ECHELON®

2.0

LNS® Plug-in Programmer's Guide
Release 4

078-0393-01A

Table of Contents

Preface

You can use LNS device plug-ins to simplify the installation of your devices for network integrators. This manual provides an overview of the LNS Plug-in API and how plug-ins and director applications such as the LonMaker Integration Tool interact. It describes how to install the LNS Plug-in Framework Developer's Kit, which implements the LNS Plug-in API and provides a set of framework assemblies that let you efficiently develop and redistribute LNS plug-ins. It explains how to write device plug-ins using .NET programming languages such as C# and Visual Basic .NET, and how to redistribute your LNS device plug-ins.

Purpose

This document describes how to write LNS plug-ins using .NET programming languages such as C# and Visual Basic .NET. After reading this document, you should understand the basic mechanics of how plug-ins work, and how to write a plug-in using the LNS Plug-in Framework.

Audience

Users of this document should have a basic understanding of the LNS Object Server, and experience programming in C#, Visual Basic .NET, or another .NET programming language.

Software Requirements

Software requirements for computers running the LNS Plug-in Framework Developer's Kit are listed below:

- Microsoft Windows Vista™, Microsoft Windows XP, Windows 2000, or Windows Server 2003. Echelon recommends that you install the latest service pack available from Microsoft for your version of Windows.
- Microsoft .NET Framework Version 2.0 Redistributable Package. To check whether your computer has this software and download it (if necessary), go to the Microsoft Windows Update Web page at *http://update.microsoft.com*.
- Microsoft Visual Studio 2005 Professional Edition or higher. Microsoft Visual Studio Express editions may work, but they have not been tested.
- LNS Server Turbo Edition (3.22 or later).
- LNS Turbo plug-in director application. This may be the LonMaker Integration Tool Turbo Edition (3.2 or later) or the LNS Turbo ADK Plug-in Director example application.

Content

This guide includes the following content:

- *Introduction*: Describes the types of plug-ins that are used with LONWORKS networks, and it describes how director applications request actions from plug-ins. Explains how the LNS Plug-in Framework allows plug-ins to function in the .NET environment and interface with COM-based director applications.
- *Creating and Redistributing LNS Device Plug-ins*: Explains how to create an LNS device plug-in project using the LNS Plug-in Framework Developer's Kit, how to debug your plug-ins, and how to create an installation project for your plug-in so that you can provide it to users.
- *How Plug-ins Work with Directors*: Summarizes 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. Provides examples of how these tasks should be performed.
- *Appendixes*: Includes listings of standard plug-in commands, properties, classes, and exceptions; a glossary that provides definitions for key terms and concepts associated with programming plug-ins; and a demonstration of plug-in behavior using the example ACME C# plug-in that is installed by the LNS Plug-in Framework Developer's Kit.

Related Manuals

The following manuals provide supplemental information to the material in this guide. You can download these documents from Echelon's Web site at *www.echelon.com*.

LNS® Programmer's Guide	Describes how to use the LNS Object Server ActiveX Control to develop an LNS application on a Microsoft Windows Vista™, Server 2003, Windows XP, or Windows 2000 host computer.
LonMaker® User's Guide	Describes how to use the LonMaker Integration Tool to design, commission, modify, and maintain LONWORKS networks.

Other Related Material

This document refers to and describes the LNS Plug-in API that all plug-ins use. Because plug-ins work together 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.

For More Information and Technical Support

The **LNS Plug-in Framework Online Help** details the **Echelon.LNS.Plugin** and **ACME.DevicePlugin** namespaces. The **Echelon.LNS.Plugin** namespace contains a framework of interfaces, classes and other types for creating .NET-based LNS Plug-ins. The **ACME.DevicePlugin** namespace contains example device plug-ins written in C# and Visual Basic that use the LNS Plug-in Framework. To view the LNS Plug-in Framework Online Help, click **Start**, point to **Programs**, point to **Echelon LNS Plug-in Framework Developer's Kit**, and then select **LNS Plug-in Framework Online Help**.

The **ReadMe First** file for the LNS Plug-in Framework provides descriptions of known problems, if any, and their workarounds. To view this ReadMe file, click **Start**, point to **Programs**, point to **Echelon LNS Plug-in Framework Developer's Kit**, and then select **ReadMe First**. For additional information, you can go to Echelon's Development Tools Web pages at *www.echelon.com/products/development*.

If you have technical questions regarding the programming of LNS plug-ins not answered by this document, the **LNS Plug-in Framework Online Help**, or the **ReadMe First** file, you can contact technical support. To receive technical support from Echelon, you must purchase support services from Echelon or an Echelon support partner. See *www.echelon.com/support* for more information on Echelon support and training services.

You can obtain technical support via phone, fax, or e-mail from your closest Echelon support center. The contact information is as follows:

Region	Languages Supported	Contact Information
The Americas	English Japanese	Echelon Corporation Attn. Customer Support 550 Meridian Avenue San Jose, CA 95126 Phone (toll-free): 1-800-258-4LON (258-4566) Phone: +1-408-938-5200 Fax: +1-408-790-3801 *lonsupport@echelon.com*

Region	Languages Supported	Contact Information
Europe	English German French Italian	Echelon Europe Ltd. Suite 12 Building 6 Croxley Green Business Park Hatters Lane Watford Hertfordshire WD18 8YH United Kingdom Phone: +44 (0)1923 430200 Fax: +44 (0)1923 430300 *lonsupport@echelon.co.uk*
Japan	Japanese	Echelon Japan Holland Hills Mori Tower, 18F 5-11-2 Toranomon, Minato-ku Tokyo 105-0001 Japan Phone: +81-3-5733-3320 Fax: +81-3-5733-3321 *lonsupport@echelon.co.jp*
China	Chinese English	Echelon Greater China Rm. 1007-1008, IBM Tower Pacific Century Place 2A Gong Ti Bei Lu Chaoyang District Beijing 100027, China Phone: +86-10-6539-3750 Fax: +86-10-6539-3754 *lonsupport@echelon.com.cn*
Other Regions	English	Phone: +1-408-938-5200 Fax: +1-408-328-3801 *lonsupport@echelon.com*

You can submit a feedback form with suggestions on how to improve the product's functionality and documentation at *www.echelon.com/company/feedback*. This feedback form is not forwarded to technical support and should not be used to submit technical or product support related issues. Please send technical support questions to your Echelon support center.

1

Introduction

This chapter describes the types of plug-ins that are used with LONWORKS networks, and it describes how director applications request actions from plug-ins. It explains how the LNS Plug-in Framework allows plug-ins to function in the .NET environment and interface with director applications.

Introduction

An LNS plug-in is an out-of-process automation server that implements the COM-based LNS Plug-in API so that it may be instantiated and controlled by an LNS plug-in director application. 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 simplifies the configuration, monitoring, or control of their devices (plug-ins that are specific to a particular type of device are sometimes referred to as *device plug-ins*). Plug-ins can also add new functionality to LNS applications such as alarming, logging, and trending. The LNS Plug-in API is further described in Chapter 3, *How Plug-ins Work with Directors.*

An *LNS plug-in director application* (herein referred to as a *director*) is a complex, general-purpose network manager that is extensible in its functionality, because it can call LNS plug-ins. For example Echelon's LonMaker Integration Tool is an LNS plug-in director application that can call LNS plug-ins to extend its functionality—typically to perform manufacturer-specific device configuration.

When a plug-in is installed on a computer, the setup program installs a number of items for the plug-in to the computer's registry. This enables directors to determine which plug-ins a user has installed on their computer and call the plug-ins at appropriate times. Directors can make access to plug-ins completely transparent to the end-user. This means that 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.

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 manufacturers also benefit from plug-ins. For tool manufacturers, plug-ins make 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 have a competitive edge: they are cheaper to install and service, and they are easier to use.

The LNS Plug-in Framework Developer's Kit allows plug-ins to function in the .NET environment and interoperate with COM-based directors. It includes the .NET components needed for interfacing with the COM-based LNS API in the .NET environment. It provides a set of example software and framework assemblies that let you efficiently develop plug-ins with the latest .NET programming tools and re-distribute your plug-ins. It provides the plug-in API and COM interaction so that you do not need to provide it in your .NET application.

Overview of Plug-ins

A plug-in is a type of LNS application that implements the COM-based LNS Plug-in API. The Plug-in API consists of the LNS Plug-in Commands (see Appendix A, *Standard Plug-in Commands*), the LNS Plug-in Properties (see Appendix B, *Standard Plug-in Properties*), and LNS Plug-in Exceptions (see Appendix D, *Standard Plug-in Exceptions*). A plug-in is defined by the actions that it can perform (for example, 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: *device plug-ins* and *system plug-ins*.

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 allows users to set

configuration properties and enforce limits on configuration property values. You can create a device plug-in executable that operates on multiple device types. 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, and let the plug-in determine which object to act upon based on the object that has been passed as a parameter.

System plug-ins apply to an entire 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. This document focuses on device plug-ins; however, you can also implement a system plug-in based on the instructions in Chapter 2, *Creating and Redistributing LNS Device Plug-ins*.

The LNS Plug-in API defines a single method, **SendCommand()**, and a number of properties that a plug-in must implement.

LNS Plug-in Commands

The **SendCommand()** method 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 path). 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 (for example, 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 for 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 controlling the device in addition to configuring it. The device plug-in can also decide whether to display the same user interface for these two commands.

Plug-ins can also implement custom commands. The values for custom command IDs 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. 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, *Standard Plug-in Properties*. Read access must be provided to all properties. Write access may optionally be provided for most properties; however, some properties must be read-only or must be read-write as noted in Appendix B, *Standard Plug-in Properties*.

Overview of the LNS Plug-in Framework Developer's Kit

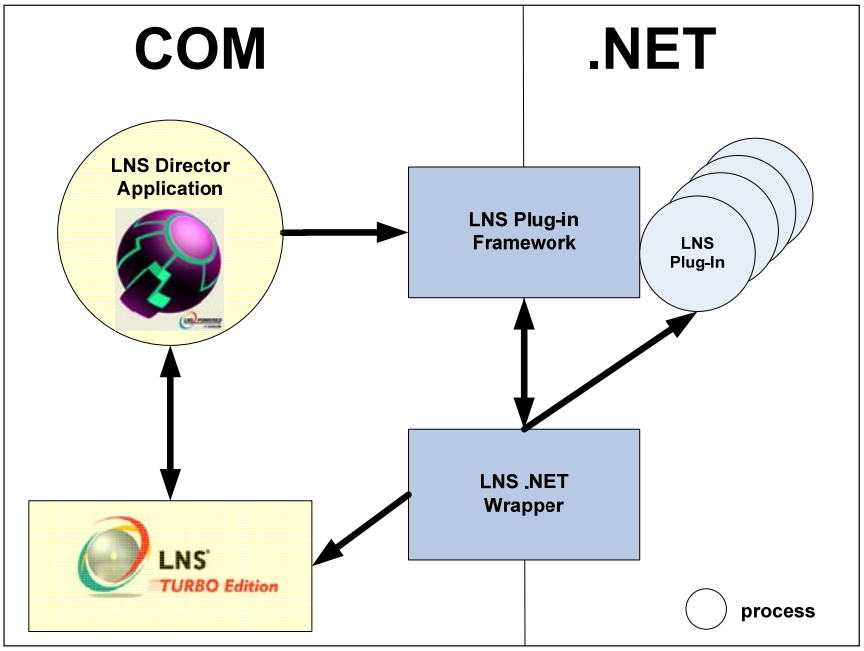
The LNS Plug-in Framework Developer's Kit includes the following components needed for interfacing with the COM-based LNS API in the .NET environment: the LNS Plug-in Framework and the LNS .NET wrapper.

The *LNS Plug-in Framework* is a library of classes, interfaces, and value types that provides access to LNS Plug-in functionality and is designed to be the foundation on which .NET-based plug-ins are built. The LNS Plug-in Framework provides the following components:

- A COM Callable Wrapper (CCW) that lets a COM director application, such as the LonMaker tool, call into the .NET plug-in assembly.
- An implementation of the full LNS Plug-in API interface (**ILnsPlugin**).

- A simple out-of-process server task model that supports plug-ins in the .NET environment (LNS plug-ins are out-of-process COM servers, for which support in the .NET environment is not well documented).

