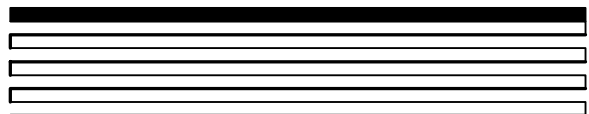


# NodeBuilder® User's Guide

Release 3.1  
Revision 3



078-0141-01E

Echelon, LON, LONWORKS, LonTalk, Neuron, LONMARK, 3120, 3150, NodeBuilder, ShortStack, the LonUsers logo, the Echelon logo, and the LONMARK logo are registered trademarks of Echelon Corporation. LonPoint, LonPoint Schedule Maker, LonMaker and LonSupport are trademarks of Echelon Corporation.

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

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

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

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

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

Model Number 65150

Printed in the United States of America.  
Copyright ©1997-2003 by Echelon Corporation.  
Echelon Corporation  
[www.echelon.com](http://www.echelon.com)

# Table of Contents

<b>Introduction</b>	<b>1-1</b>
Introduction to LONWORKS Networks.....	1-2
<b>Introduction to the NodeBuilder Tool</b>	<b>2-1</b>
Introduction to the NodeBuilder Tool.....	2-2
New Features in Release 3.1 .....	2-2
PL Smart Transceiver Support.....	2-2
Toshiba TMPN3150FR4F Neuron Chip Support.....	2-3
Neuron Chip Operation at 6.5536MHz.....	2-3
Configuration Property Arrays.....	2-4
Enhanced Configuration Property Sharing .....	2-4
Enhanced Changeable Type Network Variable Support .....	2-4
Resource Editor Enhancements .....	2-5
Resource Report Generation .....	2-5
Neuron C Enhancements.....	2-6
Enhanced Support for Clone Domain Configurations.....	2-7
What's Included with the NodeBuilder Tool.....	2-8
Documentation .....	2-9
LTM-10A Platform.....	2-10
LNS DDE Server CD.....	2-11
LonMaker Integration Tool CD.....	2-12
NodeBuilder Development Tool CD.....	2-13
NodeBuilder Gizmo 4 I/O Board .....	2-14
Using a LonBuilder Emulator.....	2-16
What's Required to use the NodeBuilder Tool .....	2-17
Installing the NodeBuilder Tool.....	2-18
Installing the NodeBuilder Software.....	2-19
Installing the NodeBuilder Hardware .....	2-22
Getting More Information and Technical Support.....	2-23
NodeBuilder Quick-Start Tutorial.....	2-24
Goals.....	2-24
NodeBuilder Quick-Start Tutorial: Introduction .....	2-25
NodeBuilder Quick-Start Tutorial: Creating a LonMaker Network .....	2-25
NodeBuilder Quick-Start Tutorial: Creating a NodeBuilder Project .....	2-27
NodeBuilder Quick-Start Tutorial: Creating a NodeBuilder Device Template .....	2-27
NodeBuilder Quick-Start Tutorial: Automatically Generating Neuron C Source Code.....	2-31
NodeBuilder Quick-Start Tutorial: Editing Your Neuron C Source Code.....	2-37
NodeBuilder Quick-Start Tutorial: Compiling, Building, and Downloading Your Application .....	2-38
NodeBuilder Quick-Start Tutorial: Testing Your Device Interface .....	2-41
NodeBuilder Quick-Start Tutorial: Debugging Your Device Application.....	2-42
NodeBuilder Quick-Start Tutorial: Installing and Testing your Device in a Network.....	2-45
NodeBuilder Quick-Start Tutorial: Generating Visual Basic Code for an LNS Device Plug-in .....	2-48
NodeBuilder Quick-Start Tutorial: Testing Your LNS Device Plug-in .....	2-51

<b>Developing a LONWORKS Device</b>	<b>3-1</b>
Introduction to Developing a LONWORKS Device.....	3-2
Sign and Return the OEM License .....	3-2
Design the Device Application and Hardware .....	3-2
Develop the Device Hardware .....	3-3
Define the Device Interface.....	3-3
Create a LonMaker Network.....	3-4
Create a NodeBuilder Project .....	3-5
Create a NodeBuilder Device Template .....	3-6
Create the Neuron C Application .....	3-6
Compile, Build, and Download the Application .....	3-7
Test the Device Interface .....	3-7
Debug the Device Application.....	3-8
Install and Test Your Device in a Network.....	3-8
Create a LonMaker Stencil .....	3-8
Create an LNS Device Plug-in .....	3-8
Develop an Operator Interface.....	3-9
Apply for LONMARK Certification .....	3-9
Create an Installation Application for your Device .....	3-9
<b>Creating and Opening NodeBuilder Projects</b>	<b>4-1</b>
Introduction to NodeBuilder Projects.....	4-2
Introduction to the NodeBuilder Project Manager .....	4-2
Using the Project Pane.....	4-3
Creating a NodeBuilder Project.....	4-4
Creating a New Project.....	4-5
Specifying New Project Name .....	4-5
Specifying the Project Default Settings.....	4-6
Starting the NodeBuilder Tool from the New Device Wizard.....	4-8
Opening a NodeBuilder Project.....	4-10
Selecting a NodeBuilder Project File .....	4-11
Copying a NodeBuilder Project to Another Computer.....	4-11
Copying a NodeBuilder Device Template to Another Computer.....	4-13
Viewing and Printing NodeBuilder XML Files.....	4-13
<b>Creating and Using Device Templates</b>	<b>5-1</b>
Introduction to Device Templates.....	5-2
Using Device Templates .....	5-2
Using the New Device Template Wizard.....	5-5
New Device Template Wizard: New Device Template .....	5-5
New Device Template Wizard: Program ID .....	5-7
New Device Template Wizard: Hardware Templates .....	5-10
Using the Standard Program ID Calculator .....	5-11
Using Device Template Targets .....	5-16
Inserting a Library into a Device Template .....	5-17
Using Hardware Templates .....	5-18
Creating and Editing Hardware Templates.....	5-20
<b>Generating Neuron C Code Using the Code Wizard</b>	<b>6-1</b>
Introduction to the NodeBuilder Code Wizard.....	6-2
Starting the Code Wizard .....	6-2
Defining the Device Interface.....	6-4
Editing Properties in the Code Wizard .....	6-32
Generating Code with the Code Wizard.....	6-32
Files Created by the Code Wizard .....	6-33
Code Generated by the Code Wizard.....	6-35

Modifying Code Generated by the Code Wizard .....	6-37
Neuron C Version 2 Features Not Supported by the Code Wizard .....	6-40
<b>Editing Resource Files</b> .....	<b>7-1</b>
Introduction to Resource Files .....	7-2
Starting the Resource Editor .....	7-5
Setting Resource Editor Options .....	7-6
Introduction to Resource Folders .....	7-7
Browsing the Resource Catalog .....	7-8
Adding a Resource Folder .....	7-10
Removing a Resource Folder .....	7-10
Moving a Resource Folder .....	7-10
Refreshing the Resource Catalog .....	7-11
Searching for a Resource .....	7-11
Creating and Editing a Resource File Set .....	7-13
Creating and Editing Resources .....	7-17
Creating and Editing a Network Variable or Configuration Property Type .....	7-18
Creating and Modifying a Functional Profile .....	7-27
Creating and Modifying an Enumeration Type .....	7-37
Creating and Editing a Language String .....	7-40
Copying Resources .....	7-55
Removing and Obsoleting Resources .....	7-56
Purging a Resource File Set .....	7-57
Converting a Resource File Set .....	7-58
Viewing Resource File Properties .....	7-61
Generating Resource Files .....	7-62
Resource Reports .....	7-63
<b>Editing Neuron C Source Code</b> .....	<b>8-1</b>
Introduction to Editing .....	8-2
Using Syntax Highlighting .....	8-2
Searching Source Files .....	8-3
Using Bookmarks .....	8-7
Setting Editor Options .....	8-8
<b>Compiling, Building, and Loading Applications</b> .....	<b>9-1</b>
Building an Application Image .....	9-2
Files Created When You Build An Application Image .....	9-3
Excluding Targets from a Build .....	9-4
Cleaning Build Output Files .....	9-5
Viewing Build Status .....	9-5
Setting Build Options .....	9-6
Loading an Application Image .....	9-8
Programming 3150 Off-chip Memory .....	9-9
Programming 3150 On-chip Memory .....	9-10
Programming 3120 On-chip Memory .....	9-13
Adding Targets .....	9-13
Adding a Target with the LonMaker Tool .....	9-14
Adding a Target with the Project Manager .....	9-19
Using Targets in the Project Manager .....	9-19
Editing Target Device Settings .....	9-20
<b>Using the NodeBuilder Debugger</b> .....	<b>10-1</b>
Using the Debugger .....	10-2
Starting and Stopping an Application .....	10-4

Setting and Using Breakpoints.....	10-5
Stepping Through Applications.....	10-6
Using the Watch List.....	10-6
Using the Call Stack.....	10-9
Using the Debug Device Manager Pane.....	10-10
Peeking and Poking Memory.....	10-11
Executing Code in Development Targets Only.....	10-12
Using the Debug Error Log Tab.....	10-12
Editing Source Code While Debugging.....	10-13
Setting Debugger Options.....	10-13
<b>Testing a NodeBuilder Device Using the LonMaker Tool</b>	<b>11-1</b>
Testing a NodeBuilder Device.....	11-2
<b>Creating Custom LonMaker Shapes</b>	<b>12-1</b>
Creating a New LonMaker Stencil.....	12-2
Creating a Custom Shape for a Device.....	12-2
Creating Custom Shapes for Functional Blocks.....	12-3
Creating Complex Custom LonMaker Shapes.....	12-4
<b>Creating an LNS Device Plug-in for a NodeBuilder Device</b>	<b>13-1</b>
Introduction to LNS Device Plug-ins.....	13-2
Starting the LNS Device Plug-in Wizard.....	13-2
Registering and Running your LNS Device Plug-in.....	13-3
Deregistering your LNS Device Plug-in.....	13-4
<b>Creating a Human-Machine Interface</b>	<b>14-1</b>
Human-Machine Interfaces.....	14-2
LonMaker Integration Tool.....	14-2
Third-Party HMI's and the LNS DDE Server.....	14-3
<b>Creating a Software Installation</b>	<b>15-1</b>
Creating a Software Installation.....	15-2
<b>Appendix A</b>	<b>A-1</b>
<b>NodeBuilder Example</b>	<b>A-1</b>
Introduction to the NodeBuilder Example.....	A-2
NodeBuilder Example Task 1: Setting Up The Project.....	A-3
NodeBuilder Example Task 2: Configuring the Node Object.....	A-4
NodeBuilder Example Task 3: Adding Digital I/O.....	A-5
NodeBuilder Example Task 4: Analog Input and Output.....	A-8
NodeBuilder Example Task 5: Simple Translator.....	A-12
NodeBuilder Example Task 6: Enhancing the Translator.....	A-14
NodeBuilder Example Task 7: Temperature Sensor.....	A-16
NodeBuilder Example Task 8: Real Time Keeper.....	A-19
NodeBuilder Example Task 9: Wheel Input.....	A-22
Continuing with the NodeBuilder Example.....	A-26
<b>Appendix B</b>	<b>B-1</b>
<b>Converting a NodeBuilder 1.5 Project to a NodeBuilder 3 Project</b>	<b>B-1</b>
Converting a NodeBuilder 1.5 Project to a NodeBuilder 3 Project.....	B-2
Converting a Neuron C Version 1 Application	
to a Neuron C Version 2 Application.....	B-4
Step 1: Build the old application.....	B-4
Step 2: Create a new device template.....	B-4

Step 3: Create Resource Files.....	B-5
Step 4: Create code using the code wizard .....	B-5
Step 5: Move global declarations.....	B-5
Step 6: Move global utility functions and system event handlers .....	B-5
Step 7: Move functional block-specific state management.....	B-5
Step 8: Set resource scopes.....	B-6
Step 9: Test #1 .....	B-6
Step 10: Move input network variable handler.....	B-6
Step 11: Move declarations and handlers for timer and I/O-related events .....	B-7
Step 12: Move application messaging code .....	B-7
Step 13: Test #2.....	B-7
NodeBuilder Project Conversion Tips .....	B-8
Running NodeBuilder 1.5 and NodeBuilder 3 Concurrently.....	B-9
<b>Appendix C</b>	<b>C-1</b>
<b>The Command Line Project Make Utility</b>	<b>C-1</b>
Using the Command Line Project Make Utility .....	C-2
<b>Appendix D</b>	<b>D-1</b>
<b>Using the LonBuilder Emulator</b>	<b>D-1</b>
Using the LonBuilder Emulator.....	D-2
<b>Appendix E</b>	<b>E-1</b>
<b>Using Source Control</b>	<b>E-1</b>
Using Source Control .....	E-2
<b>Appendix F</b>	<b>F-1</b>
<b>NodeBuilder Software License Agreement</b>	<b>F-1</b>
NOTICE .....	F-2
SOFTWARE LICENSE AGREEMENT .....	F-2
DEFINITIONS .....	F-2
LICENSE .....	F-3
TERMINATION .....	F-5
TRADEMARKS.....	F-5
LIMITED WARRANTY AND DISCLAIMER .....	F-5
LIMITATION OF LIABILITY .....	F-6
SAFE OPERATION .....	F-6
LANGUAGE.....	F-7
SUPPORT .....	F-7
GENERAL.....	F-7
<b>Index</b>	<b>i</b>





# 1

## Introduction

This chapter explains the basics of LONWORKS networks, LONWORKS devices, and the Nodebuilder tool. For more detailed information on the LONWORKS platform, see the *Introduction to LONWORKS* document included in the NodeBuilder program folder (to open, click the Windows **Start** menu, point to **Programs**, point to **Echelon NodeBuilder Software**, and then click **Introduction to LONWORKS**).

---

# Introduction to LONWORKS Networks

A LONWORKS network consists of intelligent *devices* (such as sensors, actuators, and controllers) that communicate with each other using a common *protocol* over one or more *communications channels*. Network devices are sometimes called *nodes*.

Devices may be *Neuron hosted* or *host-based*. Neuron hosted devices run a compiled Neuron C application on a Neuron Chip or Smart Transceiver. Host-based devices run applications on a processor other than a Neuron Chip or Smart Transceiver. Host-based devices may run applications written in any language available to the processor. A host-based device may use a Neuron Chip or Smart Transceiver as a communications processor, or it may handle both application processing and communications processing itself or using a different processor.

Each device includes one or more processors that provide its intelligence and implement the LonTalk® protocol. Each device also includes a component called a *transceiver* to provide its electrical interface to the communications channel. All devices on a channel share the bandwidth provided by the channel. Devices need to supply addressing information when sending information on a LONWORKS network to ensure that the message is received by the correct device or devices.

A device publishes information as instructed by the application that it is running. The applications on different devices are not synchronized, and it is possible that multiple devices may all try to talk at the same time. Meaningful transfer of information between devices on a network, therefore, requires organization in the form of a set of rules and procedures. These rules and procedures are the *communication protocol*, often called the *protocol*. The protocol defines the format of the messages being transmitted between devices and defines the actions expected when one device sends a message to another. The protocol normally takes the form of embedded software or firmware code in each device on the network. LONWORKS devices communicate using the LonTalk protocol, which is an open protocol defined by the ANSI/EIA/CEA 709.1 standard.

## Channels

The path between devices exhibits various physical characteristics and is called the *communications channel*, or simply *channel*. Different transceivers may be able to interoperate on the same channel, so channels are categorized by *channel type*, and every type of transceiver must identify the channel type or types that it supports. The choice of channel type affects transmission speed and distance as well as the network topology.

Multiple channels can be connected using *routers*. Routers are used to manage network traffic, extend the physical size of a channel (both length and number of devices attached), and connect channels that use different media (channel types) together. Unlike other devices, routers are always attached to two channels.

## Applications

Every LONWORKS device contains an *application* that defines the device's behavior. The application defines the inputs and outputs of the device. The inputs to a device can include information sent on LONWORKS channels from other devices, as well as information from the device hardware (i.e. the temperature from a temperature sensing device). The outputs from a device can include information sent on LONWORKS channels to other devices, as well as commands sent to the device hardware (i.e. a fan, light, heater, or actuator).

## Program IDs

Every LONWORKS application has a unique, 16 digit, hexadecimal standard program ID with the format FM:MM:MM:CC:CC:UU:TT:NN. This program ID is broken down into the following fields:

### Format (F)

A 1 hex-digit value defining the structure of the program ID. The upper bit of the format defines the program ID as a *standard program ID (SPID)* or a *text program ID*. The upper bit is set for standard program IDs, so formats 8 – 15 (0x8 – 0xF) are reserved for standard program IDs. Program ID format 8 is reserved for LONMARK certified devices. Program ID format 9 is used for devices that will not be LONMARK certified, or for devices that will be certified but are still in development or have not yet completed the certification process. Program ID formats 10 - 15 (0xA – 0xF) are reserved for future use. Text program ID formats are used by network interfaces and legacy devices and, with the exception of network interfaces, should not be used for new devices. The NodeBuilder tool can be used to create applications with program ID format 8 or 9, can be used to create network interfaces using text program IDs, and is also compatible with legacy applications using text program IDs.

### Manufacturer ID (M)

A 5 hex-digit ID that is unique to each LONWORKS device manufacturer. The upper bit identifies the manufacturer ID as a *standard manufacturer ID* (upper bit clear) or a *temporary manufacturer ID* (upper bit set). 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 is easily identified. Standard manufacturer IDs are never reused or reassigned. Temporary manufacturer IDs are available to anyone on request by filling out a simple form at [www.lonmark.org/mid](http://www.lonmark.org/mid). If your company is a LONMARK member, but you do not

know your manufacturer ID, you can find your ID in the list of manufacturer IDs at [www.lonmark.org/spid](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. This list is described in *Using the Standard Program ID Calculator* in Chapter 5.

**Device Class (C)**

A 4 hex-digit value identifying the primary function of the device. This value is drawn from a registry of pre-defined device class definitions. If an appropriate device class designation is not available, the LONMARK Association Secretary will assign one, upon request.

**Usage (U)**

A 2 hex-digit value identifying the intended usage of the device. The upper bit specifies whether the device has a changeable interface. The next bit specifies whether the remainder of the usage field specifies a standard usage or a functional-profile specific usage. The standard usage values are drawn from a registry of pre-defined usage definitions. If an appropriate usage designation is not available one will be assigned upon request. If the second bit is set, a custom set of usage values is specified by the primary functional profile for the device.

**Channel Type (T)**

A 2 hex-digit value identifying the channel type supported by the device's LONWORKS transceiver. The standard channel-type values are drawn from a registry of pre-defined channel-type definitions. A custom channel-type is available for channel types not listed in the standard registry.

**Model Number (N)**

A 2 hex-digit value identifying the specific product model. Model numbers are assigned by the product manufacturer and must be unique within the device class, usage, and channel type for the manufacturer. The same hardware may be used for multiple model numbers depending on the program that is loaded into the hardware. The model number within the program ID does not have to conform to the manufacturer's model number.

See the *LONMARK Application Layer Interoperability Guidelines* for more information about program IDs.

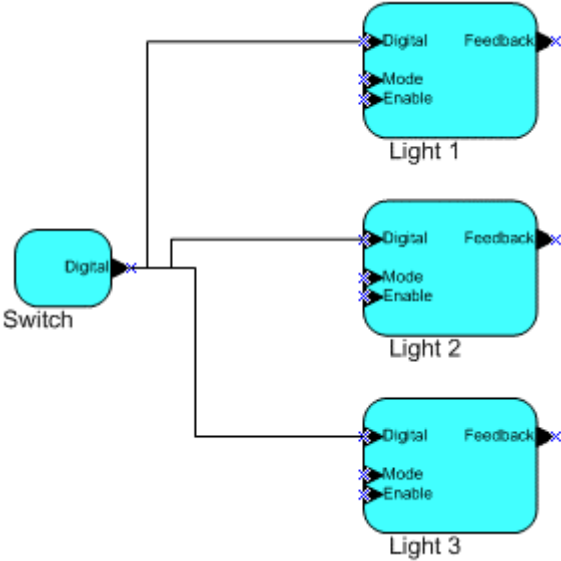
## Network Variables

Applications exchange information with other LONWORKS devices using *network variables*. Every network variable has a *direction*, *type*, and *length*. The network variable direction can be either input or output, depending on whether the network variable is used to receive or send data. The network variable type determines the format of the data.

Network variables of identical type and length but opposite directions can be connected to allow the devices to share information. For example, an application on a lighting device could have an input network variable that was of the switch type, while an application on a dimmer-switch device could have an output network variable of the same type. A network tool such as the LonMaker Integration Tool could be used to connect these two devices, allowing the switch to control the lighting device, as shown in the following figure:



The direction indicated by the triangle in the above figure indicates the direction of the network variable. A single network variable may be connected to multiple network variables of the same type but opposite direction. The following example shows the same switch being used to control three lights:



The application program in a device does not need to know anything about where input network variable values come from or where output network variable values go. When the application program has a changed value for an output network variable, it simply passes the new value to the device firmware. Through a process called *binding* that takes place during network design and installation, the device firmware is configured to know the logical address of the other device or group of devices in the network expecting that network variable's values. It assembles and sends the appropriate packets to these devices. Similarly, when the device firmware receives an updated value for an input network variable required by its application program, it passes the data to the application program. The binding process thus creates logical *connections* between an output network variable in one device and an input network variable in another device or group of devices. Connections

may be thought of as “virtual wires.” For example, the dimmer-switch device in the dimmer-switch-light example could be replaced with an occupancy sensor, without making any changes to the lighting device.

## Configuration Properties

LONWORKS applications may also contain *configuration properties*. Configuration properties allow the device’s behavior to be customized using a network tool such as the LonMaker tool or a customized plug-in created for the device (you can create a custom plug-in using the *LNS™ Device Plug-in Wizard* included with the NodeBuilder tool). For example, an application may allow an arithmetic function (add, subtract, multiply, or divide) to be performed on two values received from two network variables. The function to be performed could be determined by a configuration property. Like network variables, configuration properties have types that determine the type and format of the data they contain.

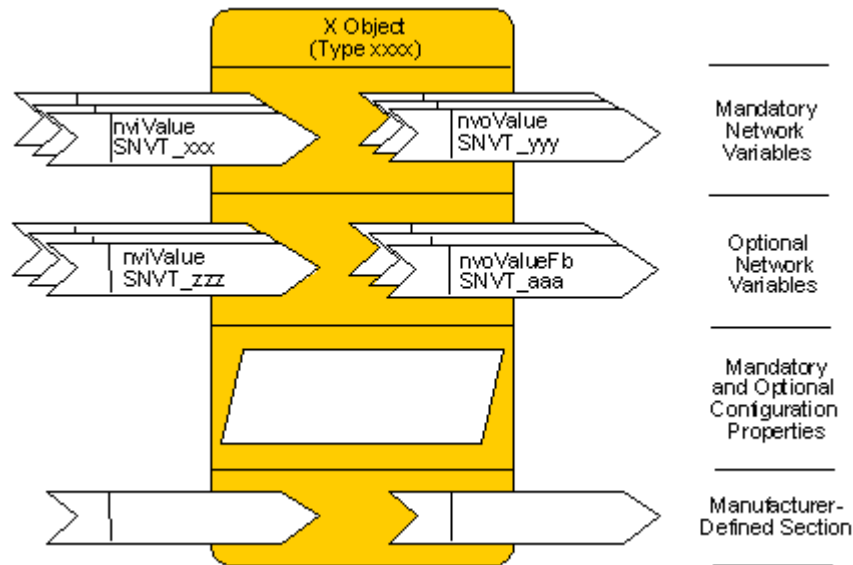
## Functional Blocks

Applications in devices are divided into one or more *functional blocks*. A functional block is a collection of network variables and configuration properties, which are used together to perform a task. These network variables and configuration properties are called the *functional block members*. For example, a LonPoint DI-10 module has four digital input functional blocks that contain the configuration properties and output network variable members for each of the four hardware digital inputs on the DI-10 device.

A functional block is an implementation of a functional profile.

## Functional Profiles

A *functional profile* defines mandatory and optional network variable and configuration property members for a type of functional block. For example, a functional profile for a light controller could have a mandatory network variable for a switch input, and an optional network variable to determine the light color (since not all lights will have multiple color options). The following diagram shows the components of a functional profile:



When a functional block is created from a functional profile, the application designer can determine which of the optional configuration properties and network variables to implement. The application designer can also choose to implement members that are not defined in the functional profile. These are called *implementation-specific members*.

## Hardware Templates

A *hardware template* is a file with a “.nbHwt” extension that defines the hardware configuration for a device. It specifies hardware attributes including platform, transceiver type, Neuron Chip or Smart Transceiver model, clock speed, system image, and memory configuration. Several hardware templates are included with the NodeBuilder tool. You can use these or create your own. Third-party development platform suppliers may also include hardware templates for their platforms.

## Neuron C

*Neuron C* is a programming language, based on ANSI C, used to develop applications for devices that use a Neuron Chip or Smart Transceiver as the application processor. Neuron C includes extensions for network communication, device configuration, hardware I/O, and event-driven scheduling.

## Device Templates

A *device template* defines a device type. The NodeBuilder tool uses two types of device templates. The first is a *NodeBuilder device template*. The NodeBuilder device template is a file with a “.NbDt” extension that specifies the information required for the NodeBuilder tool to build the application for a device. It contains a list of the application Neuron C source files and the hardware template name. When you build the application, the NodeBuilder tool automatically produces an *LNS device template*. The LNS device template defines the external interface to the device, which is called the

*device interface* (XIF), and is used by LNS tools such as the LonMaker tool to configure and bind the device.

## Device Interface Files

A *device interface file* (also known as a XIF file or an *external interface file*) is a file that specifies the external interface of a device. It includes a list of all the functional blocks, network variables, configuration properties, and configuration property default values defined by the device application. LNS tools such as the LonMaker tool use device interface files to create an LNS device template. This enables the tool to be used to create network designs without being connected to the physical devices. A text device interface file with a “.xif” extension is required by the *LONMARK Application Layer Interoperability Guidelines*. A text device interface file is automatically produced by the NodeBuilder tool when you build an application. The NodeBuilder tool also automatically creates binary (“.xfb” extension) and optimized-binary (“.xfo” extension) versions of the device interface file that speed the import process for LNS tools such as the LonMaker tool.

## Resource Files

Resource files define network variable types, configuration property types, and functional profiles. Resource files for standard types and profiles are distributed by the LONMARK Interoperability Association. The standard resource files define standard network variable types (SNVTs), standard configuration property types (SCPTs), and standard functional profiles. For example, **SCPTlocation** is a standard configuration property type for configuration properties containing the device location as a text string, and **SNVT\_temp\_f** is a network variable type for network variables containing a temperature value represented as a floating-point number. The standard network variable and configuration property types are defined in the *LONMARK SNVT and SCPT Guide* included with the NodeBuilder tool (to see this guide, click the Windows **Start** menu, point to **Programs**, point to **Echelon NodeBuilder Software**, and then click **LONMARK SNVT and SCPT Guide**). As new SNVTs and SCPTs are defined, updated resource files and documentation are posted to the LONMARK Web site ([www.lonmark.org](http://www.lonmark.org)). Standard functional profiles are included with the NodeBuilder tool; their documentation is available on the *Design Guidelines* area of the LONMARK Web site. Device manufacturers may also create user resource files that contain manufacturer-defined types and profiles called user network variable types (UNVTs), user configuration property types (UCPTs), and user functional profiles .

You can create applications that only use the standard types and profiles, in which case you will not have to create user resource files. If you need to define any new user types or profiles, you will use the *NodeBuilder Resource Editor* to create new user resource files as described in Chapter 7.

## Targets

A *target* is a LONWORKS device whose application is built by the NodeBuilder tool. There are two types of targets, *development* targets and *release* targets. Development targets are used during development; release targets are used when development is complete and the device will be released to production. Each NodeBuilder device template specifies the definition for a development



target and a release target. Both target definitions use the same source code, program ID, interface, and resource files, but can use different hardware templates and compiler, linker, and exporter options. The source code may include code that is conditionally compiled based on the type of target.



# 2

## Introduction to the NodeBuilder Tool

This chapter provides an overview of the NodeBuilder tool, installation instructions, and a quick-start tutorial.

---

## Introduction to the NodeBuilder Tool

The NodeBuilder tool is a hardware and software platform that is used to develop applications for Neuron Chips and Echelon Smart Transceivers. The NodeBuilder tool enables you to do the following tasks:

- View standard resource file definitions for SNVTs, SCPTs, and standard functional profiles.
- Create your own resource files with your UNVTs, UCPTs, and user functional profiles.
- Automatically generate Neuron C code that implements your device interface.
- Edit your Neuron C code to implement your device functionality.
- Compile and build your application, and download it to the LTM-10A Platform or to your own devices.
- Test with prototype I/O hardware on the Gizmo 4 I/O Board, use the Gizmo 4 board to prototype and test your own I/O hardware, or use your own custom device.
- Install your device into a LONWORKS network and test your device interoperating with other LONWORKS devices.
- Automatically generate Visual Basic code that implements an LNS plug-in for your device.
- Test your LNS plug-in with the LonMaker tool to ensure that your device is easy to configure and install.

---

## New Features in Release 3.1

Release 3.1 of the NodeBuilder Tool includes enhancements in the following areas:

- PL Smart Transceiver support
- Toshiba TMPN3150FR4F Neuron Chip support
- Configuration property arrays
- Enhanced Configuration property sharing
- Enhanced changeable type network variable support
- Resource editor enhancements
- Resource report generation
- Neuron C enhancements
- Enhanced support for clone domain configurations

---

## *PL Smart Transceiver Support*

You can develop applications for Echelon's power line Smart Transceivers. The PL Smart Transceiver can be used to implement low-cost, high-reliability devices that use power line communication. You can develop devices with PL Smart Transceivers that communicate on a PL-20A, PL-20C, or PL-20N channel.

To use one of these chips, select the appropriate transceiver type and model from **Transceiver Type** and **Neuron Chip Model** in the **Hardware** tab of the **NodeBuilder Hardware Template Properties** dialog. See the *PL Smart Transceiver Data Book* for more information on these chips.

If you are developing a PL-20A power line device with version 14 Neuron firmware, you can use a new 6.5536MHz Neuron input clock speed to share a crystal between your Neuron core and transceiver. Sharing a crystal may reduce your device cost and reduce the board space required to implement your device.

To use the new clock speed, select the appropriate clock frequency from **Clock Speed** in the **Hardware** tab of the **NodeBuilder Hardware Template Properties** dialog. The **Clock Speed** list shows clock speeds available for the selected Neuron Chip and transceiver, or the selected Smart Transceiver. Only A-band power line transceivers can be used with Neuron Chips or Smart Transceivers running at the new clock rate. Not all transceiver and Neuron Chip combinations support these new clock rates; see your Neuron Chip or Smart Transceiver data book for more information.

---

## *Toshiba TMPN3150FR4F Neuron Chip Support*

You can develop applications for Toshiba's TMPN3150FR4F Neuron Chip. This chip has a 2KB extended RAM, providing a total of 4KB of on-chip RAM. To use this chip, select **TMPN3150FR4F** from **Neuron Chip Model** in the **Hardware** tab of the **NodeBuilder Hardware Template Properties** dialog. See the Toshiba *TMPN3150FR4F Neuron Chip Data Book* for more information on this chip.

Toshiba includes extended RAM support. *Extended RAM* is on-chip RAM beyond the 2KB RAM in most Neuron 3150 Chips. If you are using the Toshiba TMPN3150FR4F chip, you can enable the extended RAM and assign its starting address to any available page boundary. You cannot use extended RAM if you use off-chip RAM, but you can use extended RAM with off-chip EEPROM, flash memory, or memory-mapped I/O. See *Using On-Chip Extended RAM* in Chapter 5 for more information.

---

## *Neuron Chip Operation at 6.5536MHz*

If you are developing a PL-20A power line device with version 14 (or newer) Neuron firmware, you can use the 6.5536MHz Neuron input clock speed to share a crystal between your Neuron core and transceiver. Sharing a crystal may reduce your device cost and reduce the board space required to implement your device.

To use this new clock speed, select 6.5536MHz as the clock frequency in the **Hardware** tab of the **NodeBuilder Hardware Template Properties** dialog. The **Clock Speed** field shows clock speeds available for the selected Neuron Chip and transceiver, or the selected Smart Transceiver. Only A-band power line transceivers can be used with Neuron Chips running at the new clock rate. Not all power line transceivers or transceiver and Neuron Chip combinations support this new clock rate; see your Neuron Chip or Smart Transceiver data book for more information.

---

## Configuration Property Arrays

You can define a functional profile that contains a configuration property array, and you can implement a functional block that contains a configuration property array using either the NodeBuilder Code Wizard or manually in Neuron C. A *configuration property array* is a set of configuration properties, organized as a single-dimensional array where you can access any member of the array by name and index. A single configuration property array can, circumstances permitting, accommodate the entire Neuron address space (its maximum size is limited to 64KB). For example, you can use a configuration property array to implement tabular configuration data such as a schedule or telephone directory.

Implementing a configuration property array is similar to implementing a scalar (non-array) configuration property. The configuration property array can apply to a single functional block, a single network variable, a functional block array, or a network variable array.

Unless the configuration property array is shared, each of the objects the configuration property array applies to will have its own, private copy of the entire array. See *Adding a Configuration Property Member to a Functional Profile* in Chapter 7 for information on adding a configuration property array to a functional profile template and *Implementing Optional Configuration Properties* in Chapter 6 for information on adding configuration property arrays to a device template.

---

## Enhanced Configuration Property Sharing

You can share a configuration property among multiple functional blocks or network variables. This was possible using the NodeBuilder 3 Neuron C compiler, but NodeBuilder 3.1 adds support for sharing configuration properties using the NodeBuilder Code Wizard. Sharing configuration properties can simplify device configuration by reducing the number of configuration properties that must be set by an integrator, and can also reduce the memory required for the device application. See *Using Global Configuration Property Sharing* in Chapter 6 for more information.

---

## Enhanced Changeable-Type Network Variable Support

You can use a changeable-type network variable to implement a generic functional block that works with different types of inputs and outputs. For example, you can create a general-purpose device that can be used with a variety of sensors or actuators, and then create a functional block that allows the integrator to select the network variable type depending on the physical sensor or actuator attached to the device. Another example is a scheduler that can control a variety of device types by allowing the integrator to change the type of the output of the scheduler. The NodeBuilder 3 tool supported changeable-type network variables, but the NodeBuilder 3.1 tool adds enhanced support for changeable types in the Neuron firmware, Neuron C compiler, and the NodeBuilder Code Wizard. The Code Wizard generates

code that contains a framework for supporting changeable-type network variables.

The new Neuron firmware version 14 included with the NodeBuilder 3.1 tool implements a new method for changing the size of a network variable. This new method uses an *NV length override system image extension* that is managed by the application. Whenever the firmware needs the length of a network variable, it calls the NV length override system image extension to get it. This new method provides more reliable updates to network variable sizes, since the old method could cause a device to go applicationless if a power failure occurred in the middle of a network variable size update. The new system image extension method only works with version 14 firmware, or newer. Since the LTM-10A platform does not use version 14 firmware, you can develop an application that supports both methods, enabling only one of the methods for each type of platform.

See *Using a Changeable-Type Network Variable* in Chapter 6 and *Changeable Type Network Variables* in Chapter 3 of the *Neuron C Programmer's Guide* for more information and a sample changeable-type network variable implementation.

