` tab of the project properties dialog) to `Project Compatibility` or `No Compatibility`. When project compatibility is set to either of these options, Visual Basic automatically generates a new GUID for your server class every time you compile. The other way to avoid this problem is to give the new version of your plug-in a new ActiveX class name. This is done by changing the name of the registered server (i.e. the name of the Visual Basic project) using either the plug-in wizard or the Visual Basic Project Properties dialog. The new plug-in can have the same executable name as the old plug-in; only the project name needs to be changed.

The second rule you must follow has to do with LNS. If the new version of your plug-in does not implement the same commands as the old version (i.e., if the command set is not backwards compatible) you must be sure to deregister old versions of your plug-in as part of registration of your new plug-in. That is, when your plug-in receives an `LcaCommandRegister` (50) command from a director, before you add `ComponentApp` objects for your new plug-in’s commands, you must remove any `ComponentApp` objects that list your plug-in as the `RegisteredServer` and that indicate commands that you no longer support. Since you know what commands have become obsolete in the new version, it is a straight-forward task to find and remove them. See the `DeregisterSupportedCommands`() function in the `MMain` module of the plug-in framework for an example.

4

How Plug-Ins Work — The Big Picture

This chapter presents an overview of the life-cycle of a plug-in. It describes how plug-ins make themselves known to directors, how plug-ins let directors know what they can do, and how directors and plug-ins interact. Once you understand the basic interaction between directors and plug-ins, you will find that writing a plug-in using Visual Basic is quite straightforward. The plug-in framework generated by the plug-in wizard implements the interfaces described in this chapter. You can skip this chapter if you are using the plug-in wizard and you do not need to modify the standard plug-in interfaces.

How Plug-ins Are Represented In The LNS Object Server

Before discussing how plug-ins are installed on a user's computer and how they are registered with the LNS Object Server, it helps to understand what the end result of these actions is. That is, when all is said and done, how does the LNS Object Server represent a plug-in and its functionality?

The LNS Object Server represents a plug-in and its functionality through `ComponentApp` objects. There is one `ComponentApp` object for each action that a plug-in implements and one `ComponentApp` object for the plug-in itself. Each `ComponentApp` object defines, among other things, an action, which is a command/object class pair. That is, it defines a specific command (such as `LcaCommandTest`) that the plug-in can perform on a specific class of object (such as an `AppDevice`). This means that if a plug-in can perform the same command on many different classes of objects, it represents this by multiple `ComponentApp` objects (all with the same command ID, but each with a different object class ID). For example, a plug-in might implement two actions, a test command of `AppDevice` objects and a test command of `Router` objects. Each of these actions would be represented by a `ComponentApp` object (one with a command ID/object class pair of [`LcaCommandTest` (33), `LcaClassIdAppDevice` (7)] and the other with command ID/object class pair of [`LcaCommandTest` (33), `LcaClassIdRouter` (9)].

Similarly, if a plug-in can perform many different commands on a particular class of object, each of these commands is represented by a separate `ComponentApp` object (all with the same object class ID, but each with a different command ID). For example a plug-in might implement two actions, a Test command of `AppDevice` objects and a Configure command of `AppDevice` objects. Each of these actions would be represented by a `ComponentApp` object (one with a command ID/object class pair of [`LcaCommandTest` (33), `LcaClassIdAppDevice` (7)] and the other with command ID/object class pair of [`LcaCommandConfigure` (13), `LcaClassIdAppDevice` (9)].

The plug-in itself is also represented in the LNS Object Server by a `ComponentApp` object. For the plug-in itself, the command ID of the `ComponentApp` object is always `LcaCommandRegister` (50).

Four classes of objects in the LNS Object Server contain `ComponentApps` collections — the `ObjectServer` object, `System` objects, `DeviceTemplate` objects, and `LonMarkObject` objects. The `ComponentApps` collection to which a `ComponentApp` object belongs defines the scope of the item.

If an item is in the `ObjectServer` object's `ComponentApps` collection, the item applies to all systems managed by the LNS Object Server. If an item is in a System object's `ComponentApps` collection, the item applies only to that system. If an item is in a `DeviceTemplate` object's `ComponentApps` collection, the item applies only to devices of that type in that particular system. If an item is in a `LonMarkObject` object's `ComponentApps` collection, the item applies only to functional blocks on devices of that type in that particular system.

How Plug-ins Are Installed On A User's Computer and Registered with LNS

To the user, installing a plug-in is a simple process. As with all Windows applications, the user runs a setup program (such as one created using Visual Basic Application Setup Wizard) that copies all of the files required by the application onto their machine and that makes whatever changes in the Windows registry the application requires. The next time that the user run a director, the new plug-in will be listed as available[4] and the user can then register the plug-in with LNS. Once it is registered, the user can then use the plug-in's functionality and, depending upon how the director has exposed plug-ins, may not even be aware that they are doing so. As with all software, much is going on behind the scenes to make this process easy for the end-user.

When the user runs the plug-in's setup program to install the plug-in onto their computer, a number of things happen. As with all Windows applications, the setup program copies all of the files needed to run the plug-in onto the user's computer while checking to make sure that it does not overwrite newer versions. Next, the setup program adds a number of items to the Windows registry. Some of these items are required because plug-ins are implemented as ActiveX servers[5]. Other entries are required by LNS.

LNS requires that a plug-in register a registration command for itself in the Windows registry[6]. This item provides background information about the plug-in (such as its version number and the name of the company that wrote it) along with operational information (such as the name of the plug's ActiveX class that implement the LNS Plug-in API and a description of what the plug-in does). Directors use this information (along with the other registry information required of all ActiveX components) to complete the plug-in installation process.

There are several different ways that a plug-in's setup program can add all of this required information to the Windows registry. One way is for the setup program to explicitly create all of the required registry information. Some setup toolkits can create setup applications that can do this, but the Visual Basic Package and Deployment Wizard does not support this. A plug-in can also be registered by being run in stand-alone with the `/regplugin` switch[7] after installing all of its required

[4] How a director treats new plug-ins is up to the director. Some may display them in list boxes and let the user select which ones are imported; others may automatically import all new plug-ins (although this not recommended since it can result in lots of time spent registering plug-ins that the user does not want to use).

[5] Windows requires that ActiveX components register a number of items about themselves. This includes registering the GUID, the name of the ActiveX class that implements the LNS Plug-in API (the registered server), and the full path and name of the application that implements the registered server. The GUID is in the `CLSID` value under the keys `HKEY_CLASSES_ROOT\<server class name>` and `HKEY_LOCAL_MACHINE\SOFTWARE\Classes\<server class name>`, such as `HKEY_CLASSES_ROOT\MyPlugIn.LNSPlugInAPI` and `HKEY_LOCAL_MACHINE\SOFTWARE\Classes\MyPlugIn.LNSPlugInAPI`. The executable name that implements the class is registered in the `LocalServer32` value under the key `HKEY_CLASSES_ROOT\CLSID\<GUID>`, such as `HKEY_CLASSES_ROOT\CLSID\{52816344-5E5B-11D1-865D-0060974528A0}`.

[6] The required information is described fully in Chapter 1, Introduction.

[7] ActiveX servers can run in two modes, server or stand-alone. A plug-in is running in server mode when it is launched by another application (i.e., a director) though ActiveX automation. It is running stand-alone when it was launched without ActiveX automation (for example, by the user double clicking on the plug-in's icon).

files[8]. LNS requires that plug-ins add their registration command to the Windows registry when they are run stand-alone with the `/regplugin` switch. Plug-ins created using the Plug-in Wizard automatically show a dialog that allows them to be registered and unregistered in the Windows registry when they are run stand-alone.

After the plug-in is registered with Windows, the next time the user runs a director, the director will see the new plug-in. The way this is implemented by the director is as follows. When started, directors check the Windows registry to see which plug-ins have registration commands listed. For each plug-in registration command listed in the Windows registry, the director checks to see if a more recent version of that plug-in already exists in the LNS Object Server. It does this by looking to see if there is already a `ComponentApp` object for the plug-in's registration command at the appropriate scope. If the Windows registry gives the scope as 1 (LcaScopeObjectServer), the director looks in the `ObjectServer` object's `ComponentApps` collections to see if a registration command already exists for this plug-in. If the scope is 2 (LcaScopeSystem), the director looks in the `ComponentApps` collections of the `ActiveNetwork`'s `System` object. If the plug-in is not already registered or if the registered version is older, the director may automatically add the plug-in or the director may allow the user to select which plug-ins are added[9].

In either case, at some point the user (or the director) decides that it wants to register the new plug-in. To do this, the director launches the plug-in (see *How Directors Launch And Manipulate Plug-ins* later in this chapter for details) and sends it a `LcaCommandRegister` (50) command.

When a plug-in receives this command, it registers all of the actions that it supports by adding `ComponentApp` objects for each action that it implements to the appropriate `ComponentApps` collections based upon the action's scope and carries out any other initialization that it requires. The plug-in does not register its registration command; the director automatically adds a `ComponentApp` object for this command if the plug-in is successfully registered. Registration is considered successful if the plug-in didn't crash. Failure to register any of the supported commands (for example if unable to create a device template) is not considered a plug-in registration failure.

How Directors Launch And Manipulate Plug-ins

The first question a director must answer is *when* should it launch a plug-in. During initial registration, a plug-in is launched to complete the registration process. In this case, the plug-in is launched either automatically by the director or in response to the user requesting to add the plug-in to the system. Directors may pre-launch a plug-in when opening a system if the plug-in indicates that pre-launch is supported in the Windows registry by adding a Prelaunch subkey and setting it to 1 (see Prelaunch in

[8] The Application Setup Wizard included with Visual Basic does not support registration of plug-ins in this way either. However, since Microsoft provides the source code for the `Setup1` program, it is possible to modify the program to work in this way. A better option, however, is to install your plug-in using a more full-featured setup toolkit, such as InstallShield.