The *LNS .NET wrapper* is the Primary Interop Assembly, which is generated by Echelon and distributed with LNS. It provides a Run-time Callable Wrapper (RCW) that allows your .NET plug-in to interface with the COM-based LNS API. Note that the LNS .NET wrapper is required for plug-in development with .NET, but is not considered part of the LNS Plug-in Framework because it may also be used by LNS applications that are not plug-ins. The LNS Primary Interop Assembly is included with the LNS runtime components starting in LNS 3.23. If your plug-in is targeted for LNS 3.23 or later, you do not need to include the LNS 3.22 Primary Interop Assembly that is included with the LNS Plug-in Developer's Kit, version 1.1.



The LNS Plug-in Framework is implemented as a set of base classes that your plug-in will inherit and extend. These classes are documented by the **LNS Plug-in Framework Online Help**. The use of these classes is demonstrated in the *Creating the Plug-in Source Files* section in Chapter 2.

- **PluginObjectBase**. This class contains the basic framework implementation of the LNS Plug-in API methods and properties.
- **PluginServerBase**. This class works in conjunction with **PluginObjectBase**. It contains the process model handling for the out-of-process server.
- **PluginFormBase**. This class contains the base class for the optional user interface instantiation of the plug-in. This may not be necessary in all plug-ins, since some plug-ins may not have user interfaces.

2

Creating and Redistributing LNS Device Plug-ins

This chapter explains how to create an LNS device plug-in using the LNS Plug-in Framework Developer's Kit, how to debug your plug-ins, and how to create an installation project for your plug-in so that you can provide it to users.

Creating and Redistributing Plug-ins Overview

You can use the LNS Plug-in Framework Developer's Kit to create an LNS device plug-in and create an installation project for your plug-in so that you can provide it to users.

To create a new plug-in, you need to first install the Microsoft .NET Framework Version 2.0 Redistributable Package. You then need to install the LNS Plug-in Framework Developer's Kit and an LNS Turbo plug-in director application such as the LonMaker Tool Turbo Edition (3.2 or later) if they are not already installed on your computer. You need a director application to test your plug-in during development.

After the required software is installed on your computer, you can begin creating your plug-in. To do this, you create a new C# or VB.NET plug-in project with Microsoft Visual Studio 2005 (or later), create the plug-in source files (for the plug-in server, plug-in object, and plug-in form derived classes), and optionally implement command registration for the **DeviceTemplate** and **LonMarkObject** scoped plug-in commands (note that this is not automatically implemented for you by the LNS Plug-in Framework, version 1.1).

During the development of your plug-in, you can debug it by generating trace information and using the **OLE/COM ObjectViewer** to display the registration information.

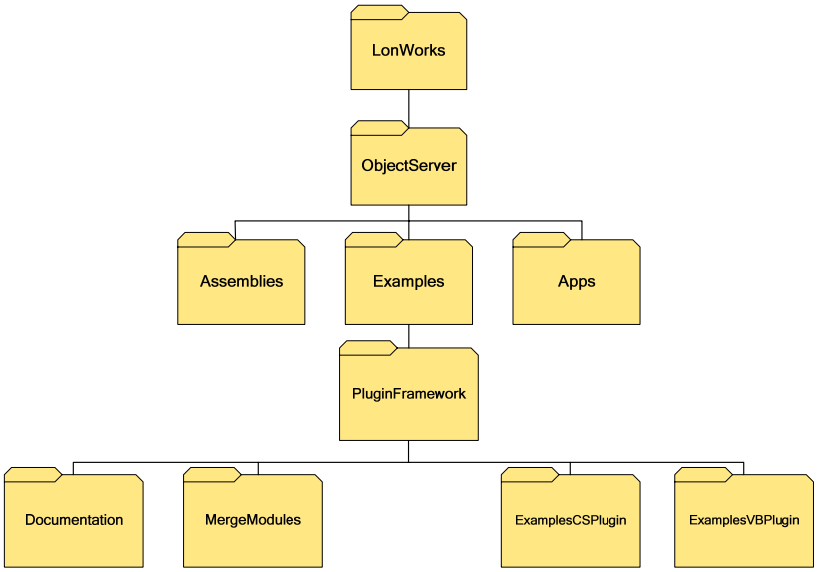
After you create your device plug-in, you can create an LNS plug-in installer based on the framework so that you can redistribute your plug-in.

Note: During development, you can also refer to the ACME plug-in example that is installed on your computer by the LNS Plug-in Framework Developer's Kit. The ACME plug-in example is located in the LonWorks\ObjectServer\Examples\PluginFramework folder on your computer, and it is provided in both C# and VB.NET programming languages. For more information on using the ACME C# plug-in example to view the behavior of a plug-in, see Appendix F, *Running the ACME Example C# Plug-in*.

Installing the Developer's Kit

The LNS Plug-in Framework Developer's Kit is automatically installed by the NodeBuilder FX tool, and it is also available as a single-executable installer that you can download from the Echelon Web site. To install the developer's kit, follow these steps:

1. Install the Microsoft .NET Framework Version 2.0 Redistributable Package (or newer) if it is not already installed on your computer. The LNS Plug-in Framework and example plug-in will not install or run properly unless the .NET 2.0 Framework (or newer) is installed on your computer. See *Redistributing Your Plug-in* later in this chapter for details. To check whether your computer has this software and download it (if necessary), go to the Microsoft Windows Update Web page at *http://update.microsoft.com*.
2. Install the LNS director application that will be used for plug-in testing. If not already set, this will set the root directory for the LNS components (for most Echelon products, this directory will appear by default at **[Windows System Drive]\LonWorks**, and is known as the **LonWorks** directory). It will also install the LNS runtime that is required to successfully run the ACME example plug-in.
3. If you do not have the NodeBuilder FX tool installed on your computer, download the LNS Plug-in Developer's Kit from the Echelon Web site at http://*www.echelon.com/support/downloads*, and then install it.
4. After the LNS Plug-in Framework Developer's Kit and other requirements have been installed (either by installing the NodeBuilder FX tool, or downloading it from the Echelon and installing it standlaone), the following folders will reside in the **LonWorks** directory on your computer.



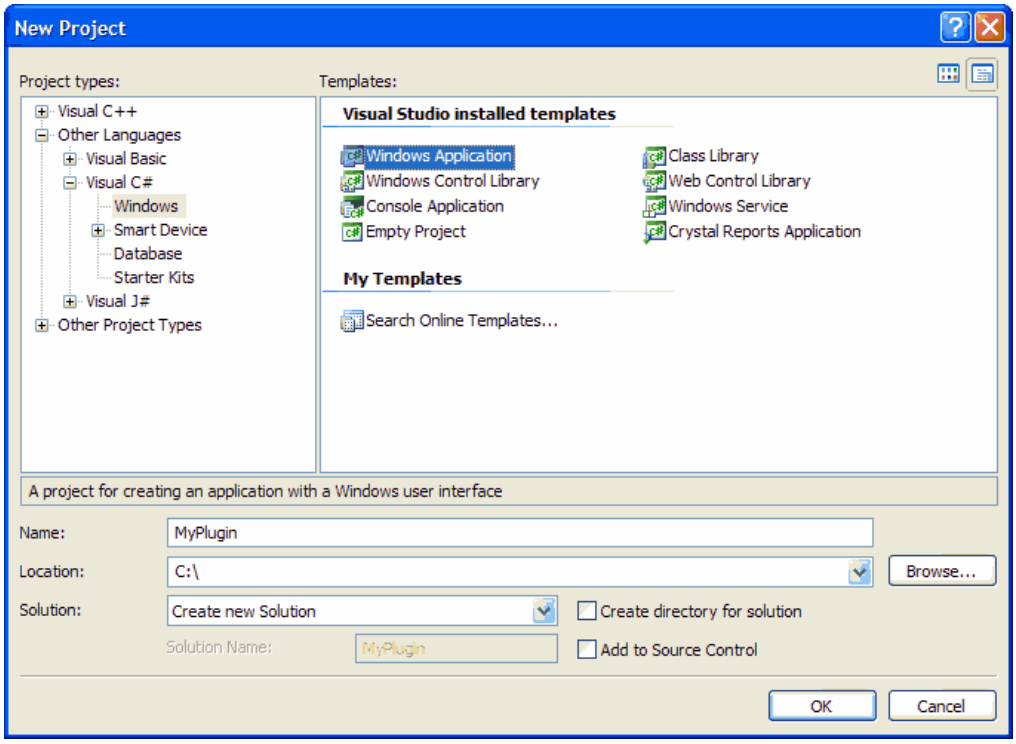
- The **ObjectServer** folder is common to all computers with the LNS Server runtime installed.
- The **Assemblies** folder contains the .NET assemblies referenced by plug-ins that are built using the framework. Visual Studio is informed of this directory by a registry entry added during the installation. This makes the assembly information accessible to the development environment. These assemblies are also installed directly into the Global Assembly Cache for runtime support of framework-based plug-ins.
- The **Apps** folder is used by convention to redistribute LNS plug-ins. The sub-directory under the **Apps** folder should give the plug-in manufacturer's name (for example, this folder contains an "ACME" subdirectory containing the provided example plug-in). Under the manufacturer's directory, there should be a directory specific to each individual plug-in created by your company. For example, there is an **Example** folder under the LonWorks\Apps\ACME directory representing one plug-in created by the ACME company. If the ACME company created another plug-in named "Example2", it would be stored in an **Example2** folder under the LonWorks\Apps\ACME directory.
- The **Examples** folder contains example LNS applications.
- The **Documentation** folder contains a ReadMe file, a PDF of this manual, the LNS Plug-in Framework Online Help, and a Visio drawing illustrating the plug-in instantiation models.
- The **MergeModules** folder contains merge modules for the two .NET assemblies referenced by framework-driven plug-ins. These merge modules should be included in your installation when you redistribute a plug-in developed with the Kit. See *Redistributing Your Plug-ins* later in this chapter for details.
- The **ExampleCSPlugin** folder and sub-directories contain source code and Visual Studio 2005 project files for the ACME Example C# Plug-in.
- The **ExampleVBPlugin** folder and sub-directories contain source code and Visual Studio 2005 project files for the ACME Example VB.NET Plug-in. This example was created by translating the C# version of the example plug-in source code to the VB.NET language.

5. After the installation has successfully completed, an **Echelon Plug-in Framework Developer's Kit** program folder will appear in the Windows program folders. Shortcuts within this folder provide access to the ReadMe file, a PDF of this manual, the LNS Plug-in Framework online help file, the standalone example plug-in executable, and the C# and VB.NET examples.

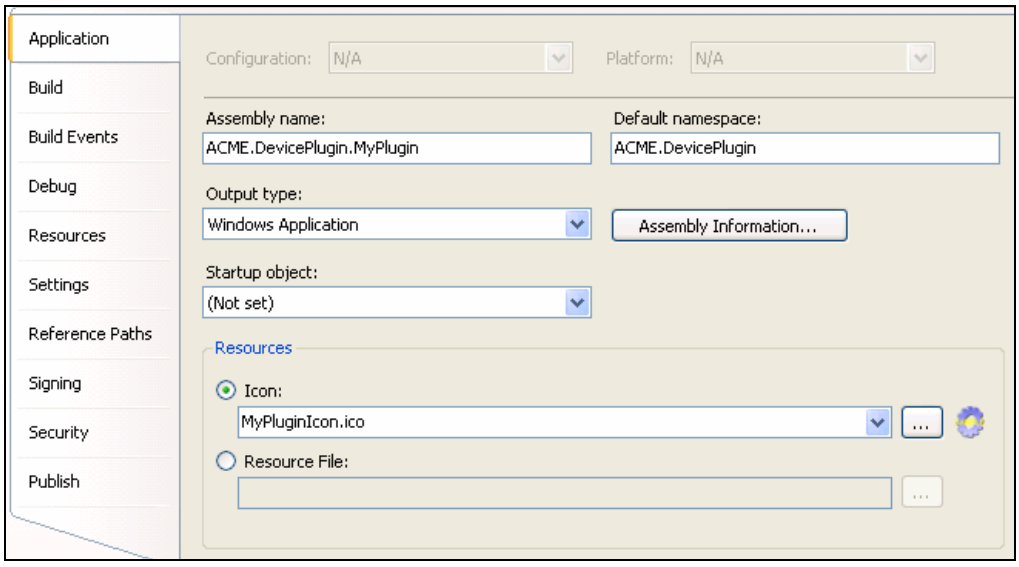
Creating the Plug-in Project

You can create a C# or VB.NET plug-in project using Microsoft Visual Studio 2005 (or later) following these steps (note that the example shown in this section is a C# plug-in project created with Microsoft Visual Studio 2005 Professional Edition):

1. Create a new C# or Visual Basic **Windows Application** project. The project must create an EXE target. Specify a name and location for your project. Click **OK**.