---

## Resource Editor Enhancements

NodeBuilder 3.1 includes a number of resource editor enhancements, including:

- *Resource file conversion.* You can convert resource file sets to earlier format versions to provide compatibility with legacy tools. See *Converting a Resource File Set* in Chapter 7 for more information.
- *Resource deletion.* You can purge resources that you have deleted from a resource file, removing them completely from the file. See *Purging a Resource File Set* in Chapter 7 for more information.
- *Resource file string search.* See *Searching for a Language String* in Chapter 7 for more information.
- *Format priority modification.* When creating or modifying a format, you can now indicate that the selected format is the default for the type, or the default for the type in the specified measuring system. See *Creating and Modifying a Format* in Chapter 7 for more information.
- *Testing scaling factors.* Scaling factors can now be tested with sample data when defining a network variable or configuration property type. See *Creating and Editing a Network Variable or Configuration Property Type*.

---

## Resource Report Generation

You can create a resource report that contains a summary of all the resources in a resource file set, or in multiple resource file sets. You can use a resource report during development as a reference guide for your resource definitions. You can also define supplementary documentation that is automatically included in your resource report. See [types.lonmark.org](http://types.lonmark.org) and [types.echelon.com](http://types.echelon.com) for two examples of resource reports.

**WARNING:** The resource report generator is included as an unsupported component of the NodeBuilder 3.1 product. It has not undergone the same

level of testing as the remainder of the NodeBuilder tool. However, you may find it to be a useful aid to your product development.

See the *Resource Report Generator User's Guide* for more information on creating resource reports. To access this document, click **Start**, point to **Programs**, point to **Echelon NodeBuilder Software**, and then click the **Resource Report Generator Guide**.

---

## Neuron C Enhancements

This section describes new features in the Neuron C programming language. These new features are described more fully in the *Neuron C Programmer's Guide*, and *Neuron C Reference Guide*.

### Compacting the Template File

You can compact the template file that defines the configuration properties implemented within configuration files for your application. You can either compact the template file by merging adjacent CP family members, or by re-ordering and merging CP family members. You can control the template file compacting and ordering using new extensions to the `#pragma codegen` directive.

### CP Files Off-chip and On-chip

You can control the memory location of the CP template and value files using new extensions to the `#pragma codegen` directive.

### IO11 Pin

You can access the new **IO11** pin on the PL Smart Transceivers. To access this pin, use the **bit** I/O object and related I/O functions such as `io_set_direction()`. The `#pragma enable_io_pullups` directive also enables an internal **pullup** for the `IO_11` pin.

### New I/O Models

The following new I/O models are described in the *Neuron C Reference Guide*:

- Hardware SCI (UART) I/O
- Hardware SPI I/O
- Extended touch input/output
- Single timer/counter edgelog I/O
- I<sup>2</sup>C I/O enhancements
- Infrared pattern output
- Magcard/bitstream input

### Other Neuron C Changes

You can create functional blocks without any network variable members. You can use such a functional block to collect various configuration properties into a unit of control.



---

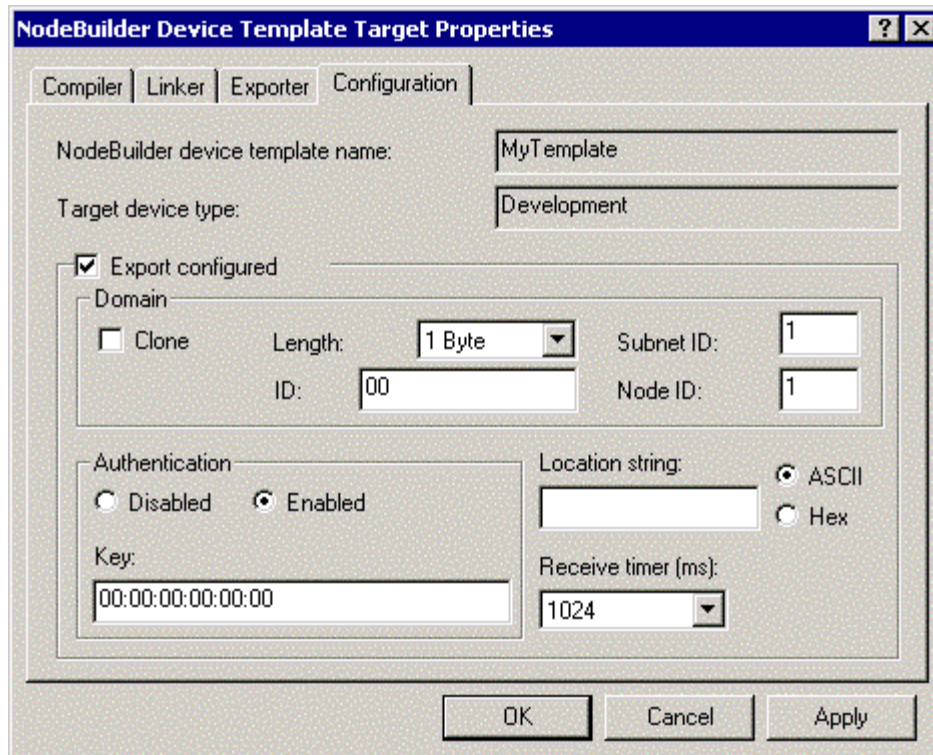
## Enhanced Support for Clone Domain Configurations

If you are exporting your application image as a configured image, you can configure the domain as a clone domain. A *clone domain* is a domain ID within a device that specifies that the device can receive messages from other devices with the same network address. A clone domain is typically only used in self-installed devices where multiple devices within a network may have the same address. Devices using a clone domain have the following reduced capabilities:

- Devices using a clone domain can no longer receive messages in that domain using subnet/node addressing. Some other addressing mode must be used (Neuron ID, group, or broadcast). Use only group and broadcast addressing for self-installed devices since the use of Neuron ID addressing makes systems more difficult to maintain.
- The device cannot receive acknowledgements and responses. The device will, however, continue to send acknowledgements and responses with proper subnet/node information.
- Authentication cannot be used in a clone domain because the reply to a challenge is sent using subnet/node addressing regardless of the addressing format of the original message.
- Devices are no longer protected against receiving their own messages in looping topologies. This must be considered when designing the application. For example, if a device sends out a network variable update, and it also had an input network variable defined with the same network variable selector, its input network variable will get updated if the message is reflected or routed back, which may not be the intention.

To create a clone domain, follow these steps:

1. From the NodeBuilder Project Manager, open a device template, right-click the **Development** or **Release** target, and then select **Settings** from the shortcut menu. The **NodeBuilder Device Template Target Properties** dialog appears.
2. Select the **Configuration** tab, as shown in the following figure:



3. Set the **Clone** checkbox.
4. Set **Length** to anything other than **<None>**. Setting **Length** to **<None>** deactivates the **Clone** checkbox.

---

## What's Included with the NodeBuilder Tool

The NodeBuilder Development Tool is a hardware and software platform that is used to develop applications and LNS device plug-ins for Neuron Chip and Echelon Smart Transceiver based devices. There are three editions of the NodeBuilder tool: the *Full Edition*, the *Classroom Edition*, and the *Upgrade Edition*. The Classroom Edition is for educational-use only. The Upgrade Edition is a product available to anyone who has licensed any previous version of the NodeBuilder or the LonBuilder Development Tool.

The three editions of the NodeBuilder tool consist of the components listed in the following table:

<b>Component</b>	<b>Full Edition</b>	<b>Classroom Edition</b>	<b>Upgrade Edition</b>
Printed Documentation	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
Online Documentation	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
LTM-10A Platform	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
LNS DDE Server OEM Edition	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
LNS DDE Server Demo Edition	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
LonMaker Integration Tool Professional Edition CD	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
LonMaker Integration Tool Standard Edition CD	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
Microsoft Visio Professional	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Microsoft Visio Standard	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
NodeBuilder Development Tool CD	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
NodeBuilder Gizmo 4 I/O Board	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

This section describes each of the components.

---

## **Documentation**

All documentation for the NodeBuilder tool is provided as online Adobe Acrobat PDF files and Windows Help files. Several of the manuals are also provided in printed versions with the Full and Upgrade Editions. The manual you are reading now, the *NodeBuilder User's Guide*, should be your starting point for using the NodeBuilder tool.

- *Gizmo 4 User's Guide*. Describes how to use the I/O devices on the Gizmo 4 I/O Board, and how to use the Gizmo 4 I/O Board to prototype your own I/O hardware.
- *LNS DDE Server User's Guide*. Describes how to use the LNS DDE Server as an I/O driver for human-machine interface (HMI), supervisory control and data acquisition (SCADA), operator interface, and visualization applications.
- *LNS Plug-in Programmer's Guide*. Describes how to develop an LNS device plug-in using the NodeBuilder tool.

- *LTM-10A User's Guide*. Describes how to use the LTM-10A Platform for testing your applications and I/O hardware prototypes. Also describes how you can design the LTM-10A Flash Control Module into your products.
- *Neuron C Programmer's Guide*. Describes how to write programs using the Neuron C programming language.
- *Neuron C Reference Guide*. Provides reference information for the Neuron C programming language.
- *NodeBuilder Errors Guide*. Provides reference information for Neuron C errors, NodeBuilder build process errors, and Neuron firmware errors. This is shipped as an Adobe Acrobat PDF file only.
- *Resource Report Generator User's Guide*. Provides information on using the Resource Report Generator utility to automatically generate resource file documentation. This is shipped as an Adobe Acrobat PDF file only.
- *NodeBuilder User's Guide*. This document. Describes how to install and use the NodeBuilder tool to develop applications and LNS device plug-ins for Neuron Chip and Echelon Smart Transceiver based devices.

---

## LTM-10A Platform

The LTM-10A Platform is a complete LONWORKS device with downloadable flash memory and RAM that you can use for testing your applications and I/O hardware prototypes.



LTM-10A Platform

The LTM-10A Platform includes an LTM-10A Flash Control Module that you can design into your prototypes and products. The LTM-10A module includes a Neuron Chip, 64KByte flash memory, 32Kbyte static RAM, 10MHz crystal oscillator, and custom Neuron firmware. The custom firmware allocates the memory to the Neuron Chip 64Kbyte address space and automatically initializes the transceiver interface for standard transceivers.



LTM-10A Flash Control Module

The NodeBuilder tool can load your application image into the RAM or flash memory of the LTM-10A module. An application image loaded into the flash memory is preserved when the module is powered down. An application image loaded into the RAM is preserved when the module is reset, but not when it is powered down. You can use the Neuron C Debugger to debug applications running in the RAM or flash memory.

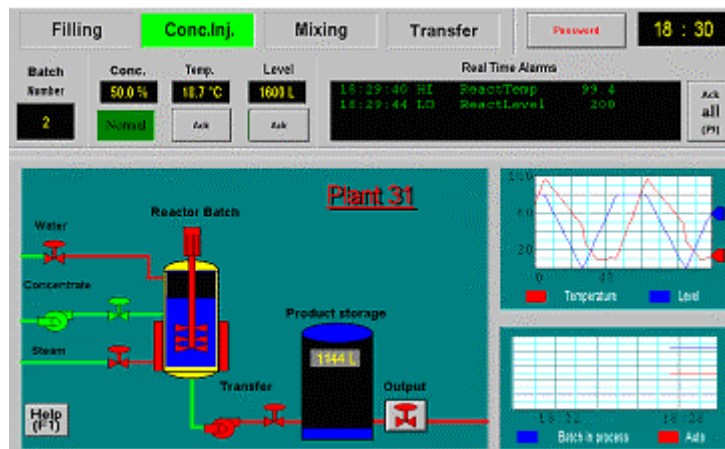
The LTM-10A Platform also includes a transceiver for attaching the platform to a LONWORKS network. The Full and Classroom Editions of the TP/FT-10 NodeBuilder tool include an FTM-10A free topology twisted pair transceiver. The Full Edition of the TP/FT-10 NodeBuilder tool also includes a TPM/XF-1250 twisted pair transceiver that you can use to convert the LTM-10A Platform for TP/XF-1250 application development. The Full Edition of the PL-20 NodeBuilder tool includes a PLM-22 power line transceiver with external power line coupler. Two power line couplers are included, one for 0 - 120V line-to-earth coupling and one for 0 - 240V line-to-neutral coupling.

The LTM-10A module also includes a Microprocessor Interface Program (MIP) function that enables the LTM-10A module to be used as a LONWORKS network interface for any microprocessor or microcontroller of your choice. The NodeBuilder software supports development of MIP-based devices using LTM-10A modules. You can develop custom network interfaces by downloading the ShortStack Developer's Kit from [www.echelon.com/shortstack](http://www.echelon.com/shortstack), or by licensing the MIP/P20 and MIP/P50 Developer's Kit or the MIP/DPS Developer's Kit.

See the *LTM-10A User's Guide* for more information on the LTM-10A Platform and Flash Control Module.

## LNS DDE Server CD

The LNS DDE Server is an I/O driver for human-machine interface (HMI), supervisory control and data acquisition (SCADA), operator interface, and visualization applications. The LNS DDE Server is not required by the NodeBuilder software, but it provides an easy and high-performance way to access your LONWORKS networks and devices from applications that support a DDE, Fast DDE, or SuiteLink interface such as Wonderware InTouch. The following figure shows a typical operator interface display built with an HMI application running with the LNS DDE Server.



## HMI Using LNS DDE Server

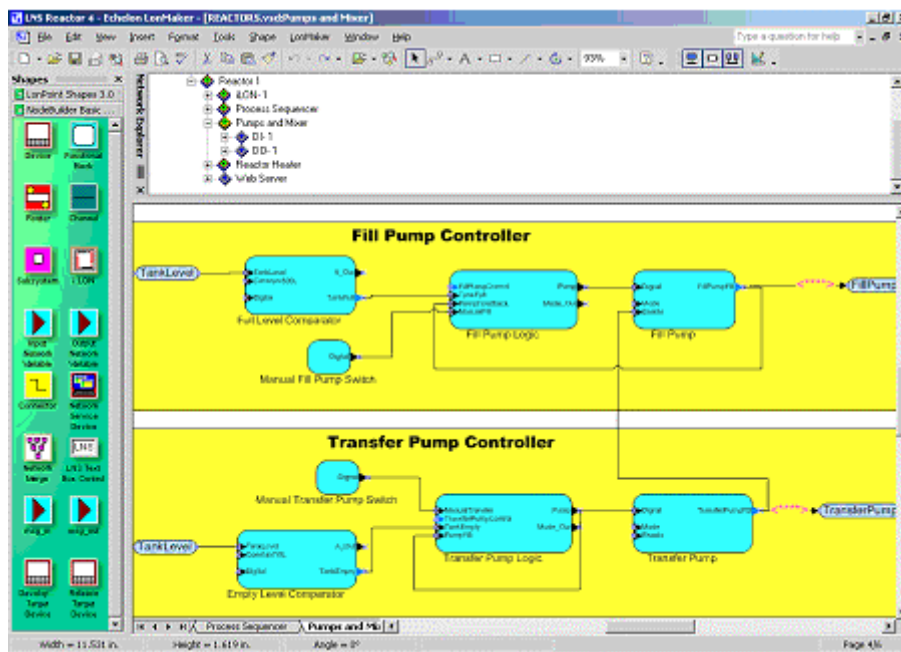
The Full Edition of the NodeBuilder tool includes the OEM Edition of the LNS DDE Server, which is the full edition of the LNS DDE Server. The Classroom and Upgrade Editions of the NodeBuilder Tool include the Demo Edition of the LNS DDE Server. The Demo Edition can be upgraded to the full edition by purchasing a key as described in the LNS DDE Server documentation. See the *LNS DDE Server User's Guide* for information on installing and using the LNS DDE Server.

---

## LonMaker Integration Tool CD

The LonMaker Integration Tool is a software application for designing, installing, operating, and maintaining multi-vendor, open, interoperable LONWORKS networks. Based on the LNS network operating system, the LonMaker tool combines a powerful client-server architecture with an easy-to-use Microsoft Visio™ user interface. The result is a tool that is sophisticated enough to be used to design, commission, operate, and maintain a LONWORKS network yet economical enough to be left behind as an operation and maintenance tool.

The LonMaker tool is available in Professional, Standard, and Upgrade Editions. The NodeBuilder tool Full Edition includes the LonMaker Professional Edition, which includes Microsoft Visio Professional. The NodeBuilder tool Classroom Edition includes the LonMaker Standard Edition, which includes Microsoft Visio Standard. The NodeBuilder tool Upgrade Edition does not include the LonMaker tool—it must be ordered separately. You already have the LonMaker tool if you are upgrading from the NodeBuilder 3 (or newer) tool.



LonMaker Integration Tool

The LonMaker tool is an integral part of your NodeBuilder tool. The NodeBuilder Project Manager is an LNS plug-in that is called from the

LonMaker tool, much like your own LNS device plug-ins may be called from the LonMaker tool.

The LonMaker tool lets you do the following tasks:

- *Network Design* — The LonMaker tool allows you to design a network without being connected to it. This allows network design to be done off site (engineered system installation scenario). The LonMaker tool also allows network design to take place on site (ad-hoc installation scenario), which is what you will use with the NodeBuilder tool, and is also desirable for smaller networks or networks in which the network topology is unknown until on-site. The LonMaker tool can learn the design from an existing network; this process is called recovery. The LonMaker tool also enables an engineered, ad-hoc, or recovered network to be changed at any time.
- *Network Installation* — The LonMaker tool allows an engineered network to be rapidly installed once the network design is brought on site. The engineered device definitions can be quickly and easily associated with their corresponding physical devices to reduce on-site commissioning time. The LonMaker Browser provides complete access to all network variables and configuration properties. The LonMaker Manage window allows you to test and manage your devices. The LonMaker Browser and Manage windows are very useful both for development and for field use.
- *Network Documentation* — Since the LonMaker tool creates a Visio drawing in parallel with the network design and installation process, this drawing accurately represents the installed network, making it an essential component of as-built reports.
- *Network Operation* — The LonMaker tool supports the operation of a network using operator interface pages contained within the LonMaker drawing.
- *Network Maintenance* — The LonMaker tool allows devices, routers, channels, subsystems, and connections to be easily added, tested, removed, modified, or replaced to support system maintenance.

This guide describes many of the LonMaker functions that you will use with the NodeBuilder tool. See the *LonMaker User's Guide* for more information on the LonMaker tool and to learn how it can be used to install, operate, and maintain your operational networks in addition to your development networks.

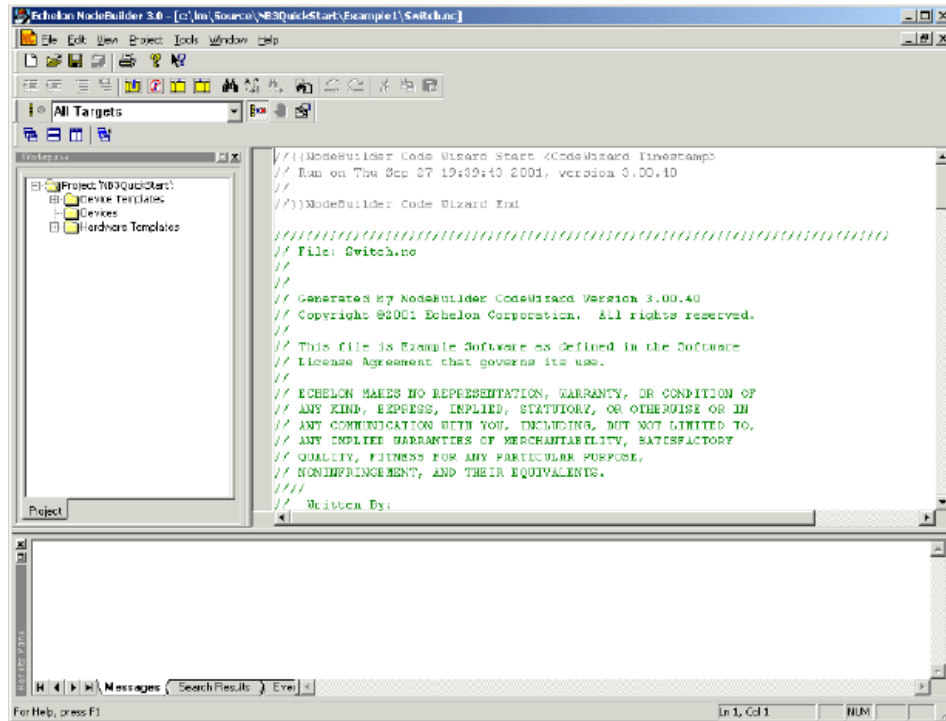
---

## ***NodeBuilder Development Tool CD***

The NodeBuilder Development Tool CD contains the software required to develop and debug Neuron C applications and LNS device plug-ins for your LONWORKS devices. This software includes the following components:

- *NodeBuilder Quick-Start Tutorial* – Learn how to use the NodeBuilder tool to develop devices and LNS device plug-ins with this animated tutorial.
- *NodeBuilder Resource Editor* – Define your device interface by selecting appropriate standard types and functional profiles, or create your own if you need types or profiles not included in the standard resource files.
- *NodeBuilder Code Wizard* – Automatically generate Neuron C source code that implements your device interface.
- *NodeBuilder Project Manager* – Customize the generated Neuron C source code; build and download your application image to the LTM-10A Platform or

your own hardware; and debug your application running on the LTM-10A Platform or your own hardware with a source-level view of your application as it executes.



NodeBuilder Project Manager

- *LNS Device Plug-in Wizard* – Develop LNS device plug-ins to configure your devices.

---

## NodeBuilder Gizmo 4 I/O Board

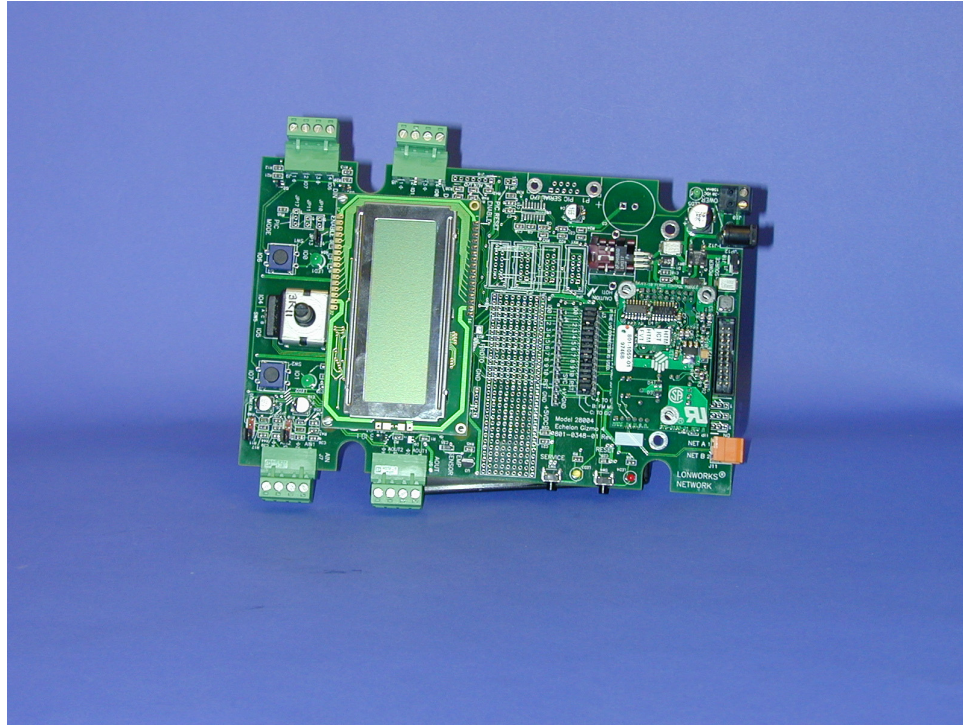
The NodeBuilder Gizmo 4 I/O Board is a collection of I/O devices that you can use with the LTM-10A Platform for developing prototype devices and I/O circuits, developing special-purpose devices for testing, or running the NodeBuilder examples. The following figure illustrates the Gizmo 4 plugged into the LTM-10A Platform.





#### Gizmo 4 Plugged-into LTM-10A Platform

You can also plug a TP/FT-10 or TP/FT-10F Control Module into the Gizmo 4 to create a self-contained LONWORKS device. This requires separate purchase of the TP/FT-10 or TP/FT-10F Control Module. The following figure illustrates this configuration.



Gizmo 4 with a TP/FT-10F Flash Control Module

The Gizmo 4 includes the following I/O devices:

- 4 line x 20 character LCD display
- 2 10-bit resolution analog inputs with screw terminal connector
- 2 8-bit resolution analog outputs with screw terminal connector
- 2 digital inputs with screw terminal connector and pushbutton inputs
- 2 digital outputs with screw terminal connector and LED outputs
- Digital shaft encoder
- Piezoelectric transducer
- Real-time clock
- Temperature sensor

A Gizmo 4 I/O library is included with the NodeBuilder software that provides easy-to-use high-level functions for accessing the display, analog I/O, piezo transducer, real-time clock, and temperature sensor.

This user's guide describes how to use the NodeBuilder 3 examples with the Gizmo 4 board. See the *Gizmo 4 User's Guide* for a description of the I/O devices on the Gizmo 4 board and a description of the Gizmo 4 I/O library.

---

## Using a LonBuilder Emulator

The NodeBuilder 3.1 software can be used to download applications to a LonBuilder Emulator and debug applications on the emulator. The NodeBuilder software does not use the debugging hardware support built into the emulator. Instead, it uses the emulator as any other downloadable

device and interfaces to the emulator using the NodeBuilder debug kernel linked with your application. See Appendix D, *Using the LonBuilder Emulator* for details.

---

## What's Required to use the NodeBuilder Tool

You will need a LONWORKS network interface and a computer to use the NodeBuilder tool.

You can use any LNS compatible network interface. Your computer must be able to communicate with your target device. A high-performance network interface is recommended, such as a PCC-10 or PCLTA-20 with the PCCVNI firmware selected in the Windows Control Panel. You can accomplish this by using compatible transceivers in both devices, or you can accomplish this by using a router between your PC and the target device. The NodeBuilder tool has been tested with the following network interfaces and routers:

- *i.LON 1000 Internet Server*. Enables development from a computer with a standard IP LAN card. The i.LON 1000 is used as a router to your development network.
- *i.LON 100 Internet Server*. Enables development from a computer with a standard IP LAN card. The i.LON 100 is used as a remote network interface, connecting your computer with your devices over the LAN or the Internet. You can also use the i.LON 100 as a Web server, scheduler, data logger, and alarm monitor for your devices.
- *i.LON 10 Ethernet Adapter*. Enables development from a computer with a standard IP LAN card. The i.LON 10 is used as a remote network interface, connecting your computer with your devices over the LAN or the Internet.
- *PCLTA-20 Network Interface*. Enables development from a desktop computer with a PCI bus.
- *PCLTA-10 Network Interface*. Enables development from a desktop computer with an ISA bus.
- *PCC-10 Network Interface*. Enables development from a laptop computer with PC Card slot.

**Note:** Running the NodeBuilder software on a different computer than the LNS Server is not supported.

The NodeBuilder tool Full and Classroom editions include drivers for the *i.LON 100*, *i.LON 10*, *PCLTA-20*, *PCLTA-10*, and *PCC-10* interfaces. If you are using an *i.LON 1000 Internet Server*, the *i.LON* software must be installed on your computer, and the LONWORKS/IP channel must be configured using the *i.LON Configuration Server* software. The *i.LON 1000* software is available at [www.echelon.com/ilon](http://www.echelon.com/ilon).

If you are using an *i.LON 100* or *i.LON 10* interface with the NodeBuilder 3.1 Upgrade Edition, you must download and install LNS 3 Service Pack 8 or newer. LNS service packs are available at [www.echelon.com/downloads](http://www.echelon.com/downloads).

Your computer must meet the following minimum requirements:

- Microsoft Windows XP, Windows 2000, or Windows 98 (Windows XP or Windows 2000 recommended)
- Pentium 200MHz or faster (Pentium II 350MHz or faster recommended)

- 128MB RAM minimum (256MB or more recommended)
- 440MB free hard disk space. The LonMaker tool requires 350MB of free space, and the NodeBuilder tool requires 90MB of free space.
- CD-ROM drive
- Super VGA (800 × 600) or higher-resolution display with 256 colors
- Mouse or other Windows compatible pointing device

If you will be using the LonMaker tool to design or manage large networks, there are additional computer requirements described under *Enhancement for Larger Networks* in Chapter 2 of the *LonMaker User's Guide*.

If you licensed the NodeBuilder upgrade, you will also need a development platform. You can use the LTM-10 platform if you originally licensed a NodeBuilder 1.5 tool, or you can use the LonBuilder Development Station if you originally licensed a LonBuilder 3.01 tool.

---

## Installing the NodeBuilder Tool

To install your NodeBuilder tool, follow these steps:

**WARNING:** You must install the LonMaker software before installing the NodeBuilder software, as described in the following procedure.

1. You will need a manufacturer ID to use many of the functions of the NodeBuilder tool. 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 is easily identified. Standard manufacturer IDs are never reused or reassigned. If you need a temporary manufacturer ID, fill out the simple form at [www.lonmark.org/mid](http://www.lonmark.org/mid) and you will be assigned a temporary ID instantly while you are online. If your company is a LONMARK member, but you do not know your manufacturer ID, find your ID in the list of manufacturer IDs at [www.lonmark.org/spid](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. This list is described in *Using the Standard Program ID Calculator* in Chapter 5.
2. If you will be developing an LNS device plug-in for your device, install Microsoft Visual Basic 6 (with SP5 or newer) as described in the Visual Basic documentation. You can install this software later, but you will need to reinstall the NodeBuilder software after installing Visual Basic to activate the LNS Device Plug-in Wizard.
3. If you will be using an i.LON device as a network interface, install the i.LON software and hardware as described in the *i.LON* documentation. If you are using the *i.LON* 1000, you must be using at least version 1.01 of the *i.LON* software. If you have the 1.0 *i.LON* 1000 software, download the 1.01 (or newer) upgrade from [www.echelon.com/ilon](http://www.echelon.com/ilon).
4. Install the LonMaker software as described in the *LonMaker User's Guide*. If you will be using a PCLTA-20, PCLTA-10, PCC-10, PCNSI, or SLTA-10 network interface, install the driver from the LonMaker CD as described in the *LonMaker User's Guide*.

5. Install the NodeBuilder software as described in the next section, *Installing the NodeBuilder Software*.
6. If you will be using the LNS DDE Server, install the LNS DDE Server as described in the *LNS DDE Server User's Guide*. You do not need to reinstall the LNS runtime or any of the drivers from the LNS DDE Server installation. The LNS DDE Server is not required to use the NodeBuilder software and can be installed at any time.
7. Install the NodeBuilder hardware as described in *Installing the NodeBuilder Hardware* later in this chapter.
8. Install your network interface hardware. If you will be using a PCLTA-20, PCLTA-10, PCC-10, PCNSI, or SLTA-10 network interface, install the hardware as described in the *LonMaker User's Guide*. If you will be using an SLTA-10 network interface, install the hardware as described in the *SLTA-10 User's Guide*. If you will be using an i.LON device as a network interface, ensure that you have a standard IP interface such as an Ethernet NIC installed in your PC and install the i.LON interface as described in the i.LON documentation.
9. Your licensed copy of the LonMaker software includes 64 free LonMaker credits. A *LonMaker credit* is a token representing a prepaid fee to commission a device. You can use LonMaker credits in one network or in multiple networks. LonMaker credits are associated with the LonMaker application and the PC running it and are stored in a file called the LonMaker license file. The LonMaker tool keeps track of the number of credits you have available. When you initially install the LonMaker tool, you have 64 free LonMaker credits to start your development. You can order up to 500 free credits for development use per year per device type that you develop. To get started, order 500 development credits as described in Chapter 8 of the *LonMaker User's Guide*.
10. Your licensed copy of the LNS DDE Server requires an application key to operate in unlimited mode. A credit for a free application key is included with the NodeBuilder tool Full Edition. To order your free application key—or to order an application key for the Upgrade or Classroom Edition—generate and send an application key order as described in the *License Settings* section in Chapter 3 of the *LNS DDE Server User's Guide*.

---

## *Installing the NodeBuilder Software*

To install the NodeBuilder software, follow these steps:

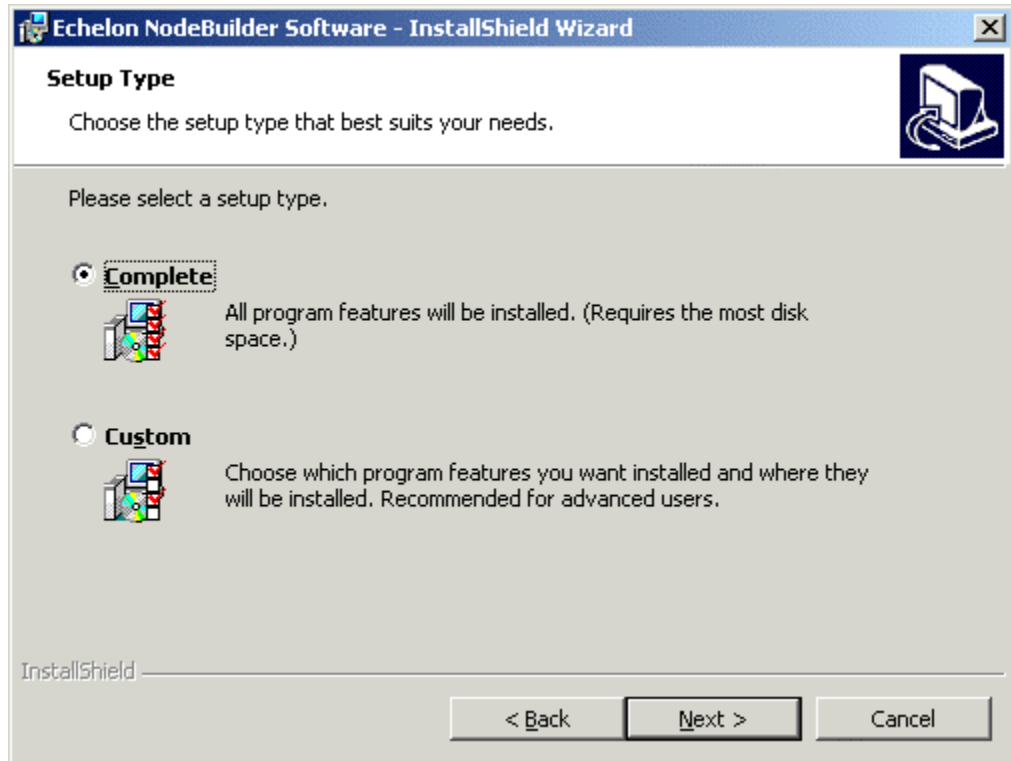
1. If you are planning to use the LNS Device Plug-in Wizard included with the NodeBuilder tool, install Microsoft Visual Basic 6 before installing the NodeBuilder software.
2. Insert the NodeBuilder CD into your CD-ROM drive. The NodeBuilder setup application should start automatically. If it does not, open the Windows **Start** menu, and then click **Run**. Browse to the **Setup** application in the root folder of the NodeBuilder CD then click **Run**. A Welcome window opens.
3. Click the **Next** button to continue. The License Agreement window opens. This window contains the NodeBuilder Software License.