[9] If the scope of the plug-in's registration command is 1, the plug-in has asked to be registered in the global `ComponentApps` collection under the `ObjectServer` object. Plug-ins registered at this level only need to be registered once. After they are registered, they are available to all systems managed by this server. If the scope is 2, the plug-in has asked to be registered in the `System` object's `ComponentApps` collection. Because each `System` object has its own `ComponentApps` collection, plug-ins registered at this level must be registered once per `System`.

Table 3-1, earlier in this chapter). During on-going operation, plug-ins can be launched (or made visible, if already pre-launched) in response to a user selecting an item from a menu; for example, right-clicking a device in the LonMaker tool and selecting Configure from the shortcut menu starts the plug-in and invokes the Configure command. The director determines what plug-ins apply to what objects by searching the appropriate ComponentApps collections[10] for ComponentApp objects that have a ComponentClassId of LcaClassIdAppDevice (7).

When a director decides to launch a plug-in to perform an action on an object, it follows the flow outlined in Table 4-1.

Table 4-1. Interaction Between a Director and a Plug-In

Director Does This	Plug-in Responds Like This
Creates an instance of the plug-in by creating an instance of the plug-in's ActiveX class that implements the LNS Plug-in API. The director gets the class's name from either the Windows registry (for initial registration) or from a ComponentApp object (for subsequent commands).	The Class_Initialize method of the class that implements the LNS Plug-in API runs. The plug-in increments its usage count. If this is the first user of the plug-in, the plug-in also performs whatever initialization it requires.
If this is a remote client, sets the Remote property to True and the sets then NetworkInterfaceName property[11] to identify the interface that the plug-in should use to communicate with the LNS Object Server. All plug-ins must support these properties.	Stores the state of the Remote property and the name of the network interface for future use.
Sets the NetworkName property to identify the network on which the plug-in will act. All plug-ins must support this property.	Opens the network, using the specified network interface name (if one was previously given).
Optionally sets additional properties, such as the Visible property (to cause the plug-in to show itself).	Responds appropriately.
Invokes the SendCommand method to tell the plug in the command that it should perform and the target object on which to perform it. The command is identified by ID, as listed in Table 1-1. The target object is identified by class ID and by object name, as described in later in this chapter.	Validates that it supports the requested command and, if so, executes it. Some command execution may be deferred until the Visible property is set to True.
Optionally sets additional properties, such as the Visible property.	Responds appropriately.
Releases the instance of the plug-in it created by setting the variable that references the plug-in to Nothing.	The Class_Terminate method of the class that implements the LNS Plug-in API runs. The plug-in decrements its usage count. If this is the last user of the plug-in and the plug-in is not visible (i.e., the Visible property is False), the plug-in exits. See *How Plug-Ins Know When To Exit* later in this chapter for more details.

[10] The appropriate collections are those that are in scope. In this case, that would be the ObjectServer object's ComponentApps collection, the ComponentApps collection of the ActiveNetwork's System object, and the ComponentApps collection of the device's DeviceTemplate object.

[11] After creating an instance of a plug-in, the director can set any property, at any time, with two exceptions. If they are set, the NetworkInterfaceName and Remote properties must be set before the NetworkName property.

How Directors Support Prelaunch

Director can support pre-launch, a feature that allows a plug-in to initialize and open the network when the director is started (see Table 1-3, earlier in this Chapter). A pre-launch sequence initiated by the director performs all plug-in launch operations except setting the plug-in visible. A director will launch a plug-in that supports pre-launch immediately after opening the network, according to the following algorithm:

1. If a plug-in supports pre-launch, launch it. This is determined by the existence of the `Prelaunch` subkey in the Windows registry with an entry value of 1.
2. Since a plug-in that supports pre-launch also exposes the Prelaunch property in its LnsPluginAPI class, the director sets this property to 1 (pre-launch in progress).
3. After the director is finished setting properties for the pre-launch operation, the director sets the Prelaunch property in the plug-in's LnsPluginAPI class to 0 (end of pre-launch property-setting sequence).
4. The plug-in performs its pre-launch sequence, based upon the properties that the director has set. For instance, any time-consuming activity that you want to be performed during the pre-launch period, such as open a network, should be contained in a property-setting method of the plug-in such as the SetNetworkName() method. Leave the minimum possible activity in the SendCommand() and SetVisible() methods, which will be called during the actual plug-in launch that the user sees. There must be no progress dialogs or message boxes displayed during the pre-launch period—the server must run hidden. The plug-in must also terminate if it is never made visible and it is released by the director.

If the plug-in has the Prelaunch property set to 1, the director will periodically check if the plug-in process is still available and attached to its reference, and will immediately pre-launch a new instance if the reference becomes invalid or if it was released.

For example, the NodeBuilder tool has Prelaunch subkey set to 1 in the Windows registry. Once the NodeBuilder plug-in is registered with the LonMaker tool, the NodeBuilder tool will launch in the background when the LonMaker tool starts.

How Directors Support MultiObject and SingleInstance

The following algorithm is used by the director to decide when to keep the reference to the plug-in, when to release it, and when to launch a new instance:

1. If this plug-in has the optional MultiObject subkey in the Windows registry, and the MultiObject entry value in the registry is set to 1, (see Table 2-1 in chapter 2), and the director is sending several commands in a batch, the director sets the plug-in's `MultiObject` property (in the ActiveX automation server class that implements the LNS Plug-in API) to 1 (batch started).
2. The director then sends the commands with multiple invocations of the `SendCommand` method.

3. After the last SendCommand of the batch, the director sets the plug-in's MultiObject property to 0 (batch complete) to have the plug-in execute the cached commands in the order they were received.
4. The director then sets the Visible property to True.
5. If the plug-in doesn't have the SingleInstance or MultiObject subkeys entry values set to 1 in the Windows registry, release the reference to it. SingleInstance functionality is described in Chapter 4, *How Plug-Ins Work — The Details*.

The director holds a reference to the plug-in so that it can send additional objects (in the case of MultiObject support) or can activate the existing instance (in the case of SingleInstance support). In these two cases, the director will always keep a reference to the plug-in until the director has terminated.

The desired instancing is achieved in the plug-in source code by setting the ActiveX Instancing property of the LnsPluginAPI class to MultiUse for a plug-in supporting MultiObject or SingleInstance functionality, and to SingleUse for a plug-in which supports neither of these properties, in combination with the algorithm used by the director to release the plug-in reference and creating a new instance. The desired effect requires coordinated behavior between the director and the plug-in.

The device plug-in source code generated by the LNS Device Plug-in Wizard does not entirely support SingleInstance functionality, as it may have problems when multiple SendCommands are called because cached objects are not cleared out. If you intend to use SingleInstance, you must add code to handle re-entry calls. For example, you could add the following code to the ProcessDeviceCommand() function:

```
Static BeenHereBefore as Boolean

If BeenHereBefore Then
   //if Single Instance is set, it is safe to exit
If g_SingleInstance = 1 then Exit Sub
End IF

BeenHereBefore = True
//Continue with the code here
```

How Directors Pass Object Names

The objectName parameter of the SendCommand method specifies the location of the target object in the LNS object hierarchy. The name includes any qualification required to find the appropriate object in the hierarchy. The qualification is provided with a path name, where each element of the path name consists of a string in the following format:

```
className:objectName
```

Multiple elements are separated by slashes, so the complete syntax is:

```
className:objectName[/className:objectName…]
```

The class name is optional if it is unambiguous. The class ID of the target object is passed as a parameter in SendCommand, so the class name for the last object in the path (which is the target) is always unambiguous. Network, System, Subsystem, and AppDevice names are also unambiguous.

Collection object names are not specified in the path name. Subsystem paths can be specified with a shorthand syntax using periods to separate the system name and subsystem names.

For example, the following object name specifies an application device named "Motor Controller 5" in the "Belt 2" subsystem of the "Assembly Line 1" subsystem in the "240 Main Street" network:

```
240 Main Street/240 Main Street.Assembly Line 1.Belt 2/Motor
Controller 5
```

The same name, with all optional class names included, would be:

```
LcaNetwork:240 Main Street/LcaSystem:240 Main
Street/LcaSubsystem:Assembly Line 1/LcaSubsystem:Belt
2/LcaAppDevice:Motor Controller 5
```

Appendix C, *Standard Plug-in Classes*, lists the addressing syntax and class ID of each object class. Classes that appear in multiple places in the object hierarchy have more than one addressing syntax. Optional class names are left off of the addressing syntax descriptions.

Object names that include an Interface object show <interface> as the first element of the path name. This represents one of the four options for specifying an Interface object as shown for the LcaInterface class.

How Plug-ins Let Directors Know About Errors

Plug-ins inform directors of errors by raising ActiveX exceptions. If an error occurs during an operation, a plug-in can raise one of the standard exceptions listed in Appendix D or it can raise a custom exception. Custom exception codes must be greater than or equal to LcaErrRangeUserStart (22000).

How Plug-Ins Know When To Exit

Because a plug-in can receive commands from both directors and users, the question of when to exit is not as clear-cut as it might at first seem. From a director's point of view, the life cycle of the plug-in is as described in Table 4-1. The plug-in comes into existence when the director creates the plug-in by creating an instance of it's class that implements the LNS Plug-in API and ceases to exist when the director releases the reference to the class.

From the user's point of view, the plug-in does not come into existence until it appears on the screen (i.e., not until the director sets the Visible property to True and the plug-in displays a visible window). The user does not expect the plug-in to terminate until the user tells it to do so, for example by clicking on an Exit button. This also means that the user has no concept of the life-cycle of "helper" plug-ins that may run only in the background (that is, of plug-ins that do all their work without becoming visible).

These two perceived life-cycles can overlap. A user might want to close a plug-in while one or more directors still have references to it. If the plug-in were to exit at this time, the directors would all now have references to invalid objects. This means that all subsequent attempts by the directors to interact with the (now terminated) plug-in will result in exceptions in the directors. The desired behavior would be for the plug-in to become invisible (instead of exiting) when the user clicked the exit button and to delay exiting until the last director released its reference to the plug-in.