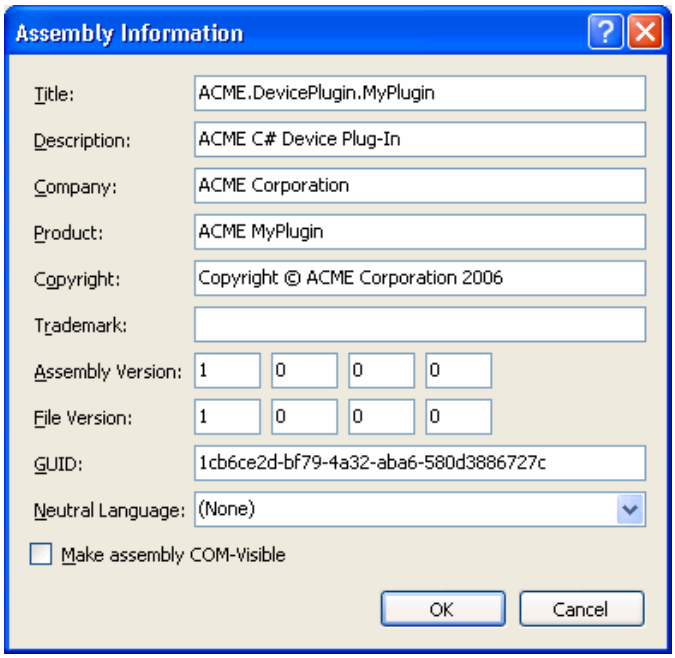
2. Click **Project** and then click your application's **Properties** option. The **Application** tab opens.



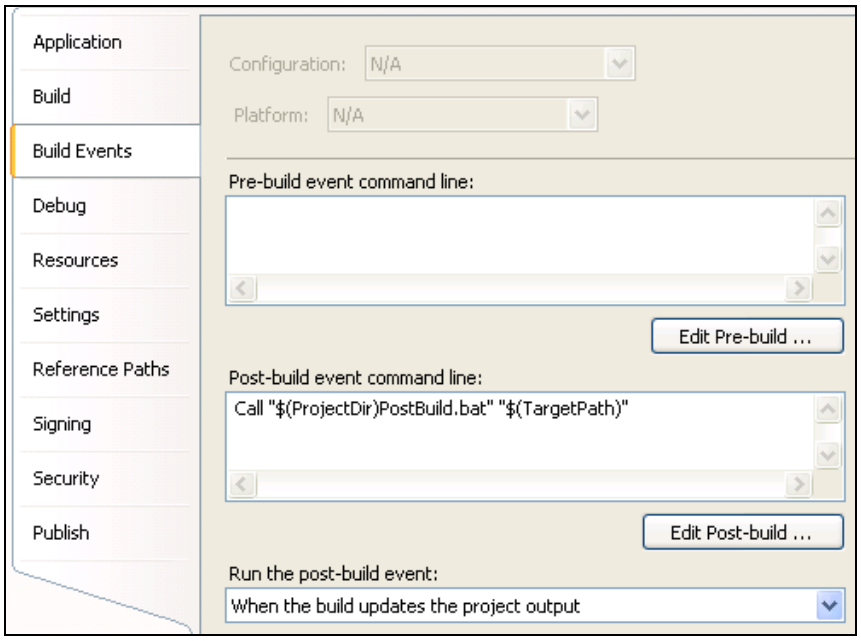
3. Modify and confirm the following project properties:

- In the **Default namespace** property, change the namespace to *<YourCompany>*.*<YourProject>*.
- In the **Assembly name** property, enter a unique and descriptive assembly name that matches your selected namespace (for example, *<YourCompany>*.*<YourProject>*.*<YourPlugin>*).
- In the **Output type** property, verify that **Windows Application** is specified.
- In the **Icon** property under the **Resources** box, specify an icon.

4. Click **Assembly Information**. The **Assembly Information** dialog opens.



5. Enter the information describing your plug-in assembly, and then click **OK** to save the information and return to the **Application** tab.
6. Click the **Build Events** tab.



7. Copy the **PostBuild.bat** file from the LonWorks\ObjectServer\Examples\PluginFramework\ExampleCSPlugin to your project folder.

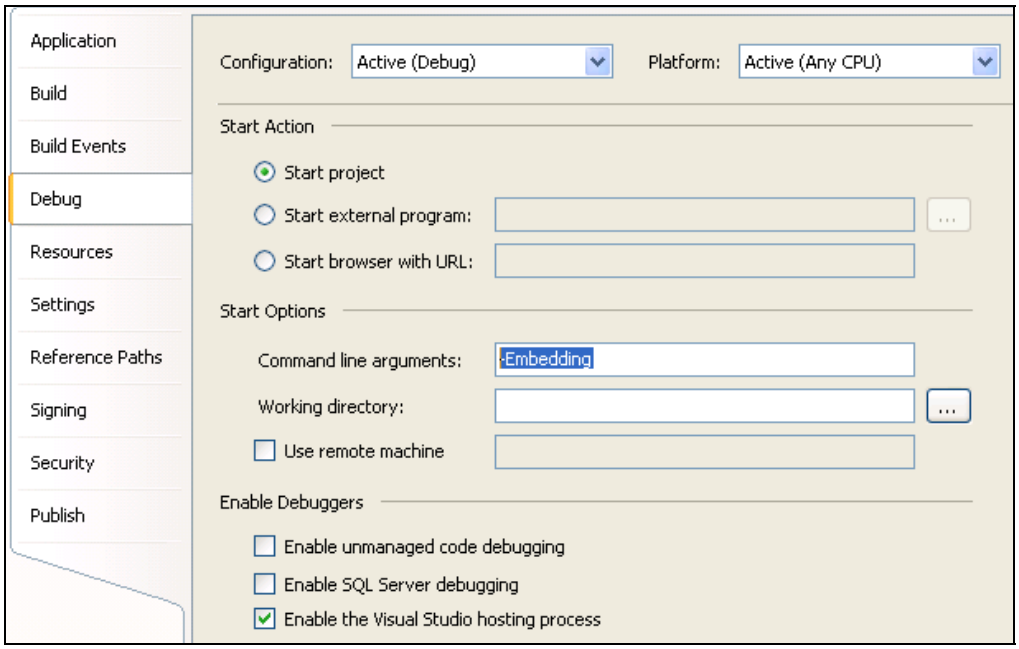
8. In the **Post-build event command line** property, enter the following line:

   Call "$(ProjectDir)PostBuild.bat" "$(TargetPath)"

   The **PostBuild.bat** file will carry out the post-build actions necessary to register your plug-in during the development process. Specifically, it generates a *COM Type Library* and registers the assembly for use with COM using **RegAsm /tlb**, and it registers the plug-in with LNS directors by calling the plug-in with a **/RegPlugin** argument.

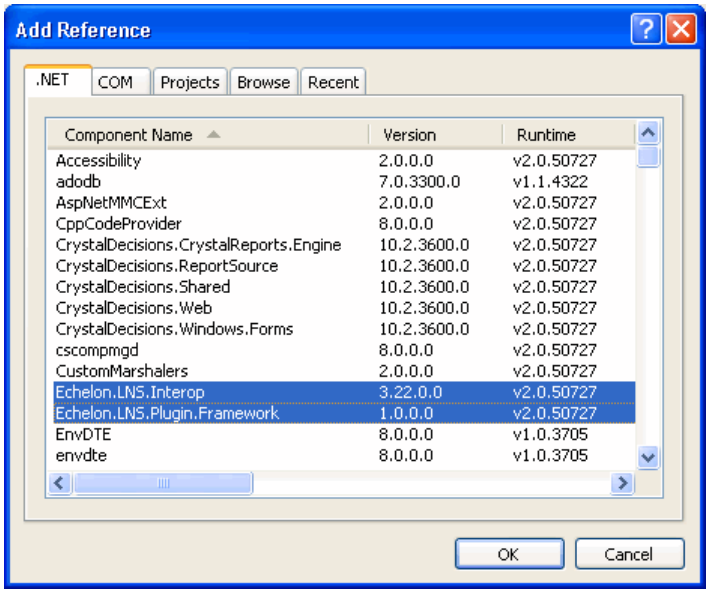
9. In the **Run the post-build event** property, select the **When the build updates the project output** option.

10. Click the **Debug** tab.



11. In the **Command line arguments** property, enter **-Embedding**. This enables debugging of the server.

12. Click **Project** and then click **Add Reference**. The **Add Reference** dialog opens.



13. Select the **Echelon.LNS.Interop** and the **Echelon.LNS.Plugin.Framework** .NET assemblies, and then click **OK**.

    - **Echelon.LNS.Interop** enables your application to interact with LNS.
    - **Echelon.LNS.Plugin.Framework** provides the base functionality for your plug-in.

Creating the Plug-in Source Files

After you create your LNS plug-in project, you need to create the plug-in source files. To do this, you create three classes: plug-in server, plug-in object, and plug-in form, which are to be derived from the **PluginServerBase**, **PluginObjectBase**, and **PluginFormBase** base classes, respectively. After you create these classes, you then modify them as described in the subsequent sections.

To create the plug-in server, plug-in object, and plug-in form classes, follow these steps:

1. In the **Solution Explorer** pane, delete the automatically generated **Program** and **Form1** files.
2. Create the plug-in server class. To do this, follow these steps:
    a. Click **Project** and then click **Add Class**. The **Add New Item** dialog opens.
    b. In the **Name** property, enter <*YourPlugin*>**Server**, where *YourPlugin* is the name of your plug-in, and then click **Add**.
3. Create the plug-in object class. To do this, follow these steps:
    a. Click **Project** and then click **Add Class**. The **Add New Item** dialog opens.
    b. In the **Name** property, enter <*YourPlugin*>**Object**, where *YourPlugin* is the name of your plug-in, and then click **Add**.
4. Create the plug-in form class, which is a Windows Form item. To do this, follow these steps:
    a. Click **Project** and then click **Add Windows Form**. The **Add New Item** dialog opens.
    b. In the **Name** property, enter <*YourPlugin*>**Form**, where *YourPlugin* is the name of your plug-in, and then click **Add**.

The following subsections describe how to implement your plug-in server, plug-in object, and plug-in form classes.

Implementing the Plug-in Server Class

To implement your plug-in server class, modify the source as follows:

1. Add **using System.Runtime.InteropServices** and **using Echelon.LNS.Plugin** directives.
2. Add a **ComVisible(false)** attribute to the class. You will not need to create this class directly from COM clients.
3. Derive your plug-in server class from the **PluginServerBase** base class (located in the **Echelon.LNS.Plugin** namespace).
4. Add a static **Main** implementation with an **STAThread** attribute to the class. Main should create an instance of your plug-in server class, and then call the base class **StartPluginServer** method on this instance, passing in the command-line arguments that were passed into **Main** and an array of **Type** objects, one for each plug-in object class that you will create (typically just the one).

The following code demonstrates how your plug-in server class should appear (note that MyPlugin represents the name of your plug-in):

```
using System;
using System.Runtime.InteropServices;
using Echelon.LNS.Plugin;

namespace YourCompany.DevicePlugin
{
    [ComVisible(false)]
    class MyPluginServer : PluginServerBase
    {
        [STAThread]
        public static void Main(string[] args)
        {
            Type[] plugIns = { typeof(MyPluginObject) };

            MyPluginServer server = new MyPluginServer();
            server.StartPluginServer(args, plugIns);
        }
    }
}
```

Implementing the Plug-in Object Class

To implement your plug-in object class, modify the source as follows:

1. Add **using System.Runtime.InteropServices**, **using Echelon.LNS.Plugin**, and **using Echelon.LNS.Interop** directives.
2. Derive your plug-in object class from **PluginObjectBase** (located in the **Echelon.LNS.Plugin** namespace).
3. Add a **ComVisible(true)** attribute and a **public** keyword to your plug-in object class to expose it to COM.
4. Add the additional attributes shown in the following table to your plug-in object class.

Attribute	Description	Value
Guid	Uniquely identifies the plug-in object to COM.	Generate a unique GUID using the Create GUID function from the Tools menu.
ProgId	Human readable version of plug-in identity.	YourCompany.YourProject.MyPlugin, for example
ClassInterface	Disables generation of a default class interface.	**ClassInterfaceType.None**

5. Add a **PluginInfo** public static property that returns a reference to a **PluginInfo** object that describes your plug-in. For efficiency, you should return a reference to a previously created read-only object (for example, using a statically initialized field) instead of creating a new instance each time.