4. Read the terms of the software license. If you agree with the terms of the license, set the **I Accept** option, and then click **Next**. The Customer Information Screen opens, as shown in the following figure:

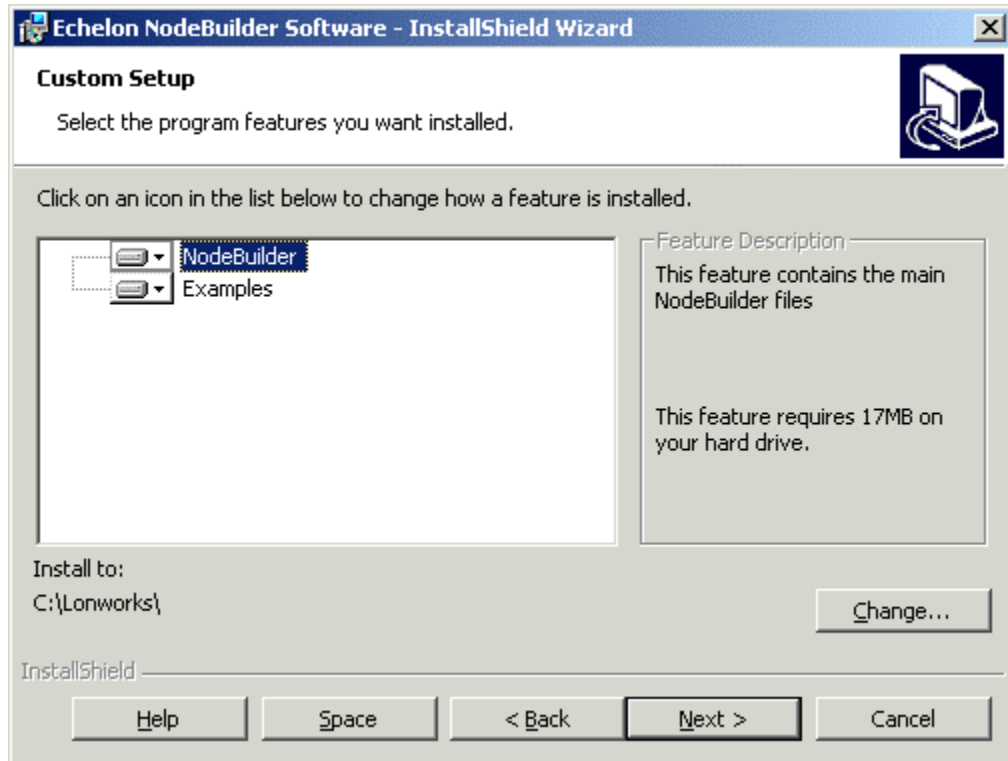
5. Enter the following information. Much of this information is automatically entered into resource files that you will create with the NodeBuilder tool, so it will save you time later if you enter complete information now.

<b>User Name</b>	The name of the person who will be using the NodeBuilder tool on this computer.
<b>Organization</b>	The name of the company for whom the user works.
<b>Phone Number</b>	A phone number where a contact can be reached.
<b>Email Address</b>	An email address where a contact can be reached.
<b>Web Address</b>	The Web site of the individual or company using the NodeBuilder tool.
<b>LonMark Mnfr. ID</b>	If your company has a LONMARK manufacturer ID, enter it here. If you do not have a manufacturer ID, get a free temporary manufacturer ID from <a href="http://www.lonmark.org/mid">www.lonmark.org/mid</a> .
<b>Serial Number</b>	The serial number printed on the back of the NodeBuilder CD. Save this number since it will be required to install future upgrades of the NodeBuilder software. Echelon will save a copy of your serial number if you send in your registration card, or if you register online as described on the registration card.

6. Click **Next**. The Program Group window opens.
7. This window allows you to determine where the NodeBuilder program group will appear in the Windows **Start** menu. By default, it will appear in Echelon NodeBuilder Software. Select a Program Group and click **Next**. The Setup Type window opens, as shown in the following figure:



8. Choose whether you will perform a complete or a custom installation. Set the Complete option unless you need to save disk space or select the installation folder. Click **Next**. If you chose **Complete**, skip to step 6. If you chose Custom, the Custom Setup window opens, as shown in the following figure:



9. Select which parts of the NodeBuilder installation to install. There are two components, *NodeBuilder* and *Examples*. The *NodeBuilder* component is required. If you do not want to install the *Examples*, click the *Examples* component and select **This Feature will not be Available**. You can install the examples at a later time by running the installation again. Click the **Space** button for a status report on the amount of hard drive space required on the computer. Click **Next** to continue. An Installation Ready window opens.
10. Click **Install** to begin the installation. The NodeBuilder software will be installed on your computer. The NodeBuilder software is automatically installed into your computers LONWORKS directory. By default, this is C:\LONWORKS. The LonMaker installation allows you to choose a LONWORKS directory; this location cannot be changed by the NodeBuilder installation. Once the installation has completed, you will be given the option to view the ReadMe file. See the ReadMe file for updates to the NodeBuilder documentation.

---

## *Installing the NodeBuilder Hardware*

To install the NodeBuilder hardware, follow these steps:

1. Install the LTM-10A Platform as described in the *LTM-10A User's Guide*. If you licensed the NodeBuilder 3 Upgrade and you have a LTM-10 Platform, you can optionally upgrade it to an LTM-10A Platform by purchasing and installing an LTM-10A Flash Control Module. This upgrade is described in the *LTM-10A User's Guide*.
2. Plug the Gizmo 4 into the LTM-10A Platform using the cable included with



the Gizmo 4. The following figure illustrates the Gizmo 4 attached to an LTM-10A Platform.



Gizmo 4 Plugged-into LTM-10A Platform

You can still use the NodeBuilder software if you licensed the NodeBuilder 3 Upgrade and you do not have an LTM-10A Platform, but you may have to modify some of the examples to try them on actual hardware.

---

## Getting More Information and Technical Support

If you have technical questions that are not answered by the documentation, on-line help, or Echelon Support Web site at [www.echelon.com/support](http://www.echelon.com/support), you can get technical support from Echelon. Your LonMaker distributor may also provide customer support. You can also enroll in training classes at Echelon to learn more about how to use the NodeBuilder tool. To receive technical support from Echelon for the NodeBuilder tool, you must register your copy with Echelon and you must purchase one of Echelon's incident-based support services. Detailed information about Echelon's support and training services may be found on the Echelon Support home page at [www.echelon.com/support](http://www.echelon.com/support). There is no charge for software installation-related questions during the first 30 days after you receive the NodeBuilder CD. You can obtain technical support via phone, fax, or email from your closest Echelon support center. The contact information is listed in the following table, and is also available at [www.echelon.com/support](http://www.echelon.com/support).

---

**Caution**     *The support programs and the information in the following table are subject to change. See the Echelon Services home page at [www.echelon.com/support](http://www.echelon.com/support)*

*for a description of the current offerings and support contracts. Your LonMaker distributor may provide you with alternate contacts for support.*

---

	London	San Jose	Tokyo
<b>Language</b>	English/French/ German/Italian	English	Japanese
<b>Hours (Mon-Fri*)</b>	0900-1700 London Time	8:30am-4:30pm PDT	0900-1700 Tokyo Time
<b>Telephone</b>	+44 (0) 1923 430200	+1-408-938-5200 +1-888-ECHELON (888-324-3566; US and Canada only)	+81 3 3440 7781
<b>Fax</b>	+44 (0) 1923 430300	+1-408-328-3801	+81 3 3440 7782
<b>Email</b>	lonsupport@echelon.co.uk	lonsupport@echelon.com	lonsupport@echelon.co.jp

\*Excluding holidays at center location

---

## NodeBuilder Quick-Start Tutorial

The objective of the Quick-Start Tutorial is to walk through the steps used to create a device and its LNS device plug-in using the NodeBuilder 3 Development Tool. It introduces NodeBuilder 3 features and provides you with a quick overview of the NodeBuilder interface.

---

### Goals

The goal of this tutorial is for the developer to be able to:

- Use the NodeBuilder Project Manager from within the LonMaker tool
- Create a new NodeBuilder project
- Add a new NodeBuilder device template
- Use the Code Wizard
- Use the project manager
- Use the standard program ID (SPID) calculator
- Use the debugger
- Use the Neuron C features for developing LONMARK compliant devices
- Use the LonMaker tool to install and load target devices
- Use the NodeBuilder Plug-in Wizard to create an LNS device plug-in

The NodeBuilder 3 Development Tool dramatically reduces the time required to develop LONWORKS devices, while at the same time producing devices that are easier to install, configure, and maintain. The result is lower development cost and more competitive products. The NodeBuilder tool can be used to create electric meters, VAV controllers, thermostats, washing machines, card-access readers, refrigerators, lighting ballasts, blinds, pumps, and many other types of devices. These devices can be used in a variety of

systems including building controls, factory automation, home automation, and transportation.

This tutorial is divided into the following sections:

- *Introduction to the Quick-Start Tutorial*
- *Create a LonMaker network*
- *Create a NodeBuilder project*
- *Create a NodeBuilder device template*
- *Automatically generate Neuron C source code*
- *Edit your Neuron C source code*
- *Compile, build, and download your application*
- *Test your device interface*
- *Debug your device application*
- *Install and test your device in a network*
- *Generate Visual Basic code for an LNS device plug-in*
- *Test your LNS device plug-in*
- *Congratulations!*

---

## ***NodeBuilder Quick-Start Tutorial: Introduction***

This tutorial will take you through the process of developing a device and its LNS plug-in with the NodeBuilder tool.

The first step required to develop a device is to define the requirements for the device. For this tutorial, you will develop a device with two sensors and an actuator. The first sensor is a simple sensor that monitors a push button and toggles a network variable output each time the button is pressed. The second sensor is a temperature sensor that reports the temperature value on a network variable output. The actuator drives the state of an LED based on the state of a network variable input. The hardware platform will be the LTM-10A Platform and the Gizmo 4 I/O Board included with the NodeBuilder tool.

This tutorial will include the steps required to develop and configure the temperature interface, but will not include the complete implementation of the code required to read the temperature sensor. The Neuron C example included with the NodeBuilder tool demonstrates how to read the temperature sensor on the Gizmo 4.

The next section describes how to *Create a LonMaker Network*.

---

## ***NodeBuilder Quick-Start Tutorial: Creating a LonMaker Network***

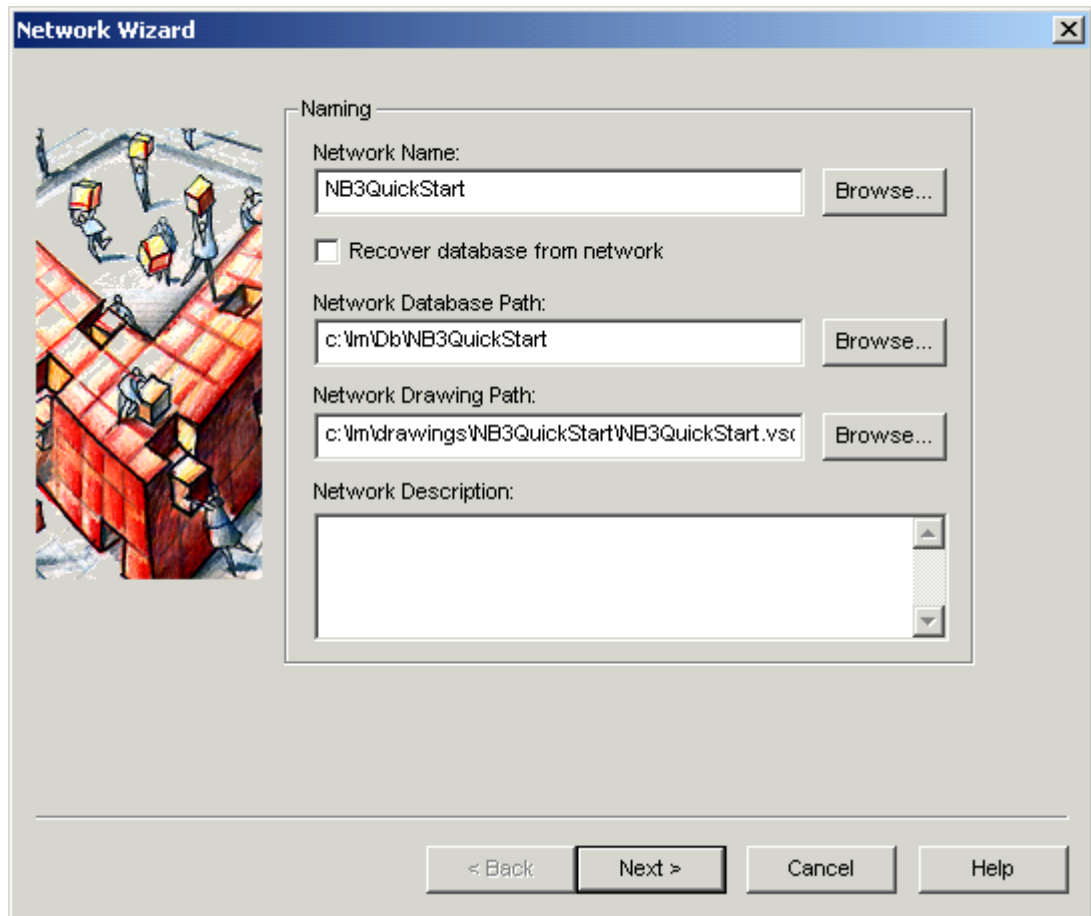
The *LonMaker Integration Tool* is a software tool for designing, installing, operating, and maintaining multi-vendor open, interoperable LONWORKS networks. You will use it to install devices that you develop in a network and to test your devices and their interactions with other devices. After your

development is complete, you can continue to use the LonMaker tool to install your production devices.

The LonMaker tool incorporates Microsoft Visio, providing an easy-to-use drawing tool for designing and documenting your networks.

To create a new LonMaker network, follow these steps:

1. Click the Windows **Start** menu, point to **Programs**, and click **LonMaker for Windows**. The LonMaker Design Manager appears.
2. Click the **New Network** button. The Network Wizard appears.
3. Enter NB3QuickStart for the **Network Name** in the following dialog, and then click **Next**.



4. Set **Network Attached** and select your network interface under **Network Interface Name**, and then click **Next**.
5. Select the **OnNet** Management Mode, and then click **Next**.
6. Click **Finish**. The LonMaker tool creates and opens a new network drawing. The next section describes how to *Create a NodeBuilder Project*.

---

## NodeBuilder Quick-Start Tutorial: Creating a NodeBuilder Project

A *NodeBuilder project* collects all the information about a set of devices that you are developing. You can use the same NodeBuilder project with multiple LonMaker networks, and you can use a LonMaker network with multiple NodeBuilder projects. However, a LonMaker network can only be used with one NodeBuilder project at a time.

You will create, manage, and use NodeBuilder projects from the *NodeBuilder Project Manager*. The project manager provides an integrated view of your entire project and provides the tools you will use to define and build your project.

To create a NodeBuilder project, follow these steps:

1. Open the LonMaker menu and click **NodeBuilder**.
2. Set the **Create a New NodeBuilder Project** option, and then click **Next**.
3. Use the default name, which is the same name as the network drawing.
4. Click **Next**.
5. Click **Finish** to close the New Project wizard. The NodeBuilder New Device Template wizard starts. This wizard is described in the *next section*.

---

## NodeBuilder Quick-Start Tutorial: Creating a NodeBuilder Device Template

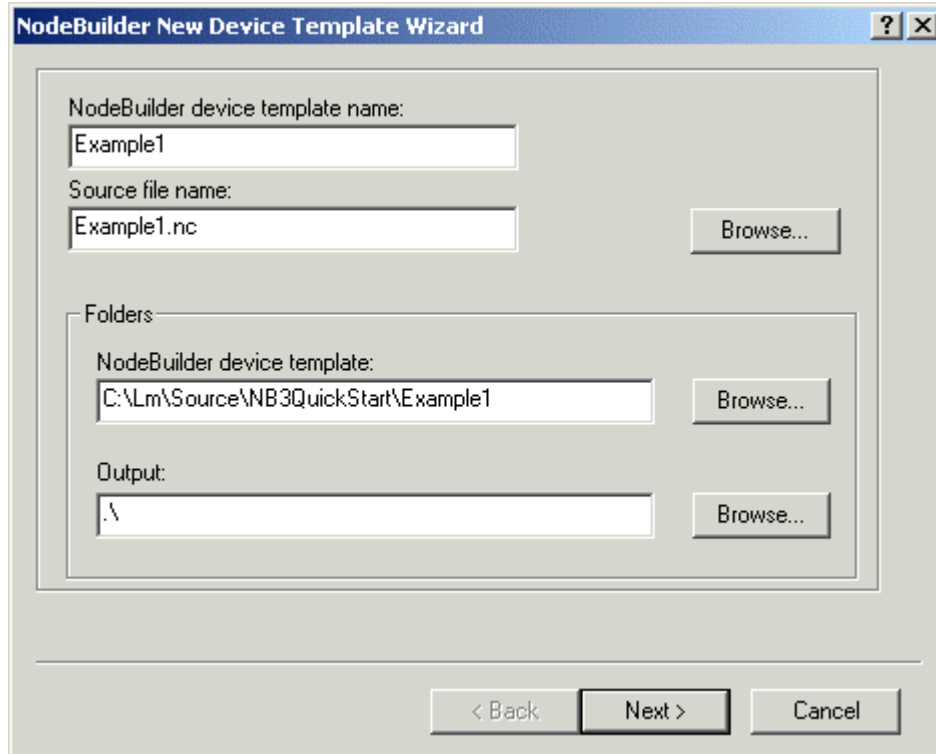
Each type of device that you develop with the NodeBuilder tool is defined by two *device templates*. The first is a *NodeBuilder device template*. The NodeBuilder device template specifies the information required for the NodeBuilder tool to build the application for a device such as a list of the source code files and up to two hardware platforms for the device. The second is an *LNS device template*. The LNS device template defines the external interface to the device, and is used by LNS tools such as the LonMaker tool to configure and bind the device.

Each pair of device templates is identified by a unique *program ID*. Every device on a network with the same program ID must have the same external interface.

**For NodeBuilder 1.5 Users:** NodeBuilder 3 now supports the ability to have more than one device template in a project. This allows the development of a network of different devices to be easily accomplished with one LonMaker drawing, and one project in NodeBuilder. To add a new device template at any time, right-click the Device Templates folder in the project pane, and then click New.

This section shows how to create a NodeBuilder device template. The LNS device template will be created automatically when you build the application. To create the NodeBuilder device template, follow these steps:

1. Enter `Example1` for the **NodeBuilder Device Template Name** as shown in the following figure, and then click **Next**. The Program ID window appears.



**For NodeBuilder 1.5 Users:** The legacy method of defining the program ID by compiler `#pragma` directives is no longer recommended. Instead they are managed as part of the device template. This allows the re-use of Neuron C source-code for different device templates that have different program IDs (perhaps due to different channel types), and allows for automatic program ID management.

2. Click the **Calculator** button next to the program ID. The Standard Program ID Calculator appears.
3. Enter the values shown in the following figure. The manufacturer ID that you entered during installation of the NodeBuilder tool is shown by default. You can use your manufacturer ID, use a new manufacturer ID that is instantly assigned at [www.lonmark.org/mid](http://www.lonmark.org/mid), or select the Examples manufacturer ID if you are developing an example or training device.

LonMark Standard Program ID Calculator - Data File Version 6

Manufacturer (M:MM:MM):  
 Examples [dropdown] 1048575 [text] [OK]

Device class (CC:CC):  
 Multi I/O module [dropdown] 05 [text] 01 [text] [Cancel]

Usage (UU):  
 General [dropdown] 10 [text]

Channel type (TT)  
 TP/FT-10 [dropdown] 4 [text]

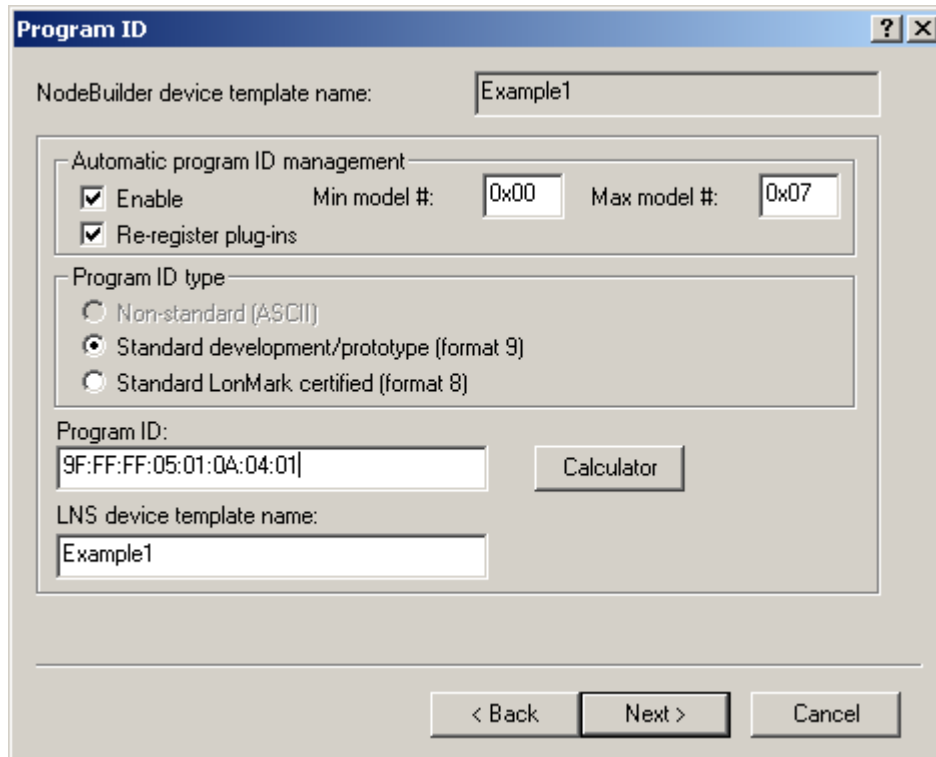
Model number (NN):  
 x 01 [text]

Standard development program ID  
 Has changeable interface  
 Usage field values defined by functional profile

Program ID:  
**FM:MM:MM:CC:CC:UU:TT:NN**  
**9F:FF:FF:05:01:0A:04:01**

The manufacturer IDs, device classes, usage values, and channel types are defined in the **spidData.xml** file that is included with the NodeBuilder tool in the LONWORKS Types folder. Updated versions are available on the LONMARK website ([www.lonmark.org](http://www.lonmark.org)). This file will be updated as the LONMARK Association adds new device classes, usage values, channel types, and manufacturer IDs.

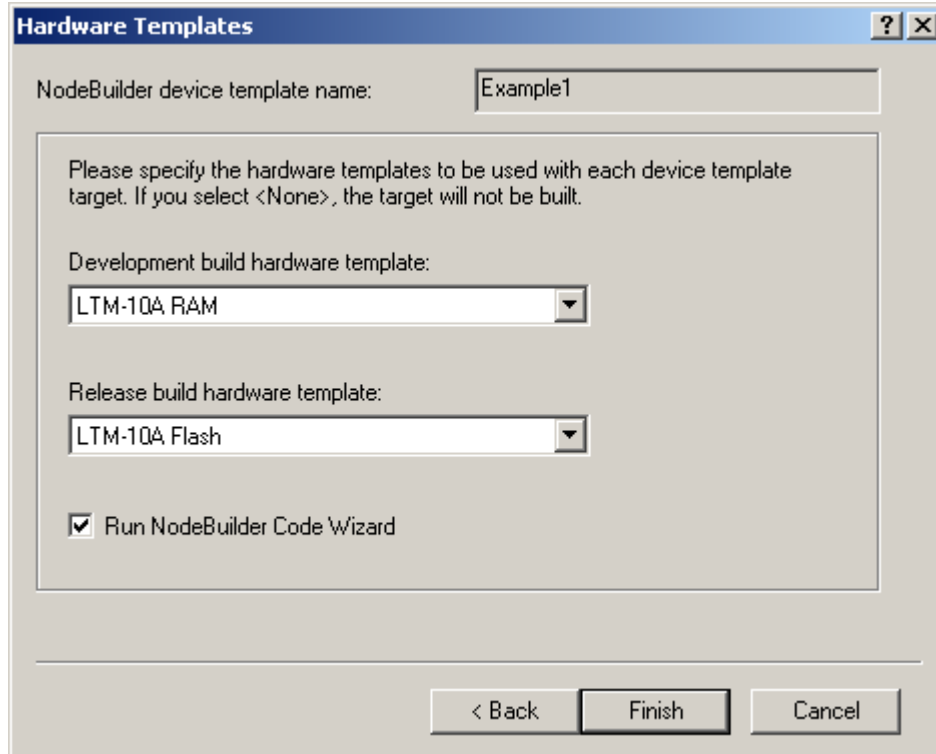
4. Click **OK**. The Standard Program ID Calculator closes, and the **Program ID** field displays the program ID you selected, as shown in the following figure.



Leave **Automatic Program ID Management** enabled so that whenever the device interface is changed, the program ID model field will automatically be incremented. This allows for the easy development of a device with a changing external interface during development. The program ID will cycle through the range of specified model numbers to avoid two devices having the same program ID but different interfaces.

5. Click **Next**. The Target Platforms window appears, as shown in the following figure.



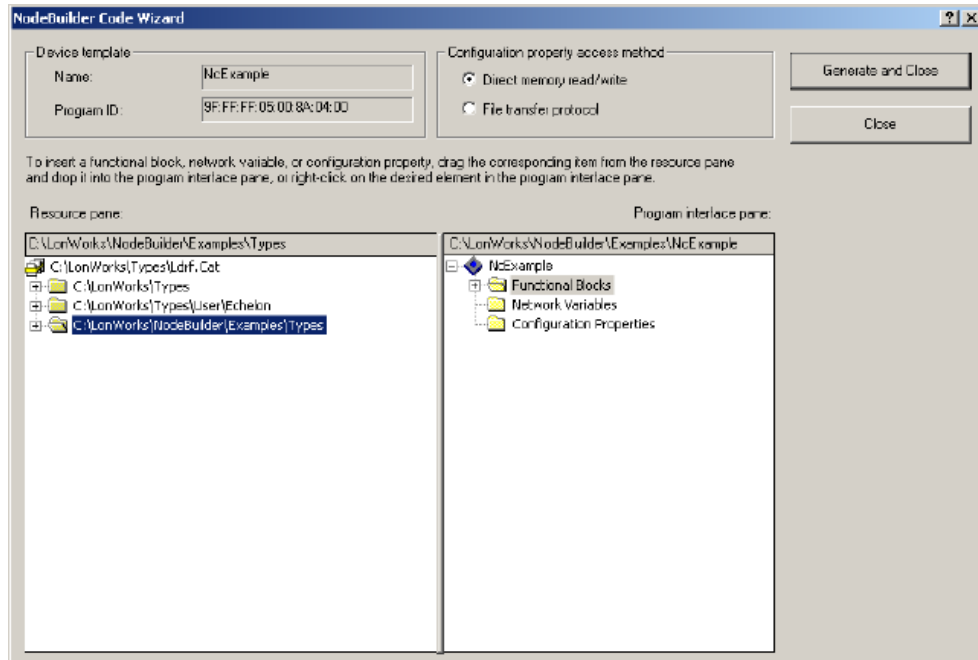


6. Choose LTM-10A RAM for the **Development Build Hardware Template** and LTM-10A Flash for the **Release Build Hardware Template**, and then click **Finish** to close the New Device Template wizard. The NodeBuilder Code Wizard starts. There will be an initial pause as it reads the available resource files. This wizard is described in the *next section*.

---

## *NodeBuilder Quick-Start Tutorial: Automatically Generating Neuron C Source Code*

You will develop applications with the NodeBuilder tool using the *Neuron C* programming language. Neuron C is based on ANSI C, with extensions for network communication, device configuration, hardware I/O, and event-driven scheduling.



The NodeBuilder tool includes the NodeBuilder Code Wizard, shown in the previous figure. The Code Wizard automatically generates Neuron C Version 2 source code that defines the *device interface* of your device. The device interface includes all the functional blocks, network variables, and configuration properties for your device. The Code Wizard also generates much of the code for a standard functional block called the Node Object. The Node Object functional block is used by network tools to test and manage the other functional blocks on your device and is also used to report alarms generated by your device.

The left pane is the Resource pane, used to display the resources that are available for your application. The right pane is the Interface pane, used to display and modify your device interface. You will drag profiles and types from the Resource pane to the Interface pane to define your external interface.

After you run the Code Wizard, you will modify the generated code to implement your device's functionality. You can rerun the Code Wizard at any time to modify your device interface, while maintaining any changes that you have implemented in the source code.

This section shows how to automatically create Neuron C source code for a device with the following functional blocks:

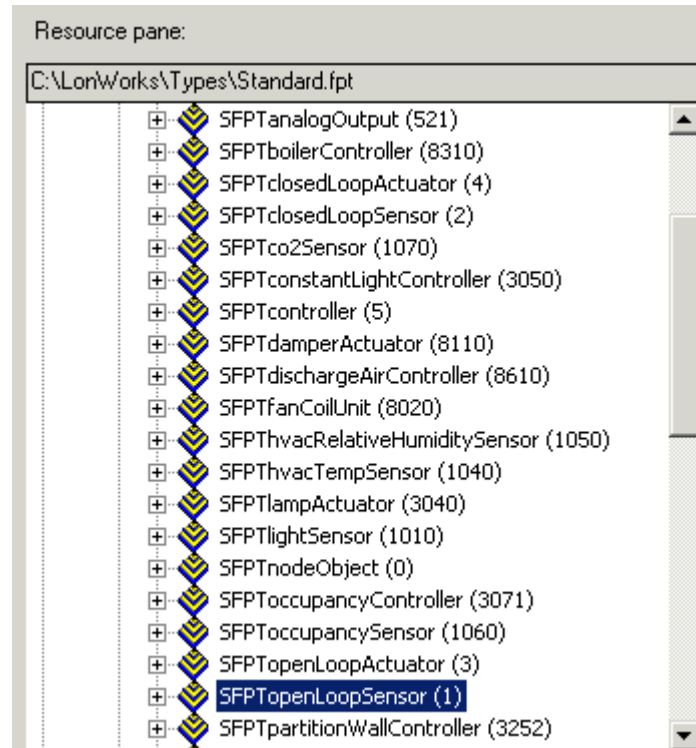
- A Node Object with no configuration properties.
- An open-loop sensor with no configuration properties
- An open-loop sensor with configuration properties
- An open-loop actuator with no configuration properties

To automatically create Neuron C source code using the Code Wizard, follow these steps:

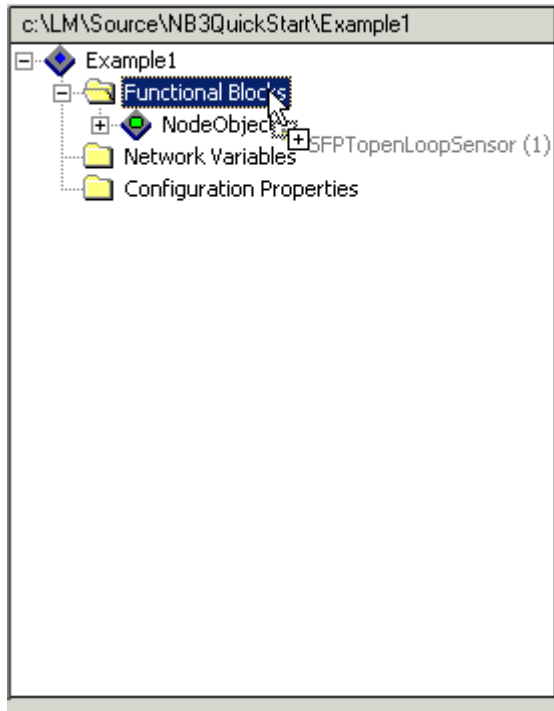
1. Expand the **Standard** scope 0 resource file set in the LONWORKS Types

folder, then expand the **functional profiles** folder. To expand a folder, click the plus sign (+) next to the folder. The standard functional profiles are displayed.

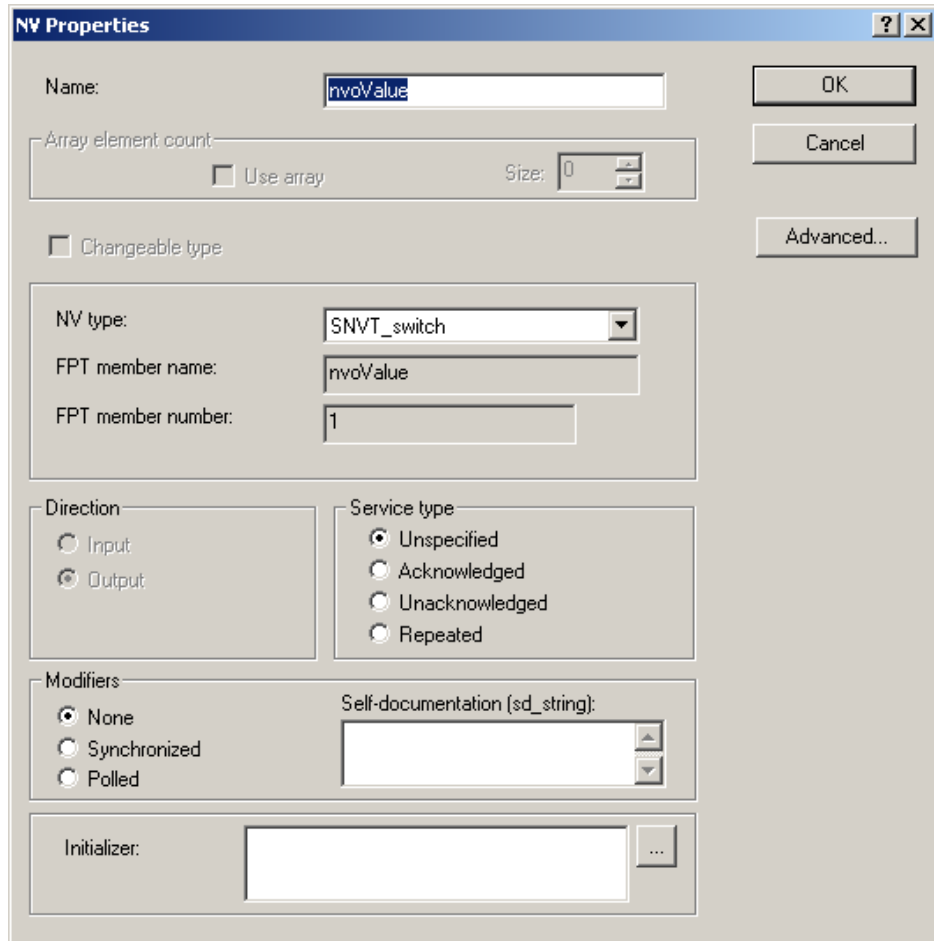
2. Scroll the Resource pane until you see the SFPTopenLoopSensor functional profile (SFPT stands for standard functional profile *template*, which is another name for a standard functional profile), as shown in the following figure.



3. Drag the SFPTopenLoopSensor functional profile to the **Functional Blocks** folder in the Interface pane on the right, as shown in the following figure.



4. Right-click the new openLoopSensor functional block in the Interface pane and click **Rename** on the shortcut menu.
5. Enter Switch as the new name, then press Enter. A warning message appears warning that new source files will be generated.
6. Click **OK** to ignore the warning message.
7. Expand the Switch functional block in the Interface pane.
8. Expand the **Mandatory NVs** folder under the Switch functional block.
9. Double-click the nvoValue Mandatory NV. The NV Properties dialog opens, as shown in the following figure.



10. Change **NV type** to `SNVT_switch`, and then click **OK**.
11. Drag another `SFPTopenLoopSensor` functional profile from the Resource pane to the **Functional Blocks** folder in the Interface pane. When asked if you would like to create a Functional Block array, click **No**, because the two sensors will be different.
12. Repeat steps 4 through 10 for the `openLoopSensor1` functional block, changing the functional block name to `Temperature`, the name of the network variable to `nvoTemp`, **NV type** to `SNVT_temp_p`, and then click **OK**.
13. Right-click the **Optional CPs** folder under the `Temperature` functional block, and then click **Implement Optional CP** on the shortcut menu.
14. Select `nciMinDelta` for the **FPT Member Name**, and then click **OK**.