The overlap can also be in the other direction. A plug-in might not be visible when the last director releases its reference to it. Some plug-ins might be designed to always run as helpers in the background, without a user-interface. Even plug-ins that normally run as visible might be invisible when the director releases the reference. For example, the director might have started the plug-in to gather information from it (such as its default window height) or to register it (by sending it an `LcaCommandRegister` (50) command) and then released the reference without making the plug-in visible. In this case, there is no way for the user (short of killing the plug-in from the Windows task manager) to end the plug-in. In this case, the desired behavior is for the plug-in to exit if it is invisible when the last director releases its reference to the plug-in.

Both of these overlaps can be handled in the same way. A plug-in should keep track of how many directors are using it. The plug-in should not exit until the last director has released its reference and either the user has asked the plug-in to exit or the plug-in is invisible (and thus not available for the user to ask it to exit).

The question now becomes one of what is the best way for a plug-in to track how many directors are using it. For ActiveX automation servers (such as LNS plug-ins) Windows automatically keeps track of how many applications are referencing the server and automatically terminates the server when there are no more references. While this might seem like the solution (and one with no work too), it has a fundamental shortcoming.

Applications implemented in Visual Basic count *all* references to them[12], including references from the application itself. If the plug-in has one or more of its forms loaded, it counts as a reference and Windows will not release the plug-in, even if all directors have released it. The problem is that it is impossible for an invisible plug-in to know when to unload its forms. For example, as described earlier, the director might start the plug-in to gather information from it (such as its default window height) and release the plug-in without ever making it visible. Unfortunately, the plug-in's main window will be loaded as a side effect of reading the `Height` property. The plug-in will thus not automatically exit when the director releases it because the plug-in has a reference to itself.

The solution, therefore, is for the plug-in to keep track of usage counts on its own. Fortunately, this is quite easy to do. Each time a director creates a new reference to the plug-in the `Class_Initialize` method of the class that implements the LNS Plug-in API runs. The plug-in can increment a global usage counter at this time. Each time a director releases a reference to the plug-in, the `Class_Terminate` method of the class that implements the LNS Plug-in API runs. The plug-in can decrement a global usage counter at this time. If the usage count goes to zero and the plug-in is invisible, the plug-in can explicitly exit by closing the LNS Object Server and calling the Visual Basic `End` method. If the plug-in is visible, it continues

[12] This is done automatically by the Visual Basic run-time support system. This is not the case for all languages. For example, it is not done automatically Visual C++ applications.

to run. When the user clicks the plug-in's Exit button, instead of exiting, the plug-in should check the global usage count. If it is non-zero, the plug-in should set itself to invisible instead of exiting (or at least warn the user that there are active reference to the plug-in and give the user the option to select what the plug-in should do). If the global usage count is zero, the plug-in should explicitly exit by closing the LNS Object Server and calling the Visual Basic `End` method.

Using the Prelaunch, MultiObject, and SingleInstance registry keys and LNS Plug-in API properties can make the determination of when to exit more complicated.

What Plug-ins Do When They Run Stand-alone

Plug-ins must provide an option to run stand-alone. When they are started with a command line option of `/regplugin`, such as `MyPlugIn /regplugin`, they must register themselves in the Windows registry. If started without this option, a plug-in must, at a minimum, either automatically register its registration command in the Windows registry or provide an option to allow the user to request that it register itself. An optional command line option is /deregplugin; if started with this option a plug-in should remove its LNS registration and automation server registration in the Windows registry. This optional command line option is implemented in the plug-in framework. The plug-in may also provide other portions of its normal functionality, but it is not required to do so.

Uninstallation Issues

If an LNS device plug-in need to be uninstalled, the registration command in the Windows registry needs to be removed first. A typical plug-in setup program created with Visual Basic Plug-in wizard does not automatically delete the LNS registration command from the Windows registry as part of the uninstallation procedure. Before running the uninstall program, you should therefore run the LNS device plug-in stand-alone with /deregister, then run the uninstall program. You may choose to modify the uninstall program, or to provide your own uninstall program, so that the deletion of the registration command occurs as part of the uninstall procedure. Furthermore, a plug-in should offer a means to unregister from LNS network databases and thus to revert to being registered with Windows, but not with the LNS database.

After running the uninstall program (and deleting the registration command in the Windows registry as described above), the LNS device plug-in is properly uninstalled from a Windows operating system point of view. However, within one or more LNS databases, there may be ComponentApp objects that refer to this now-uninstalled plug-in. An LNS director may choose to remove the ComponentApp references to an LNS device plug-in within an LNS database if it determines that the plug-in has been uninstalled from the computer. If these references are not removed, a director will get an ActiveX exception when it attempts to launch the now uninstalled plug-in and can display an appropriate error message to the user.

Uninstalling a Plug-in should not remove manufacturer resource files since other devices from the manufacturer may use these.

The Plug-in framework supports three levels of deregistration, minimal, comprehensive, and exhaustive. See Chapter 3, *Modifying Code Generated by the LNS Device Plug-in Wizard*, for more information.

5

How Plug-Ins Work — The Details

For each of the tasks introduced in the previous chapter, this chapter provides more detail. For each task it describes why and how the task is done and describes where and how the task is implemented in the plug-in framework generated by the LNS Device Plug-in Wizard. The plug-in framework generated by the plug-in wizard implements the interfaces described in this chapter. You can skip this chapter if you are using the plug-in wizard and you do not need to modify the standard plug-in interfaces.

Introduction

Table 5-1 describes the interaction between a director such as the LonMaker Integration Tool and a plug-in generated by the plug-in wizard. None of the advanced activation flags are enabled in this interaction. The following sections describe this interaction in more detail.

Table 5-1. Interaction Between a Director and the Plug-in Framework

Director Does This	Plug-in framework Responds Like This
Creates an instance of the plug-in by creating an instance of the plug-in's ActiveX class that implements the LNS Plug-in API. The director gets the class's name from either the Windows registry (for initial registration) or from a ComponentApp object (for subsequent commands).	The Class_Initialize method of the plug-in's LNSPlugInAPI class runs. If this is the first user of the plug-in, the plug-in initializes its global variables and calls the FillCommandTable() function. This function fills in the g_CommandTable global table, which defines all of the commands supported by this plug-in.
If this is a remote client, sets the NetworkInterfaceName property13 to identify the interface that the plug-in should use to communicate with the LNS Object Server. All plug-ins must support this property.	Stores the name of the network interface in the m_networkInterfaceName variable of the LNSPlugInAPI class.
Sets the NetworkName property to identify the network on which the plug-in will act. All plug-ins must support this property.	Opens the network specified by the LNSPlugInAPI NetworkName property, using the specified network interface name (if one was previously given). After the network is opened, the framework calls the NetworkOpened() user function in the modFramework module so that you can do any additional processing that your plug-in may require.
Optionally sets additional properties, such as the Visible property (to cause the plug-in to show itself).	All mandatory properties are automatically handled in the LNSPlugInAPI class. If you wish to add any custom properties, you will need to add them to the LNSPlugInAPI class file.
Invokes the SendCommand method to tell the plug in the command that it should perform and the target object on which to perform it.	If this is a registration command, the plug-in calls the RegisterSupportedCommands() function in the MMain module, which

[13] After creating an instance of a plug-in; the director can set any property, at any time, with one exception. If it is set, the `NetworkInterfaceName` property must be set before the `NetworkName` property.

	registers all the commands supported by this plug-in. After each command is registered, the plug-in calls the CommandRegistered() user function in the modFramework module so that you can do any additional processing that your plug-in may require. If the command is something other than registration, the framework validates the command by making sure that it is in the g_CommandTable table. If the command is found, the framework calls the ParseObjectName() function to parse the object name into its constituent pieces and then calls getObjectNameObject() to attempt to find the object referenced by the command14. So that you can process the command, the framework then calls the CommandReceived() user function in the modFramework module, passing the command (by index into the command table), the object that the command acts upon (or Nothing if the object was not found), and the number of parameters passed to the command. If the object is passed as Nothing, you can call the getObjectNameParameterString() utility function to retrieve the various parameters passed to the command so that you can locate the object for yourself. If the command operates on a device, the plug-in wizard adds code to call the ProcessDeviceCommand() function in the fMain form. If the command operates on a functional block, the plug-in wizard adds code to call the ProcessLonMarkObjectCommand() function in the fMain form.
Optionally sets additional properties.	All mandatory properties are automatically handled in the LNSPlugInAPI class. If you wish to add any custom properties, add them

[14] The function only attempts to locate objects of the `lcaAppDevice` and `lcaLonMarkObject` types. If your plug-in operates on other object types, you will need to locate the object yourself. The parser function also does not support subsystem names that are passed as multiple strings. If a director passes object names in this way, you will also need to locate the object yourself.

	to the LNSPlugInAPI class file.
Releases the instance of the plug-in it created by setting the variable that references the plug-in to Nothing.	The Class_Terminate method of the plug-in's LNSPlugInAPI class runs. The plug-in decrements its usage count. If this is the last user of the plug-in and the plug-in's main form is not visible, the plug-in exits.

Registering Your Plug-in's Registration Command

Purpose Of This Step

As described in the previous chapter, when a plug-in is first installed on a computer, it must register its registration command in the Windows registry. Directors use this information to determine which plug-ins a user has installed on their computer and to determine how to add the plug-in to LNS.

All plug-ins must store their registration command under the main key `HKEY_LOCAL_MACHINE\SOFTWARE\LonWorks\LCA\Plug-Ins`. Each plug-in creates its own key, with the key name the same as the plug-in's name, under this main entry. For example, a plug-in named "MyCo Plug-in" would store its registration command under the key `USER_MACHINE\SOFTWARE\LonWorks\LCA\Plug-Ins\MyCo Plug-in`. Under this key, the plug-in stores a set of subkeys and associated entry values, as described in Table 5-2. The entry values are required unless the description of the Entry Value description indicates that it is optional.