   The **PluginInfo** object provides the registration information for the plug-in that will be written to the Windows registry for access by directors. The following table lists the fields of the **PluginInfo** class and indicates whether each field is required or optional:

Field	Description
PluginName	Unique identifying name of the plug-in. Used as the Windows Registry key for the plug-in. For example, for the Echelon LonPoint plug-in, this is “EchelonLonPointConfiguration”. **Required**
Description	Description of the plug-in that is displayed by directors. This information is typically displayed in a tooltip or status bar. For example, for the Echelon LonPoint plug-in, this is “Echelon LonPoint Configuration”. **Required**
LcaVersion	Minimum version of LNS Object Server that this plug-in requires. This must be 3.22 or higher because the LNS Plug-in Framework does not support LNS releases prior to LNS Turbo Service Pack 2 (3.22). The version number is specified as <major release>.<minor release>. The minor release should contain two digits; however, it 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. This means that “3.2” is assumed to mean “3.20.” **Required**
ManufacturerName	Name of the plug-in manufacturer, which may be displayed by directors. **Required**
Name	Name of the plug-in as displayed to the user by directors. 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." For example, for the Echelon LonPoint plug-in, this is “Echelon LonPoint Configuration”. **Required**
Scope	The scope at which the plug-in should be registered. This is known as the plug-in scope or registration scope. This must be set to **ComponentAppScope.System**. This should not be confused with command scope (see the *Glossary* in Appenidx E for details). **Required**
Version	The version number of this plug-in. The version number is in the same format as **LcaVersion**. **Required**

Field	Description
MultiObject	Indicates to a director whether the plug-in supports 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** field is set to **Feature.Supported** (1), which indicates that MultiObject functionality is supported), then the plug-in must support MultiObject as described in *How Directors Support MultiObject and SingleInstance* in Chapter 3. The **MultiObject** field can also be set to **Feature.Disabled** (0), which indicates that this feature is disabled at the present time although the plug-in does include MultiObject support. The **MultiObject** field can also be set to **Feature.Unsupported** (–1), which indicates that this feature is unsupported. **Optional** (defaults to **Feature.Unsupported**)
SingleInstance	Indicates to a director whether 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 activeIf the **SingleInstance** field is set to **Feature.Supported** (1) (which indicates that SingleObject functionality is supported), then the plug-in must support SingleInstance as described in *How Directors Support MultiObject and SingleInstance* in Chapter 3. The **SingleInstance** field can also be set to **Feature.Disabled** (0), which indicates that this feature is disabled at the present time although the plug-in does include SingleInstance support. The **SingleInstance** field can also be set to **Feature.Unsupported** (–1), which indicates that this feature is unsupported by the plug-in. **Optional** (defaults to **Feature.Unsupported**)

Field	Description
Prelaunch	Indicates to a director whether 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. The **Prelaunch** field can be set to **Feature.Supported** (1), which indicates that Prelaunch functionality is supported. The **Prelaunch** field can also be set to **Feature.Disabled** (0), which indicates that this feature is disabled at the present time although the plug-in does include pre-launch support. The **Prelaunch** field can also be set to **Feature.Unsupported** (–1), which indicates that this feature is unsupported by the plug-in. **Optional** (defaults to **Feature.Unsupported**)
PluginFactoryType	For advanced use only. **Optional**

6. Add a **PluginCommands** public (instance) property that returns a reference to an array of **PluginCommand** objects that describes your plug-in's commands. For efficiency, you should return a reference to a previously created read-only object (for example, using a statically initialized field) instead of creating a new instance each time. The following table lists the fields of the **PluginCommand** class (see *How Plug-ins Implement the Registration Command*, in Chapter 3, for usage considerations):

Field	Description
Name	Name of the plug-in command displayed by directors. For example, this may be "Configure" or "Browse".
Description	Description of the plug-in command, which may be displayed by directors (for example, as a tooltip).
CommandId	An **EnumCommandIds** or **ConstCommandIds** value identifying the command. See *How Plug-ins are Installed and Made Visible to LNS* in Chapter 3 for more information.
ComponentClassId	An **EnumClassIds** or **ConstClassIds** value identifying the LNS objects to which this command can be applied. See *How Directors Launch And Manipulate Plug-ins* in Chapter 3 for more information.
DefaultAppFlag	Whether this command should be the default command for this type of LNS object.

Field	Description
Scope	A **ComponentAppScope** value identifying the scope of the plug-in's command. This is known as the command scope, which should not to be confused with the registration scope or plug-in scope (see the *Glossary* in Appenidx E for details). **Note**: The LNS Plug-in Framework currently only automatically supports the registration of commands at the **ObjectServer** and **System** command scopes; however, you will typically want to use the **DeviceTemplate** and/or **LonMarkObject** scopes for device plug-ins. See *Registering DeviceTemplate and LonMarkObject Scoped Plug-in Commands* later in this chapter for a description of the code your plug-in must implement to support plug-in commands at these scopes.

7. If your plug-in will include a user interface (typical), create a hidden instance of your **PluginForm** class in the **PluginObject** constructor, and assign it to the **PluginObjectBase.PluginForm** base class property.

The following code demonstrates how your plug-in object class should appear (note that MyPlugin represents the name of your plug-in):

```
using System;
using System.Runtime.InteropServices;
using Echelon.LNS.Interop;
using Echelon.LNS.Plugin;

namespace YourCompany.DevicePlugin
{
    [
        ComVisible(true),
        Guid("3CCF87F7-AEFD-4c8e-9936-FEEAB3BBA5A9"),
        ProgId("YourCompany.DevicePlugin.MyPlugin"),
        ClassInterface(ClassInterfaceType.None)
    ]
    public class MyPluginObject : PluginObjectBase
    {
        // describe the plug-in
        private static readonly PluginInfo _PluginInfo = new PluginInfo(
            // standard LNS Plug-in properties
            "YourCompanyMyPlugin",                              // PluginName
            "Description of YourCompany's MyPlugin",            // Description
            "3.22",                                             // LcaVersion
            "YourCompany Corporation",                          // ManufacturerName
            "YourCompany C# Device Plug-in",                    // Name
            ComponentAppScope.System,                           // Scope
            "1.0",                                              // Version

            // advanced LNS Plug-in properties (optional)
            Feature.Supported,      // MultiObject    (default: Unsupported)
            Feature.Supported,      // SingleInstance (default: Unsupported)
            Feature.Unsupported     // Prelaunch      (default: Unsupported)
        );

        public static PluginInfo PluginInfo
        {
            get { return _PluginInfo; }
        }

        // describe the plug-in commands
        private static readonly PluginCommand[] _PluginCommands = new PluginCommand[]
        {
            new PluginCommand(
                "YourCompany Test Device",                      // Name
```

```
                "Tests device.",                        // Description
                EnumCommandIds.Test,                    // CommandId
                EnumClassIds.AppDevice,                 // ComponentClassId
                false,                                  // DefaultAppFlag
                ComponentAppScope.System                // Scope
            )

            // TODO: add additional commands here
        };

        public override PluginCommand[] PluginCommands
        {
            get { return _PluginCommands; }
        }


        // constructor
        public MyPluginObject()
        {
            // create and 'register' the plug-in user interface
            PluginForm = new MyPluginForm();
            PluginForm.Visible = false;
        }
    }
}
```

Note: The **PluginObject** class implements much of the plug-in behavior; therefore, if you are creating more complex plug-ins, you may need to take the following additional steps in your **PluginObject** class:

- Override the **ActivatePlugin** virtual method in your **PluginObject** class and perform the command actions specified in the base class **Actions** property. Typically, though, your plug-in will show a user interface, and you will override the equivalent method in the **PluginForm** class instead (see the following section, *Implementing the Plug-in Form Class*).
- Override any **ILnsPlugin** properties or methods, if extra processing is required before or after the framework implementation is executed (for example, before setting the base class **Network** property, which will open the LNS network, the plug-in may set LNS attributes that need to be set before any networks are opened). Typically this is not necessary.
- Override the base class **RegisterClass** and **UnregisterClass** virtual methods to perform additional plug-in specific registration and unregistration actions. Typically this is not necessary.

Implementing the Plug-in Form Class

To implement your plug-in form class, modify the source code as follows:

1. Add **using System.Runtime.InteropServices**, **using Echelon.LNS.Plugin**, and **using Echelon.LNS.Interop** directives.
2. Derive your plug-in form class from **PluginFormBase** (located in the **Echelon.LNS.Plugin** namespace) instead of **Form. PluginFormBase** is itself derived from **Form**.
3. Override the **ActivatePlugin** virtual method in your plug-in form class and implement the code to perform the command actions specified in your plug-in object base class **PluginObjectBase.Actions** property. There is a reference to your plug-in object stored in the **PluginFormBase.Plugin** property. The **Actions** property is an array of **PluginAction** objects that contain the fields shown in the following table:

Field	Description
CommandId	A value identifying the command.
ComponentClassId	A value identifying the LNS objects to which this command can be applied.

Field	Description
ObjectName	The fully specified name of the LNS object on which to perform the command, as specified by the director.
LcaObject	A reference to the LNS object on which to perform the command, as decoded by the LNS Plug-in Framework (or **null** if unable to decode).

Note: The **Actions** array will contain only one item unless the plug-in MultiObject feature is specified as **Supported**. You can also obtain a reference to the LNS Object Server through the framework's **LcaClass.Instance** method, or to the currently open **Network** and **System** objects through the **LcaClass.Network** and **LcaClass.System** properties, respectively.

The following code demonstrates how your plug-in form class should appear (note that MyPlugin represents the name of your plug-in):

```
using System;
using System.Windows.Forms;
using System.Runtime.InteropServices;
using Echelon.LNS.Plugin;
using Echelon.LNS.Interop;

namespace YourCompany.DevicePlugin
{
    public partial class MyPluginForm : PluginFormBase
    {
        public MyPluginForm()
        {
            InitializeComponent();
        }

        protected override void ActivatePlugin()
        {
            // carry out each queued command action
            foreach (PluginAction action in Plugin.Actions)
            {
                // extract information from command action
                EnumCommandIds commandId = action.CommandId;
                EnumClassIds classId = action.ComponentClassId;
                string objectName = action.ObjectName;
                object lcaObject = action.LcaObject;

                // perform the command action
                switch (commandId)
                {
                    case EnumCommandIds.Test:
                        // TODO: add command handler here
                        break;

                    // TODO: add other command handlers here
                }
            }
        }
    }
}
```

Note: Typically, a device plug-in is launched from the director, and it then allows the user to select various device configuration options from a user interface that is customized to the device functionality. After creating your basic **PluginForm class**, you can add most of your plug-in functionality to this user interface class, adding controls using Visual Studio's Form Designer and updating your controls from your command handlers.