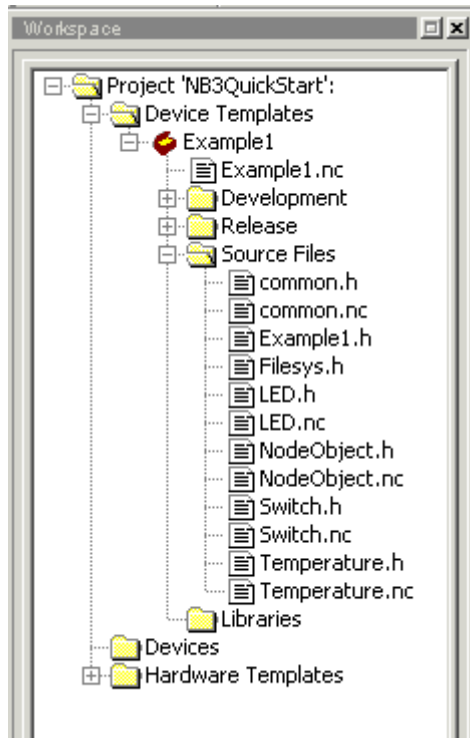
The **SCPTsndDelta** configuration property is used to specify the minimum change in the output that must occur before the device will send the new value to the network. Network integrators use this configuration property to configure devices with very precise measuring capabilities to update the network variable only when a significant change in the sensed value occurs.

The configuration property provides the value used to tell the device what the minimum change must be, however it is left to the developer to write the code that uses this value and actually limits the rate at which the network variable is updated. This will not be implemented during this tutorial, but is left as an exercise for the more advanced user to try.

15. Drag an SFPTOpenLoopActuator functional profile from the Resource pane to the Interface pane.
16. Repeat steps 4 through 10 for the openLoopActuator functional block, changing the functional block name to LED, **NV type** to SNVT\_switch, and then click **OK**.

You have completed designing your device interface. You will now use the NodeBuilder Code Wizard to generate the source files for you.

17. Click the **Generate and Close** button in the top right of the Code Wizard window. The Code Wizard generates Neuron C source files that implement your specified external interface. The Code Wizard closes and you are returned to the Project Manager window. The following Project pane within the project manager displays the files and templates defined for your project.



18. Double-click the Example1.nc file in the Project pane to open the main Neuron C file for this new device template.
19. Open the following header files and look at the declarations of the functional blocks and the configuration properties:
  - Switch.h
  - Temperature.h
  - LED.h
20. Open the Neuron C files and look at the default implementation of the director function (named SwitchDirector or equivalent):
  - Switch.nc
  - Temperature.nc
  - LED.nc

**NodeBuilder 1.5 Users:** The director function is a new element of the

Neuron C Version 2 language starting with NodeBuilder 3. The director function is one of several features added to ease the development of LONMARK compliant devices and to promote modular design of Neuron C code.

The director function is a mechanism that allows the developer to easily dispatch events to all the functional blocks in a device with a single function call. For instance, during reset, the `when(reset)` clause can dispatch the reset event for each functional block in the device when it is done initializing the global pieces in the device. This is done using the following line of code:

```
executeOnEachFblock(FBC_WHEN_RESET);
```

The next section describes how to edit the Neuron C source code.

---

## *NodeBuilder Quick-Start Tutorial: Editing Your Neuron C Source Code*

The Neuron C source code generated by the NodeBuilder Code Wizard implements your device interface, but not your device functionality. You will edit the code generated by the Code Wizard to implement your device functionality, including any required interaction with the I/O hardware within your device.

For this section, you will add Neuron C I/O declarations to the following files:

- `Switch.h`
- `LED.h`

Then you will need to implement your desired I/O functionality in the following files:

- `Switch.nc`
- `LED.nc`

The code in this section is designed to work with the LTM-10A Platform and the Gizmo 4 I/O Board. You can modify the code if you are using different hardware. To modify code, follow these steps:

1. Double-click the `Switch.h` file in the Project pane to edit the source file. To declare the I/O hardware for the Switch, find the following line of code in the Editor window:

```
//}}NodeBuilder Code Wizard End
```

Add the following line after it:

```
IO_6 input bit ioSwitch1;
```

2. Double-click the `Switch.nc` file in the Project pane. Then add the following when clause to the end of the `Switch.nc` file, before the `#endif // _Switch_NC_` line:

```
when(io_changes(ioSwitch1))
{
    Switch::nvoValue.state = !input_value;
```

```
Switch::nvoValue.value = input_value ? (short) 0: 200;
}
```

3. Double-click the `LED.h` file in the Project pane. To declare the I/O hardware for the LED, add the following line after the `NodeBuilder Code Wizard` End line in the `LED.h` file:

```
IO_0 output bit ioLamp = 1;
```

4. Double-click the `LED.nc` file in the Project pane. Then edit the `LEDprocessNV()` function in the `LED.nc` file as follows:

```
void LEDprocessNV(void)
{
  io_out(ioLamp,
    !(nviValue.value && LED::nviValue.state));
}
```

5. Open the **File** menu then click **Save All** to save all your changes to the source files.

The next section describes how to compile, build, and download your application.

---

## *NodeBuilder Quick-Start Tutorial: Compiling, Building, and Downloading Your Application*

The NodeBuilder tool includes a complete set of tools for compiling your Neuron C application, building an application image that can be loaded into your device, and downloading your application image to your device.

When you build your application, the NodeBuilder tool will create a set of files called the *device file set*. The device file set includes an application image file that can be downloaded to your device and a device interface (XIF) file that describes your device interface. The device interface file is used by network tools to determine how to bind and configure your device. The device interface file is also used by the NodeBuilder tool to automatically create the LNS device template.

The NodeBuilder tool can create two device sets for each device that you build, one for a development version of your device and one for a release, or production, version of your device. The default project directory for your NB3QuickStart project is `c:\Lm\Source\NB3QuickStart`. The two device file sets are written to different directories, `Example1\Development` and `Example1\Release` directory, both within your project directory.

To compile, build, and download your application, follow these steps:

1. Right-click the `Example1` device template icon in the Project pane, and then click **Build** on the shortcut menu.
2. If you receive any build errors, double-check that the code you entered matches the above (you will receive some warnings, this is normal).
3. Click the `LonMaker` button in the Windows taskbar to switch to the



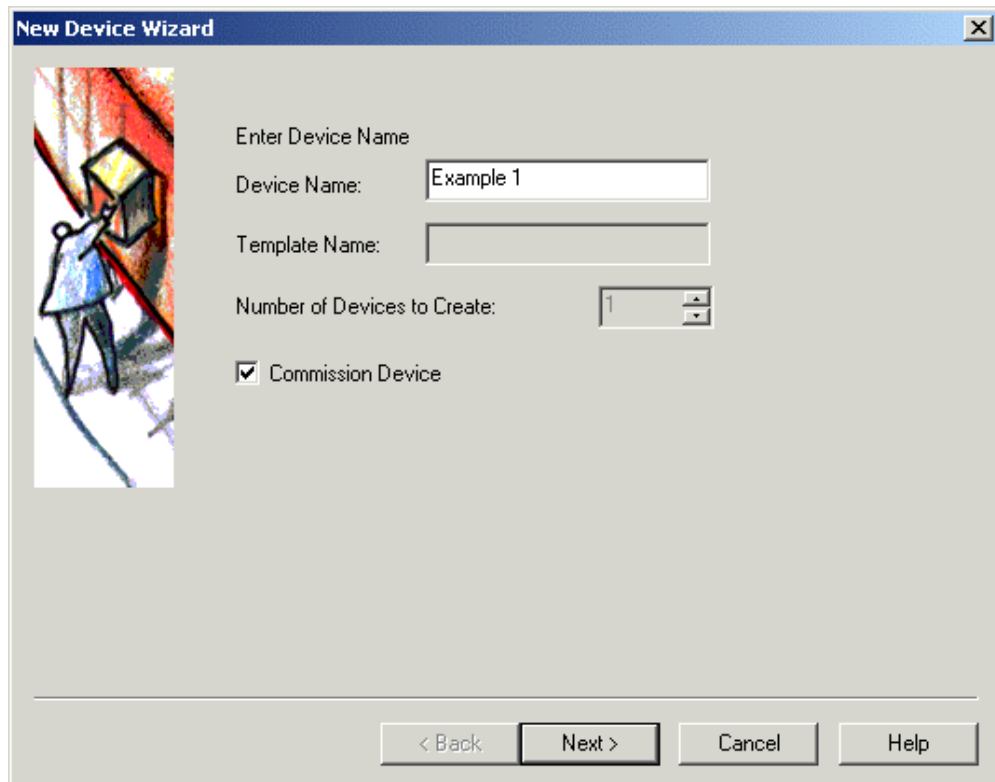
LonMaker window.

You will use the LonMaker Integration Tool to install, bind, configure, and test the devices in your project. The LonMaker tool displays a network drawing that shows the devices, functional blocks, and connections in your network.

The LonMaker tool also displays *stencils* that contain shapes that you can drag to your LonMaker drawing. The LonMaker tool includes a NodeBuilder Basic Shapes stencil with shapes that you will use to add new devices, functional blocks, and connections to your network drawing. The NodeBuilder Basic Shapes stencil contains shapes that can be used with any device. You can also create custom stencils with shapes customized for your devices and networks.

The NodeBuilder Basic Shapes stencil contains two shapes that you will use to define your devices during development. They are the *Development Target Device* shape and the *Release Target Device* shape. These special device types help distinguish between other devices on the network and the target devices used by the NodeBuilder tool. The NodeBuilder tool allows you to create a mixed network of development hardware (LTM-10A Platforms or LonBuilder Emulators), release hardware (your own hardware), and other devices.

4. Drag a **Development Target Device** shape from the **NodeBuilder Basic Shapes** stencil to your network drawing. You can drop the shape anywhere, but a good location is near the Network Interface shape on your drawing. The New Device Wizard starts, as shown in the following figure:



New Device Wizard

Enter Device Name

Device Name: Example 1

Template Name:

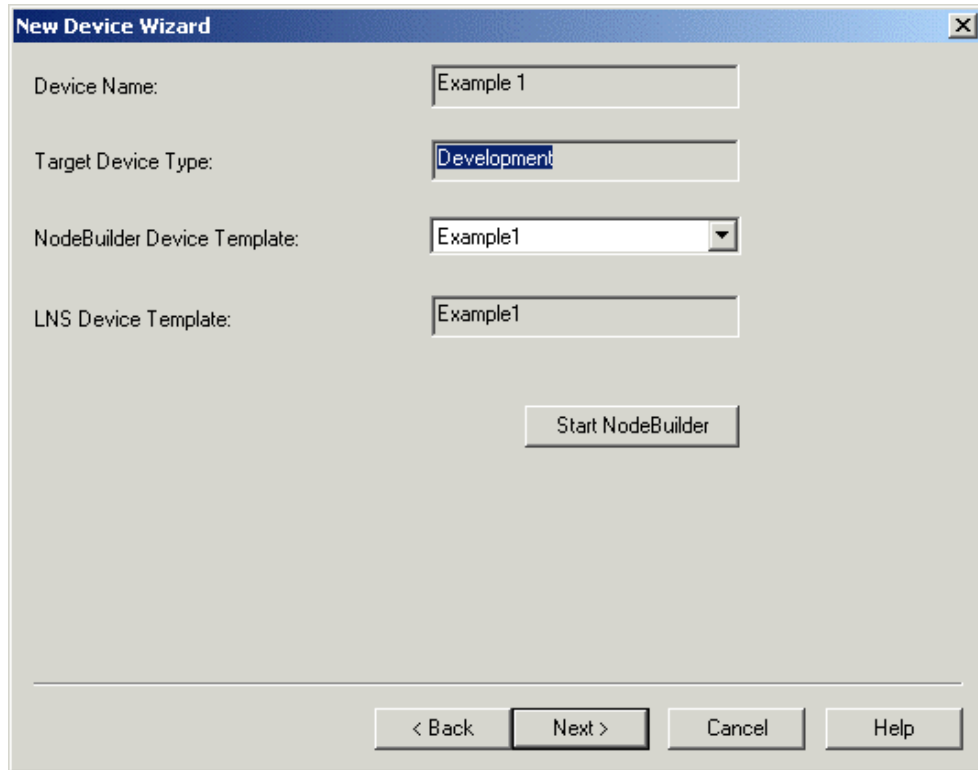
Number of Devices to Create: 1

Commission Device

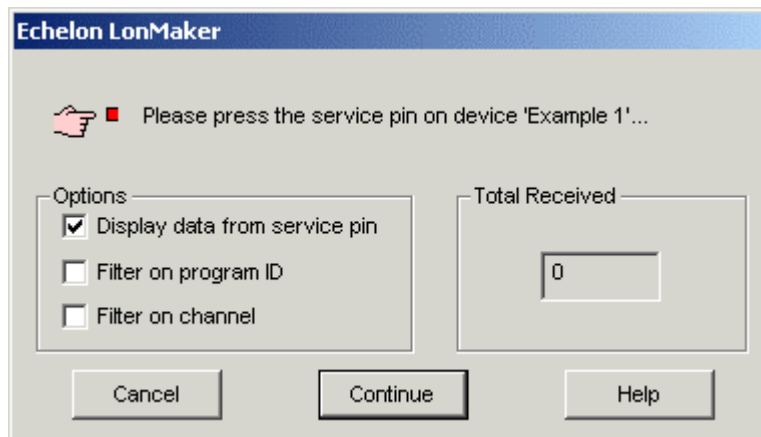
< Back Next > Cancel Help

5. Enter Example 1 as the device name.
6. Set the **Commission Device** option, and then click **Next**. A window opens

that allows you to select the NodeBuilder device template to use for this target device, as shown in the following figure. You can select a NodeBuilder device template from a list of all NodeBuilder device templates that you have built for this project.



7. Select `Example1` as the **NodeBuilder Device Template**, and then click **Next**. The Channel Selection window appears.
8. Click **Next** for this window and for the next two windows, selecting all defaults. A window appears with a **Load Application Image** option.
9. Set the **Load Application Image** checkbox, and then click **Next**. The final window of the New Device Wizard appears.
10. Select the **Online** device state option to start your device online, and then click **Finish**. The Press Service Pin window appears.





5. Click the Value for the LED `nviValue` network variable. The value of the `nviValue` network variable is copied to the **Value** field in the Browser toolbar.
6. Click the **Value** field to highlight the value, then enter `100 1` and press Enter. The LonMaker browser updates the `nviValue` input network variable with a value of 100% and On. The left LED at the bottom of the Gizmo 4 lights.
7. Click the **Value** field to highlight the value, then enter `0 0` and press Enter. The LonMaker browser updates the `nviValue` input network variable with a value of 0% and Off. The left LED at the bottom of the Gizmo 4 turns off. The LED functional block appears to be functioning correctly.

The next section describes how to *Debug your Device Application*.

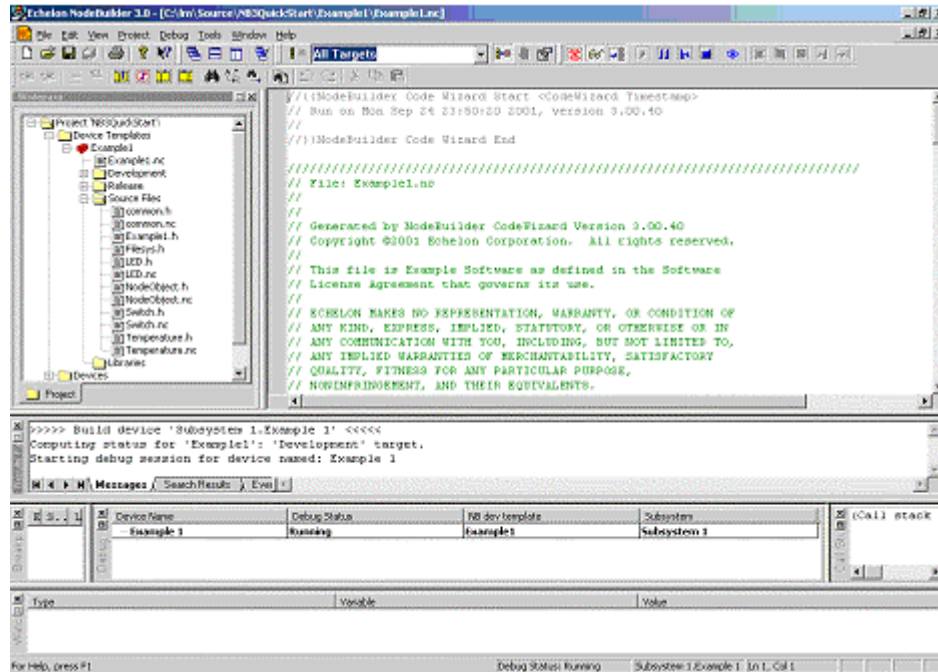
---

## *NodeBuilder Quick-Start Tutorial: Debugging Your Device Application*

If everything works as expected when you test your device interface, you can skip this step and go on to the next step of connecting your device to other devices and testing your device as part of a system. If things do not go as expected, you can use the NodeBuilder debugger to get an inside view of the Neuron C source code executing within your device.

To debug your device's application with the LonMaker Debugger, follow these steps:

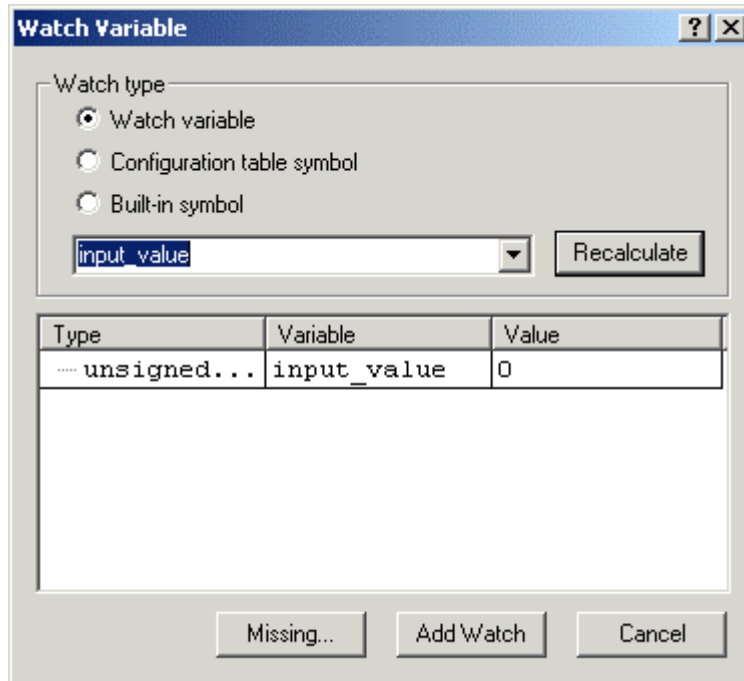
1. Click the LonMaker button in the Windows taskbar to switch to the LonMaker window.
2. Right-click the `Example 1` device shape, and then click **Debug** on the shortcut menu. The NodeBuilder Project Manager appears, and a debug session for the device starts. There is a short pause as the debug session is started while the NodeBuilder tool establishes communication with the device's debug kernel.




3. Double-click the Switch.nc file in the Project pane. A Debug window appears for the Switch.nc file.
4. Find the when(io\_changes(ioSwitch)) clause that you added near the end of the file.
5. Right-click the following line, and then click **Toggle Breakpoint** on the shortcut menu. A breakpoint marker (●) appears next to the line, and the line is added to the Breakpoint List pane.

```
Switch::nvoValue.state = !input_value;
```


6. Press and release the bottom left button on the Gizmo 4. Program execution stops at your breakpoint.
7. Right-click the input\_value variable, and then click **Watch Variable** on the shortcut menu. The Watch Variable window appears.



8. Click **Add Watch** in the Watch Variable window. The variable is added to the Watchlist pane at the bottom of the NodeBuilder Project Manager. This pane displays each of the variables added to the watchlist and their current values.

9. Using the Step Into button () in the debug toolbar, single-step through the code in the function until you reach the end of the when statement. The `input_value` variable is 0.

10. Use the Step Into button to observe that the function executes a second time. The `input_value` variable is 1.

11. Click the Resume button (). Your application resumes normal execution.

12. Open the Debug menu, click **Stop Debugging**, and then select **All Devices** from the shortcut menu.


The debugger has shown that an event occurs when the button is pressed, then a second event occurs when the button is released. To implement the desired behavior, you will modify the code to toggle the output value when the button is pressed.

13. Change the following lines:

```
Switch::nvoValue.state = !input_value;
Switch::nvoValue.value = input_value ? (short) 0: 200;
```

to the following:

```
    if (!input_value) {
Switch::nvoValue.state = !Switch::nvoValue.state;
Switch::nvoValue.value = Switch::nvoValue.state ? (short)
200 : 0;
    }
```

14. Verify that the  Load after Build option is set.
  15. Right-click the Example1 device template in the Project pane, and then click **Build** on the shortcut menu. The NodeBuilder tool rebuilds the Example1 application and downloads it to all devices using the Example1 device template.
  16. Repeat the steps described under *Testing Your Device Interface* to test the change. The device should now be working correctly.
- The next section describes how to *Install and Test your Device in a Network*

---

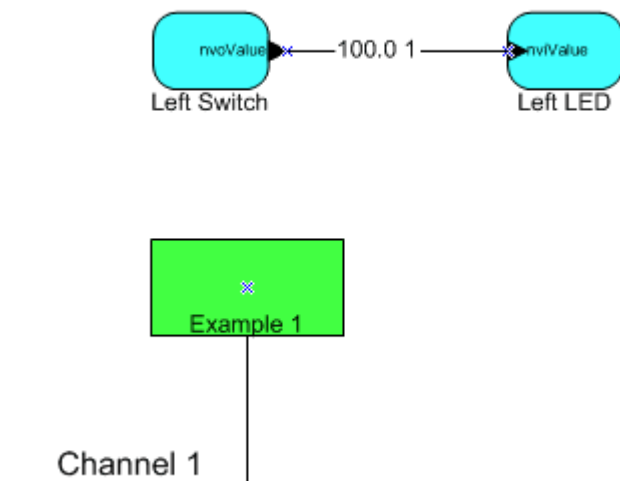
## NodeBuilder Quick-Start Tutorial: Installing and Testing your Device in a Network

Now that your device appears to be functioning correctly by itself, it is time to test it as part of a network. You will use the LonMaker tool to connect your development devices to other devices and to verify their operation within a network.

If compatible, the network variable outputs of a device may be connected to network variable inputs of the same device. These are called *turnaround connections*. For this tutorial, you will create a turnaround connection so that the switch on the Gizmo 4 controls the LED on the Gizmo 4. The procedure is exactly the same for creating connections between different devices.

To create Functional Block shapes with Network Variable shapes for each of your functional blocks, and then connect the network variables, follow these steps:

1. Click the LonMaker button in the Windows taskbar to switch to the LonMaker window. In this section, you will update the LonMaker drawing as shown in the following figure:



2. Drag a **Functional Block** shape from the **NodeBuilder Basic Shapes**

stencil on the left of the LonMaker window to the drawing. The New Functional Block wizard appears. You will use this wizard to associate the new Functional Block shape with the **Example 1** device and the Switch functional block. The **Example 1** device is already selected under **Device**.

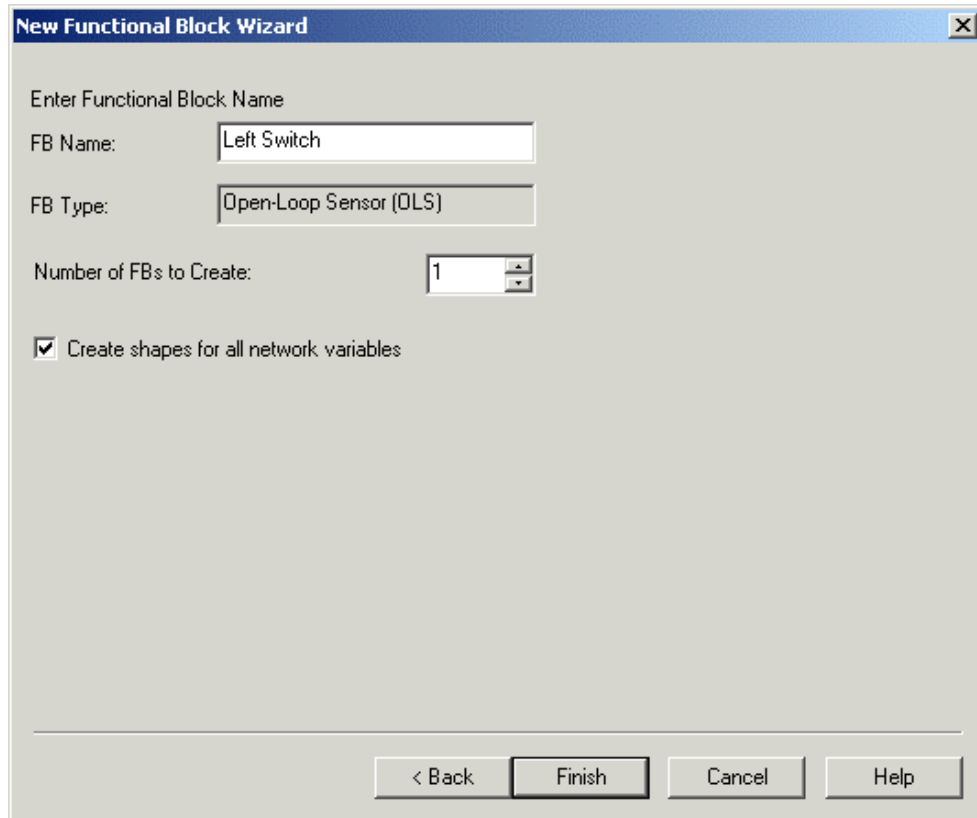
The screenshot shows the 'New Functional Block Wizard' dialog box with the following fields and values:

- Select Device and Functional Block Instance**
  - Source FB Name: Func Block 1
  - FB Type: (empty)
- Subsystem**
  - Name: Subsystem 1
  - Browse... button
- Device**
  - Type: Example1
  - Name: Example 1
- Functional Block**
  - Type: Open-Loop Sensor (OLS) ID: 1
  - Name: Switch

Navigation buttons at the bottom: < Back, Next >, Cancel, Help.

3. Select the **Switch** functional block under **Functional Block Name**, and then click **Next**.





4. Enter **Left Switch** for the **FB Name**.
5. Set the **Create Shapes for All Network Variables** option, and then click **Finish**. The New Functional Block wizard closes and the LonMaker drawing appears. A new **Switch** shape appears on the drawing.
6. Repeat steps 2 through 5 to create a Functional Block shape for the **LED** functional block, and its **nviValue** network variable. Enter **Left LED** for the functional block name.
7. Drag the **Connector** shape from the **NodeBuilder Basic Shapes** stencil to the drawing. Position the left end of the shape over the tip of the **nvoValue** output network variable of the **Left Switch** functional block before releasing the mouse button. A red box appears around the end of the **Connector** shape when you have positioned it correctly over the **Network Variable** shape.
8. Drag the other end of the **Connector** shape to the **nviValue** network variable input of the **Left LED** shape until it snaps into place and a square box appears around the end of the **Connector** shape. There is a brief pause as the LonMaker tool updates the **Example 1** device over the network.
9. Double-click the new **Connector** shape to enable connection monitoring. The current value of the **LED** functional block **nviValue** network variable displays on the connector.
10. Test the connection by pressing the left button on the Gizmo 4. The left LED turns on and off each time you press the button. The current value of the **nviValue** network variable on the **Connector** shape updates each time you press the button.

The next section describes how to generate visual basic code for an LNS Device Plug-in.

---

## *NodeBuilder Quick-Start Tutorial: Generating Visual Basic Code for an LNS Device Plug-in*

Once your device is functioning correctly, you will want to make your device easy to install and configure. To do this, you will create an LNS device plug-in. AN LNS device plug-in is a small application whose sole function is to set-up your device's initial configuration when it is installed in a network. Integrators will start your LNS device plug-in from the LonMaker tool, or from any other installation tool that supports LNS device plug-ins.

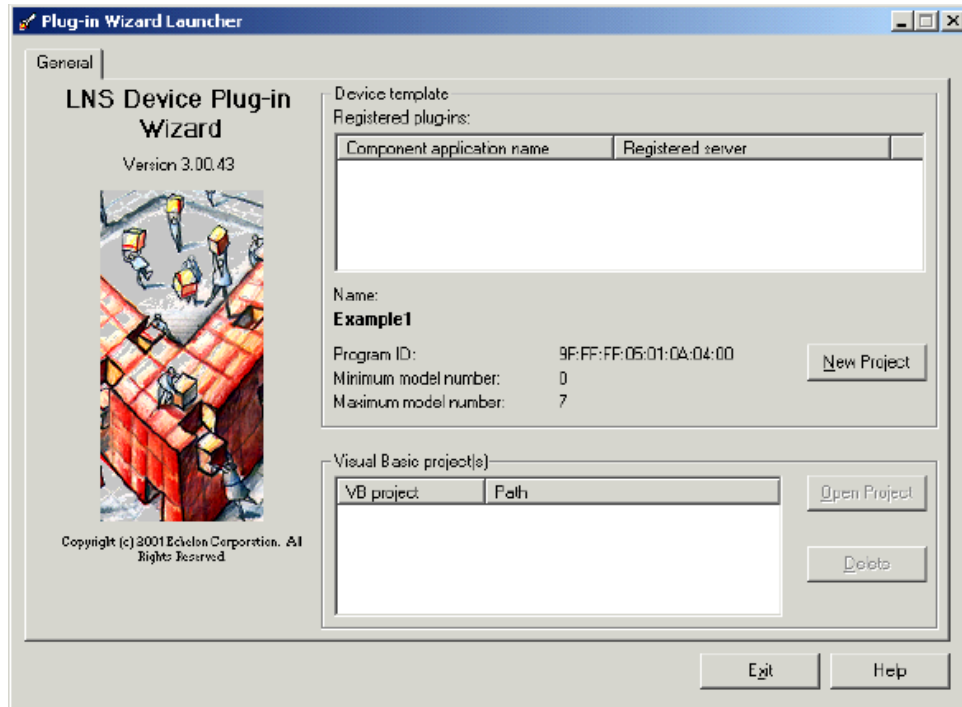
LNS device plug-ins may be written in Visual Basic 6 or Visual C++. The LNS Device Plug-in Wizard included with the NodeBuilder tool generates an initial LNS device plug-in written in Visual Basic 6. You may update this LNS device plug-in to make it easier for integrators to install and configure your device.

This section shows how to create an LNS device plug-in for your example device.

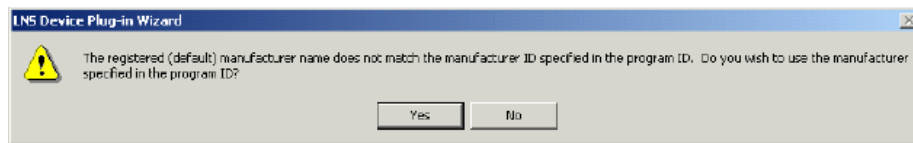
**LNS Developers:** This is a different plug-in wizard than that which ships with the LNS 3 Application Developer's Kit for Windows. The LNS Device Plug-in Wizard included with the NodeBuilder 3 tool is an update of the LNS 3 Application Developer's Kit for Windows LNS plug-in wizard, except that it cannot be used to create system plug-ins.

**NOTE:** The LNS Device Plug-in Wizard requires Microsoft Visual Basic 6 with service pack 5 or better. If you do not have Visual Basic 6, skip this section. If you install Visual Basic 6 after installing the NodeBuilder 3 software, reinstall the NodeBuilder 3 software after installing Visual Basic 6.

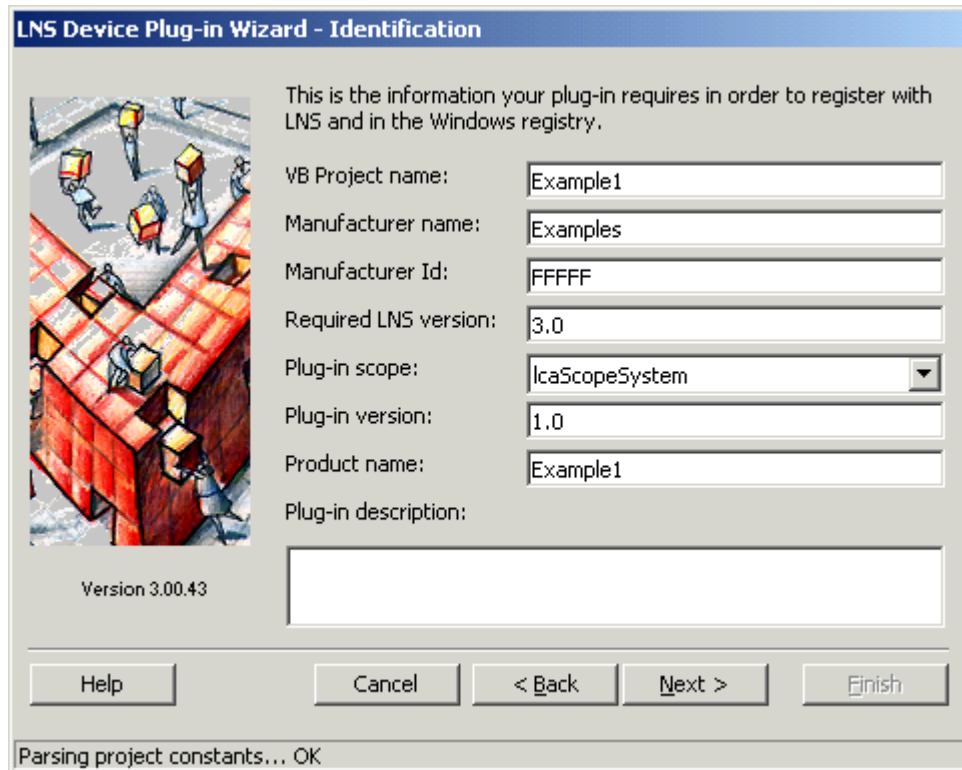
1. Click the NodeBuilder button in the Windows taskbar to switch to the NodeBuilder window.
2. Right-click the `Example1` device template in the Project pane, and then click **Plug-in Wizard** in the shortcut menu. The LNS Device Plug-in Wizard Launcher appears.




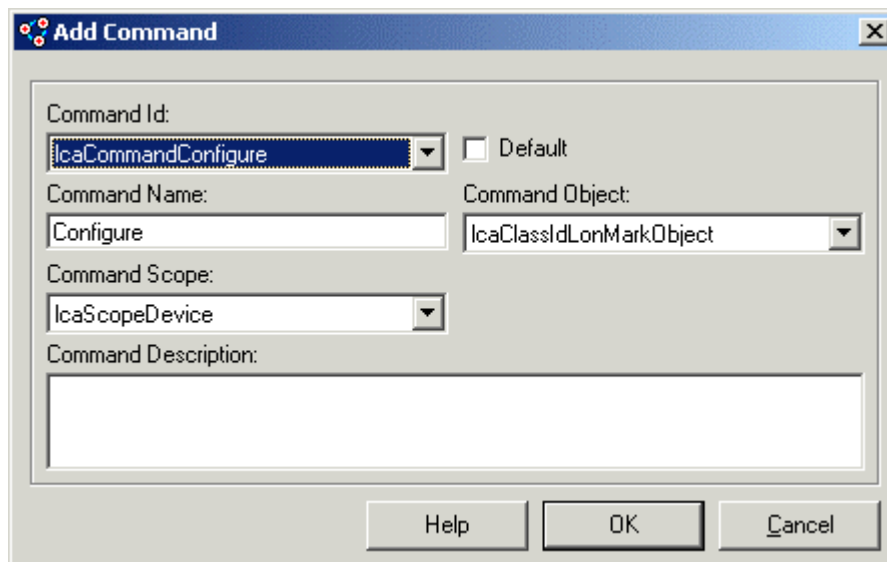
3. Click the **New Project** button to create a new Visual Basic project. A prompt appears that asks if you want to create the default folder.
4. Click **Yes** to create the folder. A New LNS Device Plug-in Project window appears.
5. Click **Save** to save the project file in the default location (the Plug-in folder within the device template's folder). Visual Basic 6 starts, then the LNS Device Plug-in Wizard starts and the wizard's Introduction window appears.
6. Click **Next**. The following warning dialog may appear if you used the Examples manufacturer ID to create the NodeBuilder device template as recommended above.