Table 5-2. Values Stored In The Windows Registry For A Plug-in's Registration Command

Registry Subkey	Entry Value
(default)	The name of the ActiveX class that implements the LNS Plug-in API. In Visual Basic, this name is always the name of your project followed by a dot and the name of the class that implements the LNS Plug-in API.
Description	A description of what the plug-in does. Typically a director will display this information in a tooltip or status bar.
LcaVersion	The minimum version of the LNS Object Server that this plug-in requires. The version number is specified as <major release>.<minor release>. The minor release can contain either one or two digits. If the minor release contains only one digit, it is assumed that the second digit is a zero. That is, “1.5” is assumed to mean “1.50.”
ManufacturerName	The name of the company that wrote this plug-in.
MultiObject	The optional MultiObject subkey allows a plug-in to indicate to a director whether or not it can support MultiObject functionality. When supported, the MultiObject functionality allows the plug-in to cache multiple SendCommand method invocations from a director and to defer the execution of those requests until told to do so by the director. If the MultiObject subkey is present in the registry and the entry value is set to 1 (which indicates that MultiObject

	functionality is supported), then the plug-in must include the MultiObject property in the ActiveX automation server class that implements the LNS Plug-in API, and must behave as described in How Directors Support MultiObject and SingleInstancein Chapter 4. The MultiObject subkey entry value can also be set to 0, which indicates that this feature is disabled at the present time although the plug-in does include MultiObject support. Finally, the MultiObject subkey entry value can also be set to –1, which indicates that this feature is unsupported. If the MultiObject subkey is not present in the Windows registry, the director assumes that MultiObject functionality is not supported.
Name	The name of this action. Typically a director will display this name in a menu. The standard name for the registration action is <company> <plug-in name>, such as "The Plug-In Company Sample Plug-In."
Prelaunch	The optional Prelaunch subkey allows a plug-in to indicate to a director whether or not the director may launch the plug-in in the background before the plug-in is needed. Supporting this property can improve the perceived startup performance of your plug-in, as the plug-in can open the specified network and perform any required initialization before receiving the information from the director when the director decides to invoke a command that the plug-in supports. The plug-in should remain running in the background until a command to that plug-in is sent, at which point the plug-in should become visible (when the director sets the Visible property to True) and behave normally. If the Prelaunch subkey is present in the registry and the entry value is set to 1 (which indicates that Prelaunch functionality is supported), then the plug-in must include the Prelaunch property in the ActiveX automation server class that implements the LNS Plug-in API, and must behave as described in *How Directors Support Prelaunch* in Chapter 4. The Prelaunch subkey entry value can also be set to 0, which indicates that this feature is disabled at the present time although the plug-in does include pre-launch support. Finally, the Prelaunch subkey entry value can be set to –1, which indicates that this feature is unsupported by the plug-in. If the Prelaunch subkey is not present, the director assumes that the plug-in does not support pre-launch functionality.
Scope	The scope at which the plug-in should be registered. Set to 1 to indicate that the plug-in should be registered in the `ObjectServer` object's `ComponentApps` collection. Set to 2 to indicate that the plug-in should be registered in the `ComponentApps` collection of the `ActiveNetwork`'s `System` object. Specify system-level scope if your plug-in requires special initialization for each system. For example, if you are creating a device plug-in, your plug-in will need to create a device template for the device type you support if one does not already exist in the system. It must do this for every new

	system that is created. If your plug-in does not require any system-specific initialization, you can register it at either scope. If you want to allow the user to decide, on a system-by-system basis, whether or not to use your plug-in, specify the scope as system-level. If you want to ensure that all applications accessing all systems have access to your plug-in, select the Object Server scope.
SingleInstance	The optional SingleInstance subkey allows a plug-in to indicate to a director whether or not a single instance of the plug-in should be used per object that the plug-in operates on. The purpose of the SingleInstance functionality is to specify to the director that if multiple commands are to be performed on the same LNS object, that an existing instance of the plug-in (if there is one) operating on that object should be passed the new command. If SingleInstance functionality is not supported, multiple commands on the same object would result in multiple instances of the plug-in being launched. From the user's point of view, rather than have multiple plug-in instances operating on the same object (and possibly causing interaction with each other), a more friendly approach is to pass the new command to the existing instance of the plug-in instance operating on that object if it is still active, or launch a new plug-in instance if one is not currently active. For plug-ins which accept commands on a single object, the plug-in must return success to a SendCommand for the same command/object if is still active (and the director's re-issuing of the SendCommand should have no other effect). If the SingleObject subkey is present in the registry and the entry value is set to 1 (which indicates that SingleObject functionality is supported), then the plug-in must behave as described in How Directors Support MultiObject and SingleInstancein Chapter 4. The MultiObject subkey entry value can also be set to 0, which indicates that this feature is disabled at the present time although the plug-in does include SingleObject support. Finally, the SingleObject subkey entry value can also be set to –1, which indicates that this feature is unsupported by the plug-in. If the SingleObject subkey is not present in the Windows registry, the director assumes that the plug-in does not support SingleObject functionality.
Version	The version number of this plug-in. The version number is in the same format as LcaVersion. The version number may be followed with a space, and then optional text information. For example: "1.01 Controller Device Configuration Applet." The text information is not required, and directors may ignore it.

When And How It Is Done

Your plug-in's registration command should be added to the Windows registry when your plug-in is installed on the user's computer as part of your setup program as described in the previous chapter. In this way, the plug-in will be available to directors (i.e., they will see it in the registry) immediately after the plug-in is installed.

Your plug-in must register itself when it is started stand-alone with a command-line switch of `/regplugin`, such as `MyPlugIn /regplugin`. It must also provide the ability (either automatically or at the user's discretion) to register itself when it is run stand-alone without this command-line switch. A Visual Basic application can determine if it is running stand-alone by checking the `StartMode` property of the `App` object in it `Main` procedure. If the value is `vbSModeStandalone`, the application was launched stand-alone. A Visual Basic application can retrieve the command line arguments used to start it by calling the `Command()` function. Access to the Windows registry is provided through a number of functions in the Windows API. The plug-in framework, described in the next chapter, provides simplified access to these functions through the `CRegistry` class.

Note: Before adding your registration command to the Windows registry, you **must** check to see if a newer version of the plug-in is already registered. If so, your plug-in **must not** overwrite the existing registration command.

How It Is Done In The Plug-in Framework

The plug-in framework created by the Plug-in Wizard registers the registration command in the `RegisterRegistrationCommand()` function, which is called when the user clicks the `Register Plug-in` button of the `fRegisterPlugIn` form or starts the plug-in with the `/regplugin` switch. The `RegisterRegistrationCommand()` function uses the setting in Table 5-3. It uses the `CRegistry` class to simplify access to the Windows registry and the `GetVersionNumber()` function to parse a version string into major and minor version numbers. The code for `RegisterRegistrationCommand` can be found in the MMain module.

Table 5-3. Where the Sample Plug-In Framework Stores the Settings for its Registration Command

Item	Data Source
(default)	Global constant g_RegisteredServer in the modFramework module.
Description	Global constant g_PlugInDescription in the modFramework module
LcaVersion	Global constant g_MinimumRequiredLCAVersion in the modFramework module
ManufacturerID	Global constant g_ManufacturerID in the modFramework module
ManufacturerName	Global constant g_ManufacturerName in the modFramework module
Name	Global constant g_PlugInProductName in the modFramework module
Scope	Global constant g_PlugInScope in the modFramework module
Version	Global constant g_PlugInVersion in the modFramework module
Prelaunch	Global constant g_Prelaunch in the modFramework module
MultiObject	Global constant g_MultiObject in the modFramework module
SingleInstance	Global constant g_SingleInstance in the modFramework module

Registering Your Plug-in's Other Commands

Purpose Of This Step

As described earlier, registration is a two-step process. The first step is registering your plug-in's registration command, as described above. The second step, described here, is registering the set of actions that your plug-in supports.

When And How It Is Done

Plug-ins must register the actions that they support when they receive a `LcaCommandRegister` (50) command from a director (i.e., when a director invokes the `SendCommand` method with a `CommandId` of `LcaCommandRegister` (50)).

The plug-in must do the following for each action (i.e., command/object pair) to be registered:

1. Fetch the appropriate `ComponentApps` collection, based on the scope of the action. If the scope of the action is device or functional-block specific (i.e., the action will be added to a `DeviceTemplate`'s `ComponentApps` collection or a `DeviceTemplate.LonMarkObject`'s `ComponentApps` collection), you may need to create a `DeviceTemplate` before you can add the `ComponentApp` object. To see if the system already contains a device template for your device type, use the `ItemByProgramId` method. If a device template is not found, and the plug-in is not running as a remote client, create one by adding a new `DeviceTemplate` object to the `DeviceTemplates` collection and importing the appropriate device interface file into the newly created `DeviceTemplate` object.

   A plug-in running on a remote LNS client (where either the LNS Server or LNS Remote Client redistribution is also installed) can successfully create a device template if the device interface text and binary files (".xif" and ".xfb" extensions) are located in the same directory both on the client and on the server system and a binary (".xfb extension") file exists on the server system. Since there is no way to guarantee this unless the remote client actually runs on the same system as the server, the Plug-in framework will not allow a plug-in running on a remote client system to create a new device template, and will not register a command using such a template. One workaround is to create a template on the server system prior to running a remote client. If you hard-code the locations of device interface files and ensure that they will be in the same directory on the client and server systems, you can remove the blocking code in the plug-in source code.

2. Fetch or create a new `ComponentApp` object for your action. To do this, search the `ComponentApps` collection to see if a `ComponentApp` object already exists that specifies your server as a provider of the desired command ID for the desired class ID. If your server is already registered as a provider of this action, move to step 4. If it is not, you need to add a new `ComponentApp` object to the collection.

   When you add the new object, you must give it a name. Typically directors will display this name in a context menu as appropriate. Since `ComponentApps` collections use the name as a key into the collection, the name that you choose for your action must be unique within the collection. Therefore, before adding your new action, you must check to see if a `ComponentApp` object with your desired name already exists in the collection. For example, if your action has a very general name, such as `Test`, it is quite likely that another plug-in already provides an action with this name. If your action name is more specific, such as `Test My Device Type`, it is less likely (although still possible) that another plug-in has already registered an action with this name.