Registering DeviceTemplate and LonMarkObject Scoped Plug-in Commands

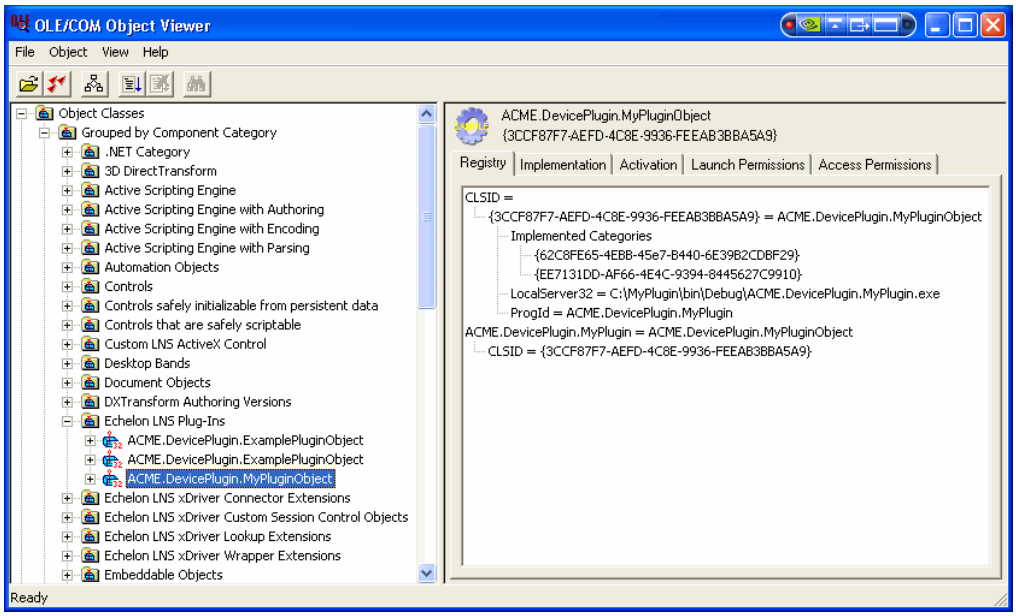
The LNS Plug-in Framework does not currently support registration of **DeviceTemplate** and **LonMarkObject** scoped plug-in commands; however commands at one or both of these scopes are essential to device plug-ins. You can override the framework function and implement registration of **DeviceTemplate** and **LonMarkObject** scoped plug-in commands by following these steps:

1. Set your plug-in command's **PluginCommand.Scope** property to **ComponentAppScope.DeviceTemplate** or **ComponentAppScope.LonMarkObject**.
2. Override the **PluginObjectBase.PreRegisterPluginCommand** and **PluginObjectBase.PreUnregisterPluginCommand** methods and hook the commands with scopes of **ComponentAppScope.DeviceTemplate** and/or **ComponentAppScope.LonMarkObject**. Other command scopes should be sent back to the base class implementation.
3. Implement your plug-in registration command for the **ComponentAppScope.DeviceTemplate** and/or **ComponentAppScope.LonMarkObject** scopes by adding the logic described in *How Plug-ins Implement the Registration Command* in Chapter 3. Upon completion of your processing, return **null** to the **PluginObjectBase.PreRegisterPluginCommand**.
4. Implement your plug-in **unregistration** command for the **ComponentAppScope.DeviceTemplate** and/or **ComponentAppScope.LonMarkObject** scopes by undoing the LNS network database changes that you made when registering your plug-in.

Debugging Plug-ins

You can collect trace information from the LNS Plug-in Framework by copying the **app.config** file from the LonWorks\ObjectServer\Examples\PluginFramework\ExampleCSPlugin to your plug-in's project folder. Visual Studio will automatically copy this file to your plug-in's target folder changing the name of the file to match that of your plug-in. You can then modify this file as desired.

To check that your plug-in has been successfully registered, you can use the **OLE/COM ObjectViewer** to display the registration information associated with your plug-in.



Redistributing your Plug-in

After you have developed and tested your plug-in, you can create an installation project for it so that you can distribute your application to your users. See the documentation for your selected installer for details on how to do this.

You should adhere to the following guidelines when creating an LNS plug-in installer based on the framework:

- A plug-in based on the LNS Plug-in Framework requires Microsoft .NET Framework 2.0 (or newer) to be installed. You can either embed the .NET Framework 2.0 within your plug-in installation, or you can inform your users that but that they must have NET Framework 2.0 installed on their computers. Note that embedding .NET Framework 2.0 within your plug-in adds 40MB to the size of the installation, which may eliminate the possibility for some users to download your plug-in over the Internet.
- Add the two merge modules provided with the LNS Plug-in Framework Developer's Kit to your installation to provide the LNS Interop and LNS Plug-in Framework assemblies. These merge modules will always install to the Windows Global Assembly Cache to provide global access to the assemblies. Note that changing the installation location of these assemblies does not modify the target directory. If your plug-in needs to support LNS 3.23 or later, and not LNS 3.22, you do not need to include the LNS Interop merge module because it is included in LNS 3.23 and higher.
- Register the plug-in with LNS and COM on installation, and unregister the plug-in on uninstallation. This means that the plug-in executable should be run with the **/RegPlugin** command switch near the end of a successful installation, and the plug-in executable should be run with the **/UnRegPlugin** command switch before the plug-in components have been removed during an uninstallation. The fact that registration is handled dynamically, by installed components, means that the runtime of the plug-in must be complete at installation time. For example, if the .NET Framework 2.0 is not yet installed, the plug-in registration will fail.
- Determine the LONWORKS directory location at installation time by reading the value of the **HKEY_LOCAL_MACHINE\Software\LonWorks\LonWorks Path** Windows registry string. The path string will always be absolute, including the drive letter, but will not end in a slash "\" character (for example, "**c:\LonWorks**").
- Install your plug-ins in the **LonWorks\Apps**[*ManufacturerName*]\[*MyPluginName*] directory, as this is the convention. For example, if the LONWORKS directory is set to the default of **C:\LonWorks**, Echelon is the manufacturer, and the plug-in is for the LonPoint plug-in, the plug-in directory should be **C:\LonWorks\Apps\Echelon\LonPoint**.
- Install device files associated with your LNS device plug-ins, such as XIF, XFB, and APB files, in the **LonWorks \Import**[*ManufacturerName*]\[*DeviceName*]\[*VersionName*] directory, as this is the convention (for example, **C:\LonWorks\Import\Echelon\LonPoint\Version3**").
- Before your plug-in can add its registration data to the Windows registry, it must check to see if a newer version of the plug-in is already installed. If so, your plug-in must not overwrite the existing installation. Your plug-in can perform this check by using a Windows Installer product version check.

Note: The LNS Framework Developer's Kit installation and merge modules were created with Macrovision InstallShield 11.5, and require Windows Installer version 3.1 (which will be installed by the LNS Framework Developer's Kit, if necessary). Any recent installation development tool that uses Windows Installer and is designed for .NET redistribution should be sufficient for designing your installation.

3

How Plug-ins Work with Directors

This chapter details how directors and plug-ins interact. Once you understand the basic interaction between directors and plug-ins, using the LNS Plug-in Framework will be straightforward.

How Plug-ins Are Represented in the LNS Object Server

The LNS network database represents physical objects on a LONWORKS network with objects in the database. For example, network databases contain **Router**, **Channel** and **AppDevice** objects.

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 that is used to register the plug-in. Each **ComponentApp** object defines, among other things, an action, which is a command/object class pair. This means that 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** (7)].

The successful completion of plug-in registration is also represented in the LNS Object Server by a **ComponentApp** object. In this case, the command ID of the **ComponentApp** object representing successful registration is **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 command represented by the **ComponentApp**.

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 of that type on devices of that type in that particular system.

How Plug-ins are Installed and Made Visible to LNS

To the user, installing a plug-in is a simple process. As with all Windows applications, the user runs a setup program 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 runs a director, the new plug-in will be listed as available and the user can then register the plug-in with LNS. Note that the manner in which the director treats new plug-ins is up to the director. Some directors may display the new plug-ins in list boxes and let the user select which ones are registered; other directors may automatically register all new plug-ins (although this not recommended because it may result in lots of time spent registering plug-ins that the user does not want to use).

Once a plug-in is registered with LNS, 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. 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 COM servers. Other entries are required by LNS.

LNS requires that a plug-in store registration data for itself in the Windows registry (see *Implementing the Plug-in Object Class* in Chapter 2 for more information). This data 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 COM 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 COM components) to complete the plug-in registration 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. A plug-in can also be registered by being run in standalone mode with the **/regplugin** switch after installing all of its required files. In standalone mode, the user can launch the plug-in, for example, by double clicking the plug-in's icon. LNS requires that plug-ins add their registration data to the Windows registry when they are run in standalone mode with the **/regplugin** switch.

After the plug-in is registered with Windows, the next time the user runs a director, the director will see the new plug-in. This is implemented by the director as follows. When started, directors check the Windows registry to see which plug-ins have registration data listed. For each plug-in listed in the Windows registry that the user chooses to register with LNS, the director checks to see if an equivalent or higher version of that plug-in has already been registered in the LNS Object Server. It does this by checking to see if there is already a **ComponentApp** object for the plug-in's registration command at the appropriate registration command scope. There are two possible values for plug-in registration 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. Plug-ins registered at this level only need to be registered once. After they are registered, they are available to all networks managed by this server.

  Note: This is the only plug-in registration that is stored in the LNS global database. All other plug-in registrations are stored in the LNS network databases. The plug-in registration process described in the following section should *only* register commands at the **ObjectServer** scope for this type of plug-in.

- If the scope is 2 (**LcaScopeSystem**), the director looks in the **ComponentApps** collections of the **System** object belonging to the **ActiveNetwork**. Because each **System** object has its own **ComponentApps** collection, plug-ins registered at this level must be registered once per **Network** (as each network has one **System**).

To complete the registration of a plug-in, the director launches the plug-in and sends it an **LcaCommandRegister** (50) command. The plug-in's registration actions are described in the following section.

How Plug-ins Implement the Registration Command

When a plug-in receives the registration command, it adds **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 LNS Plug-in Framework, version 1.1, automatically registers the actions supported by commands that are at the **ObjectServer** or **System** scopes, based on the settings in the static **PluginCommands** array (see *Implementing the Plug-in Object Class* in Chapter 2). For commands at the **DeviceTemplate** or **LonMarkObject** scope, you must use the following algorithm to implement them:

1. The plug-in fetches the appropriate **ComponentApps** collection, based on the scope of the action. If the scope of the action is device or functional-block specific (for example, 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 **DeviceTemplates.ItemByProgramId** method. If the correct 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 **DeviceTemplate**s 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**) file exists on the server system.

2. The plug-in fetches or creates 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, skip 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, proceed 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, proceed to step 3.

3. The plug-in creates a new **ComponentApp** object via the **ComponentApps.Add** method, specifying the name for this action, the name of the plug-in's server class, and the class ID of the object to which it applies. In the framework, the name of the plug-in's server class may be found in the **ProgId** property of the **PluginObject** class. See Appendix C, *Standard Plug-in Object Classes*, for a list of all the Class IDs for all LNS objects.

4. The plug-in then sets the following additional properties of the **ComponentApp** object:

Attribute	Description
CommandId	The command that you are registering. See Appendix A for a complete list of plug-in command IDs.
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 company's manufacturer ID.

5. Finally, 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. A **ComponentApps** collection can only have one default action; therefore, 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. Note that 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).

Registration is considered successful if the plug-in returns without an exception. 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 Plug-ins Respond to Commands from a Director Other than Registration

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. This is done 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.

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 in Appendix A, *Standard Plug-in Commands*), the class ID of the target object to which it applies, and the object name (path) of the object to which it applies.

Plug-ins based on the LNS Plug-in Framework must implement the Framework-specific **ActivatePlugin()** method to provide processing for each command that it supports. For more information on the **ActivatePlugin()** method, see *Implementing the Plug-in Form Class* in Chapter 2.

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. During normal operation, directors may pre-launch a plug-in when opening a **System** if the plug-in indicates that pre-launch is supported.

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. The director determines the plug-in commands that apply to various objects by searching for the **ComponentApps** collections that are in scope.

For example, right-clicking a device in the LonMaker tool and clicking **Configure** on the shortcut menu starts the plug-in and invokes the Configure command. In this case, the director searches for **ComponentApp** objects that have a **ComponentClassId** of **LcaClassIdAppDevice** (7) in 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.

The following table describes the action-response script between a director and plug-in when a director launches a plug-in to perform an action on a given object.

Director Action	Plug-in Response
Creates an instance of the plug-in by creating an instance of the plug-in's COM 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).	If this is the first time a plug-in is used, the plug-in's out-of-process automation server (executable) will be launched (executing the the class derived from **PluginServerBase**'s **Main**() with a **–Embedding** command-line argument). This, in turn creates an instance of the class derived from **PluginObjectBase**. The constructor runs and starts the Framework-based server, awaiting COM

Director Action	Plug-in Response
	requests. The Plug-in Framework automatically manages the plug-in's reference counts. See *How Plug-ins Know When To Exit* later in this chapter for more information on how reference counts are tracked.
If this is a remote client, sets the **RemoteFlag** property to True and then sets the **NetworkInterfaceName** property to identify the interface that the plug-in should use to communicate with the LNS Object Server. All plug-ins must support these properties. **Note**: 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.	Stores the state of the **RemoteFlag** property. Stores the name of the network interface in the **NetworkInterfaceName** property of the **PlugInObjectBase** 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 using the specified network interface name (if one was previously given). You can override the **NetworkName** property put method if you want your plug-in to do additional work (for example, load device templates).
Optionally sets additional properties.	Responds appropriately. All mandatory properties are automatically handled by the derived **PlugInObjectBase** class. If you want to add any custom properties, you need to override the derived class. See Appendix B, *Standard Plug-in Properties*, for the list of standard plug-in properties and potential responses.
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 Appendix A, *Standard Plug-in Commands*. The target object is identified by class ID and by object name, as described later in this chapter.	If the command is something other than registration, the framework validates the command by making sure that it is in the **PluginCommands** array. Note that the framework 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. See Appendix C, *Standard Plug-in Object Classes*, for addressing syntax for each object type.
Optionally sets the **Visible** property.	Displays the user interface, if appropriate. The **ActivatePlugin()** method is called with an array of actions specified in the **Actions** property of the **PluginObjectBase** class. Each action contains a command and a decoded LNS object.