7. Click **Yes** to use the Examples manufacturer ID. The wizard's Identification window appears.




8. Enter Example1 as the **Product Name**, and then click **Next**. The wizard's Command Table window opens. You will use this window to specify which commands your LNS device plug-in supports.
9. Click the  button to add a new command to the LNS device plug-in.
10. Fill out the Add Command window as shown below, and then click **OK**:



When an LNS tool launches your plug-in, the `lcaCommandConfigure` Command ID will be passed to your plug-in along with the name of a device and functional block to run the command on. That is how the Command ID and the Command Object are related. The Command Scope determines

where the command will be available. Setting `CommandScope` to **lcaScopeDevice** allows it to be used on all devices that use the `Example1 Device Template`.

11. Click **Next**. The wizard's Resource Table window appears.
12. Click **Next** without setting any checkboxes since your plug-in only uses standard resource files. The wizard's Instancing Control window appears.
13. Click **Next**. The wizard's summary window appears.
14. Click the **User Interface** button in the lower right to start the editor for the plug-in's user interface.
15. Expand the **Switch**, **Temperature**, and **LED** functional blocks then drag the `nvoValue`, `nvoTemp`, and `nviValue` network variables from the Device Interface pane on the left to the User Interface pane on the right. When you drag `nvoTemp`, you will also get the `SCPTsndDelta` configuration property. This is because dragging any element of the tree over to the pane on the right will cause all the sub-elements to be moved over as well.
16. Click the  button to save the user interface definition.
17. Close the User Interface Editor window, clear the **View ReadMe.txt** checkbox, and then click **Finish** to close the LNS Device Plug-in Wizard. The LNS Device Plug-in Wizard generates the Visual Basic code for your LNS device plug-in and returns to Visual Basic. If the Visual Basic window is not visible, click the Visual Basic button in the Windows taskbar to switch to Visual Basic.
18. Open the **File** menu, and then click the **Make Example1.exe** command. Click **OK** to save the LNS device plug-in executable file in the default directory. A LONWORKS Object Server warning dialog appears with a warning message that there can be only one instance per process. You can safely ignore this message in this context, as it is caused by the way Visual Basic executables are generated.
19. Click **OK** to close the warning.

The next section, *NodeBuilder Quick-Start Tutorial: Test Your LNS device plug-in*, describes how to test your LNS device plug-in.

---

## *NodeBuilder Quick-Start Tutorial: Testing Your LNS Device Plug-in*

Once you have created a plug-in, you will test it with the LonMaker tool and your prototype device. To test the plug-in with the LonMaker tool, you must first register the LNS device plug-in with Windows so that the LonMaker tool can find it. The installation program for the plug-in typically handles this procedure so that the users of your plug-in do not have to register it manually. We will skip building a Windows installation program for your plug-in in this tutorial.

This section shows how to manually register a plug-in, and then test it with the LonMaker tool.

1. Open the Windows Start menu, click **Run**, and then click **Browse** and browse to your plug-in. By default, your plug-in will be in the `c:\lm\source\NB3QuickStart\Example1\plug-in` folder. If you have

changed the location from the default, browse to that location instead.

2. Run the `Example1.exe` program. The plug-in starts and displays the Registration window.
3. Click the **Register Plug-in** button, and then click **Exit**. Your plug-in has now been registered with Windows. A dialog box appears that confirms the registration. Click **OK**, and then click **Exit**.
4. Click the Visio button in the Windows taskbar to switch to the LonMaker drawing.
5. Open the LonMaker menu, and then click **Network Properties**.
6. Select the **Plug-in Registration tab**.
7. Select the `Example1` plug-in in the list of **Not Registered** plug-ins, and then click the **Add** button. The `Example1` plug-in is added to the **To Be Registered** list.
8. Click the **OK** button to close the Network Properties dialog and register the LNS device plug-in with the network.
9. Click **Yes** if prompted to make Configure the default action for `Example1` devices.

This completes registration of your plug-in with Windows and your development network. You will now run the plug-in from the LonMaker tool.

10. Right-click the `Left Switch` functional block, and then click **Configure** on the shortcut menu. A dialog appears reminding you to add code to the plug-in. This information dialog is being displayed by your plug-in, and indicates that LonMaker is able to successfully start your plug-in.
11. Click **OK** to close the dialog. This dialog provides a reminder of where you would start customizing the plug-in. It is found in the `ProcessDeviceCommands()` function of the `frmMain` form in the Visual Basic project.

The plug-in displays real-time values from the device's outputs, and allows you to set both the value for the configuration property and the input network variable. To change the value of either the configuration property or the input network variable, enter a new value and then click **Apply**.

If you close and re-run the plug-in, you will see that the configuration property value is still the same value you last entered, while the input value may have changed. This is because the configuration property value is stored in the database at the same time it is written to non-volatile memory in the device. The network variable is a real-time-only value, and may have been changed between closing and re-opening the plug-in.

# 3

## Developing a LONWORKS Device

This chapter provides an overview of the steps that you will follow to develop a LONWORKS device. The details of these steps will be discussed in the ensuing chapters.

---

## Introduction to Developing a LONWORKS Device

The NodeBuilder tool makes it easy to develop your device by providing tools for every step of the development process. To develop a device with the NodeBuilder tool, follow these steps:

1. Sign and return the OEM license.
2. Design the device application and hardware.
3. Develop the device hardware.
4. Define the device interface.
5. Create a LonMaker network.
6. Create a NodeBuilder project.
7. Create a NodeBuilder device template.
8. Create the Neuron C application.
9. Compile, build, and load the application.
10. Test the device interface.
11. Debug the device application.
12. Install and test the device in a network.
13. Create a LonMaker stencil.
14. Create an LNS device plug-in.
15. Develop an operator interface.
16. Apply for LONMARK certification.
17. Create an installation application for your device.

These steps are described in the following sections.

---

### *Sign and Return the OEM License*

A LONWORKS OEM License is required to purchase Neuron Chips or Echelon Smart Transceivers, and is also required to manufacture devices that contain Neuron Chips or Echelon Smart Transceivers. A LONWORKS OEM License is included with the NodeBuilder tool. Sign and return this license so that you can purchase Neuron Chips or Echelon Smart Transceivers when you are ready to start building hardware.

---

### *Design the Device Application and Hardware*

The first step in developing a LONWORKS device is planning and designing. What is the purpose of the device? What hardware inputs and outputs will it have? What information will it share with other devices in the LONWORKS network? How will the device be configured? What functional profiles, standard network variable types, and configuration property types will it use? Will the device require a Human-Machine Interface (HMI)?



---

## Develop the Device Hardware

You may choose to develop your device hardware before, after, or in parallel with your device application development. If you will be developing hardware after your device application, you will need a development platform to use for testing your device application. Even if your hardware is available during your application development, you may choose to use a development platform because the NodeBuilder debugger performs many writes to the program memory, which may stress the limits of writes to a flash memory device. To prevent this, your development platform should support writing the application to RAM. Two suitable development platforms are the LTM-10A Platform included with the NodeBuilder tool (but not the upgrade) and the LonBuilder Emulator, or you can use any other development platform that uses RAM for the program memory. .

To develop a device with the NodeBuilder tool, the hardware must contain a Neuron Chip or an Echelon Smart Transceiver that implements the LonTalk protocol. See the *Neuron Chip Data Book*, *Smart Transceiver Data Book*, *LONWORKS FTT-10A Free Topology Transceiver User's Guide*, the *LONWORKS PLT-22 Power Line Transceiver User's Guide*, the *ShortStack™ User's Guide*, and the *LONWORKS Twisted Pair Control Module User's Guide* for more information on hardware development.

To simplify hardware design and I/O software debugging, you may choose to develop prototype I/O hardware that you test with your development platform. Both the LTM-10A Platform and the LonBuilder Emulator include an I/O connector that gives you access to the 11 I/O pins of the Neuron Chip or Smart Transceiver in the platform. You can use this connector to test your prototype I/O hardware with your device application. See the *LTM-10A User's Guide* or the *LonBuilder Hardware Guide* for more information on using the I/O connector on either platform.

You can also use the Gizmo 4 I/O Board included with the NodeBuilder tool (but not the upgrade) for testing your application with I/O hardware. The Gizmo 4 includes commonly used I/O devices including analog and digital inputs and outputs, a rotary shaft encoder, a temperature sensor, a piezo transducer, a real-time clock, and a 4 line by 20 character display that you may find useful for debugging your device application. You can also prototype your own I/O hardware on a prototyping area on the Gizmo 4 board. See the *Gizmo 4 User's Guide* for more information on using the Gizmo 4 board.

---

## Define the Device Interface

The *device interface* for your device consists of the functional blocks, network variables, and configuration properties that allow the device to communicate with other LONWORKS devices and to be configured by network tools. A network variable defines an operational input or output for the device, with the structure, range, units, and format of the network variable defined by a *network variable type*. A configuration property specifies a configuration option for a network variable, functional block, or the entire device. The structure, range, units, and format of a configuration property are defined by a *configuration property type*. A functional block groups network variables and configuration properties that are related to a particular function for the

device. Functional blocks make a device easier to install and configure. Each functional block is defined by a *functional profile* that defines the network variables and configuration properties that comprise the profile.

Functional profiles, network variable types, and configuration property types are defined in *resource files* as described in *Using the Resource Editor*. Resource files are grouped into *resource file sets*, where each set defines functional profiles, network variable types, and configuration properties for a particular scope and program ID mask. A standard resource file set is included with the NodeBuilder tool. This file set defines many standard functional profiles, standard network variable types (SNVTs), and standard configuration property types (SCPTs) that you may find suitable for your device interface. If you need additional functional profiles or types that are not defined in the standard resource file set, you can define your own functional profiles and types as described in *Using the Resource Editor*.

To define your device interface, first determine the functional profiles to be implemented by your device. To select the functional profile or profiles to be implemented by your device, first look through the standard functional profiles that are available in the standard resource file set. You can view these profiles from the NodeBuilder Resource Editor. Detailed documentation for each of the standard functional profiles is available on the *Design Guidelines* area of the LONMARK Web site ([types.lonmark.org](http://types.lonmark.org)).

Each functional profile has a name and number that is unique for the scope of the resource file set. The number is called the *functional profile number*, and is also called the *functional profile key* or *FPT key*. If your device is a simple sensor or actuator, you can use functional profiles 1 through 4: the open-loop sensor (**SFPTopenLoopSensor**; profile 1), the closed-loop sensor (**SFPTclosedLoopSensor**; profile 2), the open-loop actuator (**SFPTopenLoopActuator**; profile 3), or the closed-loop actuator (**SFPTclosedLoopActuator**; profile 4). If your device is more complex, look through the other functional profiles to see if any suitable standard profiles have been defined. If you cannot find any, you will define a manufacturer-defined functional profile as described in *Creating and Modifying a Functional Profile* in the *Editing Resource Files* chapter. If you find an existing functional profile that is close to what you require, but needs minor changes or extensions, you can create a new functional profile that inherits from the standard profile. This is also described in *Creating and Modifying a Functional Profile* in Chapter 7.

---

## Create a LonMaker Network

The *LonMaker Integration Tool* is a software tool for designing, installing, operating, and maintaining multi-vendor open, interoperable LONWORKS networks. You will use it to install devices that you develop in a network, and to test your devices and their interactions with other devices. After your development is complete, you can continue to use the LonMaker tool to install your production devices.

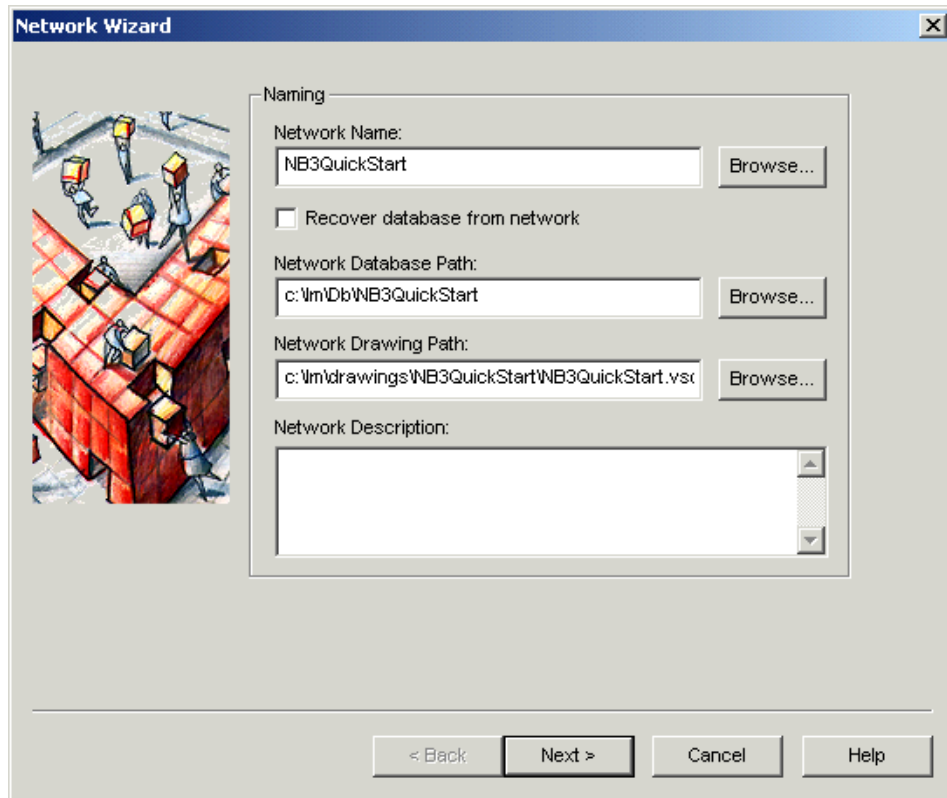
The LonMaker tool incorporates Microsoft Visio, providing an easy-to-use drawing tool for designing and documenting your networks.

To create a new LonMaker network, follow these steps:

1. Click the Windows **Start** menu, point to **Programs**, and then click

**LonMaker for Windows.** The LonMaker Design Manager appears.

2. Click the **New Network** button. The Network Wizard appears.
3. Enter a name for your development network for the **Network Name** in the following dialog, and then click **Next**.



4. If you are attached to your development network, set **Network Attached** and select your network interface under **Network Interface Name**, and then click **Next**. If you are not attached, clear **Network Attached** then click **Next**.
5. If you are attached to your development network, select the **OnNet** Management Mode, and then click **Next**. This option will not be displayed if you are not attached.
6. Click **Finish**. The LonMaker tool creates and opens a new network drawing. See the *LonMaker User's Guide* for more information on using the LonMaker tool.

---

## Create a NodeBuilder Project

A *NodeBuilder project* collects all the information about a set of devices that you are developing. You can use the same NodeBuilder project with multiple LonMaker networks, and you can use a LonMaker network with multiple NodeBuilder projects. However, you can use a LonMaker network with only one NodeBuilder project at a time.

You will create, manage, and use NodeBuilder projects from the *NodeBuilder Project Manager*. The project manager provides an integrated view of your

entire project, and provides the tools you will use to define and build your project.

To start the NodeBuilder Project Manager, select **NodeBuilder** from the LonMaker tool's **LonMaker** menu. Alternately, you can drag a NodeBuilder Target shape to a LonMaker drawing and set the **Start NodeBuilder** checkbox in the *LonMaker New Device Wizard*. You can also start the NodeBuilder Project Manager by opening the Windows **Start** menu, pointing to **Programs**, pointing to **Echelon NodeBuilder Software**, and then clicking **NodeBuilder Development Tool**. Once the NodeBuilder tool has been started, you can use the *Device Template Wizard* to define a new device template as described in the next section. See *Creating a NodeBuilder Project* for more information on creating a NodeBuilder project.

---

## Create a NodeBuilder Device Template

Each type of device that you develop with the NodeBuilder tool is defined by two *device templates*. The first is a *NodeBuilder device template*. The NodeBuilder device template specifies the information required for the NodeBuilder tool to build the application for a device such as a list of the source code files and up to two hardware platforms for the device. The second is an *LNS device template*. The LNS device template defines the device interface, and is used by LNS tools such as the LonMaker tool to configure and bind the device.

Each pair of device templates is identified by a unique *program ID*. Every device on a network with the same program ID must have the same device interface and use the same device template.

You will use the New Device Template Wizard to create a device template. You may automatically start this wizard when you create a new project as described in the previous section, or you may start the wizard at any time by right-clicking the Device Templates folder in the NodeBuilder Project Manager and then clicking **New Device Template** on the shortcut menu. See *Using the New Device Template Wizard* for more information on creating a NodeBuilder device template.

---

## Create the Neuron C Application

You will develop applications with the NodeBuilder tool using the *Neuron C* programming language. Neuron C is a programming language based on ANSI C, with extensions for network communication, device configuration, hardware I/O, and event-driven scheduling.

You can develop your Neuron C applications from scratch, by modifying an existing application, or by using the NodeBuilder Code Wizard. The Code Wizard accelerates your development by generating the declarations required to implement your device interface, and also generating some of the code used by network tools to interact with your device, saving days of development for every functional block. The *Using the Code Wizard* chapter describes how to use the Code Wizard to generate your initial application. Once you have run the Code Wizard, or if you are creating your Neuron C application from scratch or from an existing application, you will use the NodeBuilder Project Manager to edit your application's Neuron C code and to add additional files

required to complete its functionality. You can also use your own programming editor if you prefer. *Introduction to the NodeBuilder Project Manager* in the Using NodeBuilder Projects chapter describes how to use the project manager. The *Neuron C Programmer's Guide* and *Neuron C Reference Guide* describe the Neuron C programming language and how to use it.

---

## ***Compile, Build, and Download the Application***

The NodeBuilder tool includes a complete set of tools for compiling your Neuron C application, building an application image that can be loaded into your device, and downloading your application image to your device.

When you build your application, the NodeBuilder tool creates a set of files called the *device file set*. The device file set includes an application image file that can be downloaded to your device and a device interface file that describes your device interface. The device interface file is used by network tools to determine how to bind and configure your device. The device interface file is also used by the NodeBuilder tool to automatically create the LNS device template.

The NodeBuilder tool will create two device file sets for each device that you build, one for a development version of your device and one for a release, or production, version of your device. The two device file sets are written to Development and Release directories within your device template directory, which is a directory within your project directory.

See *Building an Application Image* in the *Compiling, Building and Loading Your Application* chapter for a description of how to use the NodeBuilder Project Manager and the LonMaker Integration Tool to compile, build, and download your application.

---

## ***Test the Device Interface***

The NodeBuilder tool makes it easy to test your device by itself, and to also integrate your device into a test network and test its interaction with other devices.

The first tool that you will typically use for testing is the LonMaker browser. The browser displays all the network inputs, network outputs, and configuration inputs for your device. You will typically exercise the hardware or network inputs to your device and observe the hardware and network outputs from your device.

*Testing a NodeBuilder Device* describes how you can use the LonMaker tool and LonMaker browser to verify that all of the functional blocks, configuration properties, and network variables have been created and are using the desired formats. You will typically do your initial testing on a development platform such as the LTM-10A Platform or the LonBuilder Emulator. The development platform may be connected to prototype I/O devices, or to the Gizmo 4 I/O Board if its I/O devices are suitable. If your device hardware is ready, you can alternatively load your application into it to verify that the application performs correctly in the hardware. See the *LonMaker User's Guide* for more detailed information about the LonMaker tool and LonMaker browser.

---

## *Debug the Device Application*

If everything works as expected when you test your device interface, you can skip this step and go on to the next step of connecting your device to other devices and testing your device as part of a system. If things don't go as expected, you can use the NodeBuilder debugger to get an inside view of the Neuron C source code executing within your device.

The debugger allows you to set breakpoints and flags to help you pinpoint any problems in your application. Use the debugger to troubleshoot and fix any problems contained in your Neuron C code. See *Using the Debugger* for more information on using the NodeBuilder debugger.

---

## *Install and Test Your Device in a Network*

Once your device appears to be functioning correctly in a small test network, you will test it as part of a larger network. You will use the LonMaker tool to connect your development device to other devices and to verify their operation within a production-scale network. See *Testing a NodeBuilder Device* for more information on using the LonMaker tool to install and test a network.

---

## *Create a LonMaker Stencil*

If your device will be installed by integrators, you can create a LonMaker stencil for your device. This stencil should contain custom LonMaker shapes for your device and for each functional block defined by your application. A custom stencil is not required, but if you have one your device will be easier to install for network integrators. See *Creating a New LonMaker Stencil* for more information on creating a LonMaker stencil.

---

## *Create an LNS Device Plug-in*

Once your device is functioning correctly, you will want to make your device easy to install and configure for network integrators who will be installing it. To do this, you will create an LNS device plug-in. An LNS device plug-in is an application that you will create whose sole function is to set-up your device's initial configuration when it is installed in a network. Integrators will start your plug-in from the LonMaker tool or from any other LNS based installation tool that supports LNS plug-ins.

Plug-ins may be written in Visual Basic 6 or Visual C++ 6 or any other language that supports ActiveX automation servers. Echelon supports the development of plug-ins in Visual Basic 6 and Visual C++ 6, so it is recommended that you use one of these two languages.

You must have Visual Basic 6 with service pack 5 or better installed on your PC before installing the NodeBuilder software in order to use LNS Device Plug-in Wizard. The wizard is not compatible with Visual Basic.NET, and Visual Basic.NET cannot be used to create LNS plug-ins. See *Creating an LNS Plug in* and the *LNS Plug-in Programmer's Guide* for more information on creating LNS device plug-ins.

---

## ***Develop an Operator Interface***

Once you have developed one or more LONWORKS devices designed to be used in LONWORKS networks, you may wish to develop an operator interface. You will typically do this if you are building complete systems that require some form of operator interface. If your device will be installed by integrators where each installation is unique, your integrators will develop the required operator interfaces. You or your integrator can use the LNS DDE Server in concert with an HMI development tool such as Wonderware InTouch to create a graphical operator interface to allow end users to monitor and control networks with your devices. You can also use the LonMaker tool to create simple operator interfaces based on the LNS Text Box. See the *Human-Machine Interfaces* chapter, the *LNS DDE Server User's Guide*, and the *LonMaker User's Guide* for more information on creating operator interfaces.

---

## ***Apply for LONMARK Certification***

The LONMARK Interoperability Association is an independent, non-profit organization that defines guidelines for developing interoperable LONWORKS devices. If your device will be installed by integrators, you will want to apply for LONMARK certification for your device since most integrators require LONMARK certified devices for their projects. LONMARK certified devices are assured to be compliant with the LONMARK guidelines and can be easily integrated into LONWORKS networks with other LONWORKS devices from multiple vendors. For information on having your device LONMARK certified, see [www.lonmark.org/products/lmcerti.htm](http://www.lonmark.org/products/lmcerti.htm) on the LONMARK Web site.

---

## ***Create an Installation Application for your Device***

If your device will be installed by integrators, they will need a number of different files to successfully install the device into a LONWORKS network, including the device application (if your device uses downloadable application memory), the device interface files, any new resource files, and optionally the LonMaker stencil, the device plug-in, and the device operator interface. The *Creating a Software Installation* chapter describes how to create an installation application that installs these files in the proper directories on the network integrator's computer.





# 4

## Creating and Opening NodeBuilder Projects

This chapter describes how to create a new NodeBuilder project or open an existing project. You will typically create NodeBuilder projects from the LonMaker tool, but you may also create NodeBuilder projects standalone from the NodeBuilder tool.

---

## Introduction to NodeBuilder Projects

A *NodeBuilder* project collects all the information about a set of devices that you are developing. You can use the same NodeBuilder project with multiple LonMaker networks, and you can use a LonMaker network with multiple NodeBuilder projects. However, you can use a LonMaker network with only one NodeBuilder project at a time.

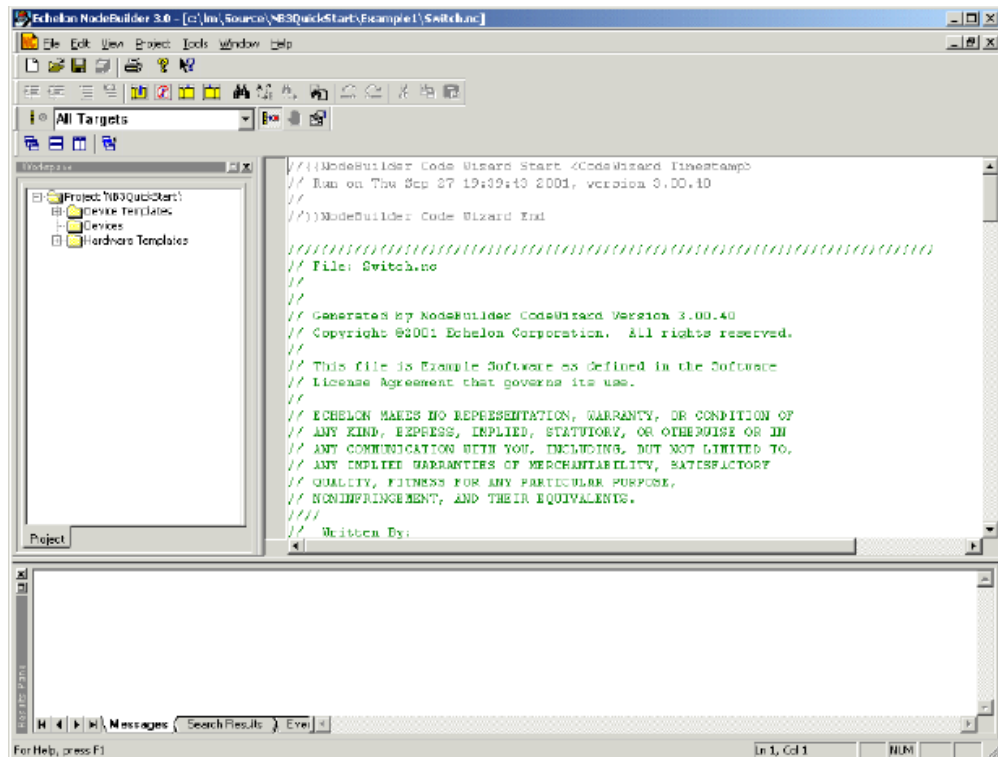
You will create, manage, and use NodeBuilder projects from the *NodeBuilder Project Manager*. The project manager provides an integrated view of your entire project, and provides the tools you will use to define and build your project.

A NodeBuilder project is defined by two XML files called the NodeBuilder *project files*. The base name of each file is the project name. The extensions are “.NbPrj” and “.NbOpt”. The NodeBuilder Project Manager automatically creates and updates these files. You should not modify these files directly, but you may print them using Internet Explorer, Microsoft XML Notepad, or any text editor if you would like a printed record of your project. See *Viewing and Printing NodeBuilder XML Files* for more information.

---

## Introduction to the NodeBuilder Project Manager

The NodeBuilder Project Manager is a software tool that allows you to create, edit, build, and debug a NodeBuilder project. The NodeBuilder Project Manager appears as shown in the following figure:



The NodeBuilder Project Manager initially contains three panes, the *Project pane*, the *Edit pane*, and the *Results pane*. You can move and resize these panes, and you can close the Project and Results panes. By default the three panes appear as shown in the figure above.

The *Project pane* is located on the left side of the Project Manager window by default. It contains a hierarchical display of all the components of a NodeBuilder project.

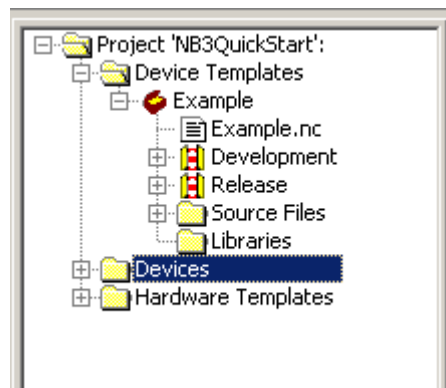
The *Edit pane* is located on the right side of the Project Manager window by default. It allows you to edit any of the Neuron C source files or header files that are used in the project. See the *Editing Neuron C Source Code* chapter for more information.

The *Results pane* is located on the bottom of the Project Manager window by default. This pane contains **Messages**, **Search Results**, and **Event Log** tabs. The **Messages** tab displays compiler and other messages generated when you build the application image for a NodeBuilder device template (see *Building an Application Image in Building, Compiling, and Loading*). If any errors or warnings are generated during the build, you can double-click them to open the file containing the error or warning and go to the line of code that generated the error or warning. The **Search Results** tab displays the results of a **Find in Files** search. You can double-click any of these results to open the file containing the search text and go to the line containing the search text (see *Searching Source Files*, in the *Editing Neuron C Source Code* chapter). The **Event Log** contains debugger event messages (see the *Using the NodeBuilder Debugger* chapter).

---

## Using the Project Pane

The Project pane allows you to browse the files used in the NodeBuilder Project. The Project pane appears as shown in the following figure:



The top level of the Project pane is always a project folder labeled **Project '<Project Name>'**: where <Project Name> is the name of your project. Right-click the project folder to see a shortcut menu with the following options:

- |                   |   |
|-------------------|---|
| <b>Settings</b>   | Displays project settings.  |
| <b>Properties</b> | Displays file properties of the NodeBuilder project file (.NbPrj extension). The properties include the |

file name, location, size, and dates the file was created, last modified, and last accessed.

The Project folder normally contains **Device Templates**, **Devices**, and **Hardware Templates** folders. The **Device Templates** folder contains all of the device templates that have been created or referenced in this NodeBuilder project, and is described in the *Creating and Using Device Templates* chapter. The **Devices** folder contains a list all devices in LonMaker drawings that have been associated with device templates in this NodeBuilder project and is described in *Files Created When You Build An Application Image* in the *Compiling, Building, and Loading Applications* chapter. The **Devices** folder will not appear if the NodeBuilder project is not associated with a LonMaker network. The **Hardware Templates** folder contains a list of the hardware templates available in this NodeBuilder project, and is described in Chapter 4, *Creating and Using Device Templates*.

---

## Creating a NodeBuilder Project

To create a NodeBuilder project, start the NodeBuilder Project Manager. You can start the NodeBuilder Project Manager from the LonMaker tool, or directly from the NodeBuilder program folder. You will typically start the project manager from the LonMaker tool since that simplifies associating the NodeBuilder project with the LonMaker network.

To create a NodeBuilder project by starting the NodeBuilder Project Manager from the LonMaker tool, follow these steps:

1. Create or open a LonMaker drawing. See the *LonMaker User's Guide* for more information on creating and opening LonMaker drawings. If you will want to load the application you develop into a device, make sure the LonMaker computer is attached to the network.
2. Open the **LonMaker** menu then click **NodeBuilder**. The NodeBuilder Project Manager starts. If you have not previously created a NodeBuilder project for this network, the New Project wizard automatically starts.
3. If you have previously created a project for this network and you want to create a new project, open the NodeBuilder **File** menu, and then click **Create Project**.
4. Enter project information into the wizard as described in the following sections.

To create a NodeBuilder project by starting the NodeBuilder Project Manager standalone from the NodeBuilder program folder, follow these steps:

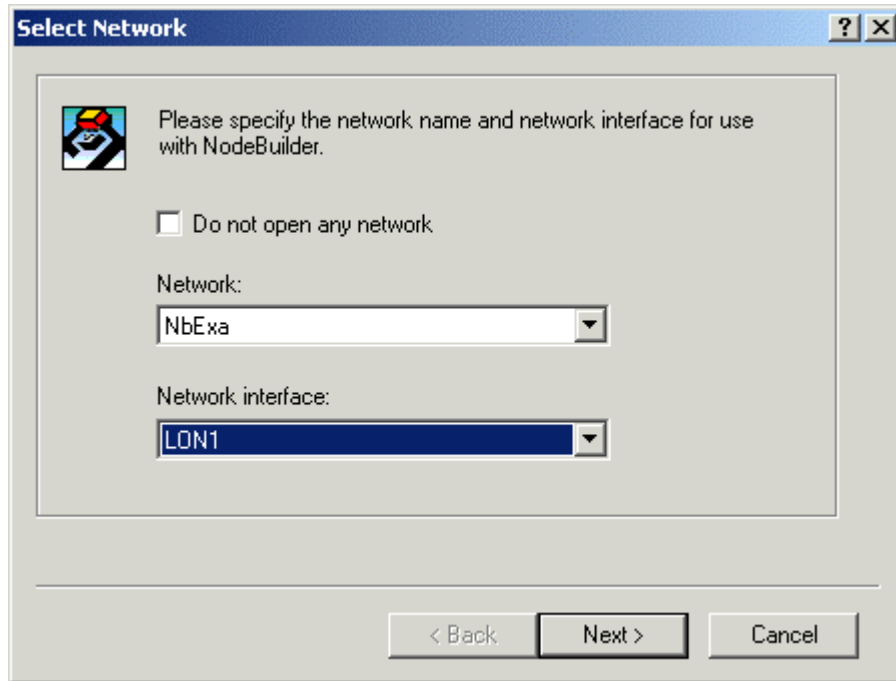
1. Click the Windows **Start** menu, point to **Programs**, point to **Echelon NodeBuilder Software**, and then click **NodeBuilder Development Tool**. The NodeBuilder Project Manager starts.
2. Open the NodeBuilder **File** menu then click **Create Project**. The New Project wizard starts.
3. Enter project information into the wizard as described in the following sections.

You can also start the NodeBuilder tool from the LonMaker tool's New Device Wizard. See *Starting the NodeBuilder tool from the New Device Wizard*, later in this chapter, for more information.

---

## Creating a New Project

If you started the NodeBuilder Project Manager standalone, the project manager opens the following window. This window is not displayed if you started the NodeBuilder Project Manager from the LonMaker tool.



This window allows you to select the LonMaker network that you want to use with this NodeBuilder project. Select the LonMaker network under **Network**. If the physical network is attached, select the network interface under **Network Interface**.

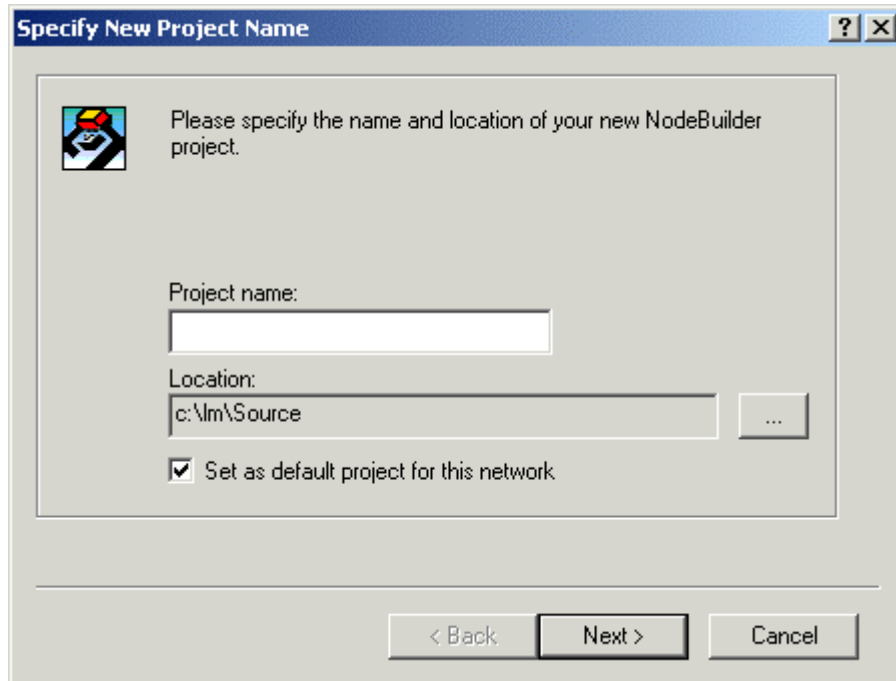
You can set the **Do Not Open Any Network** checkbox to create a project that is not associated with a LonMaker network. Setting this option will disable automatic LNS device template creation, automatic load after build, and use of the LNS Device Plug-in Wizard.