If there is no object with your desired name, move to step 3. If your desired name is already in use, you must pick a new name for your action. For example, you might add your plug-in name to the action name, such as `Test (My Plug-in)`. Continue the process until you have found an unused name. Once you have found an unused name, move to step 3.

3. Create a new `ComponentApp` object, specifying the name for this action, the class ID of the object to which it applies, and the name of your plug-in's server class. In Visual Basic, the name of your plug-in's server class is always the name of your project (as set in the project properties dialog) followed by a dot and the name of the class that implements the LNS Plug-in API.

4. Set the following properties of the `ComponentApp` object:

`commandId`	The command that you are registering
`description`	The description of the service the plug-in provides for this command. Typically a director will display this description in a tooltip or status bar.
`VersionNumber`	The version number of your plug-in.
`ManufacturerId`	Your manufacturer ID.

5. If you want your plug-in to provide the default action for this object type, set the `defaultAppFlag` property to `True`. The default application is the plug-in that the director will, by default, call when the user invokes the specified command on the specified object. Since there can only be one default action for any given `ComponentApps` collection, you must examine all other `ComponentApps` in the collection and set their `defaultAppFlag` properties to `False`.

The plug-in does not need to register its registration command; the director will automatically add the registration command by adding a `ComponentApp` object at the appropriate scope with a command ID of `LcaCommandRegister` [50]) if the plug-in is successfully registered[15].

How It Is Implemented In The Plug-in framework

The plug-in framework created by the Plug-in Wizard registers all of the commands implemented by the plug-in. When the plug-in framework receives a `LcaCommandRegister` (50) command, it calls the `RegisterSupportedCommands()` function in the MMain module. This function

[15] Directors use this `ComponentApp` object differently than the `ComponentApp` objects created by plug-ins. Rather than signifying an action that is provided by a plug-in, this `ComponentApp` object signifies that the plug-in has been successfully registered (and thus does not need to be register again).

steps through the g_CommandTable array (which has one entry per action that the plug-in supports) and registers the actions into the appropriate ComponentApps collection. The g_CommandTable array is initialized in the FillCommandTable() function contained in the modFramework module, which is called from the LNSPlugInAPI class module's Class_Initialize() function.

If the scope field of the command table entry is device-level or functional-block level, the RegisterSupportedCommands() function looks to see if a DeviceTemplate object with the required program ID (specified by the programId field of the command table entry) already exists. If one does not, the function creates a new DeviceTemplate object (using the devTemplateName field of the command table entry as the template name if possible or a derived name if the desired name is already in use) and attempts to import a device interface file into the new template. It does this by looking at every device interface file (.xif extension) under the LNS import directory (specified by the ImportDirectory property of the System object) and its subdirectories until an interface file is found that defines a device with the desired program ID.

If a matching device interface file is not found, the NeedToCreateDeviceType() function in the modFramework module is called. This function displays a dialog asking the user to locate the device interface file. The user can click Cancel, which will result in the failure of the command registration. As described in *Registering Your Plug-in's Other Commands* in Chapter 5, the RegisterSupportedCommands() method in the MMAin module caches the failed program ID and skips registration of all commands applied to that device template. This cache is not persistent, i.e. the plug-in will attempt to register this command when the registration procedure is run again. If you want to provide some other method for locating the device interface file, replace the code in the NeedToCreateDeviceType() function.

If command registration fails because the device template could not be created (for example because the device interface file can't be located), the plug-in adds the program ID to the tblFailedProgID table as shown here:

```
tblFailedProgID.Add g_CommandTable(i).programId
```

The plug-in checks the program ID table when attempting to add future commands:

```
' Check if this progID already failed
For Each varPID In tblFailedProgID
        ' If this progID is found in the table of failed program IDs
        ' then just bail, do not continue
        bMessage = False
        If (CStr(varPID) = g_CommandTable(i).programId) Then
                GoTo Err_RegisterNextSupportedCommand
Next varPID
```

After each command is registered, the plug-in calls the CommandRegistered() function in the modFramework module so that you can do any additional item-related initialization that your plug-in requires.

Responding to Property Reads and Writes

Purpose Of This Step

The LNS Plug-in API defines a set of properties that all plug-ins must implement. Directors write to these properties to tell the plug-in how it should display itself (in the case of properties such as `Left`, `Top`, `Height`, `Width`, and `Visible`) or how to interact with networks (in the case of properties such as `NetworkName` and `NetworkInterfaceName`). Directors can also read from these properties (and from an additional set of read-only properties, such as `Version` and `ManufacturerID`). You can add custom properties to a plug-in, but your plug-in must operate correctly even if a director never sets or gets any of these custom properties.

When And How It Is Done

After creating an instance of a plug-in, the director can set any property at any time, with the exception that the `NetworkInterfaceName` property, if it is set, must be set before the `NetworkName` property.

For the most part, how a plug-in should respond when a director sets and gets its properties is clear. There are, however, a few scenarios that could be handled in multiple ways, all of which are potentially correct. For these scenarios, there are conventions that define how the plug-in should behave, as discussed below.

When a plug-in is minimized, it should return its current `Top`, `Left`, `Height`, and `Width`, not the values that it would have were it to be un-minimized. If these properties are set while the plug-in is minimized, the plug-in should ignore the set and raise a `LcaComponentErrCantSetProperty` error.

If a plug-in is not visible when its minimized property is set, the plug-in should raise a `LcaComponentErrCantSetProperty` error, store the value, and apply the values the next time that the plug-in becomes visible; the plug-in should not become visible as a side-effect of the set to the minimized property.

If a director sets a plug-in's `Top`, Left, `Width`, or `Height` properties to values that cause the plug-in to not fit on the screen or to have a display area that is too small to display the plug-in's contents, the plug-in can behave in any of the following ways:

- Apply the new value without raising an error.
- If the value is too large, the plug-in can clip the value to its maximum allowable setting. In this case, the plug-in should not raise an error.
- If the value is too small, the plug-in can expand the value to its minimum allowable setting. In this case, the plug-in should not raise an error.
- The plug-in can reject the value and raise a `LcaComponentErrCantSetProperty` error.

How It Is Implemented In The Plug-in framework

The plug-in framework generated by the Plug-in Wizard defines Get and Let methods for plug-in properties in the LNSPlugInAPI class module. For example, the following code defines the Set and Let methods for the Visible property:

```
Public Property Let Visible(bNewValue As Boolean)
    fMain.Visible= bNewValue
End Property

Public Property Get Visible() As Boolean
    Visible = fMain.Visible
End Property
```

Other Let methods are more complex. For example, see the Let networkName method. This method sets lightweight (i.e. remote IP) clients to use exclusive mode:

```
         ' An IP-remote client must access data points through the Data Server operating
        ' in the LNS Server Application's process on the server PC.  The client can
access
        ' points in either exclusive or shared mode.  The mode is set by the DsMode
property
        ' of the System object.
        '
        ' Set exclusive mode if running as Remote IP Client.  Find the appropriate mode
        ' constants.
        '
        If (isIpFlag = True) Then
            g_currentSystem.DsPause = True
            g_currentSystem.DsMode = lcaDsModeExclusive
            g_currentSystem.DsPause = False
        End If
```

Responding to Commands from a Director Other than Registration

Purpose Of This Step

The primary purpose of plug-ins is to provide simplified ways of carrying out operations, usually customized to a particular type of device, application, or end-user. The way that this is done is by directors sending action requests to plug-ins, specifying the command that they should perform and the target object on which they should perform it.

When And How It Is Done

Directors invoke the SendCommand method to request that a plug-in perform a command on an object. This method has three parameters: the ID of the command to execute (as given given in Appendix A), the class ID of the target object to which it applies, and the object name (as given in Table 1-5) of the object to which it applies. When a plug-in's SendCommand method is executed, if the command is something other than LcaCommandRegister or LcaCommandUnregister, the plug-in must validate that the command passed in the commandId parameter is one that it supports. If it is, the plug-in must next locate the target object to which the command applies, using the objectClassId and objectName parameters. Once this is done, the plug-in can carry out that the requested operation. The plug-in must not display itself at this time; the plug-in is not displayed until the director sets the Visible property to True. If the command is the LcaCommandRegister (50) command, the plug-in should carry out the steps described in *Registering Your Plug-in's Other Commands*, earlier in this chapter.

How It Is Implemented In The Plug-in framework

The plug-in framework generated by the Plug-in Wizard defines the SendCommand procedure in the LNSPluginAPI class module. This procedure first checks for the LcaCommandRegister and LcaCommandUnregister commands.

```
Select Case commandId
    Case lcaCommandIdRegister       ' See if this is a registration command

        ' Set the caption of the progress box
        fProgress.Status.Caption = "Registering plug-in .  .  ."
        DoEvents

        ' Register the commands that we support with the LCA Object Server
        RegisterSupportedCommands

    Case lcaCommandIdUnregister ' See if this is a unregistration command
        ' Set the caption of the progress box
        fProgress.Status.Caption = "Unregistering plug-in .  .  ."
        DoEvents

        ' Deregister the commands that we support with the LCA Object Server
        DeregisterSupportedCommands False
```

Determining When To Exit

Purpose Of This Step

As described in Chapter 4, to avoid errors a plug-in must not exit until the last director has released its reference and either the user has asked the plug-in to exit or the plug-in is invisible (and thus not available for the user to ask it to exit).

When And How It Is Done

Each time a director creates a new reference to the plug-in, the `Class_Initialize` method of the class that implements the LNS Plug-in API runs. The plug-in should increment a global usage counter at this time. Each time a director releases a reference to the plug-in, the `Class_Terminate` method of the class that implements the LNS Plug-in API runs. The plug-in should decrement the global usage counter at this time. If the usage count goes to zero and the plug-in is invisible, the plug-in can explicitly exit by closing the LNS Object Server and calling the Visual Basic `End` method. If the plug-in is visible, it continues to run. When the user clicks the plug-in's `Exit` button, instead of exiting, the plug-in should check the global usage count. If it is non-zero, the plug-in should set itself to invisible instead of exiting (or at least warn the user that there are active reference to the plug-in and give the user the option to select what the plug-in should do). If the global usage count is zero, the plug-in should explicitly exit by closing the LNS Object Server and calling the Visual Basic `End` method.