Director Action	Plug-in Response
Releases the instance of the plug-in it created.	COM decrements the plug-in's usage count. If this is the last user of the plug-in and the plug-in is not visible (for example, the **Visible** property is **False**), the plug-in's destructor is called and the server exits.

How Directors Support Prelaunch

The pre-launch feature allows a plug-in to initialize and open the network when the director is started. A pre-launch sequence initiated by the director performs all plug-in launch operations except sending commands and setting the plug-in's **Visible** property. Pre-launch enables a faster display of the user interface when a plug-in is invoked by a user request. 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** entry in the Windows registry with an entry value of 1. The LNS Plug-in Framework automatically creates this registry entry during plug-in registration if the plug-in specifies support in its **PlugInInfo** structure.
2. Because a plug-in that supports pre-launch also exposes the Prelaunch property, 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 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 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 (the LNS Plug-in Framework automatically handles this).

If the plug-in has the Prelaunch entry set to 1 in the Windows registry, 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** entry 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:

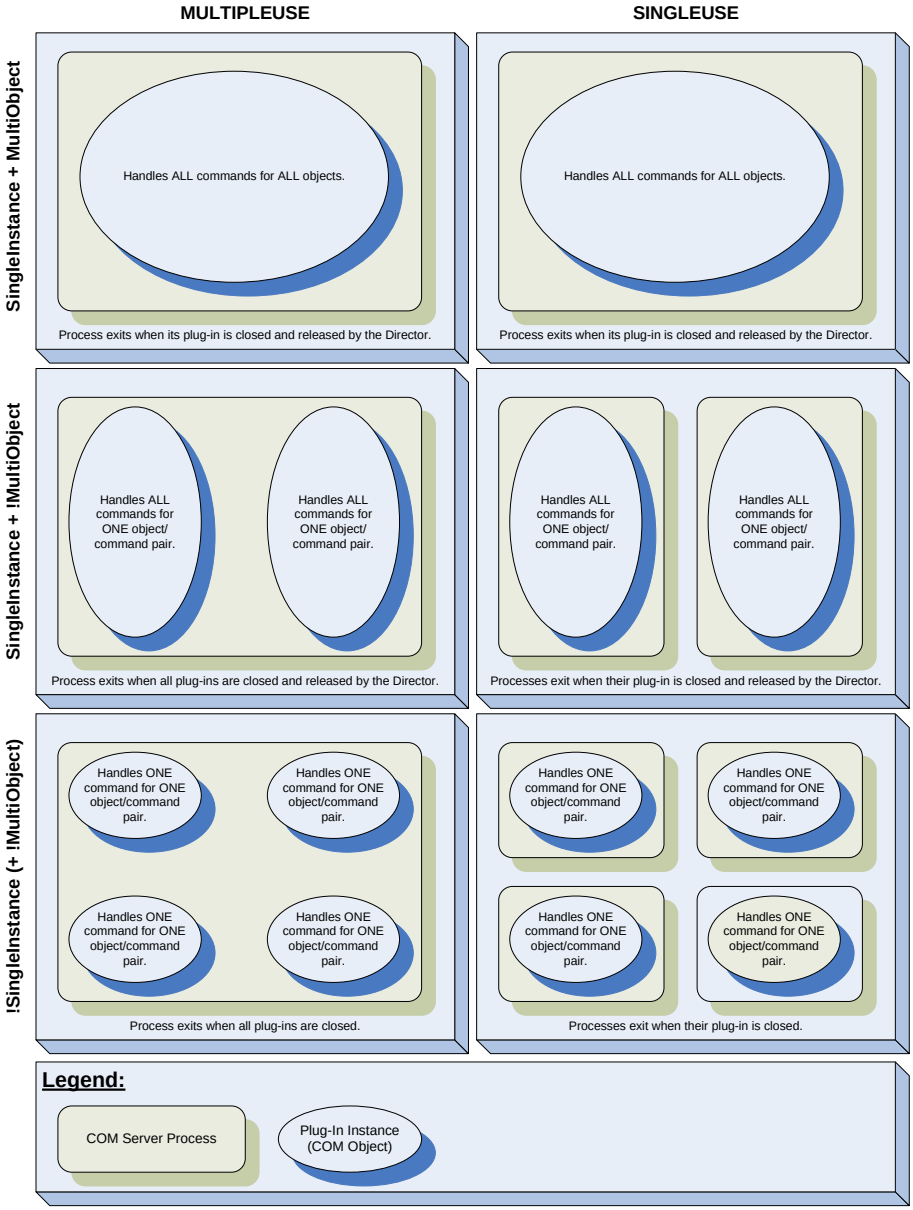
1. If the plug-in has set the optional **MultiObject** value in the Windows registry to 1, and the director is sending several commands in a batch, the director sets the plug-in's **Batch** property (in the COM automation server class that implements the LNS Plug-in API) to 1 (batch started).
2. The 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 **Batch** 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 has not set the **MultiObject** value in the Windows registry to 1, release the reference to it. SingleInstance functionality is described in *Implementing the Plug-in Object Class* in Chapter 2.

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, or explicitly released the plug-in (for example, you can use the **Plug-in Registration** dialog in the LonMaker tool to deregister plug-ins).

Note that you can create a new process for each plug-in instance by overriding the **PluginFactory** base class.

The following diagram illustrates the interaction between MultiObject, SingleInstance, and the COM factory's MULTIPLEUSE and SINGLEUSE flags.



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 COM exceptions. If an error occurs during an operation, a plug-in can raise one of the standard exceptions listed in Appendix D, *Standard Plug-in Exceptions*, 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 may not be clear. From a director's point of view, the life cycle of the plug-in is as described in *How Directors Launch and Manipulate Plug-ins* earlier in this chapter. The plug-in comes into existence when the director creates the plug-in by creating an instance of its 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 (for example, 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 (for example, 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 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 to end the plug-in (other than killing the plug-in from the Windows task manager). 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 are handled in the same way. The plug-in server should keep track of how many directors are using it. The plug-in server 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 LNS Plug-in Framework automatically handles this by maintaining a server-wide object use count and server lock count in the **PluginServerBase** class. The server will automatically shut down when the object count and lock count reach 0. Individual plug-in objects (derived from the **PluginObjectBase** class) automatically manage their object use counts. They do this by incrementing the server's object count upon construction, and decrementing it upon deconstruction.

When a plug-in object implements a user interface by deriving a class from the **PluginFormBase** class, the form itself will maintain a reference to the associated plug-in object when the form is activated (for example, when the director sets the Visible property). When the user closes the form, the reference to the plug-in object is released, allowing the server to shut down if all other references have been released.

What Plug-ins Do When They Run in Standalone Mode

Plug-ins must provide an option to run in standalone mode. When they are started with a command line option of **/regplugin**, such as **MyPlugIn /regplugin**, they must register themselves in the Windows registry. The LNS Plug-in Framework automatically does this.

Plug-ins should also support a command line option of **/deregplugin**; if started with this option a plug-in should remove its LNS plug-in data and automation server registration from the Windows registry. This command line option is implemented in the framework.

If started without this option, a plug-in must, at a minimum, either automatically write its registration data in the Windows registry or provide an option to allow the user to request registry data setup. The framework provides a user interface for this case that allows registration or de-registration options.

The plug-in may also provide other portions of its normal functionality in standalone mode, but it is not required to do so.

You can override the **RunStandalone()** method of the **PluginServerBase** class to provide standalone mode functionality in framework-based plug-ins.

Responding to Property Reads and Writes

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.

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

Uninstallation Issues

Your plug-in should implement the **LcaCommandUnregister** (51) command for LNS network databases in order to allow de-registration of plug-in function from individual network databases, while still providing the plug-in functionality on the computer.

If an LNS device plug-in needs to be uninstalled, the LNS plug-in registration data in the Windows registry may need to be removed before the COM server registration, so that the COM server will be able to perform the plug-in de-registration.

After running the uninstall program (and deleting the registration data in the Windows registry as described above), the LNS device plug-in is properly uninstalled from a Windows operating system point of view.

After Windows uninstallation, 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 a COM exception when it attempts to launch the now uninstalled plug-in and can display an appropriate error message to the user. Because the plug-in software un-installation process is independent from any individual LNS network database access, plug-in directors should implement either automatic or user-specified clean-up when **ComponentApp** references to non-existent plug-ins are encountered.

Generally, uninstalling a plug-in should not remove manufacturer resource files because other devices from the manufacturer may use these.

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.

The following table lists the 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.

Note: 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.

The following table lists the standard LNS plug-in properties. Note that **t**he property names **CharacterEncoding** and **LanguageId** are reserved for future use.

Name	**Type**	**Description**
Height	Long	The height, in pixels, of the plug-in's main window.
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.
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 *http://www.lonmark.org/technical_resources/temp_mid_request*. 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 *http://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.

Name	Type	Description
Batch	Read-Write Long	The optional **Batch** property allows plug-ins that support the MultiObject feature 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 **Batch** 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 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 **Batch** property must also set the MultiObject value in the Windows registry. The Windows registry key is described in *Implementing the Plug-in Object Class* in Chapter 2. If you are using the LNS Plug-in Framework, you can also use the **MultiObject** field of the **PluginInfo** static property of the plugin's object class to have the registry value set for you.
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. To run a plug-in as an LNS remote lightweight client, this property should be set to "Internet". Some early plug-ins require this becusase they do not implement the **RemoteTransport** property.
NetworkName	Read-Write String	The name of the network on which to operate.

Name	Type	Description
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 value in the Windows registry. The Windows registry key is described in *Implementing the Plug-in Object Class* in Chapter 2. If you are using the LNS Plug-in Framework, you can also use the **Prelaunch** field of the **PluginInfo** static property of the plugin's object class to have the registry value set for you. 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.
RemoteFlag	Boolean	Optionally set by the director to indicate if the plug-in is running on the same computer as the LNS Object Server (**RemoteFlag** is False) or on a different computer (**RemoteFlag** is True). In the remote case, the director will set this property before setting the **NetworkName** property. If **RemoteFlag** is True, the **RemoteTransport** property should also be set.
RemoteTransport	Read-write Long	The enumeration specifies the remote client type. It should only be set if the **RemoteFlag** property is True. For LNS full clients, set this property to **1**. For LNS lightweight clients, set this property to **2**.
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.

Name	Type	Description
Version	Read-Only String	Version number of the plug-in. The version number is in "<major 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. This means that "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). Alternatively, 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.

Appendix C

Standard Plug-in Object 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.

The following table lists the standard LNS plug-in classes.

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/data Value
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

Object Class	ID	Addressing Syntax
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

Object Class	ID	Addressing Syntax
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 exceptions that may be thrown by plug-ins. Plug-ins may also throw manufacturer-specific exceptions.

The following table lists the standard LNS plug-in exceptions.

Exception	Code	Description
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

Glossary

This appendix provides definitions for key terms and concepts associated with plug-ins.

Glossary

Action

A command/object class pair implemented by a plug-in. A plug-in is defined by the actions that it can perform (for example, 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.

COM

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

COM Automation Server

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

COM Class

The definition for a programmable COM object.

Class

See *Object Class*.

Class ID

A number that defines a particular LNS object class. See Appendix C, *Standard Plug-in Object Classes*, for a list of all LNS class IDs.

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 Object

The type of LNS object used to represent plug-ins and their actions in the LNS Object Server.

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. 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). Note that the term *Windows class ID* does NOT mean the same thing as the term *class ID* used in this document. When defining the GUID for a plug-in, make sure that it has a unique GUID of its own, and is not simply copied from another plug-in.

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 COM API used by LNS Plug-in directors to instantiate and command LNS plug-ins.

LonMarkObject

The type of LNS database object used to represent LonMark *functional blocks*. The LonMark terminology for this construct changed since LNS was introduced.

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 out-of-process COM 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 COM Server that implements the LNS plug-in.

Registration

The two-phase process by which a plug-in is installed into the LNS Object Server. In the first phase of registration, the plug-in installs registration data 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. This document uses the term scope in three different contexts:

> **Action Scope**, **Command Scope**
>
> For example, if the scope of a command that applies to **AppDevice** objects is **ObjectServer** (1), the command is applicable anywhere within the LNS Object Server. If the scope of the same command were **System** (2), then the command would apply to all 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 (that is, 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 type of functional block on all **AppDevice** objects that use a particular **DeviceTemplate**.