Once you have chosen a network, click **Next**. The *Specify New Project Name* dialog opens.


---

## Specifying New Project Name

This window appears as shown in the following figure:



This window contains the following fields:

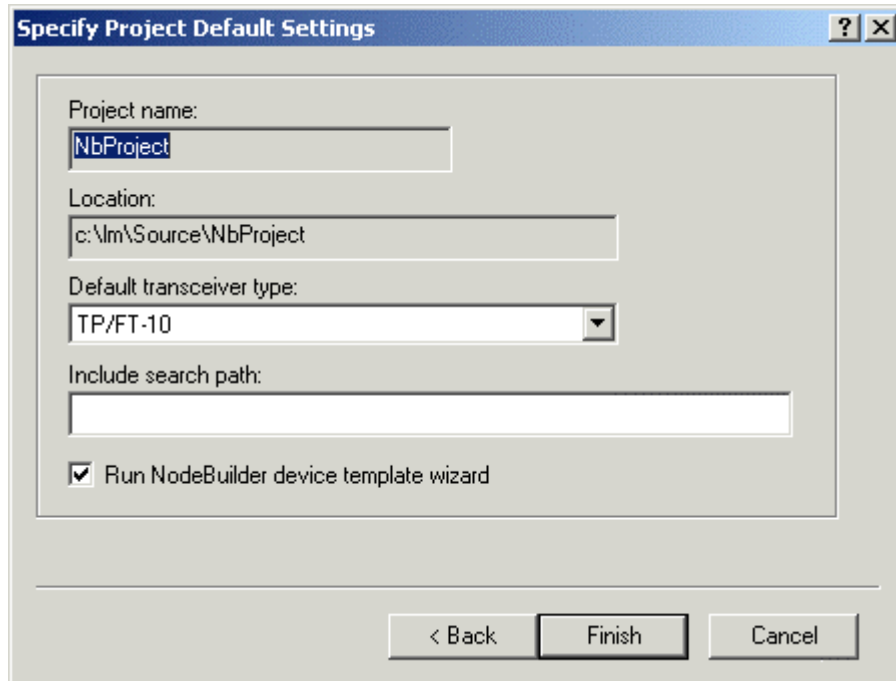
- |                               |  |
|-------------------------------|--|
| <b>Project Name</b>           | The name of the project. Project files with this name and NbPrj and NbOpt extensions will be created in the folder specified in <b>Location</b> . This field defaults to the name of the current LonMaker drawing. You can use this name or enter a new one.   |
| <b>Location</b>               | The folder containing the project file. This folder is called the <i>project folder</i> . The default location is c:\lm\Source\<<Project name>. To change the location, click the  button to browse to a new location. You can use \$LonWorks\$ to specify the PC's LONWORKS folder (this is C:\LonWorks by default). |
| <b>Set as Default Project</b> | Set this option to automatically open this NodeBuilder project when the NodeBuilder tool is started from the current LonMaker drawing.   |

Click **Next**. The *Specify Project Default Settings* window opens.

---

## *Specifying the Project Default Settings*

This window appears as shown in the following figure:



This window contains the following fields:

<b>Project Name</b>	The name of the project as specified in the <i>Specify New Project Name</i> window.
<b>Location</b>	The location of the project folder as specified in the <i>Specify New Project Name</i> window.
<b>Default Transceiver Type</b>	The transceiver type to be used for Hardware Templates that specify <b>default</b> for the transceiver type. The default value for this field is <i>TP/FT-10</i> . See <i>Setting Hardware Template Properties: Hardware</i> for more information.
<b>Include Search Path</b>	<p>An optional semi-colon separated list of directories that are to be searched for include files when a NodeBuilder project is compiled. This list is blank by default, and only the device template source file directories will be searched for include files. If relative path names are specified, they are relative to the location of the NodeBuilder project directory (location of the <code>NbPrj</code> project file). This list applies to the entire project, and affects include files specified with names in double quotes (e.g. <code>#include "abc.h"</code>). The user-defined include search path does not affect system include files, which are NodeBuilder system files specified in angle brackets (e.g. <code>#include &lt;mem.h&gt;</code>)</p> <p>You can include the phrase <i>\$LonWorks\$</i> as a machine-independent reference to the</p>

LONWORKS directory within the search path (e.g. "\$LonWorks\$\NodeBuilder\Gizmo4"). This simplifies copying projects between different computers.

**Run Device Template Wizard** If this checkbox is set, the device template wizard opens immediately after you click **Finish**. The Device Template Wizard will guide you through the process of creating the first NodeBuilder device template for this project. See *Using the Device Template Wizard* in Chapter 6 for more information.

Set the desired options, set the **Run NodeBuilder Device Template Wizard** checkbox, and then click **Finish**.

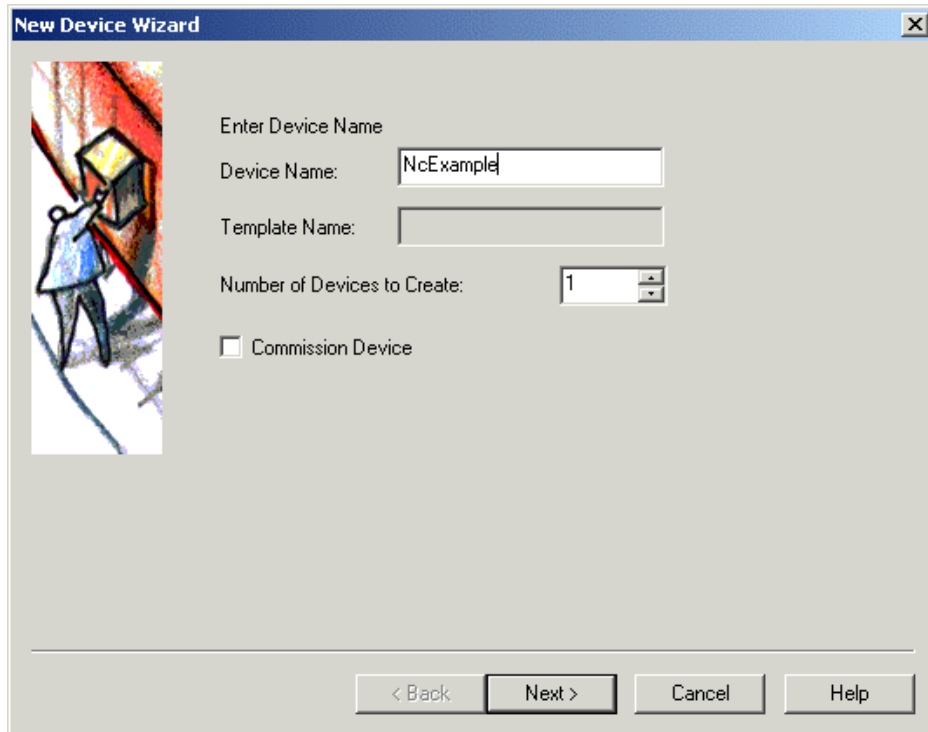
---

## *Starting the NodeBuilder Tool from the New Device Wizard*

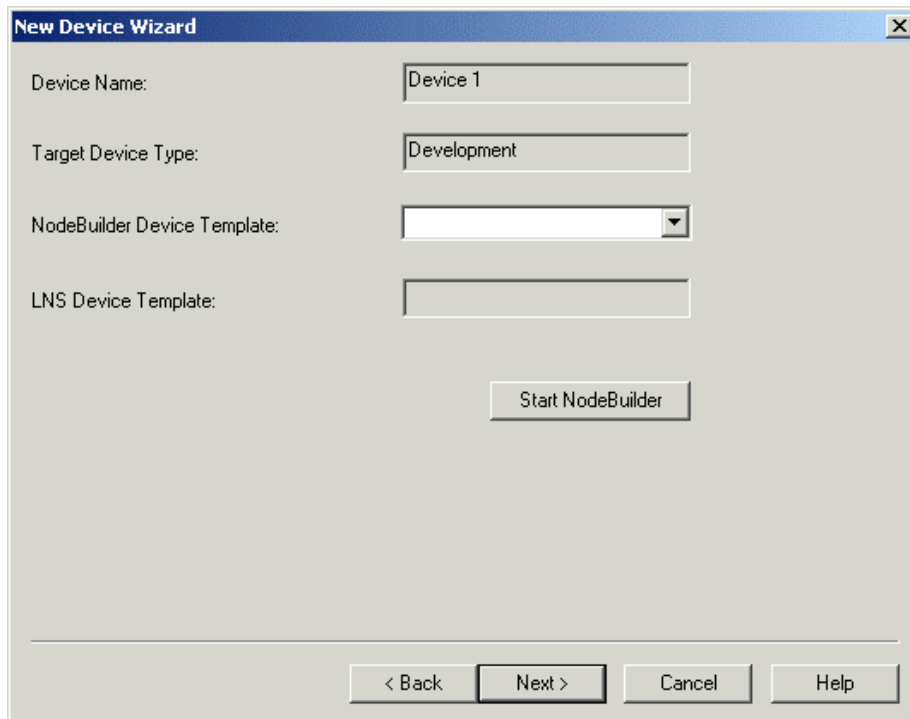
You can start the NodeBuilder tool from the LonMaker tool's New Device Wizard. This is described in the following steps:

1. Create or open a LonMaker network. See the *LonMaker User's Guide* for more information on creating and opening LonMaker drawings. If you will want to load the application you develop into a device, make sure the LonMaker computer is attached to the network.
2. Drag one of the **Target Device** shapes to the LonMaker drawing. The NodeBuilder tool includes a **Development Target Device** shape and a **Release Target Device** shape. If you are going to build for production hardware, use the **Release Target Device** shape. If you are going to build to a test platform such as an LTM-10A or LonBuilder Emulator, use the **Development Target Device** shape. The *LonMaker New Device Wizard* opens, as shown in the following figure:





3. Enter a name for the device in **Device Name** . This name determines the name of the device in the LonMaker drawing and has no effect on the name of the NodeBuilder project. If you will want to load the application into the device immediately, set the **Commission Device** checkbox. Click **Next**. The second window of the New Device Wizard opens, as shown in the following figure:



This window displays the name of the device and the target type of the

device. Click the arrow on **NodeBuilder Device Template** to select an existing NodeBuilder device template. All NodeBuilder device templates that you have successfully built will be listed. This list is updated when you click the arrow. The **LNS Device Template** box contains the name of the currently selected device template, which is typically the same as the **NodeBuilder Device Template** field.

4. Click **Start NodeBuilder** to begin creation of a new NodeBuilder project. The NodeBuilder tool will start.

---

## Opening a NodeBuilder Project

To open an existing NodeBuilder project, you must first start the NodeBuilder Project Manager if it is not already running. You can start the NodeBuilder Project Manager from the LonMaker tool, or directly from the NodeBuilder program folder. You will typically start the project manager from the LonMaker tool since that simplifies associating the NodeBuilder project with the LonMaker network.

To open a NodeBuilder project if the NodeBuilder Project Manager is already running, follow these steps:

1. Open the NodeBuilder **File** menu, and then click **Open Project**.
2. Locate and open the folder containing your project file (".NbPrj" extension).
3. Double-click the NodeBuilder project file.

To open a NodeBuilder project by starting the NodeBuilder Project Manager from the LonMaker tool, follow these steps:

1. Create or open a LonMaker drawing. See the *LonMaker User's Guide* for more information on creating and opening LonMaker drawings. If you will want to load the application you develop into a device, make sure the LonMaker computer is attached to the network.
2. Open the **LonMaker** menu then click **NodeBuilder**. The NodeBuilder Project Manager starts. If you have previously created a NodeBuilder project for this network, the default project for the network opens.
3. To open a different project, open the NodeBuilder **File** menu, and then click **Open Project**.
4. Locate and open the folder containing your project file (".NbPrj" extension).
5. Double-click the NodeBuilder project file.

You can open a project and start the New Device Template wizard at the same time by dragging a Development Target or Release Target shape from the LonMaker stencil to your drawing.

You can open specific windows within the default project by right-clicking a Development Target or Release Target shape in the LonMaker drawing, and then clicking **Edit Source**, **NodeBuilder Properties**, **Build**, or **Debug** on the shortcut menu.

To open a NodeBuilder project by starting the NodeBuilder Project Manager standalone from the NodeBuilder program folder, follow these steps:

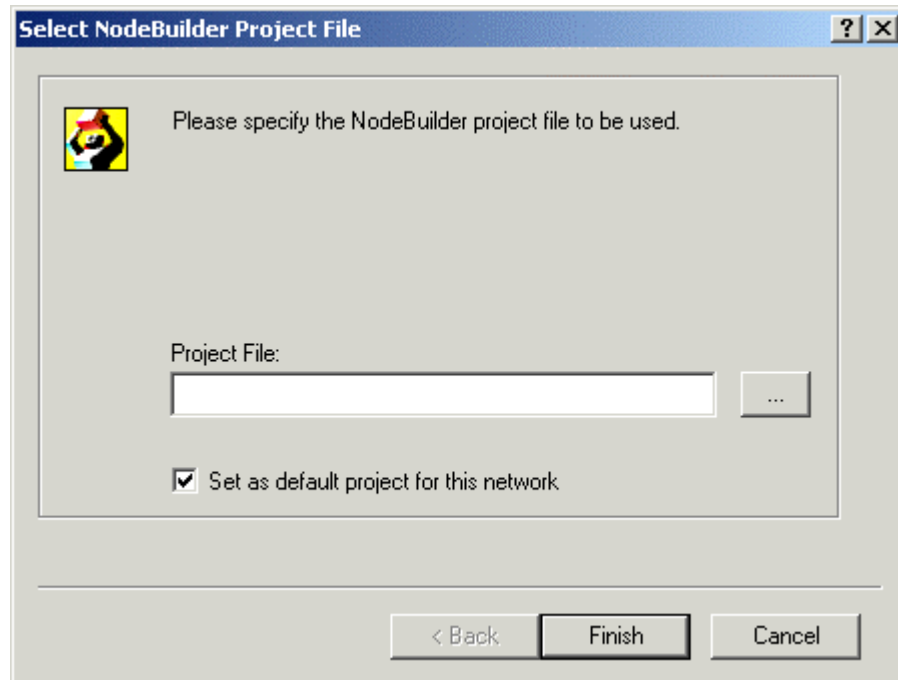
1. Click the Windows **Start** menu, point to **Programs**, point to **Echelon NodeBuilder Software**, and then click **NodeBuilder Development Tool**.


- The NodeBuilder Project Manager starts.
2. Open the NodeBuilder **File** menu and then click **Open Project**. The New Project wizard starts.
  3. Locate and open the folder containing your project file (“NbPrj extension”).
  4. Double-click the project file.

---

## Selecting a NodeBuilder Project File

This dialog appears as shown in the following figure:



Enter the path and filename of the NodeBuilder Project file (“NbPrj” extension) to be opened. Click the  button to browse to the file. Set the **Set as Default Project for this Network** checkbox to have the selected file set as the default project for the LonMaker network that was used to open the NodeBuilder Project Manager.

---

## Copying a NodeBuilder Project to Another Computer

You can copy a NodeBuilder project to another computer. To copy a NodeBuilder project, follow these steps:

1. Ensure that the target computer and the computer currently containing the NodeBuilder project have the same version of the NodeBuilder and LonMaker tools.
2. Start the LonMaker tool. The LonMaker Design Manager opens.
3. Select the LonMaker network associated with the NodeBuilder project to be copied and then click **Backup**. The LonMaker Backup dialog opens.
4. Set **Backup Drawing**, **Backup Database**, and **Backup NodeBuilder**

**Project.** See the *LonMaker User's Guide* and help file for more information.

5. Click **OK**. The LonMaker drawing and database, and the NodeBuilder project, are all archived in a single LonMaker backup file (".zip" extension).
6. Copy the LonMaker backup file to the new computer. Start the LonMaker tool and then click **Restore**. The LonMaker drawing and database and the NodeBuilder project are copied to the new computer. The NodeBuilder project is associated with the LonMaker network.
7. Copy any user hardware templates and custom libraries to the new computer. Place the user hardware templates into the LONWORKS NodeBuilder\Templates\Hardware\User folder. Place the custom libraries in the same folder they were in on the original computer. If this is not possible, re-add them to the project as described in *Inserting a Library into a NodeBuilder Device Template*.

Alternatively, you can copy all the files associated with the project manually, though this is more difficult than using LonMaker backup and restore. In order to do this, follow these steps:

1. Ensure that the target computer and the computer currently containing the NodeBuilder project have the same version of the NodeBuilder tool.
2. Copy the entire project folder and its contents to the target computer. By default, the project folder has the same name as the NodeBuilder project and is stored in the Lm\Source folder. The project folder will contain one subdirectory for each device template in the NodeBuilder project.
3. Copy any user hardware templates and custom libraries to the new computer. Place the user hardware templates into the LONWORKS NodeBuilder\Templates\Hardware\User folder. Place the custom libraries in the same folder they were in on the original computer. If this is not possible, re-add them to the project as described in *Inserting a Library into a NodeBuilder Device Template*.
4. Start the NodeBuilder tool as described in *Opening a NodeBuilder Project* earlier in this Chapter and browse to the project file to open it.

**Note:** To simplify copying a NodeBuilder project to another computer, use the \$LonWorks\$ phrase when specifying include search paths. You can use this phrase in a search path to automatically specify the LONWORKS directory. Even if two computers have the LONWORKS directory in different places, or on different drives, you can copy a project to the new computer as long as you place it in the same relative path to the LONWORKS directory.

For example, a project may include files from the Gizmo 4 library, which requires NodeBuilder\Gizmo4\gizmo4.h to be included from the LonWorks directory. This can be achieved by adding \$LonWorks\$\NodeBuilder\Gizmo4 to the project include search path (see *Specifying the Project Default Settings*, earlier in this chapter, for more information on include search paths).

Some devices share include files that reside in a headers folder which is a directory within the project directory. This could be accomplished by adding **headers** to the project include search path, or by adding **..\headers** to each device template include search path.

---

## Copying a NodeBuilder Device Template to Another Computer

You can copy a NodeBuilder device template to another computer. To do this, follow these steps:

1. Ensure that the target computer and the computer currently containing the NodeBuilder device template have the same version of the NodeBuilder tool.
2. If the NodeBuilder project that will contain the device template has not been created, create it as described in *Creating a New Project* earlier in this chapter
3. Copy the entire device template folder and its contents to the target computer. Place the device template folder inside the project folder of the NodeBuilder project that will contain the device template.
4. Copy any user hardware templates and custom libraries to the new computer. Place the user hardware templates into the LONWORKS NodeBuilder\Templates\Hardware\User folder. Place the custom libraries in the same folder they were in on the original computer. If this is not possible, re-add them to the project as described in *Inserting a Library into a NodeBuilder Device Template*.
5. Open the NodeBuilder tool if it is not already open.
6. Right-click the **Device Templates** folder and then click **Insert** on the shortcut menu. Browse to the device template folder copied in step 3, open it, and then select the NodeBuilder device template file (“.NbDt” extension). The device template will now be added to the NodeBuilder project.

---

## Viewing and Printing NodeBuilder XML Files

Many of the files created by the NodeBuilder tool are XML files. You can view and print these files using a variety of tools including Internet Explorer, Microsoft Excel, or Microsoft XML Notepad (see [msdn.microsoft.com/xml/notepad/intro.asp](http://msdn.microsoft.com/xml/notepad/intro.asp)). This can be useful for generating printed summaries of the options contained in these files. Do not change the contents of these files. To open one of these files, right-click the file in Windows Explorer and then click **Open With** on the shortcut menu. Choose Microsoft Excel, Internet Explorer, XML NotePad, or another XML browsing tool in the Open With dialog.

The NodeBuilder tool creates and maintains the following XML files :

<b>Project File (*.NbProj)</b>	Contains a project definition including the project version and a list of the device templates and the hardware templates for a project. There is one project file per project. It is kept in the project folder.
<b>Options File (*.NbOpt)</b>	Contains the NodeBuilder project options for a project. There is one options file per project. It is kept in the project folder.
<b>Device Template File (*.NbDt)</b>	Contains a device template, including the options specified for the device template and device template targets. There is one device

template file per device template. It is kept in the device template folder.

**Hardware Template File  
(\*NbHwt)**

Contains a hardware template, including the options specified for the hardware template. There is one hardware template file per hardware template. Standard hardware template files are kept in the LONWORKS Templates\Hardware\Standard folder. User hardware template files are kept in the LONWORKS Templates\Hardware\User folder. Hardware templates specific to the project can also be contained in the project folder.

# 5

## Creating and Using Device Templates

This chapter describes how to create a new NodeBuilder device template using the New Device Template Wizard and describes hardware templates, libraries, and targets.

---

## Introduction to Device Templates

A *device template* defines a device type. The NodeBuilder tool uses two types of device templates. The first is a *NodeBuilder device template*. The NodeBuilder device template is an XML file with a “.NbDt” extension that specifies the information required for the NodeBuilder tool to build the application for a device. It contains a list of the application Neuron C source files and the hardware template name. When you build an application, the NodeBuilder tool automatically produces an *LNS device template*. The LNS device template defines the interface, and is used by LNS tools such as the LonMaker tool to configure and bind the device.

You will create device templates using the New Device Template Wizard. This wizard starts automatically if you set the **Run Device Template Wizard** checkbox when you create a new NodeBuilder project. You can also start this wizard by right-clicking the Device Templates folder in the Project pane and clicking **New** on the shortcut menu.

---

## Using Device Templates

You can view and edit device templates using the NodeBuilder Project Manager. The **Device Templates** folder in the Project pane of the project manager lists all the device templates that are defined as part of the currently open NodeBuilder project. Right-click the **Device Templates** folder to see a shortcut menu with the following commands:

- |                    |  |
|--------------------|--|
| <b>New</b>         | Creates a new device template in the currently open NodeBuilder project. This starts the <i>New Device Template Wizard</i> described later in this chapter.  |
| <b>Insert</b>      | Inserts an existing NodeBuilder device template into the currently open NodeBuilder project. A dialog opens allowing you to browse to and select a NodeBuilder device template file (“.NbDt” extension). This option allows you to reuse device templates in multiple projects, and allows you to share a single device template among multiple projects to.   |
| <b>Insert Copy</b> | Creates a copy of an existing NodeBuilder device template and inserts it into the currently open NodeBuilder project. You can copy a NodeBuilder 3 or 3.1 device template (“.NbDt” extension) or convert and copy a NodeBuilder 1.5 device file (“.dev” extension). A dialog opens allowing you to browse to and select a NodeBuilder device template file (“.NbDt” or “.dev” extension). Once you have selected an existing device template, the <i>New Device Template Wizard</i> opens. Use this wizard to select a path, change the program ID, etc. All files associated with the device template |



(i.e. all files in the **Source Files** sub-folder) will be copied to the new device template.

<b>Build</b>	Builds the application images for all qualifying targets. See <i>Building an Application Image</i> in the <i>Compiling, Building, and Loading Applications</i> chapter for more information.
<b>Clean</b>	Deletes all output files created when building the currently open NodeBuilder project for all qualifying targets. See <i>Cleaning Build Output Files</i> in the <i>Compiling, Building, and Loading Applications</i> chapter for more information.
<b>Status</b>	Displays the build status for all device templates. See <i>Viewing Build Status</i> in the <i>Compiling, Building, and Loading Applications</i> chapter for more information.

Right-click one of the device templates in the Device Template folder to see a shortcut menu with the following commands:

<b>Settings</b>	Opens the NodeBuilder Device Template Properties dialog. This dialog allows you to change the options set in the Device Template Wizard's <i>New Device Template</i> and <i>Program ID</i> windows.
<b>Set Source File</b>	Sets the main source file (.nc extension) for this device template. By default the main source file is <code>&lt;Device Template Name&gt;.nc</code> .
<b>Code Wizard</b>	Starts the NodeBuilder Code Wizard for this device template. See the <i>Generating Neuron C Source Code Using the Code Wizard</i> chapter for more information about the code wizard.
<b>Plug-in Wizard</b>	Starts the LNS Device Plug-in Wizard for this device template. See the <i>Creating an LNS Device Plug-in</i> chapter and the <i>LNS Plug-in Programmer's Guide</i> for more information on the LNS Device Plug-in Wizard.
<b>Remove</b>	Removes this device template from the currently open NodeBuilder project. This does not delete the device template file or source files. Use Windows Explorer to delete these files.
<b>Build</b>	Builds the application image specified by this device template for all qualifying targets. See <i>Setting Build Options</i> in the <i>Compiling, Building, and Loading Applications</i> chapter for more information.
<b>Clean</b>	Deletes all output files created when building this device template for all qualifying targets. See <i>Cleaning Build Output Files</i> in the <i>Compiling,</i>

*Building, and Loading Applications* chapter for more information.

<b>Build Exclude</b>	Determines if this device template will be included or excluded when you click the Build command for the <b>Device Templates</b> folder. If you set this checkbox, the device template will be excluded from a device templates build, the device template name is dimmed, and a checkmark will appear next to <b>Build Exclude</b> on the shortcut menu. When you exclude a device template, you can still explicitly build the device template by right-clicking the device template and selecting Build from the shortcut menu.
<b>Status</b>	Displays the build status for this device template. See <i>Viewing Build Status</i> in the <i>Compiling, Building, and Loading Applications</i> chapter for more information.
<b>Properties</b>	Displays file properties of the NodeBuilder device template file (“NbDt” extension). The properties include the file name, location, size, and dates the file was created, last modified, and last accessed.

Each device template contains the following items:

<b>Main Source File</b>	The first item in the device template is the main Neuron C source file (.nc extension) for this device template. This file may include other source files by using Neuron C <b>#include</b> “filename” statements. By default, this file is named <i>&lt;Device Template Name&gt;.nc</i> . Double-click this file to edit it as described in the <i>Editing Neuron C Source Code</i> chapter.
-------------------------	---

<b>Source Files</b>	This folder contains all source files associated with this device template, with the exception of the main source file. When you add source files to the NodeBuilder project directly or using the NodeBuilder Code Wizard, they will be added to this folder. Double-click any source file to edit it as described in the <i>Editing Neuron C Source Code</i> chapter.
---------------------	---

To add source files, right-click the Source Files folder, and then click **Insert** on the shortcut menu. You can add any file, but you will typically add Neuron C files (“.nc” extension), header files (“.h” extension), C files (“.c” extension), text files (“.txt” extension), or other specification or documentation files. All files to be included when you build the application image must be explicitly included in the Neuron C code using **#include** “filename” statements; adding files to this folder does not automatically include them in the

build. You can add non-source code files to this folder to allow them to be easily accessed from the project.

Right-click a source file and then click **Remove** on the shortcut menu to remove it from the device template. Right-click a source file and then click **Properties** on the shortcut menu to view the location, size, and date stamps of the file.

### **Libraries**

This folder contains all libraries explicitly used by this device template. A *library* is a file containing one or more compiled ANSI C functions. When you build the application image for a device template, functions are included from libraries if they are referenced by any code included in the device template. The code for any unreferenced functions is not included in the application image. To add a library, right-click the **Libraries** folder and then click **Insert** on the shortcut menu. The **Specify Library Type** dialog opens. Select a custom or standard library type and then click **Next**. See *Inserting Libraries into a NodeBuilder Project* in the *Creating and Using Device Templates* chapter for more information.

### **Development/Release**

The development and release targets contain information specific to building application images for development and release targets, respectively. See *Using Device Template Development and Release Targets*, later in this chapter, for more information.

---

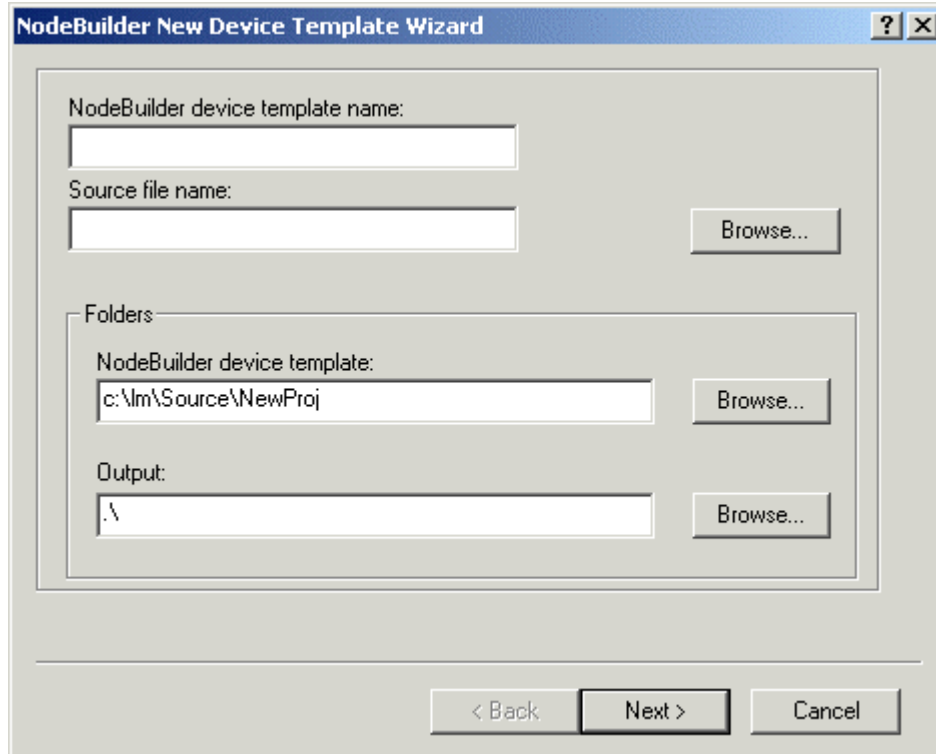
## **Using the New Device Template Wizard**

The Device Template Wizard guides you through the process of creating a new NodeBuilder device template. You will specify a device template name, working directories, a Program ID, and hardware templates. The Device Template Wizard contains three windows: *New Device Template*, *Program ID*, and *Target Platforms*.

---

### ***New Device Template Wizard: New Device Template***

The **New Device Template** window is the first window of the Device Template Wizard. This window is shown in the following figure:



This step of the Device Template Wizard allows you to assign a name and select a location for a new NodeBuilder device template. This window contains the following fields:

**Device Template Name**

The name of the device template. A NodeBuilder device template file with this name and a “.NbDt” extension will be created in the folder specified in the NodeBuilder Device Template Folder field. This entry must be non-empty and a valid Windows file name. The name can contain up to 210 characters, including spaces. The name cannot contain the following characters: \ / : \* ? " < > | .

**Source File Name**

Enter the name of the Neuron C source file for this device template. By default, this field is set to *<Device Template Name>.nc*, and the file will be created in the folder specified in the NodeBuilder Device Template Folder field. Click **Browse** to select an existing source file.

**NodeBuilder Device Template**

Enter the folder into which the device template file will be placed. This is called the *device template folder*. By default, this is a folder with the same name as the device template that is contained within the project folder. Click **Change** to browse to a different folder.

## Output

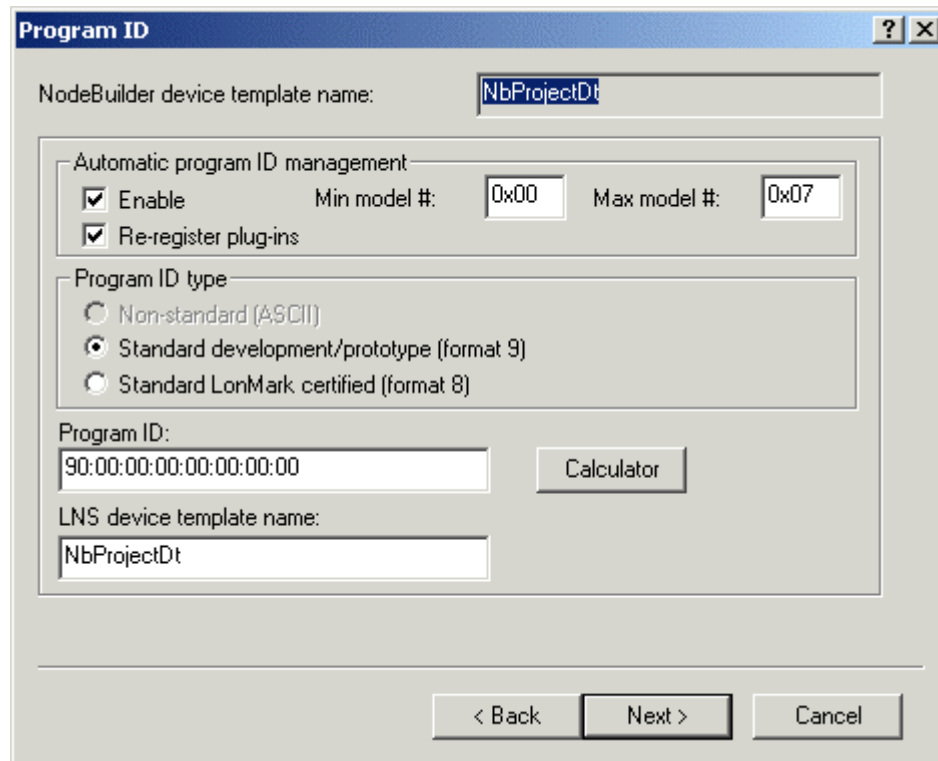
Enter the root folder for output files generated by the build process. You can enter either an absolute or relative path name. Relative paths are based on the device template folder. The default value is . \ (the build target folder).

Set your desired options, and then click **Next**. The *Program ID* window opens.

---

## New Device Template Wizard: Program ID

The Program ID window is the second window of the Device Template Wizard. You can access these options after creating a device template by right-clicking the device template and selecting **Settings**, then selecting the **Program ID** tab. This window is shown in the following figure:



This step of the Device Template Wizard allows you to specify the program ID and determine whether automatic program ID management will be used. Enter the following information:

### NodeBuilder Device Template Name

The name of the device template. This field is read-only.

### Automatic Program ID Management

When this checkbox is set, the NodeBuilder tool automatically increments the model number field within the program ID whenever the device interface changes. It also automatically modifies the LNS device template name whenever the

program ID is changed. The program ID is incremented within the range defined by the **Min Model #** and **Max Model #** fields. This ensures that the program ID uniquely identifies a device template. When selected, the **Min Model #** and **Max Model #** fields are enabled, and the **Nonstandard (ASCII) Program ID Type** is disabled. This option is set by default. If you disable this option, you must manually manage the program ID and device template name to ensure they are unique for each unique device interface.

If you set this checkbox, the NodeBuilder tool automatically upgrades all target devices using this device template if **Load After Build** is set. To upgrade the target devices, the NodeBuilder tool creates a new device template with the new name and program ID, and then downloads the new application to the target devices, preserving connections for compatible network variables.

When the **Max Model #** value is reached, the model number field of the program ID will be reset to the **Min Model #** value. The NodeBuilder tool automatically deletes old LNS device templates with the minimum model number as long as they are not in use by any devices. If the old LNS device template is in use, the NodeBuilder tool reports an error.

Clear this checkbox if you are creating a resource file for the device template and the resource file specifies a scope of 6 (model number specific). If this option is set with a scope 6 resource file, you will have to modify the program ID template in the resource file each time you change the device interface.

### **Re-register Plug-ins**

When this option is set, the NodeBuilder tool automatically re-registers LNS plug-ins whenever the program ID changes. This option is only available if the **Automatic Program ID Management** option is set.