How It Is Implemented In The Plug-in framework

The plug-in framework created by the LNS Device Plug-in Wizard implements Initialize and Terminate methods in the LNSPlugInAPI class module to keep track of the number of references the plug-in has. These functions use the global variable `g_useCount` defined in the MMain module:

```
Private Sub Class_Initialize()
    ' If this is the first user of the class, init some stuff
    If g_useCount = 0 Then
        ' Fill in the table of all the commands our plug-in supports
        FillCommandTable

        g_lcaOSIsOpen = False
        g_networkIsOpen = False
        Set g_lcaOS = fMain.lcaOS
    End If

    ' Increase the use count
    g_useCount = g_useCount + 1
End Sub

Private Sub Class_Terminate()
    ' Decrease the use count
    g_useCount = g_useCount - 1

    ' When the last director releases its reference to us,
    ' if we are invisible at this time we need to exit
    ' because the user can't do the exit (since we are invisble)
    If g_useCount = 0 And fMain.Visible = False Then fMain.EndPlugIn
End Sub
```

The fMain form created by the LNS Device Plug-in Wizard contains the ConditionalExit method, which exits the plug-in if no directors are referencing it; otherwise the plug-in is hidden, but continues to run until the last reference is removed:

```
Private Sub ConditionalExit ()
    ' If no directors are using the plug-in, exit
    ' Otherwise just hide.
    If g_useCount = 0 Then
        ' End the plug-in
        EndPlugIn
    Else
        ' Hide
        Me.Visible= False
    End If
End Sub
```

The fMain form also defines the mnuExit_Click method that allows the user to force the plug-in to exit when the Exit menu item is selected, even if it is still referenced by one or more directors:

```
Private Sub cmdExit_Click()
    Dim Warning As String

    ' If no directors are using the plug-in, exit
    ' Otherwise warn the user and ask them if they mean it
    If g_useCount = 0 Then
        ' End the plug-in
        EndPlugIn
    Else
        If g_useCount = 1 Then
            Warning = " A director is using this plug-in.  Exiting may cause the" + _
                      " director to experience errors.  Are you sure that you want to
exit?"
        Else
```

```
            Warning = "There are " + Trim$(Str(g_useCount)) + _
                      " directors using this plug-in.  Exiting may cause the directors" +
_
                      " to experience errors.  Are you sure that you want to exit?"
        End If
        If MsgBox(Warning, vbYesNo + vbDefaultButton2, g_PlugInProductName) = vbYes Then
            ' End the plug-in
            EndPlugIn
        Else
            ` Just hide the form
           Me.Hide
        End If
    End If
End Sub
```

Finally, the fMain form defines the EndPlugIn method, which terminates the plug-in and closes the network and the LNS Object Server:

```
Public Sub EndPlugIn()
    On Error Resume Next

    ' Release local object cache
    Set m_LonMarkObject = Nothing
    Set m_Device = Nothing

    'Detach from the network.
    If Not g_currentSystem Is Nothing Then If g_currentSystem.IsOpen Then
g_currentSystem.Close

    ' Force LNS to release the system
    ' When remote, must be done BEFORE closing the network
    Set g_currentSystem = Nothing

    'Close the network database.
    If Not g_currentNetwork Is Nothing Then If g_currentNetwork.IsOpen Then
        g_currentNetwork.Close
    End If

    ' Force LNS to release the network
    Set g_currentNetwork = Nothing

    ' Close the object server
    If g_lcaOS.IsOpen Then g_lcaOS.Close

    ' Set all our flags
    g_lcaOSIsOpen = False
    g_networkIsOpen = False

    ' Unload the main form (which as a side-effect will unload all other forms in use)
    Unload Me

    ' Exit the application
    End
End Sub
```

Creating an Installation for an LNS Device Plug-in

Once you have completed plug-in development, you should create an installation for your LNS Device Plug-in. Typically, this is an installation program (`setup.exe`) that can be used by the end user, your customer, to install the plug-in software and all support files needed. To create an installation, you will need to use an installation tool such as InstallShield®.

LNS Plug-ins require an LNS Server or LNS Remote Client redistribution to be available on the target machine. However, the LNS redistribution is typically installed by directors, so it is not usually necessary to install the LNS redistribution with your LNS Plug-in. If your plug-in is designed to be run stand-alone, you may need to install the LNS redistribution. Contact your nearest Echelon sales office, technical support office, or any Echelon distributor for details about redistributing the LNS Server of the LNS Remote Client. The LNS Server or LNS Remote Client redistributions are licensed software, and may not be redistributed except under a signed license agreement.

An LNS Device Plug-in installation should perform the following steps:

1. Find the LONWORKS folder. When a user first installs as LNS tool, such as the LonMaker tool, a LONWORKS folder is created. This folder is typically located in c:\LonWorks by default, but the user or installer may choose a different folder. The location of this folder is stored in the following Windows registry key:

   ```
   HKEY_LOCAL_MACHINE\SOFTWARE\LonWorks\LonWorks Path
   ```

   If the target computer does not have this registry key, your installation should either install the LNS redistribution (see above) or return a message informing the user that they must install an LNS tool that includes the LNS redistribution before installing your plug-in.

   If you are using the InstallShield tool, this can be accomplished by following these steps:

   i. Open the InstallShield power editor and select **RegLocator** to create a new `Signature_(32)` with the following properties:

Signature_(32)	LonWorksPathValue
Root(I2)	2
Key(S255)	SOFTWARE\LonWorks
Name(S255)	LonWorks Path
Type(I2)	2

   ii. Select **Directory** to create a folder with the following properties:

Directory(S32)	LONWORKS
Directory_Parent(S32)	TARGETDIR
DefaultDir(S255)	LonWorks

   iii. Select **AppSearch** and add the following property:

Property(S32)	LONWORKS
Signature_(S32)	LonWorksPathValue

2. Install your plug-in executable (.exe extension) in a folder on the target computer. This may be any folder, but plug-ins are typically installed in a folder with your manufacturer name within a Plug-Ins folder within the LONWORKS folder. For example, the default location is the following: `c:\LonWorks\Plug-Ins\<Manufacturer Name>`.

3. Install `LnsDevCtrls.ocx` in the LONWORKS `bin` folder (c:\LonWorks\bin by default). This file contains the `EchStatusBar` and `EchLog` Visual Basic controls.

4. Install any third party controls (".ocx" extension) and DLLs. These are

typically installed in the target computer's Windows system folder. You can run a dependency check to find which ones you need. A dependency check may return files that are part of the LNS Server or LNS Remote Client redistributions. You may not distribute any LNS files other than `LnsDevCtrls.ocx` without permission and a license from Echelon. If the target machine has an LNS tool, it will already have the necessary LNS controls and DLLs installed.

5. If your Plug-in requires resource files, install them in a folder on the target computer. This may be any folder, but resource files are typically installed in a folder with your manufacturer name within a Types\User folder within the LONWORKS folder. For example, the default location is the following: `c:\LonWorks \Types\User\<Manufacturer Name>`. Then use the Resource File API to add your files to the resource catalog and then to refresh the resource catalog. Uninstalling a Plug-in should not remove manufacturer resource files, since other devices from the device manufacturer may also use these files. See Chapter 14 of the *NodeBuilder User's Guide* for more information.
6. Install any other files that you need, such as bitmaps, installation videos, and help files.
7. Install the device interface (.xif, .xfb, and .xfo extensions) and application image files (.apb and .nxe extensions), if required, to a folder on the target computer. This may be any folder, but resource files are typically installed in a folder with your manufacturer name within an Imports folder within the LONWORKS folder. For example, the default location is the following: c:\LonWorks\Import\<Manufacturer Name>. See Chapter 14 of the *NodeBuilder User's Guide* for more information. The program ID of the device interface file must exactly match the program ID in the command table of the device plug-in.
8. Run the plug-in executable with the `/regplugin` command line switch. This will register the plug-in in the Windows registry.

Appendix A

Standard Plug-in Commands

This Appendix lists the standard commands that may be implemented by plug-ins. Plug-ins may also implement manufacturer-specific commands.

Table A-1. Standard LNS Plug-in Commands

Command		Description
LcaCommandBrowse	20	Monitor and control the object.
LcaCommandBuildImage	10	Build the image for the object. Only applies to AppDevice class objects.
LcaCommandCalibrate	14	Set calibration configuration properties for an object.
LcaCommandCommission	11	Load the network image into an object. Only applies to AppDevice class objects.
LcaCommandConfigure	13	Set the configuration properties for an object.
LcaCommandConnect	15	Connect an object to other objects. Only applies to AppDevice and Subsystem class objects.
LcaCommandControl	22	Control the object.
LcaCommandEditSource	2	Edit source code for the object. Only applies to AppDevice class objects.
LcaCommandLoad	12	Load an application image into an object. Only applies to AppDevice class objects.
LcaCommandMonitor	21	Monitor the object.
LcaCommandMonitorRecovery	61	Monitor recovery of an object.
LcaCommandNew	1	Create a new object of the specified class with the specified name.
LcaCommandOffline	31	Change the state of the object to offline.
LcaCommandOnline	30	Change the state of the object to online.
LcaCommandRecover	60	Recover object.
LcaCommandRegister	50	Register a component with the object.
LcaCommandReplace	41	Replace the object with a new object.
LcaCommandReport	23	Generate a report for the object.
LcaCommandReset	32	Reset the object.
LcaCommandSecurityLevel	70	Set the security level for an object
LcaCommandTest	33	Test the object.
LcaCommandUninstall	40	Uninstall the object.
LcaCommandUnregister	51	Unregister the component.
LcaCommandWink	34	Wink the object. Only applies to AppDevice and Router class objects.