The command 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.

Registration Scope, Plug-in Scope

The Registration command is a special kind of command that must be supported by every LNS plug-in It has either **ObjectServer**-wide scope (1) or **System**-wide scope (2). The scope of the Registration command is also 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)**.

Appendix F

Running the ACME Example C# Plug-in

This appendix describes the function of the ACME Example C# Plug-in provided with the LNS Plug-in Framework Developer's Kit.

Running the ACME Example C# Plug-in

After installing the LNS Plug-in Framework Developer's Kit, the ACME Example C# Plug-in will be installed on your development computer in the same way as any LNS plug-in, which means that it will be visible to all director applications. As a result, you may want to unregister the ACME Example C# Plug-in to prevent networks that you create from registering the plug-in and popping up tracing message boxes. To unregister the ACME Example C# Plug-in, run it in standalone mode from the "Run ACME Example C# Plug-in" shortcut and press the **unregister** button. This will prevent networks that you create from registering the plug-in and popping up tracing message boxes.

The ACME Example C# Plug-in demonstrates the behavior plug-ins and directors exhibit when the plug-in is initially registered on a specific computer, and then registered within a specific LNS network database on that computer. This plug-in and director behavior is described as follows:

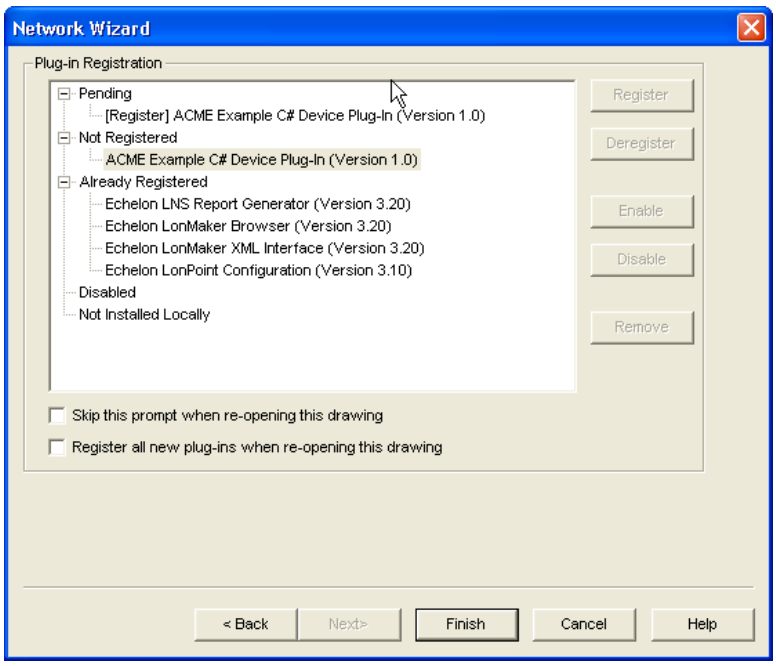
- At plug-in installation, the plug-in installer creates Windows registry entries that tell the director that the plug-in exists on the computer. As described in Chapter 2, *Creating and Redistributing LNS Device Plug-ins*, the contents of the registry entries for your plug-in will be determined by simple table entries during plug-in development.
- When a director creates a new LNS database, it can choose to call a plug-in to have it register itself to be used in that database.
- When a plug-in registers itself in a database, it puts database entries in that tell which objects the plug-in can act on and what commands it can perform on those objects.
- Once the plug-in is registered in that database, the director can call the plug-in to execute those commands on those objects

The ACME Example C# Plug-in example provides pop-up progress message boxes for each plug-in API command issued, and also for some internal actions such as creating and destroying plug-in objects. Note that this is not normally an interactive process and is only done in this example to show the code execution order.

To observe the behavior of the ACME Example C# Plug-in, follow these steps:

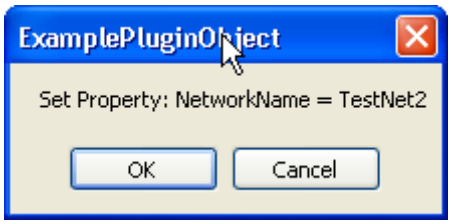
1. Start the director application. This example uses the LonMaker tool as the direction application.
2. Create a new network database or open an existing database.

   The following image shows the LonMaker tool plug-in registration window, which is shown by default during the process of opening a network database. Notice that the ACME Example C# Plug-in is shown to be in the **Registration Pending** state, which will be the case after the LNS Plug-in Framework Developer's Kit has been installed, you open a network database that already exists (and therefore may have all of the other plug-ins registered already), and press the **Register** button.

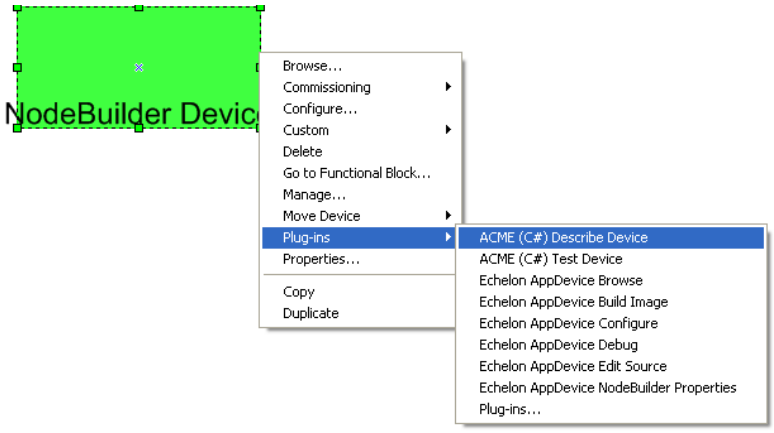


3. Register the **ACME Example C# Device Plug-in (Version 1.0)** plug-in and observe register command calls.

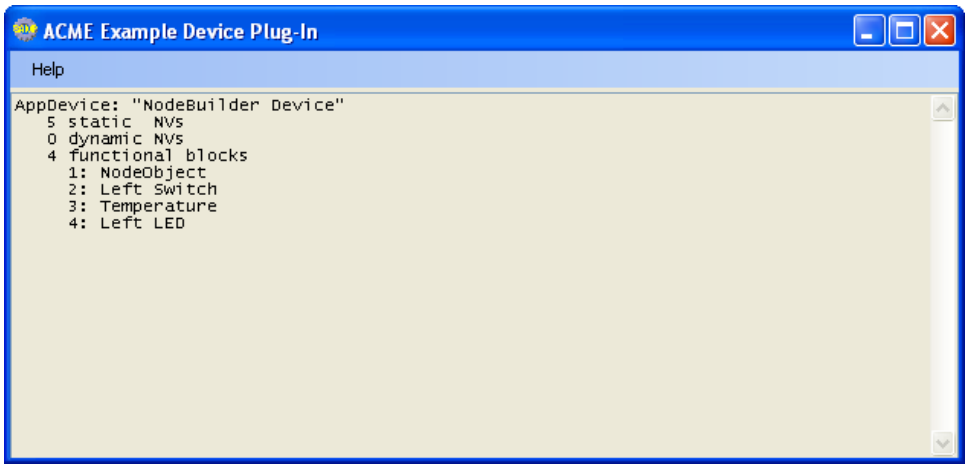
   The following image shows a representative message box seen during the instantiation of the example plug-in. You must click the **OK** button in a timely fashion to allow normal plug-in operation, otherwise the director may time out waiting for the plug-in to respond. If you choose to **Cancel** the Plug-in API call instead, the director will be notified that the Plug-in API call has failed. Note that the pop-up message box may sometimes be obscured by other application windows, but you can see and select it on the Windows task bar.



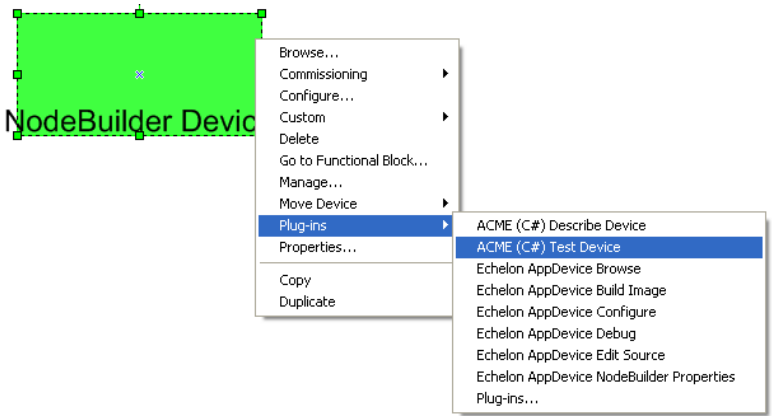
4. Execute the **ACME Describe Device** command on a device in your network. To do this with the LonMaker tool, right-click the device, point to **Plug-ins**, and then click **ACME C# (Describe Device)** on the shortcut menu.



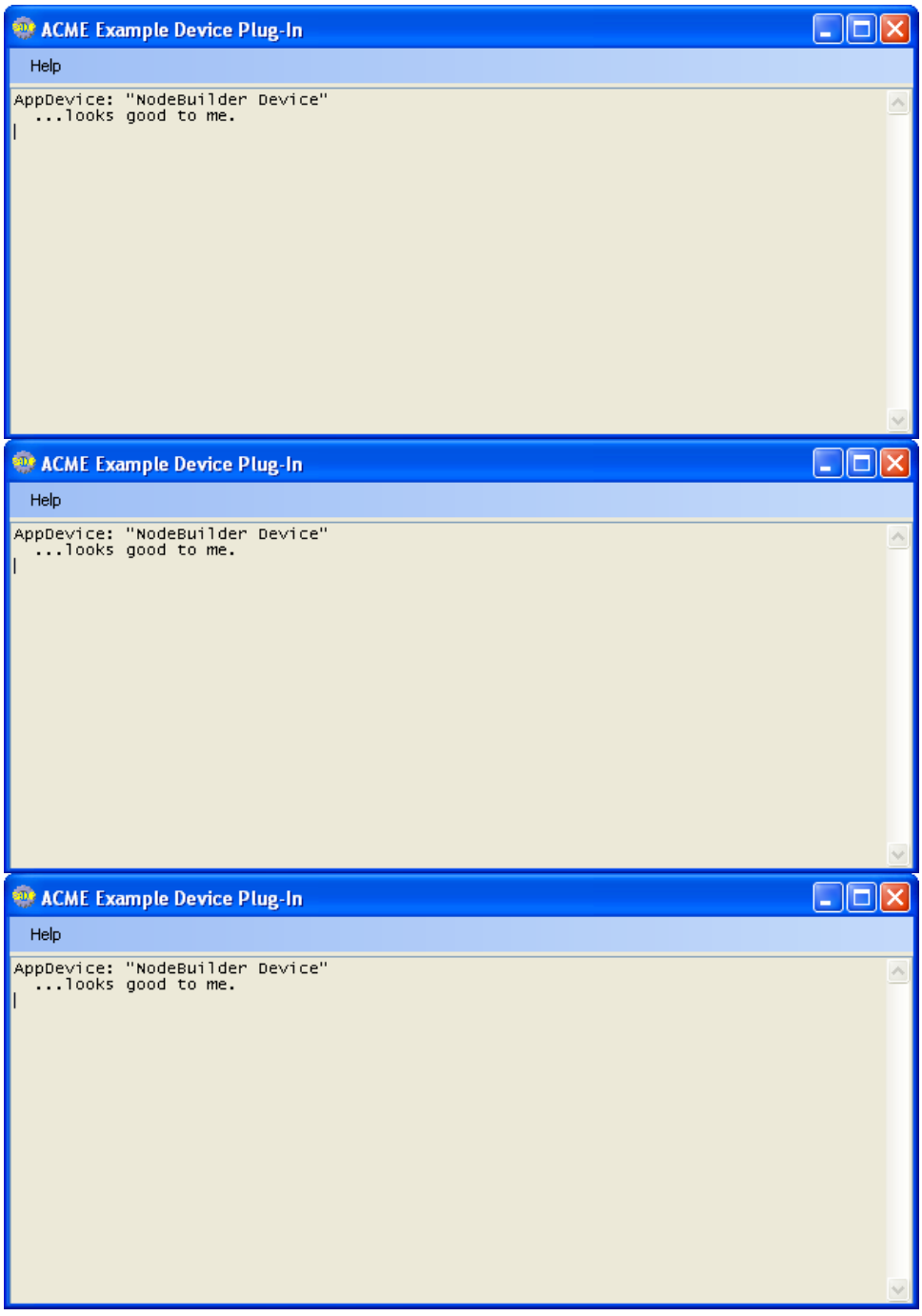
5. Observe the command calls that occur after selecting the describe device command.
6. The **ACME Example Device Plug-in** dialog opens and displays the results of the device description.