Only LNS plug-ins that were registered for the most recent previous device template will be registered (i.e. if you turn this option off for several program ID changes, then turn it back on, you will need to manually re-register LNS plug-ins for the newest version of the device template).

### **Min and Max Model #**

These fields are only active if the **Automatic Program ID Management** checkbox is set. Enter the range of model numbers allowed for automatic program ID management. After the

**Max Model #** value has been reached, it is reset to the **Min Model #** value. The **Max Model #** value must be greater than the **Min Model #** value. The fields are specified as 2-digit hex numbers (00 – FF). By default, **Min Model #** is set to 00 and **Max Model #** is set to 07..

**Program ID Type**

Select the type of program ID to be used – *Non-standard (ASCII)*, *Standard development/prototype (format 9)*, or *Standard LONMARK certified (format 8)*. The setting of this value determines the format of the data for the **Program ID** field. The Standard LONMARK Certified (format 8) option is reserved for LONMARK certified devices. With the exception of LONMARK certified, legacy, and network interface devices, most devices should use the default Standard Development/Prototype (format 9) option. You cannot use the NodeBuilder Code Wizard or the LNS Device Plug-in Wizard if you select the Non-standard (ASCII) option.

**Program ID**

Enter the program ID for the device template. The program ID is a 16-hex-digit number that uniquely identifies the device interface for a device. Network tools assume that two different devices with the same program ID have the same device interface.

Depending on the **Program ID Type** selected, this value has the following constraints:

*Non-standard (ASCII)* - The field accepts up to 8 text characters.

*Standard development/prototype (format 9)*. Enter 16 hex digits, the first of which must be a '9'. Pairs of digits are automatically delimited by colons (":") for readability; the colons are not part of the program ID.

*Standard LONMARK certified (format 8)*. Enter 16 hex digits, the first of which must be an '8'. Pairs of digits are delimited by colons (:) for readability; the colons are not part of the program ID.

Click the **Calculator** button to start the **Standard Program ID Calculator**. You can use the wizard to generate a program ID based on information you supply for each field of the program ID. See *Using the Program ID Calculator* for more information.

**LNS Device Template Name**

Enter the name of the LNS device template. This is the name that the device template will be referred to by LNS tools such as the LonMaker

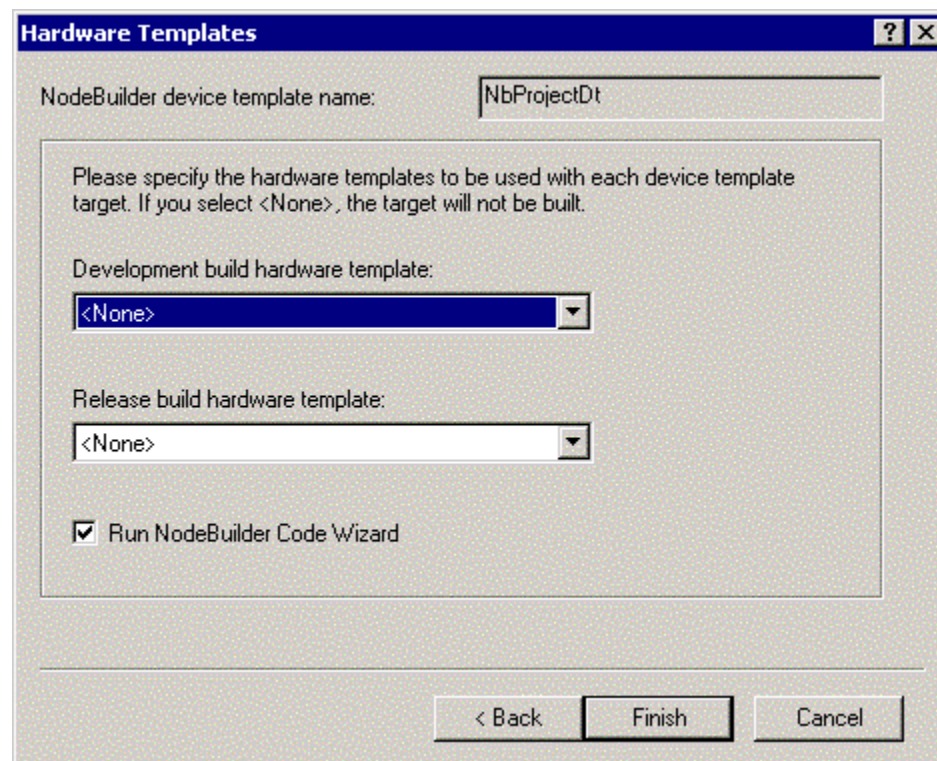
tool. Each LNS device template has a unique program ID. You may not have multiple LNS device templates with the same name in a network, so if you change the program ID of a NodeBuilder device template, you must also change the LNS device template name. If you are using automatic program ID management, the name of the LNS device template will automatically be updated in the format `<Device Template Name> [version number]`. To revert to the old LNS device template name, you must remove the LNS device template with the old name from the LNS database (for example by using the Device Templates dialog in the LonMaker tool). The default LNS device template name is the same as the NodeBuilder device template name.

Set your desired options, and then click **Next**. The **Hardware Templates** window opens.

---

## *New Device Template Wizard: Hardware Templates*

The Hardware Templates window is the third window of the Device Template Wizard. To access these options after the device template has been created, right-click the device template in the NodeBuilder Project pane and select **Settings**, then select the **Hardware Templates** tab. This window is shown in the following figure:





Specify the hardware templates used for Debug and Release targets.

A *hardware template* is a file that defines the hardware configuration for a device. It specifies hardware attributes including platform, transceiver type, Neuron Chip or Smart Transceiver model, clock speed, system image, and memory configuration.

A *target* is a LONWORKS device whose application is built by the NodeBuilder tool. There are two types of targets, *development* targets and *release* targets. Use development targets during development to facilitate debugging your application; use release targets when you complete development and you are ready to release the device to production.

You do not have to select the hardware templates at this point, but you must select them before you can build the device template. Enter the following information:

<b>NodeBuilder Device Template Name</b>	The name of the device template.
<b>Development Build Hardware Template</b>	The hardware template for development targets. Click the arrow to display all the hardware templates in the Hardware Templates folder in the Project pane.
<b>Release Build Hardware Template</b>	The hardware template for release targets. Click the arrow to display all the hardware templates in the Hardware Templates folder in the Project pane.
<b>Run Code Wizard</b>	Set this checkbox to run the NodeBuilder Code Wizard immediately after clicking <b>Finish</b> . This option is not available if the <b>Program ID Type</b> of the device template is set to <b>Non-Standard (ASCII)</b> . The <b>Run NodeBuilder Code Wizard After Creating New Device Template</b> checkbox in the NodeBuilder Project Settings dialog's <i>Options</i> tab determines whether this checkbox is set by default.

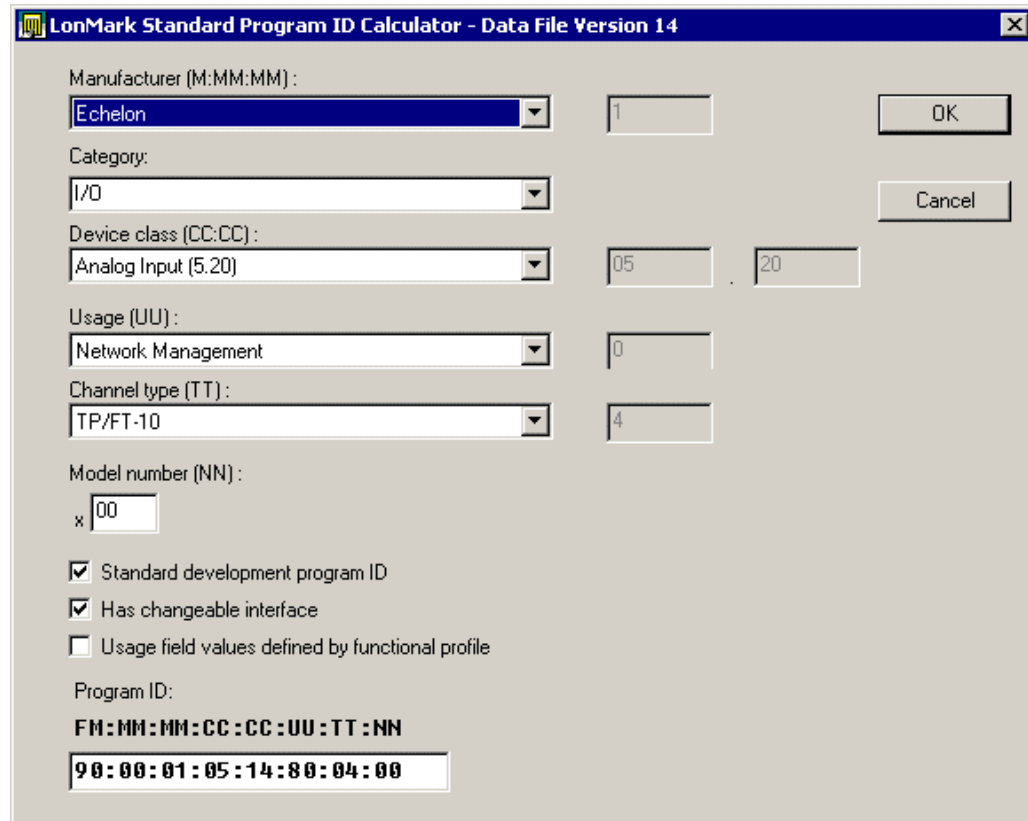
Set your desired options, and then click **Finish**. If you have set the **Run NodeBuilder Code Wizard** option, the NodeBuilder Code Wizard starts. See the *Generating Neuron C Code Using the Code Wizard* chapter for more information about the NodeBuilder Code Wizard.

---

## Using the Standard Program ID Calculator

The program ID is a 16-hex-digit number that uniquely identifies the device interface for a device. You can format the program ID as a standard or non-standard program ID. When formatted as a standard program ID, The 16 hex digits are organized as 6 fields that identify the manufacturer, classification, usage, channel type, and model number of the device. The Standard Program ID Calculator makes it easy for you to select the appropriate values for these fields by allowing you to select from lists contained in a program ID definition file included with the NodeBuilder tool and updated by the LONMARK Interoperability Association.

To start the Standard Program ID Calculator from the Device Template Wizard (see *Creating a Nodebuilder Project from the LonMaker Tool*), click **Calculator** in the Program ID window. To start the Standard Program ID Calculator from the New Resource File Set dialog (see *Creating a New Resource File Set or Editing an Existing Set* in Chapter 6), click **Calculator** in the dialog. The Standard Program ID Calculator appears as shown in the following figure:



This dialog allows you to choose a value for each part of the standard program ID. The **Program ID** field at the bottom of the dialog displays the current program ID. Enter the following values to set the program ID:

**Manufacturer**

The device manufacturer. Click the arrow to select from a list of all the LONWORKS device manufacturers who are members of the LONMARK Interoperability Association. If your company is a member of the LONMARK association but is not included in the list, download the latest program ID data from [www.lonmark.org/spid](http://www.lonmark.org/spid). If your company is not a member of the LONMARK association, get a temporary manufacturer ID from [www.lonmark.org/mid](http://www.lonmark.org/mid). If your company is a LONMARK member, but not listed in the updated program ID list, or if you have a temporary manufacturer ID, select **<Enter Number [Decimal]>** in the Manufacturer list, then enter your manufacturer ID in the field to the right of

the Manufacturer field. Enter the value in decimal, the calculator converts it to hex for the program ID. You do not have to join the LONMARK association to get a temporary manufacturer ID, the information required to get one is very minimal, and there is no fee to get one. However, if your company is not a member of the LONMARK Interoperability Association, now is a good time to join. For more information, see [www.lonmark.org](http://www.lonmark.org).

### Category

The general purpose or industry of the device. Click the arrow to select from a list of categories maintained by the LONMARK association. The **Category** determines the device classes that will be available in **Device Class**. Select **ALL** to have **Device Class** show all existing device classes. Select **Profiles By Name** to have **Device Class** show an alphabetical list of all device classes with a profile on the LONMARK website. Select **Profiles By Number** to have **Device Class** show a numerical list (sorted by device class number) of all device classes with a profile on the LONMARK website. .

### Device Class

The primary function of the device. The primary function of the device is determined by the primary functional profile implemented by your device. Your device must implement at least one functional profile, and may implement multiple functional profiles. If you implement multiple functional profiles, determine which is the primary based on the most typical usage of your device. Enter one of the following depending on your primary functional profile:

- If you are using a standard functional profile other than functional profiles 0 through 6 and the functional profile is included in the standard resource file set, click the arrow and select the functional profile name. The device class will be set to the functional profile number for the selected functional profile.
- If you are using a standard functional profile other than functional profiles 0 through 6 that has not yet been included in the standard resource file set, click the arrow and select **<Enter Number [Decimal]>**, and then enter the functional profile key in the two boxes to the right of Device Class. Enter the last two decimal digits in the second box, and the remaining decimal digits in the first box.

- If your primary functional profile is based on standard functional profiles 1 through 5 (you cannot use functional profiles 0 or 6 as the primary functional profile) or a user functional profile, click the arrow to select from a list of device classes maintained by the LONMARK association. You can update the list by downloading the latest program ID data from *www.lonmark.org/spid*. To enter a device class value that has not yet been added to the standard list, select **<Enter Number[Decimal]>** and enter a decimal value from 0 to 255 in each of the fields to the right of the Device Class field (the calculator converts the values to hex for the program ID).

### Usage

The intended usage of the device. The most significant two bits are determined by the **Has Changeable Interface** and **Use Field Values Defined By Functional Profile** checkboxes below the **Usage** box. If you are using a standard usage value, set **Defined By Functional Profile**, and then click the arrow to select from a list of standard usage values maintained by the LONMARK association. You can update the list by downloading the latest program ID data from *www.lonmark.org/spid*. If the primary functional profile implemented by your device specifies custom usage values, clear **Defined By Functional Profile**, select **<Enter Number[Decimal]>** in the Usage list, and then enter a decimal value from 0--255 in the field next to the Usage list (the calculator translates the value to hex for the program ID).

### Channel Type

The channel type supported by the device's LONWORKS transceiver. Click the arrow to select from a list of channel types maintained by the LONMARK association. You can update the list by downloading the latest program ID data from *www.echelon.com/spid*. Select **Custom** if you are using a transceiver that is not compatible with any of channel types in the list. To enter a channel type value that has not yet been added to the standard list, select **<Enter Number[Decimal]>** and enter a decimal value from 0 to 255 in the box to the right of the Channel Type box (the calculator converts the value to hex for the program ID).

### Model Number

The specific product model. Assign a unique model number for the specified manufacturer, device class, usage, and channel type. You may use the same hardware for multiple model

numbers depending on the program that is loaded into the hardware. The model number within the program ID does not have to conform to your published model number. This value can be automatically updated by setting **Automatic Program ID Management** in the *Program ID* window of the device template wizard (described earlier in this chapter). Enter a model number within the range specified in the device template wizard.

**Standard Development Program ID**

Identifies this device as a development or prototype device. Set this checkbox if the device has not been certified by the LONMARK Interoperability Association. When set, the calculator sets the **F** field of the program ID to 9. When cleared, the calculator sets the **F** field of the program ID to 8.

**Has Changeable Interface**

Set this checkbox to indicate that the device has a changeable device interface. Set this checkbox if the device has any network variables with changeable types, or if the device supports dynamic network variables.

*Dynamic network variables* are network variables that are added at installation time by a network tool. You can only implement dynamic network variables on host-based devices, so you cannot use them in any Neuron Chip or Smart Transceiver-hosted devices that you develop with the NodeBuilder tool.

Network variables with changeable types are network variables whose type can be modified at installation time by a network tool. You can implement changeable type network variables on any type of device. See the *Using the Resource Editor* chapter in this document and *The Neuron C Programmer's Guide* for more information.

**Usage Fields Defined By Functional Profile**

Set this checkbox if the primary functional profile implemented by this device defines usage values. Otherwise, clear the checkbox to specify standard usage values. When set, the **Usage** value will be set to <Enter Number>. Enter the custom usage value in the box to the right of the **Usage** box.

**Program ID**

This box is automatically updated when changes are made to the other boxes. You can also enter some or all of the program ID components directly into this box. If you enter values directly, the calculator updates the other boxes to match what you have entered.

---

## Using Device Template Targets

Each NodeBuilder device template in the project manager Project pane contains a **Development** target and a **Release** target. These device template targets define the hardware properties and file dependencies for the two types of targets that you can build for each device template. Each of these targets contains the following items:

**Hardware Template** A file that defines the hardware configuration for a target. It specifies hardware attributes including platform, transceiver type, Neuron Chip or Smart Transceiver model, clock speed, system image, and memory configuration. Double-click the hardware template to change the hardware template properties (see *Creating and Editing Hardware Templates* for more information).

**Dependencies** Lists all the files required to build the application image for this target. This list is created automatically when you build the application image for this target. The list is empty until you successfully build an application image for this target.

Right-click the **Development** or **Release** target to see a shortcut menu containing the following commands:

**Settings** Displays compiling, linking, exporting, and configuration options for the target.

**Set Hardware Template** Sets the hardware template to be used for this target. You can select from all hardware templates contained in the **Hardware Templates** folder. You can also drag a hardware template from the Hardware Templates folder to the Development or Release target to specify a hardware template. You cannot build or clean a target until it has a hardware template.

**Build** Builds the application image for this target only. See *Building an Application Image* in the *Compiling, Building and Downloading Applications* chapter for more information.

**Compile** Compiles the application for this target only. Only the compilation step of the build process is completed; the application is not linked and the application image is not created.

**Clean** Deletes all output files created when building the target. See *Cleaning Build Output Files* in the *Compiling, Building, and Loading Applications* chapter for more information.

**Build Exclude** Determines if this target will be included or excluded when you build this device template. When this option is set, the target will be

excluded from a device templates build, the target name is dimmed, and a checkmark will appear next to **Build Exclude** on the shortcut menu. When you exclude a target, you can still explicitly build the target by right-clicking the target and selecting Build from the shortcut menu. See *Excluding Targets from a Build* in the *Compiling, Building, and Loading Applications* chapter for more information

**Status** Displays the build status for this target. See *Viewing Build Status* in the *Compiling, Building, and Loading Applications* chapter for more information.

---

## Inserting a Library into a Device Template

You can add a library to a NodeBuilder device template. A *library* is a file with a “.lib” extension containing one or more compiled ANSI C functions. When you build the application image for a device template, functions are included from libraries if they are referenced by any code included in the device template. The code for any unreferenced functions is not included in the application image.

There are two types of libraries: standard and custom. The standard libraries are included with the NodeBuilder tool and are automatically included when you build a device template. Custom libraries are any libraries that you or a third party creates. Custom libraries must be explicitly included in a NodeBuilder project. You may also explicitly include standard libraries in a NodeBuilder project for documentation purposes.

The following standard libraries are included with the NodeBuilder tool, and are automatically included in every NodeBuilder project (but are not explicitly listed in the Project pane unless you add them):

**Extarith.lib** The extended arithmetic function library. Provides floating point and 32-bit integer math functions. For more information see the *Neuron C Programmer's Guide*.

**Psg.lib** The programmable serial gateway library. Provides serial I/O functions for the PSG/3 and PSG-20 programmable serial gateways. See the programmable serial gateway documentation for more information.

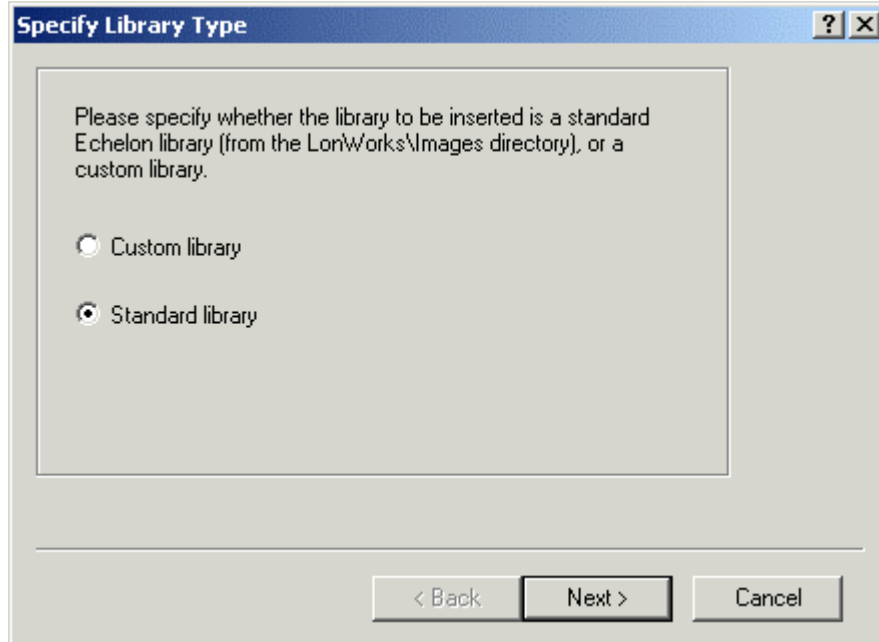
**Gen.lib** The standard Neuron C support library. Provides general support functions for Neuron C.

To see a summary of the contents of any library file, type “nlib -r <file name>” at a command line prompt. To save the summary, redirect the output to a file by typing “nlib -r <library file name> > <text file name>”.


You can create your own custom and standard libraries. See the *Neuron C Programmer's Guide* for more information on creating libraries.

To insert a library into a NodeBuilder device template, follow these steps:

1. Expand the device template in the project manager Project pane
2. Right-click the **Libraries** folder and then click **Insert** on the shortcut menu. The following dialog opens:



3. Choose whether you would like to insert a **Standard Library** or a **Custom Library**. Standard libraries are stored in the LONWORKS Images folder; custom libraries can be stored anywhere.
4. If you chose **Standard Library**, a dialog opens allowing you to choose from the libraries in the LONWORKS Images folder. If you select a standard library, when you build the application image the NodeBuilder tool first searches for this library in the folder within the Images folder that contains the system image for the target (for example: c:\LonWorks\Images\Ver12). If the library is not in that folder, the library in the LONWORKS Images folder is used (for example: c:\LonWorks\Images).

If you chose **Custom Library** a dialog opens allowing you to enter the path name of the library or libraries to be inserted. You can enter multiple library files by separating the paths with a semi-colon. To browse to a library file, click the  button and then browse to any file with the .lib extension.

---

## Using Hardware Templates

You can create, edit, and assign hardware templates with the NodeBuilder Project Manager. A *hardware template* is a file with a .nbHwt extension that defines the hardware configuration for a target device. It specifies hardware attributes including platform, transceiver type, Neuron Chip or Smart Transceiver model, clock speed, system image, and memory configuration. Several hardware templates are included with the NodeBuilder tool. You can use these or create your own. Third-party development platform suppliers may include NodeBuilder hardware templates for their platforms.



To view the currently defined hardware templates, expand the **Hardware Templates** folder in the project manager Project pane. The **Hardware Templates** folder contains **Standard Templates** and **User Templates** folders. The **Standard Templates** folder contains standard NodeBuilder hardware templates that are included with the NodeBuilder tool. The **User Templates** folder contains your custom hardware templates that you can use in any NodeBuilder projects on this computer. Any hardware templates unique to this project are located in the **Hardware Templates** folder, and are not contained in the Standard Templates or User Templates folders.

Right-click the **Hardware Templates** or **User Templates** folders to open a shortcut menu that allows you to select from the following commands:

- New** Creates a new hardware template that will be added to the selected folder. The **Hardware Template Properties** dialog opens. See *Setting Hardware Template Properties* for more information.
- If you are creating a hardware template in the **Hardware Templates** folder, you will be prompted to browse to a location for the hardware template file. If you are creating a hardware template in the **User Templates** folder, the new hardware template will be placed in the user hardware templates directory. This folder can be set in the *Options* tab of the NodeBuilder Project Properties dialog. By default, the user hardware templates directory is `c:\lm\Source\Templates\Hardware`. You can create folders in the user hardware templates folder, but the NodeBuilder tool will only show them if they contain at least one hardware template.
- Insert** Adds an existing hardware template to the **Hardware Templates** folder. A dialog appears that allows you to browse to and select an existing hardware template file (".NbHwt" extension). This command is only displayed for the **Hardware Templates** folder.
- Insert Copy** Adds a new hardware template to the **Hardware Templates** folder that is based on an existing hardware template. A dialog opens that allows you to browse to and select an existing hardware template file. Once you have a hardware template file, the Hardware Template Properties dialog opens with the values of the selected hardware template. You can change these properties as desired. Otherwise this option is identical to the **New** command. You can also drag a standard or user hardware template to the **Hardware Templates** folder to create a copy.

You cannot add hardware templates to the **Standard Templates** folder, so this folder has no shortcut menu.

You can also use this command to insert a NodeBuilder 1.5 device template file (.DTM extension). In NodeBuilder 1.5, device template files contained hardware properties.

You can add a hardware template to a device template's development or release target by dragging the hardware template from the **Hardware Templates** folder to the appropriate **Release** or **Development** folder. Each of these folders can contain only one hardware template. When you drag a hardware template to one of these folders, it replaces the old one if the folder already contained a hardware template. You can edit an existing hardware template by double-clicking it. The *Hardware Template Properties* dialog opens.

Do not modify hardware templates in the **Standard Templates** folder since any changes that you make will be overwritten by future NodeBuilder updates. To change a standard template, first insert a copy in the User Templates folder, and then make any changes to the copy. Future upgrades of the NodeBuilder tool will not modify any user templates.

You cannot remove hardware templates in the **Standard Templates** and **User Templates** folders since they may be used by other NodeBuilder projects. You can remove project-specific hardware templates in the **Hardware Templates** folder by right-clicking them and then clicking **Remove** on the shortcut menu. Removing a hardware template does not delete the hardware template file; it only removes the hardware template from the project. After removing the hardware template from the project, you can use Windows Explorer to delete the file if desired.

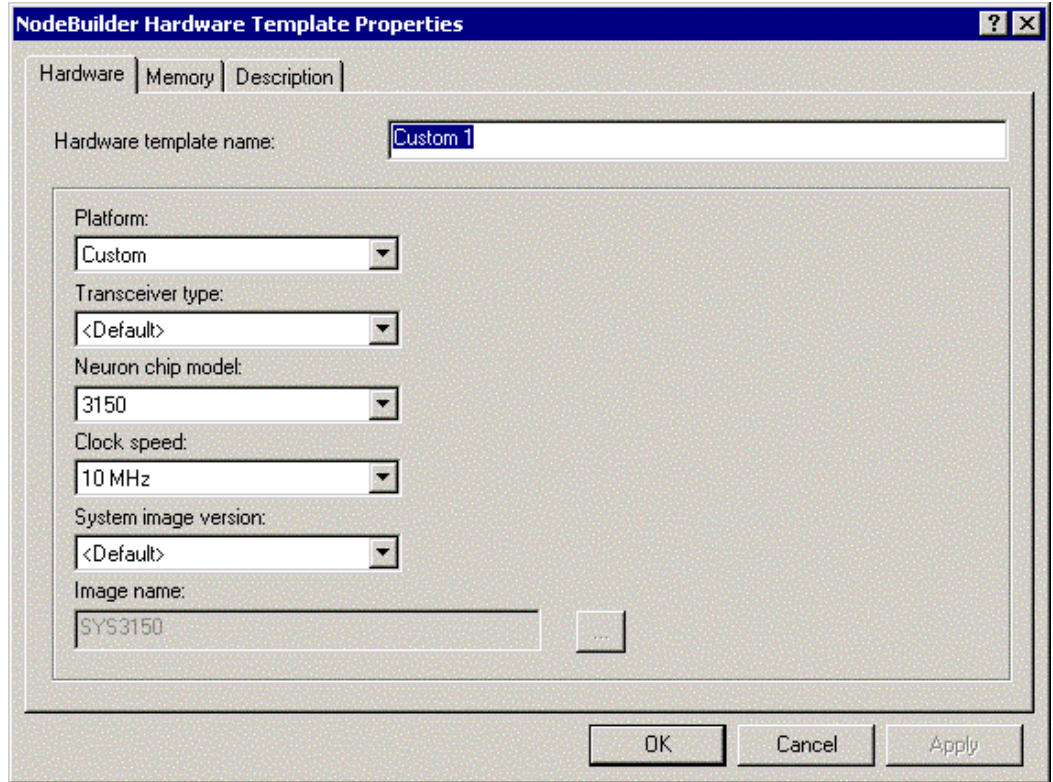
---

## *Creating and Editing Hardware Templates*

You can create and edit hardware templates that describe the hardware properties for a NodeBuilder target. To create or edit a hardware template, create a new hardware template, copy a hardware template, or open an existing hardware template as described in *Using Hardware Templates* in the previous section. Then set the hardware properties as defined in the following sections.

### Setting Hardware Properties

You can set hardware properties for a hardware template on the **Hardware** tab of the NodeBuilder Hardware Template Properties dialog. This tab appears as shown in the following figure:




You can use this tab to set the properties of the hardware template. If you open an existing template or create a new hardware template using **Insert Copy**, this tab shows the properties of the selected hardware template. If you create a new hardware template, it contains the default values shown above. This tab contains the following:

**Hardware Template Name** The name of the hardware template. By default, new hardware templates are named *Custom 1*, *Custom 2*, etc. The name may be any valid Windows file name. The name can contain up to 210 characters, including spaces. The name cannot contain the following characters: \ / : \* ? " < > |.

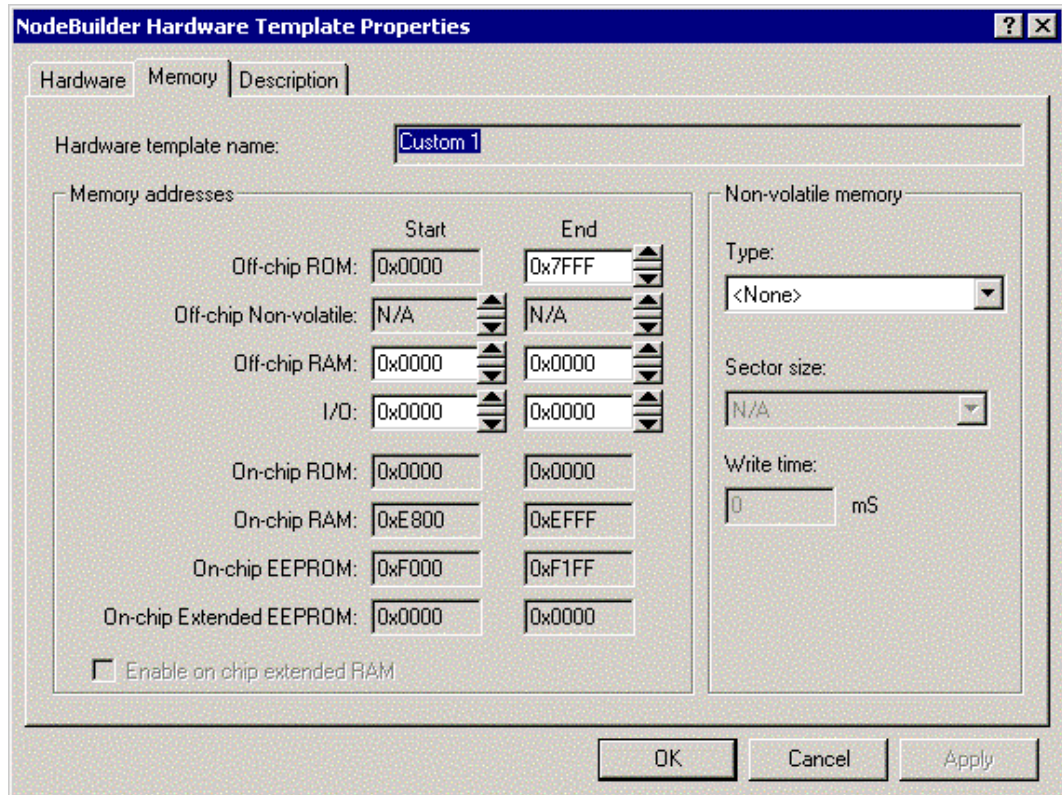
**Platform** The hardware platform. Click the arrow to select the **LTM-10**, **LTM-10A**, **LonBuilder Emulator 3150**, or a **Custom** platform. Select **Custom** if you are not using an LTM-10 Platform, LTM-10A Platform, or LonBuilder Emulator as the target platform, otherwise select your platform.

**Transceiver Type** The transceiver type. Each transceiver type identifies a unique set of transceiver parameters that are included in the application image if the boot image is included. Click the arrow to select from a list of transceiver types supported by the selected Neuron Chip or Smart Transceiver. Select **<Default>** to use the default transceiver specified in the project default settings.

<b>Neuron Chip Model</b>	The Neuron Chip or Smart Transceiver model. Click the arrow to select from a list of all Neuron Chip and Smart Transceiver models supported by the selected platform.
<b>Clock Speed</b>	The input clock speed for the Neuron Chip or Smart Transceiver. Click the arrow to select from a list of the available clock speeds for the selected Neuron Chip and transceiver, or the selected Smart Transceiver. See your Neuron Chip or Smart Transceiver data book for more information.
<b>System Image Version</b>	The system image version number. Click the arrow to select from a list of the available system image versions for the selected Neuron Chip or Smart Transceiver model. See your Neuron Chip or Smart Transceiver data book for more information. Select <b>&lt;Default&gt;</b> to use the default system image which is the most current system image version included with this version of the NodeBuilder tool and any applied service packs. Select <b>&lt;Custom&gt;</b> to specify your own custom system image in the <b>Image Name</b> field. See the <i>Neuron C Programmer's Guide</i> for information on creating custom system images.
<b>Image Name</b>	The file name of the system image. If you select <b>&lt;Custom&gt;</b> in <b>System Image Version</b> you can enter a system image file name or browse to a system image symbol file (.sym extension) by clicking the  button.

## Viewing and Setting Memory Properties

You can view the on-chip memory properties for a hardware template on the **On-chip Memory** tab of the NodeBuilder Hardware Template Properties dialog. This tab appears as shown in the following figure:



This tab provides details on how on-chip and off-chip memory is organized on the selected Neuron Chip or Smart Transceiver model.

On-chip memory values are dependent on the chip type and may not be modified with the exception of extended RAM (see *Using On-Chip Extended RAM*, later in the chapter). A value of 0x0000 will be shown for **Start** and **End** for any unavailable memory category.

Off-chip memory values are shown for chips that support off-chip memory. The Neuron 3120 Chip and 3120 Smart Transceiver do not support off-chip memory, so all fields in this tab will be set to N/A for these parts. A value of 0x0000 will be shown for **Start** and **End** for any unavailable memory. You can modify the off-chip memory start and end locations by clicking the arrows or by manually entering the hexadecimal values.

If you are using a Neuron 3150 Chip or 3150 Smart Transceiver, this tab specifies the type of non-volatile memory (**EEPROM**, **FLASH**, and **NVRAM**), if any. If **EEPROM** is selected, the **Write Time** field specifies the EEPROM write time. If **Flash** is selected, the **Sector Size** field specifies for flash memory sector size. See the Neuron Chip or Smart Transceiver data book for more information.

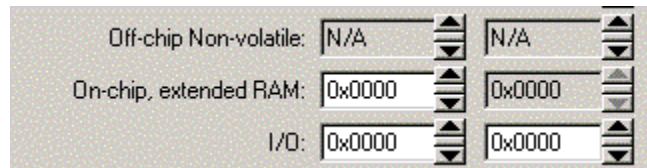
For devices where the system image is kept in non-volatile memory, select either **Flash** or **NVRAM**. EEPROM is not supported for this configuration.