Plug-ins can also implement custom commands. The values for custom command values are assigned by the plug-in, and may be any value greater than or equal to `LcaCommandUserStart` (10000).

Appendix B

Standard Plug-in Properties

This Appendix lists the standard properties that must be implemented by plug-ins, as well as optional properties that may also be implemented by plug-ins. Plug-ins may also implement manufacturer-specific properties.

Table B-1. Standard LNS Plug-in Properties

Name	Type	Description
Height	Long	The height, in pixels, of the plug-in's main window. Note that in Visual Basic, a form's height is specified in twips[16], not pixels. You can convert to pixels by multiplying the form's Height property (when setting the property) or dividing (when getting the property) by the value Screen.TwipsPerPixelY.
LcaVersion	Read-Only String	Minimum version of the LNS Server redistribution required by this plug-in.
Left	Long	The x location, in pixels, of the upper left corner of the plug-in's main window. 0 is at the leftmost of the user's display. Note that in Visual Basic, a form's left position is specified in twips, not pixels. You can convert to pixels by multiplying the form's Left property (when setting the property) or dividing (when getting the property) by the value Screen.TwipsPerPixelX.
ManufacturerID	Read-Only String	An identifier that is unique to each LONWORKS device manufacturer. Manufacturer IDs may be one of two types: *standard manufacturer IDs* and *temporary manufacturer IDs*. Standard manufacturer IDs are assigned to manufacturers when they join the LONMARK Interoperability Association, and are also published by the LONMARK Interoperability Association so that the device manufacturer of a LONMARK certified device or plug-in is easily identified. Standard manufacturer IDs are never reused or reassigned. Temporary manufacturer IDs are available to anyone on request by filling out a simple form at www.lonmark.org/mid. If your company is a LONMARK member, but you do not know your manufacturer ID, you can find your ID in the list of manufacturer IDs at www.lonmark.org/spid. The most current list at the time of release of the NodeBuilder tool is also included with the NodeBuilder software. If your company is not a LONMARK member, get a manufacturer ID at www.lonmark.org/mid.
ManufacturerName	Read-Only String	The name of the company that wrote this plug-in.
Minimized	Boolean	Specifies whether the plug-in is minimized (True) or not (False). Setting the property sets the plug-in's main window state.
MultiObject	Read-Write Long	The optional MultiObject property allows plug-ins that support the MultiObject operation to cache incoming requests from directors and defer the execution of those requests until told to do so by the director. This is known as a *batch operation*. If a director finds the MultiObject property in the interface of your plug-in, it can set it to 1 (indicating the beginning of the batch) before sending the first request and set it to 0 to indicate that the plug-in should execute all the cached requests in the order received. Not all directors support batch operations, and if not this property

[16] The Visual Basic help file defines a twip as "a screen-independent unit used to ensure that placement and proportion of screen elements in your screen application are the same on all display systems. A twip is a unit of screen measurement equal to 1/20 of a printer's point. There are approximately 1440 twips to a logical inch or 567 twips to a logical centimeter (the length of a screen item measuring one inch or one centimeter when printed)."

		is ignored. If a directory does not support batch operations, it will never send batch operations to the plug-in even though a plug-in may support them. A plug-in implementing the MultiObject property must also set the MultiObject key in the Windows registry. This can be done using the LNS Device Plug-in Wizard, as described in Chapter 2, *Generating a Plug-in with the LNS Device Plug-in Wizard*. You can also set the `g_Prelaunch` constant in the `modFramework` module of the generated plug-in framework to have this key and its entry value set in the registry for you. The Windows registry key is described in the next chapter.
Name	Read-Only String	The plug-in's name.
NetworkInterfaceName	Read-Write String	The name of the network interface object associated with the network. This name is required by applications accessing the LNS Object Server from a remote client. In the remote case, the director will set this property before setting the NetworkName property.
NetworkName	Read-Write String	The name of the network on which to operate.
Prelaunch	Long	The optional Prelaunch property allows a director to launch the plug-in in the background, typically when the director is started. Not all directors support the use of pre-launch, and if not this property is ignored, and the plug-in would never be pre-launched by that director. When the plug-in is pre-launched, the plug in should remain running in the background until a command to that plug-in is sent from the director, at which point the plug-in should become visible and behave normally. This allows plug-ins with a long launch time to be called quickly once the director is running. A plug-in implementing the Prelaunch property must also define the Prelaunch key and set its entry value in the Windows registry. This can be done using the LNS Device Plug-in Wizard, as described in Chapter 3. You can also set the `g_Prelaunch` constant in the `modFramework` module of the LNS Device Plug-in framework to have the key and its entry value set for you. The Windows registry key is described in chapter 5. A plug-in supporting pre-launch operation should not display any messages or become visible while going through the pre-launch sequence. A director can tell a plug-in to pre-launch by setting the Prelaunch property to 1.
Remote	Boolean	Optionally set by the director to indicate if the plug-in is running on the same computer as the LNS Object Server (Remote is False) or on a different computer (Remote is True). In the remote case, the director will set this property before setting the NetworkName property.
Top	Long	The y location, in pixels, of the upper left corner of the plug-in's main window. 0 is at the topmost of the user's display. Note that in Visual Basic, a form's Top position is specified in twips, not pixels. You can convert to pixels by multiplying the form's Top property (when setting the property) or dividing (when getting the property) by the value Screen.TwipsPerPixelY.
Version	Read-Only	Version number of the plug-in. The version number is in "<*major*

	String	*release*>.<*minor release*>" format. The minor release can contain either one or two digits. If it contains only one digit, it is assumed that the second digit is a zero. That is, "3.5" is assumed to mean "3.50". The version number may be followed with a space, and then optional text information. For example: "1.01 Controller Device Configuration Plug-in".
Visible	Boolean	The display state (visible or not visible) of the plug-in. Plug-ins are not required to display a window or become visible at the time that this property is set to True. For example, your plug-in might choose to wait until it has received a request to execute a command (via the SendCommand method). Or, your plug-in might be one that performs a task that never requires it to become visible. In these cases, the plug-in should not raise an exception to the director's set request — it should just cache the requested state value for future use or ignore the set request as appropriate. Reads of the Visible state should, as always, return the actual display state of the plug-in.
Width	Long	The width, in pixels, of the plug-in's main window. Note that in Visual Basic, a form's width is specified in twips, not pixels. You can convert to pixels by multiplying the form's Width property (when setting the property) or dividing (when getting the property) by the value Screen.TwipsPerPixelX.

Appendix C

Standard Plug-in Classes

This Appendix lists the standard classes of objects that may be passed to plug-ins, as well as the addressing syntax used to identify objects.

Table C-1. Class IDs and Addressing Syntax For The LNS Objects.

Object Class	ID	Addressing Syntax
LcaClassIdAppDevice	7	network/system.subsystem[.subsystem…]/appDevice
LcaClassIdAppDevices	8	network/system.subsystem[.subsystem…]
LcaClassIdBuildTemplate	34	network/system network/system/buildTemplate network/system/LcaProgramTemplate:programTemplate
LcaClassIdBuildTemplates	35	network/system
LcaClassIdChannel	12	network/channel
LcaClassIdChannels	13	network
LcaClassIdComponentApp	30	ComponentApp network/system/componentApp network/system/LcaDeviceTemplate:deviceTemplate/componentApp
LcaClassIdComponentApps	31	n/a (for ObjectServer)network/system network/system/LcaDeviceTemplate:deviceTemplate
LcaClassIdConfigProp	26	<interface>/configProp <interface>/LcaLonMarkObject:lonMarkObject/configProp <interface>/LcaNetworkVariable:networkVariable/configProp
LcaClassIdConfigProps	27	<interface> <interface>/LcaLonMarkObject:lonMarkObject <interface>/LcaNetworkVariable:networkVariable
LcaClassIdConnectDescTemplate	42	network/system/connectDescTemplate
LcaClassIdConnectDescTemplates	43	network/system
LcaClassIdConnections	18	network/system
LcaClassIdDataValue	49	<interface>/networkVariable/dataValue <interface>/LcaLonMarkObject:lonMarkObject/networkVariable/dataValue <interface>/LcaConnections:connections/networkVariable/dataValue
LcaClassIdDetailInfo	49	network/system.subsystem[.subsystem…]/appDevice network/system.subsystem[.subsystem…]/LcaRouter:router
LcaClassIdDeviceTemplate	36	network/system/deviceTemplate
LcaClassIdDeviceTemplates	37	network/system
LcaClassIdError	44	network/system
LcaClassIdExtension	50	extension (for Object Server) network/extension network/system.subsystem[.subsystem…]/appDevice/extension network/system.subsystem[.subsystem…]/LcaRouter:router/extension network/LcaChannel:channel/extension network/system/LcaDeviceTemplate:deviceTemplate/extension network/system/LcaHardwareTemplate:deviceTemplate/extension
LcaClassIdExtensions	51	n/a (for Object Server) network network/system.subsystem[.subsystem…]/appDevice network/system.subsystem[.subsystem…]/LcaRouter:router