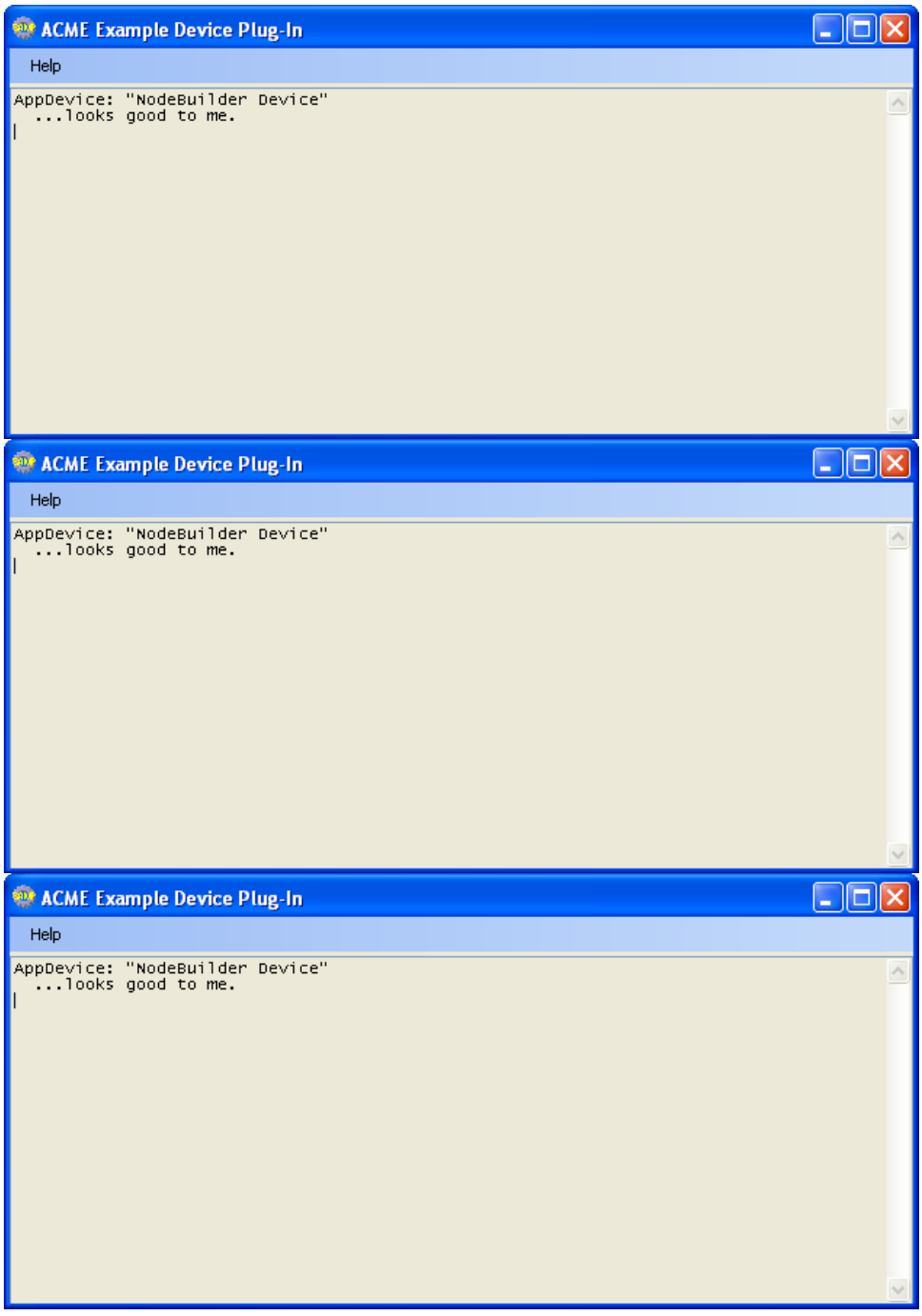


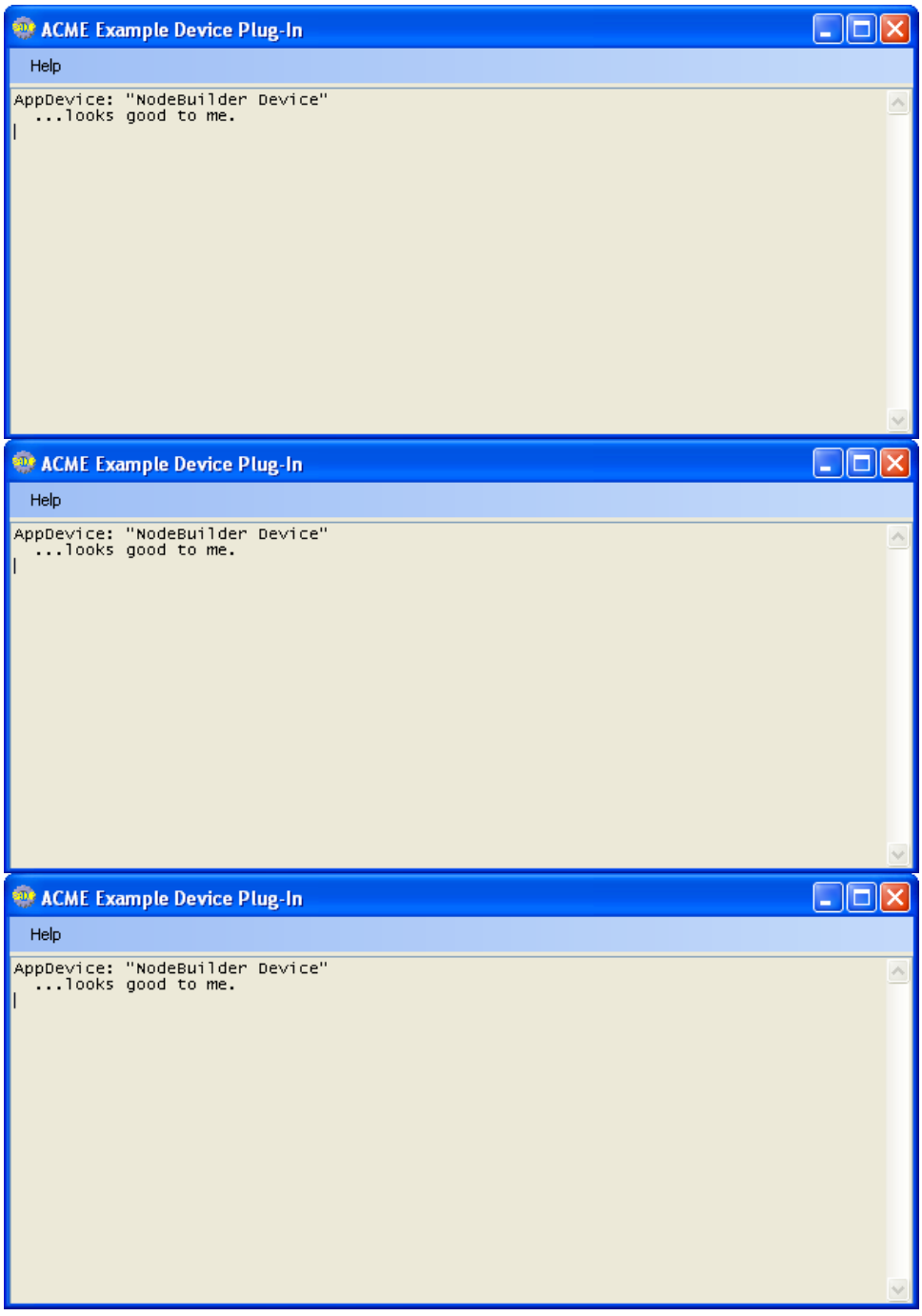
7. Execute the **ACME Test Device** command on a device in your network. To do this with the LonMaker tool, right-click the device, point to **Plug-ins**, and then click **ACME C# (Test Device)** on the shortcut menu.

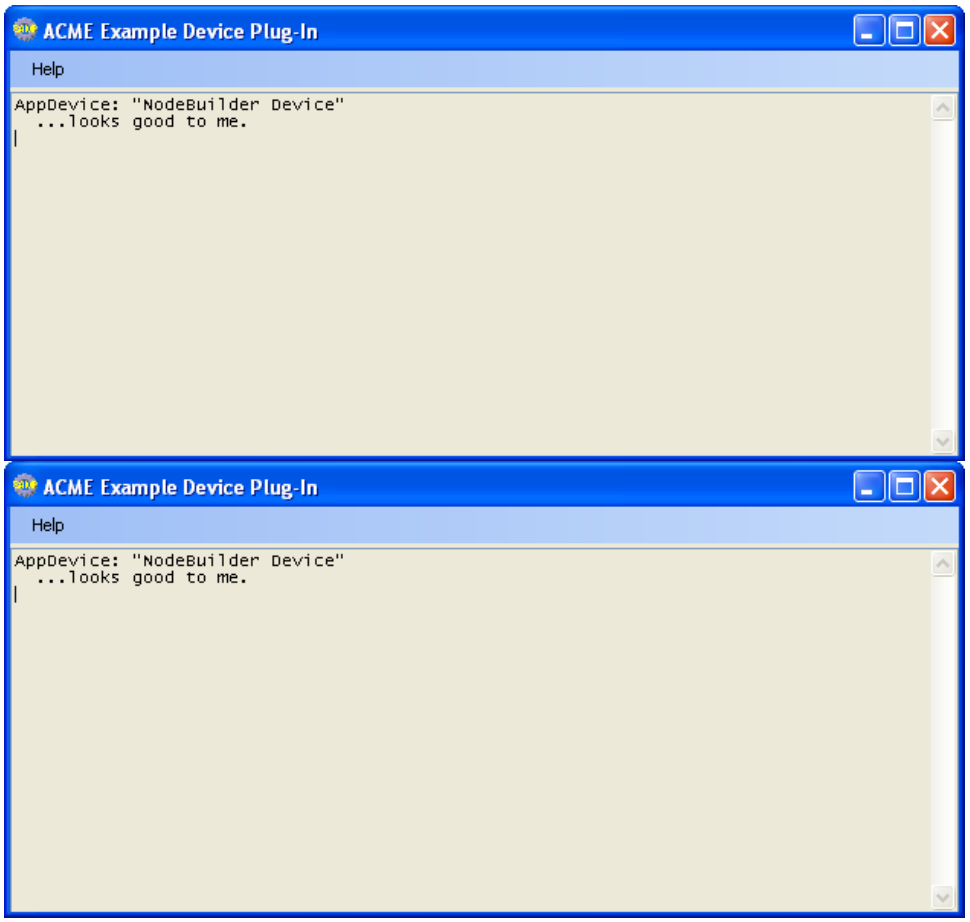


8. Observe the command calls that occur after selecting the test device command.
9. The **ACME Example Device Plug-in** dialog opens and displays the results of the device test.


ACME Example Device Plug-In
Help
AppDevice: "NodeBuilder Device"
...looks good to me.
ACME Example Device Plug-In
Help
AppDevice: "NodeBuilder Device"
...looks good to me.
ACME Example Device Plug-In
Help
AppDevice: "NodeBuilder Device"
...looks good to me.


ACME Example Device Plug-In
Help
AppDevice: "NodeBuilder Device"
...looks good to me.
ACME Example Device Plug-In
Help
AppDevice: "NodeBuilder Device"
...looks good to me.
ACME Example Device Plug-In
Help
AppDevice: "NodeBuilder Device"
...looks good to me.


ACME Example Device Plug-In
Help
AppDevice: "NodeBuilder Device"
...looks good to me.
ACME Example Device Plug-In
Help
AppDevice: "NodeBuilder Device"
...looks good to me.
ACME Example Device Plug-In
Help
AppDevice: "NodeBuilder Device"
...looks good to me.


ACME Example Device Plug-In
Help
AppDevice: "NodeBuilder Device"
...looks good to me.
ACME Example Device Plug-In
Help
AppDevice: "NodeBuilder Device"
...looks good to me.

www.echelon.com