		network/LcaChannel:channel network/system/LcaDeviceTemplate:deviceTemplate network/system/LcaHardwareTemplate:deviceTemplate
LcaClassIdHardwareTemplate	32	network/system/hardwareTemplate
LcaClassIdHardwareTemplates	33	network/system
LcaClassIdInterface	19	network/system.subsystem[.subsystem…]/appDevice network/system/LcaDeviceTemplate:deviceTemplate network/LcaNetworkServiceDevice:networkServiceDevice/interface network/system/LcaNetworkServiceDevice/interface
LcaClassIdInterfaces	20	network/LcaNetworkServiceDevice:networkServiceDevice network/system/LcaNetworkServiceDevice
LcaClassIdLonMarkAlarm	46	<interface>/LonMarkObject
LcaClassIdLonMarkObject	28	<interface>/lonMarkObject
LcaClassIdLonMarkObjects	29	<interface>
LcaClassIdMessageTag	24	<interface>/messageTag <interface>/LcaConnections:connections/messageTag
LcaClassIdMessageTags	25	<interface>
LcaClassIdNetwork	1	network
LcaClassIdNetworkInterface	14	network/LcaNetworkServiceDevice:networkServiceDevice/networkInterface network/system/networkInterface
LcaClassIdNetworkInterfaces	15	network/LcaNetworkServiceDevice:networkServiceDevice network/system
LcaClassIdNetworks	2	n/a
LcaClassIdNetworkServiceDevice	40	network/LcaNetworkServiceDevice:networkServiceDevice network/system
LcaClassIdNetworkServiceDevices	41	network
LcaClassIdNetworkVariable	22	<interface>/networkVariable <interface>/LcaLonMarkObject:lonMarkObject/networkVariable <interface>/LcaConnections:connections/networkVariable
LcaClassIdNetworkVariableField	48	<interface>/LcaNetworkVariable:nv/field <interface>/LcaLonMarkObject:lonMarkObject/LcaNetworkVariable:nv/field
LcaClassIdNetworkVariables	23	<interface>
LcaClassIdObjectServer	0	n/a
LcaClassIdObjectStatus	47	<interface>/LonMarkObject
LcaClassIdProgramTemplate	38	network/system/programTemplate
LcaClassIdProgramTemplates	39	network/system
LcaClassIdRecoveryStatus	52	network/system
LcaClassIdRouter	9	network/system.subsystem[.subsystem…]/router
LcaClassIdRouters	10	network/system.subsystem[.subsystem…]
LcaClassIdRouterSide	11	network/system.subsystem[.subsystem…]/router/LcaFarSide network/system.subsystem[.subsystem…]/router/LcaNearSide
LcaClassIdSubnet	16	network/system/subnet
LcaClassIdSubnets	17	network/system
LcaClassIdSubsystem	5	network/system.subsystem[.subsystem…]
LcaClassIdSubsystems	6	network/system[.subsystem[.subsystem…]]

LcaClassIdSystem	3	network/system
LcaClassIdSystems	4	network
LcaClassIdTemplateLibrary	21	network/system

Appendix D

Standard Plug-in Exceptions

This Appendix lists the standard properties that may be thrown by plug-ins. Plug-ins may also throw manufacturer-specific exceptions.

Table D-1. Standard Plug-in Exception Codes

Exception	Code	Meaning
LcaComponentErrCantFindObject	20005	The plug-in could not locate the object specified by the ObjectName parameter.
LcaComponentErrCantGetProperty	20007	An error occurred while the plug-in was attempting to get the property.
LcaComponentErrCantSetProperty	20006	An error occurred while the plug-in was attempting to set the property.
LcaComponentErrGeneric	20000	Some unspecified error occurred.
LcaComponentErrInit	20001	An error occurred during the plug-in's initialization.
LcaComponentErrInvalidCommandId	20003	The plug-in does not support the command specified by the CommandId parameter for the object class specified by the objectClassId parameter.
LcaComponentErrInvalidObjectType	20004	The plug-in does not support the command specified by the CommandId parameter for the object class specified by the objectClassId parameter.
LcaComponentErrWriteNotSupported	20002	The director attempted to write to a read-only property.

Appendix E

Plug-in Framework Components

This Appendix lists the standard components that are contained in the plug-in framework generated by the LNS Device Plug-in Wizard. These components include the files, event handlers, and functions contained within the framework.

Plug-in Files

Table E-1 lists the files generated by the plug-in wizard. Files listed in italics are the files that you will typically modify when customizing the plug-in for your device. Modifications are described in Chapter 3.

Table E-1. The Plug-In **Framework** Files

File	Purpose
CRegistry.cls	A class that simplifies access to the Windows registry. It is used by the RegisterRegistrationCommand() function to register the plug-in in the Windows registry. Do not modify this file.
fMain.frm	The main form for your plug-in. This form contains the tabs and monitor and control points added using the User Interface Editor window.
fHidden.frm	This form is an invisible container for the ActiveX Common Dialog control that is used to display a dialog that allows the user to select the device interface file for the plug-in. This feature is used by the RegisterSupportedCommands() function if an appropriate DeviceTemplate is not found within the current network database. Do not modify this file.
fProgress.frm	A form that provides a progress indicator when the plug-in is working. You can modify this form to customize what the user sees when the plug-in is working. You can display this form from within any function that requires a noticeable execution time such as, for example, downloading an application image to your device.
fRegisterPlugIn.frm	A form that allows the user to register and deregister the plug-in. This form is displayed when your plug-in is run standalone.
LnsPluginAPI.cls	A class that implements the required properties and methods of the LNS Plug-in API, providing the public interface to your plug-in. Whenever a director sends a command request for one of your supported commands, the class calls the CommandReceived() function in the modFramework module. You will only need to modify the LnsPluginAPI file if you wish to add custom properties or methods to your plug-in.
MMain.bas	A module file that provides various utility functions used by the other files. You will only need to modify this file if you wish to complete the getObjectNameObject method (which, as described later fetches the object identified in the SendCommand method for some but not all object types).
modFramework.bas	This module contains various constants and methods used by the plug-in. Much of the code contained in this module is managed by the plug-in wizard. Use the plug-in wizard to change this code. Do not edit the framework code in `modFramework.bas`. You can modify the callback functions at the end of the module. These functions are initially empty or mostly empty. They exist to allow you to add application-specific processing when events such as command registration or network opening occur.
WalkDir.bas	A module file that provides functions that searches for a device interface file in a specified directory (and all its subdirectories) that contains a specified program ID. For device-scope commands, this function is used by the RegisterSupportedCommands() function to import an appropriate device interface file. Do not modify this file.

Plug-in Event Handlers

Table E-2 lists the event handlers generated by the plug-in wizard. Event handlers listed in italics are the event handlers that you will typically modify when customizing the plug-in for your device. Modifications are described in Chapter 3.

Table E-2. Plug-in Framework Event Handlers.

Event Handler	File	Purpose
cmdApply_Click()	fMain.frm	Commits the changes pending in the altered text controls.
cmdCancel_Click()	fMain.frm	Cancels the changes pending in the altered text controls.
lcaOS_OnNetworkVariableUpdate()	fMain.frm	Calls the NetworkVariableUpdateReceived() function to process an incoming network variable update.
lcaOS_OnNvUpdateError()	fMain.frm	Logs errors for monitored network variables.
smnuMonitor_Click()	fMain.frm	Menu handler to control monitoring of network variables and configuration properties.
txtCp_KeyPress()	fMain.frm	Marks changes in the relevant text controls when using the wizard-generated user-interface. The plug-in code tries to commit pending changes when Apply is clicked.
txtNvi_KeyPress ()	fMain.frm	Marks changes in the relevant text controls when using the wizard-generated user-interface. The plug-in code tries to commit pending changes when Apply is clicked.

Plug-in Functions

Table E-3 lists the functions generated by the plug-in wizard. Functions listed in italics are the functions that you will typically modify when customizing the plug-in for your device. Modifications are described in Chapter 3.

Table E-3. Plug-in Framework Functions.

Function	File	Purpose
DeregisterRegistrationCommand()	MMain.bas	Removes registration entries from the Windows registry.
DeregisterSupportedCommands()	MMain.bas	Deregisters all supported commands from all device templates within the supported program ID range for complete and exhaustive deregistration modes.
DrfLookUp()	fMain.frm	Sets the functional block scope using the LNS functional block Mode property
EnableControls()	fMain.frm	Prepares the text controls that are used for monitoring.
EnableMonitor()	fMain.frm	Initializes monitoring for network variables and configuration properties.
GetClassIdName()	MMain.bas	Given a class ID, this function returns the name of the class in a string.
GetCPTByTypeIndex	fMain.bas	Given a string-encoded reference to a configuration property, this function returns the related LcaConfigProperty object.
GetCommandIdName()	MMain.bas	Given a command ID, this function returns the name of the command in a string.
GetNumObjectNameParameterStrings()	MMain.bas	Returns the number of substrings in the last object name passed to the plug-in by a director.
GetObjectNameParameterString()	MMain.bas	Returns the ith substring of the last object name passed to the plug-in by a director and optionally trims the class identifier, if any.
InitDevice()	fMain.frm	Initializes the plug-in interface to a device during plug-in startup. It reads the device state, sets the monitor tags, initializes formats, and retrieves initial values.
NetworkVariableUpdateReceived()	fMain.frm	Updates text box controls added by the plug-in wizard.
ParseWord()	MMain.bas	Given a string and a character to search for, returns all of the

		characters before the search character as the result of the function (or the entire input string if the search character is not found) and returns all of the characters after the search character (or an empty string if the search character is not found) in the RemainingString parameter.
ProcessDeviceCommand()	fMain.frm	A handler that processes a command that applies to a device (LcaAppDevice). The default implementation contains code to perform a complete resource file look-up to adjust the scope of the implemented functional blocks. It also contains pre-defined code to set the device template label and to initialize the device. The default implementation also contains a MsgBox() function call, reminding you that these handlers are subject to review and enhancement.
ProcessLonMarkObjectCommand()	fMain.frm	A handler that processes a command that applies to a functional block (LcaLonMarkObject). The default implementation for each function contains code to perform a complete resource file look-up to adjust the scope of the implemented functional blocks. It also contains pre-defined code to set the device template label and to initialize the device. The default implementation also contains a MsgBox() function call, reminding you that these handlers are subject to review and enhancement.
ReadInitValues()	fMain.frm	Adjust formatting options for network variables and configuration properties, and retrieve initial values for configuration properties.
RegisterRegistrationCommand()	MMain.bas	Registers the plug-in and the commands it supports within the Windows registry.
RegisterSupportedCommands()	MMain.bas	Registers all supported commands within the current LNS network database. The commands are registered for all existing device templates within the configured program ID range.
SelectActiveTab()	fMain.frm	Sets the active tab during start-up.
SetDevTemplateLabel()	fMain.frm	Sets the caption for the Device Template label. Called during plug-in startup.

Index