### *Using On-Chip Extended RAM*

You can use the 2KB extended RAM on the Toshiba TMPN3150FR4F Neuron Chip. *Extended RAM* is on-chip RAM beyond the 2KB RAM in most Neuron 3150 Chips. If you are using the Toshiba TMPN3150FR4F chip, you can enable the extended RAM and assign its starting address to any available

page boundary. You cannot use extended RAM if you use off-chip RAM, but you can use extended RAM with off-chip EEPROM, flash memory, or memory-mapped I/O. To use extended RAM on a Toshiba TMPN3150FR4F chip, follow these steps:

1. Create a new Hardware Template or edit an existing one.
2. Select the **Hardware** tab, and then set the **Neuron Chip Model** to **TMPN3150FR4F**.
3. Select the **Memory** tab. The **Enable On-chip Extended RAM** checkbox will be set. When this checkbox is set, the **Off-chip RAM** field changes to **On-chip, Extended RAM**, as shown in the following figure:



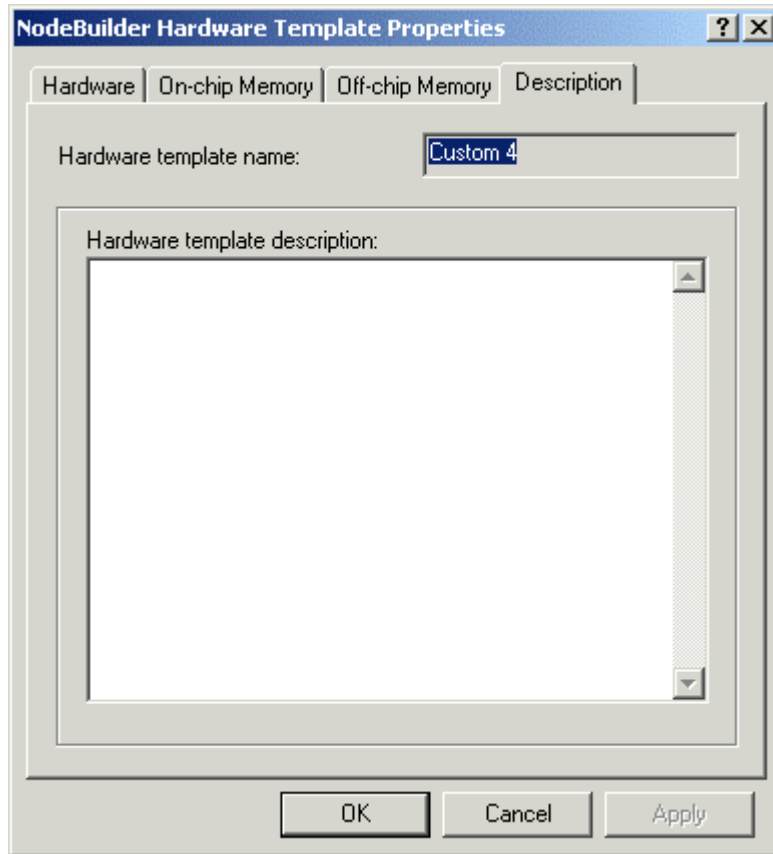
4. Set the starting address for the on-chip extended RAM. This address must be a multiple of 256 bytes (i.e. a page boundary).

If you are using off-chip RAM with a Toshiba TMPN3150FR4F chip, you must disable the extended RAM. To disable extended RAM, follow these steps:

1. Create a new Hardware Template by right-clicking the **Hardware Templates** or **User Templates** folder and selecting **New** from the shortcut menu (see *Using Hardware Templates*, earlier in this chapter). The **NodeBuilder Hardware Template Properties** dialog opens.
2. Select the **Hardware** tab, and then set the **Neuron Chip Model** to **TMPN3150FR4F**.
3. Select the **Memory** tab. Clear the **Enable On-chip Extended RAM** checkbox.

## Setting the Hardware Template Description

You can enter a description for a hardware template using the **Description** tab of the NodeBuilder Hardware Template Properties dialog. This tab appears as shown in the following figure:



Enter an optional description of the hardware template. This description will be saved in the hardware template file and will be available if this hardware template is used in other NodeBuilder projects.





# 6

## Generating Neuron C Code Using the Code Wizard

This chapter describes how to define your device interface and generate source code that implements the device interface. It explains how to use the NodeBuilder Code Wizard to define the network variables, configuration properties, and functional blocks to be implemented by the device.

---

## Introduction to the NodeBuilder Code Wizard

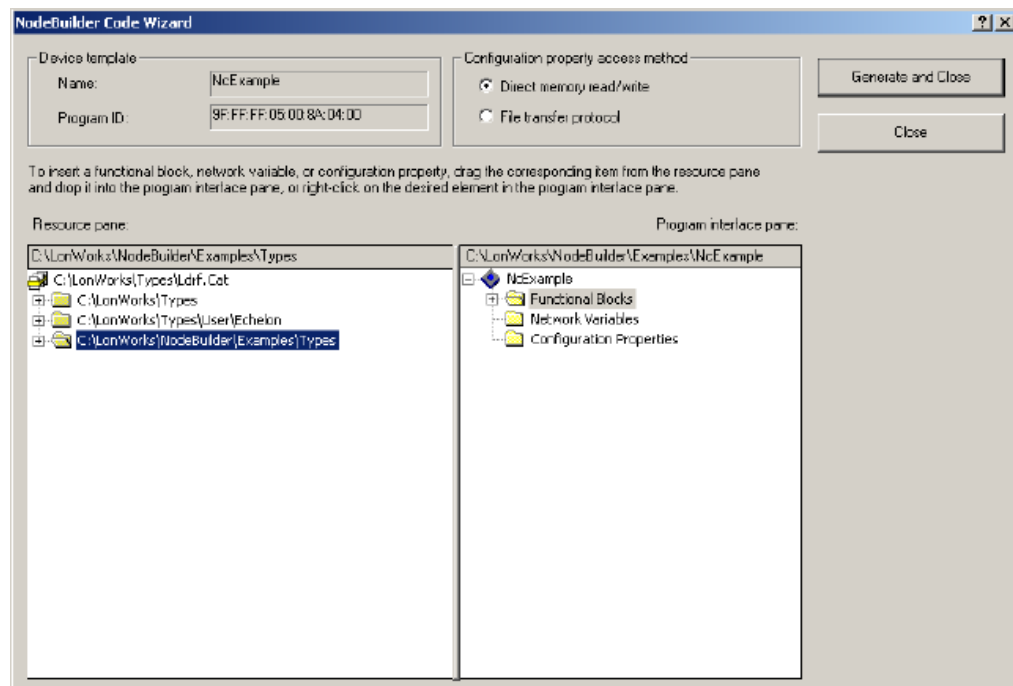
The NodeBuilder Code Wizard generates Neuron C source code that implements a device interface that you specify. The device interface defines the functional blocks, network variables, and configuration properties that are implemented by your device. As described in *Introduction to LONWORKS Networks*, network variables define the information the device can share with other LONWORKS devices on the network. Configuration properties define the information that can be configured by an LNS plug-in or a configuration property browser such as the LonMaker Browser. Functional blocks group network variables and configuration properties into functional units.

---

### Starting the Code Wizard

You can start the NodeBuilder Code Wizard when you are creating a new device template with from the *New Device Template Wizard*. To do this, set the **Start NodeBuilder Code Wizard** checkbox in the *Target Platforms* window of the *New Device Template Wizard*.

You can also start the Code Wizard at any time from within the NodeBuilder Project Manager by right-clicking a device template in the Project pane and clicking **Code Wizard** on the shortcut menu.



Enter the following information:

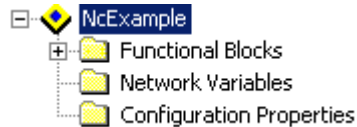
<b>Device Template</b>	The name and program ID of the NodeBuilder device template. To change the device template program ID, right-click the device template in the Project pane and select <b>Properties</b> from the shortcut menu.
<b>Configuration Property Access Method</b>	<p>Configuration properties may be accessed using read and write network management commands, or they be accessed using the LONWORKS File Transfer Protocol (FTP). The <b>Direct Memory Read/Write</b> option is the default, requires less space and code on the target device, and is the recommended option.</p> <p>If you select <b>Direct Memory Read/Write</b>, the Code Wizard automatically implements the <b>NodeObject</b> functional block's <b>nvoFileDirectory</b> optional network variable. It will not be allowed to implement the <b>nviFileReq</b>, <b>nviFilePos</b>, or <b>nvoFileStat</b> optional network variables.</p> <p>If you select <b>File Transfer Protocol</b>, the Code Wizard automatically implements the <b>NodeObject</b> functional block's <b>nviFileReq</b>, <b>nviFilePos</b>, and <b>nvoFileStat</b> optional network variables. It will not be allowed to implement the <b>nvoFileDirectory</b> optional network variable.</p>
<b>Resource Pane</b>	Lists all the resources in the resource catalog. Also lists all the resources that are available in these resource files. You will drag resources from the Resource pane to the Interface Pane to define your device interface. The Resource pane behaves identically to the <i>NodeBuilder Resource Editor</i> . The resource catalog is typically stored in the LONWORKS Types\LDRF.CAT file.
<b>Interface Pane</b>	The functional blocks, network variables, and configuration properties in the device interface. You will drag resources from the Resource pane to the Interface Pane to define your device interface. See <i>Defining the Device Interface</i> for more information.
<b>Generate and Close</b>	Click this button to generate a Neuron C code framework and close the Code Wizard. The Neuron C code generated by the Code Wizard is described under <i>Code Generated by the Code Wizard</i> later in this chapter.
<b>Close</b>	<p>Click this button to close the Code Wizard. If you have made changes, you will be asked whether you want to save them or not.</p> <p>Use this button to save changes you made without generating the Neuron C source code. You are</p>

responsible for synchronizing the Neuron C source and Code Wizard's view of the device interface. Click **Generate and Close** to have the source code generated that matches the interface defined in the Code Wizard.

---

## Defining the Device Interface

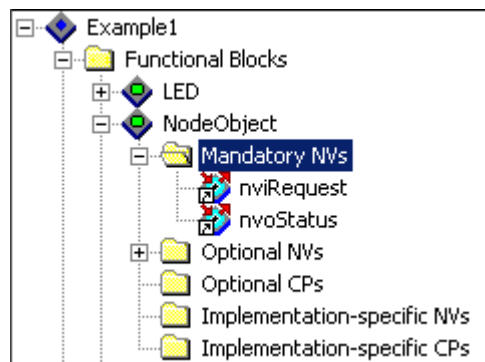
When you open the NodeBuilder Code Wizard with a new device template, the **Interface** pane looks as follows:



You can use the **Interface** pane to browse the device interface. At the top level is the device template name, called *NcExample* in this example, and beneath it are three folders labeled **Functional Blocks**, **Network Variables**, and **Configuration Properties**.

The **Functional Blocks** folder contains all the functional blocks contained in this device interface. The **Network Variables** folder contains all the *device network variables* for this device interface. The **Configuration Properties** folder contains all *device configuration properties* for this device interface. Device network variables and device configuration properties belong to the device as a whole; they are not contained by any functional block.

If an item in the Interface pane has a  $\boxplus$  next to it, the item contains other items. Click the plus to expand an item. If you expand the initial NodeObject functional block and its **Manadatory NVs** folder, the **Interface** pane looks like this:



The Code Wizard requires every device interface to contain a Node Object functional block with *nviRequest* and *nvoStatus* network variables. The Node Object functional block is used by network tools to test and manage the other functional blocks on a device. The Node Object functional block is required by the Code Wizard, though you can manually delete it from the generated code. If you remove the Node Object functional block with the Code Wizard before generating code, you cannot generate code with the Code Wizard. See the *LONMARK Application Interoperability Guidelines* and the Node Object functional profile for more information about the Node Object.

Right-click the device template to open a shortcut menu containing the following commands:

<b>CP Access Method</b>	Specifies a configuration property access method, as described in <i>Opening the Code Wizard</i> .
<b>Use External FB Name</b>	Toggles whether network tools, such as the LonMaker tool, will see the functional block name set in the Code Wizard. If this option is turned off, network tools will see the functional profile name.
<b>Generate and Close</b>	Generate codes and closes the Code Wizard.
<b>Refresh Catalog</b>	Refreshes the Code Wizard. If you made any changes in the Resource pane to network variable types, configuration property types, or functional profiles used by the device template, they will now be reflected in the <b>Interface</b> pane. If you change the name of a network variable type, configuration property type, or functional profile in the <b>Resource</b> pane, it will be removed from the <b>Interface</b> pane when the Code Wizard is refreshed and must be re-added.  You can also refresh an individual functional block or the device's <b>Network Variables</b> and <b>Configuration Properties</b> folders by right-clicking them and selecting <b>Refresh Catalog</b> from the shortcut menu.
<b>Properties</b>	Select this option to view <i>device template properties</i> .

## Adding a Functional Block to the Device Interface

Each functional block represents a specific function your device performs. For example, the LonPoint DI-10 device contains 4 hardware digital inputs, and each has its own functional block. To add a functional block to a device template, follow these steps:

1. Find the functional profile that defines the functional block in the Resource pane. Functional profiles are contained in functional profile folders within each resource file set. The Standard (Scope 0: Standard) resource file set contains the standard functional profiles. If you have defined your own functional profiles, they will be located in your resource file sets.
2. Drag the functional profile from the Resource pane to the Functional Blocks folder in the Interface pane. A new functional block with the same name as the functional profile (without the SFPT or UFPT prefix, and truncated to 16 characters or less) is added to the device interface. For example, dragging a SFPTsccChilledCeiling functional profile to the Interface pane creates a functional block named sccChilledCeilin. If additional functional blocks are created from the same functional profile, a number is appended to the name to make the name unique. The functional blocks are sorted by name.
3. If you have selected a functional profile of the same type and scope as an existing one, you will be asked whether you want to create an array. Click

**Yes** to create an array of functional blocks. Click **No** to create a new functional block using the same functional profile.

You can use a functional block array for two similar functional blocks rather than two separate functional blocks to reduce RAM and code space requirements.

4. Right-click the new functional block name then select **Rename** from the shortcut menu to change the name of the functional block. LNS network tools use this name to identify the functional block. This name is not case sensitive; however, creating a functional block, then removing it and creating another with different capitalization can cause compilation errors; to avoid this problem, be sure to delete the old Neuron C file (".nc" extension) before creating the new functional block. You may choose to use the functional profile name to identify the functional block instead. To do this, right-click the device template in the Code Wizard, and clear **Use External FB Name**.
5. If you need more than one instance of a functional block in the device interface, use an array of functional blocks. This will save RAM, code space, and **when** clauses. To create a functional block array, right-click the new functional block name then select **Properties** from the shortcut menu. Set **Use Array** and enter the size of the array.
6. When you add a new functional block, any mandatory network variables specified by the functional profile are automatically added to the **Mandatory NVs** folder within the functional block. This folder exists only if the functional profile contains mandatory network variables. Similarly, any mandatory configuration properties specified by the functional profile are automatically added to the **Mandatory CPs** folder within the functional block. This folder exists only if the functional profile contains mandatory configuration properties. You cannot delete mandatory items from the functional block.
7. If any of the mandatory network variables do not have a default type set by the functional profile, set the network variable type for those network variables. To set the type, double-click the network variable or right-click the network variable and select **Properties** to open the *Edit Network Variable Properties* dialog. If the **NV Type** field is not set, choose a network variable type.
8. Add any optional or implementation-specific network variables or configuration properties for this functional block. See *Implementing Optional Network Variables*, *Implementing Optional Configuration Properties*, *Adding Implementation-specific Network Variables*, *Adding Implementation-specific Configuration Properties*, *Adding Network Variables to the Device Interface*, and *Adding Configuration Properties to the Device Interface* for more information.

You can also add a functional block by right-clicking the Functional Blocks folder in the Interface pane and selecting **Add Functional Block** from the shortcut menu. See *Adding a Functional Block to the Device Interface Using the Add Functional Block Command* in the help file for details.

## Implementing Optional Network Variables

Functional profiles may specify mandatory network variables that must be implemented by any implementation of the profile, and may also specify optional network variables that may be implemented but are not required. When you add a functional profile to the device interface in the NodeBuilder

Code Wizard, the wizard adds all the mandatory members of the functional profile to the device interface but does not add any of the optional members. To implement an optional network variable on a functional block, follow these steps:

1. Right-click the **Optional NVs** folder for the functional block in the Code Wizard **Program Interface** pane and select **Implement Optional NV** from the shortcut menu. Alternately, you can drag a network variable from the functional profile's **Optional NVs** folder in the **Resource** pane to the functional block's **Optional NVs** folder in the **Interface** pane. If this functional profile does not have any optional network variables defined, the **Optional NVs** folder will not exist. A dialog appears that is similar to the following figure:

2. Use this dialog to select the optional network variable to implement from the optional network variables available in the functional profile. This dialog contains the following information:

<b>Name</b>	The name of the optional network variable as it will appear to the network integrator. This name defaults to the name specified by the functional profile. This name must be unique within the device, can be a maximum of 16 characters, and is case sensitive.
-------------	--


Once you add a network variable, you can rename it by right-clicking it and then clicking **Rename** on the shortcut menu.

<b>Array Element Count</b>	Indicates if the functional block containing this network variable is an array, and if so how many members are in the array. The optional network variable will be implemented for each functional block in the array. This information can be useful when determining how many network variables have been created on the device.
<b>Type</b>	The type of network variable to implement.
<b>FPT Member Name</b>	The name of the network variable member as specified in the functional profile. Click the arrow to display all optional network variables for this functional profile that have not yet been implemented for this functional block. If you change this value, the <b>Type</b> and <b>FPT Member Number</b> values are updated automatically.
<b>FPT Member Number</b>	Each mandatory and optional network variable in a functional profile is assigned a unique member number for the profile. This field indicates the member number of the currently selected optional network variable.
<b>Direction</b>	The direction of the currently selected optional network variable ( <b>Input</b> or <b>Output</b> ) as specified in the functional profile.
<b>Service Type</b>	The service type of the network variable ( <b>Unspecified</b> , <b>Acknowledged</b> , <b>Repeated</b> , or <b>Unacknowledged</b> ). This selection is only available for output network variables. You can specify the service type of the network variable if the functional profile has not specified one. See <i>Adding Implementation-specific Network Variables</i> , later in this chapter for more information about service types.
<b>Modifiers</b>	Indicates whether this network variable has the <b>Synchronized</b> or <b>Polled</b> modifiers. This selection is only available for output network variables. See <i>Adding Implementation-specific Network Variables</i> , later in this chapter, for more information about network variable modifiers.
<b>Self-document (sd_string)</b>	Optional additional text to be appended to the self-documentation string for this network variable. Network variable members of



functional blocks use a standard self-documentation format that is detailed in the *LONMARK Application Layer Interoperability Guidelines*. The Neuron C Compiler automatically generates all required self-documentation information. You can enter a text string to provide additional notes, which can be accessed from a network tool. The total length of the self-documentation string can be up to 1024 characters, including the characters automatically generated by the Neuron C Compiler.

### **Initializer**

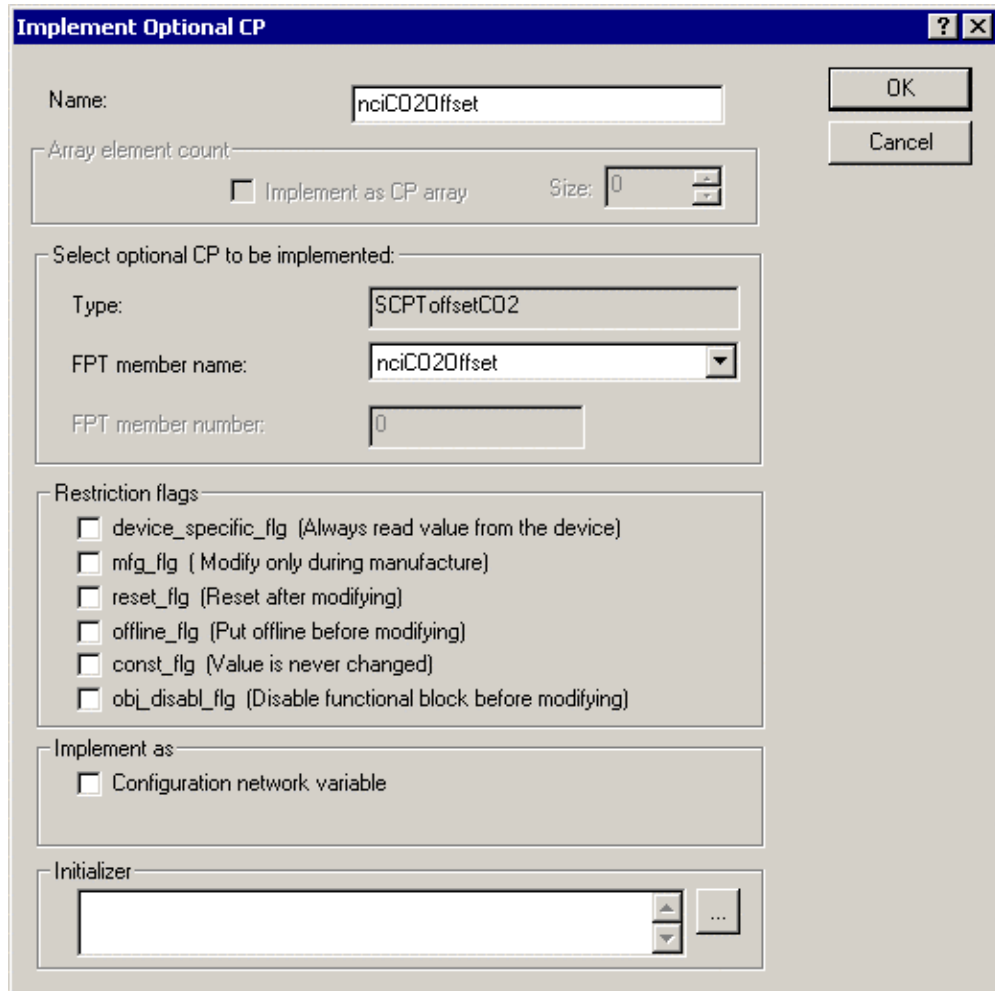
An optional initializer value for the network variable. This is the value that will be set when the device is reset. If this network variable is a structure, union, float, signed 32-bit, or enum type, click  to open the Edit Initializer dialog to get more information about the network variable type (see *Editing the Initializer for Network Variables and Configuration Properties* for more information).

Set your desired options, and then click **OK**. The optional network variable appears in the **Optional NVs** folder.

## **Implementing Optional Configuration Properties**

A functional profile may specify mandatory configuration properties that must be implemented by any implementation of the profile, and may also specify optional configuration properties that may be implemented but are not required. When a functional profile is added to the device interface in the NodeBuilder Code Wizard, the wizard adds all the mandatory members of the functional profile to the device interface but does not add any of the optional members. To implement an optional configuration property, follow these steps:

1. Right-click the **Optional CPs** folder for the functional block in the Code Wizard Interface pane and then click **Implement Optional CP** on the shortcut menu. Alternately, you can drag a configuration property from the functional profile's **Optional CPs** folder in the **Resource** pane to the functional block's **Optional CPs** folder in the **Interface** pane. If this functional profile does not have any optional configuration properties defined, the **Optional CPs** folder will not exist. A dialog appears that is similar to the following figure:



2. Use this dialog to select the optional configuration property to implement from the optional configuration properties available on the functional profile. This dialog contains the following information:

**Name**

The name of the configuration property as it will be used in the Neuron C program. This name defaults to the name specified by the functional profile. This name must be unique within the device, can be a maximum of 16 characters, and is case sensitive.

Once a configuration property has been added, you can rename it by right-clicking it and then clicking **Rename** on the shortcut menu.

**Implement as CP Array**


Set this checkbox to implement this configuration property as an array. The functional profile template may indicate that this configuration property must be implemented as an array or that it may not be implemented as an array; in either of these cases, this checkbox will be set appropriately and deactivated.

	See the <i>Neuron C Programmer's Guide</i> and <i>Neuron C Reference Guide</i> for more details on configuration property arrays.
<b>Size</b>	The size of the configuration property array if the <b>Implement as CP Array</b> checkbox is set. The functional profile template may specify a minimum or maximum size for the configuration property array; if you set <b>Size</b> to a value outside of this range, it will be reset to the minimum or maximum value (depending on whether the value you set was too low or too high). The functional profile may specify a fixed size for the array; in this case <b>Size</b> will be set appropriately and deactivated. A configuration property array can be up to 65500 bytes.
<b>Type</b>	The type of configuration property to implement. If you are implementing a CP array, this is the type of each element of the array.
<b>FPT Member Name</b>	The name of the configuration property as specified in the functional profile. Click the arrow to display all optional configuration properties for this functional profile that have not yet been implemented for this functional block. If you change this value, the <b>Type</b> value is automatically updated.
<b>FPT Member Number</b>	Not applicable to configuration properties.
<b>Restriction Flags</b>	Sets the configuration property flags. Network tools are responsible for checking these flags and handling configuration properties appropriately.  See the <i>Neuron C Programmer's Guide</i> and <i>Neuron C Reference Guide</i> for more information about configuration property restriction flags.
<b>Implement As</b>	Specifies implementation options for the configuration property. Specifies the following options:  <i>Configuration Network Variable</i> – If this option is set, the configuration property is implemented as a configuration network variable. This takes up network variable resources on the device but the configuration property can be read, written, and bound just like a network variable. If this option is cleared, the configuration property is implemented within a configuration file.

Configuration properties implemented within configuration files do not use up any network variable resources, but they cannot be bound to output network variables on other devices.

*Static CP* – This option is available only if the configuration property is in a functional block array. Setting this option creates a single configuration property to be shared among all functional blocks in the array. Modifying the value of the configuration property for any functional block on the array will modify all of them - there is actually only one variable allocated. If this option is cleared, a separate configuration property will be created for each functional block in the array. To share a configuration property between several functional blocks or network variables that are not part of an array, see *Sharing a Configuration Property*, later in this chapter.

### Initializer

An optional initializer value for the configuration property. This is the value that will be set when the device is reset. If this configuration property is a structure, union, float, signed 32-bit, or enum type, click  to open the Edit Initializer dialog to get more information about the configuration property type (see *Editing the Initializer for Network Variables and Configuration Properties* for more information).

3. Set your desired options, and then click **OK** to implement the configuration property on the functional block. The configuration property appears in the **Optional CPs** folder.

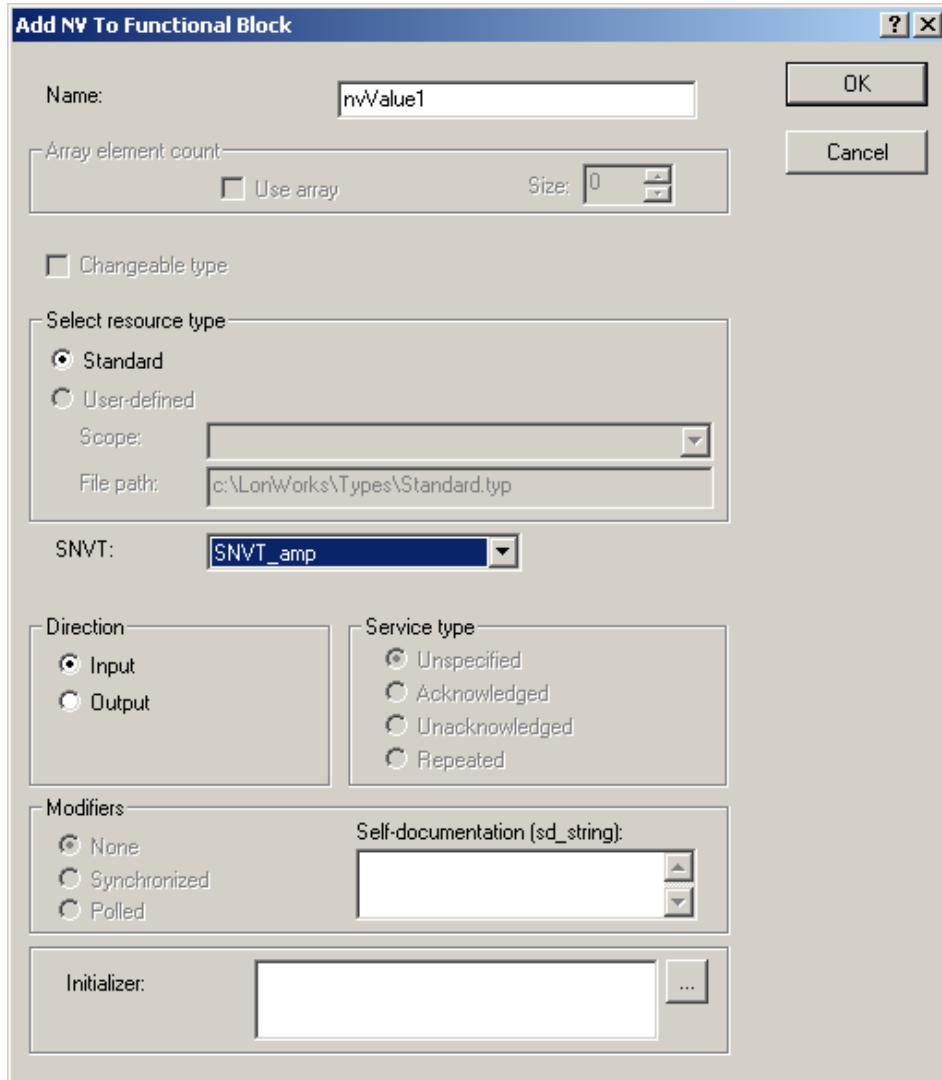
## Adding Implementation-specific Network Variables

You can add a network variable member that is not defined by the functional profile. This is called an *implementation-specific network variable*. Implementation-specific network variables should be avoided as part of a device's interoperable interface since they are not documented by a functional profile. An alternate method to add members to a functional profile is to define a new functional profile that inherits from an existing profile. This method results in a new functional profile that you can easily reuse in new devices and is described in *Creating or Modifying a Functional Profile* in the *Editing Resource Files* chapter.

In order to add an implementation-specific network variable to a functional block, the scope of the network variable type must be less than or equal to the scope of the functional profile upon which the functional block is based. For example, a UNVT could not be added to a standard function profile, but a SNVT may be added to a user functional profile.

To add an implementation-specific network variable to a functional block or device, follow these steps:

- To add the network variable to a functional block, right-click the **Implementation-specific NVs** folder contained by the functional block in the Code Wizard **Interface** pane and select **Add NV** from the shortcut menu. Alternatively, you can drag a network variable from a **Network Variables** folder in the Resource pane to the functional block's **Implementation-specific NVs** folder to add the network variable to the device. A dialog appears that is similar to the following figure:



- Specify the type, direction, and other information about the new network variable using this dialog. This dialog contains the following information:

**Name**

The name of the network variable to be created as it will appear to the network integrator. This name must be unique within the device. This name may contain only letters, numerals, and the underscore character, and can be a maximum of 16 characters.

**Array Element Count**

This option cannot be set for an implementation-specific network variable on a

functional block that is not part of a functional block array. It can only be enabled for network variables created directly on the device interface, as described in *Adding Network Variables to the Device Interface*. If the functional block containing the network variable is part of an functional block array, the size of the network variable array is set to the size of the functional block array.

### **Changeable Type**

Set this option to allow the network integrator to change the type of this network variable. This allows you to create a network variable that can send or receive different kinds of information, depending on how the device is used. For example, a generic PID controller device can be implemented using `SNVT_temp_f` as the initial type, but allow this type to be changed by a network integrator to a range of other types to allow the PID controller to control, light, pressure, or other types. You must implement additional code in your application to support changeable types. See *Using a Changeable-Type Network Variable*, later in this chapter, and the *Neuron C Programmer's Guide*, for more information.

This option will only be enabled if the program ID indicates a changeable interface (e.g. by setting the **Has Changeable Interface** checkbox in the *Standard Program ID Calculator*).

### **Select Resource Types**

Specifies the resource file containing the network variable type definition for this network variable. To use a standard network variable type (SNVT) for this network variable, select the **Standard** option. To use a user network variable type (UNVT), select the **User-defined** option. In order to use a UNVT, you must first add the resource file containing the UNVT to the resource catalog, as described in *Editing Resource Files*.

### **Scope**

The scope of the resource file containing the UNVT definition. You can only change the scope if you select **User-defined** under **Select Resource Types**. Each resource file has a scope and a program ID template. In order to use a UNVT from a user resource file, you must select the scope of that resource file, and the program ID template of the resource file must match the program ID of the application to the degree specified by the

scope (e.g. if the scope selector is 4, the manufacturer and device class components of the two program IDs must match). Possible scope selector values for user-defined resources are the following:

**3 – Manufacturer**

**4 – Manufacturer and Device Class**

**5 – Manufacturer, Device Class, and Usage**

**6 – Manufacturer, Device Class, Usage, and Device Model**

Scope selectors 0, 1, and 2 are reserved for use by the LONMARK association

Devices that use scope selector 6 should not use automatic program ID management (see *Device Template Wizard: Program ID* in the *Creating and Using Device Templates* chapter for more information).

**File Path**

The path of the resource file containing the selected SNVT or UNVT definition.

**SNVT/UNVT**

The label and contents of this field will change depending on the option selected in **Select Resource Types**. If you select **Standard**, this field will be labeled **SNVT** and the list displayed by clicking the arrow will contain all the SNVTs. If you select **User-defined**, this field will be labeled **UNVT** and the list displayed by clicking the arrow will contain all the **UNVTs** in resource files of the scope specified in **Scope** that match the program ID template to the degree specified by the scope. The added network variable's type must have a scope that is equal to or lower than the scope of the functional profile upon which the functional block is based.

If the network variable has a changeable type, this value represents the default type.

**Direction**

Specifies whether this network variable used to receive or send information. Network variables can only be connected to network variables of the opposite direction.

**Service Type**

If **Direction** is set to **Output**, this value specifies the default LONWORKS messaging service that will be used to send network variable updates for this output network variable. The network integrator can override

the value selected here. The following options are available:

*Unspecified* — There is no specified service type. A network tool will determine what service type is used.

*Acknowledged* — This service causes the device that receives the network variable message to send a response message confirming that it was received. If the response is not received, the message will be sent again, up to a configurable maximum number of retries. This service should only be used for critical data sent to a single device or small fan-out connection where missing an update will cause an application failure. This is the default service.

*Unacknowledged* — This service causes the network variable message to be sent once with no verification of it being received. This service should be used for non-critical updates that can tolerate occasional loss such as a sensor value with a heartbeat output.

*Unacknowledged Repeat* — This service type — typically called the *Repeated* service — causes network variable messages to be sent a number of times configurable by the network integrator. There is no verification of the network variable being received. If the sending device is not programmed to do something about network variable update failures, this service provides reliability equal to the Acknowledged service while consuming less network bandwidth for fan-out connections to more devices than the retry count. This is because each device must send at least one acknowledgment for acknowledged connections. For example, an acknowledged message to 10 devices will generate a minimum of 11 packets (one update, and 10 acknowledgements). A repeated message with a repeat count of four to the same ten devices will generate exactly four packets and will provide the same probability of delivery as an Acknowledged message with four retries..

**Note:** The Repeated service prevents backlog estimation from functioning correctly. since Repeated messages do not indicate the number of repeated packets that will be generated. Backlog estimation is a traffic-



prediction mechanism built into the LonTalk protocol. Repeated messaging in lieu of Acknowledged messaging can significantly reduce the number of acknowledgement packets, and typically provides better system performance even with the decrease in backlog estimation accuracy.

## **Modifiers**

Indicates if the network variable is synchronous or polled. These options are only available on output network variables. If **Polled** is specified for the network variable in the functional profile, this option will be set to **Polled** and cannot be modified. Select one of the following options:

*None* — The new network variable is neither synchronous nor polled.

*Synchronized* — The new network variable is a synchronous network variable. Normally, when a device application updates an output network variable multiple times before the device has a chance to send all the values to the network, the device sends only the most recent value. Similarly, when a device receives several updates from an input network variable before the device application can process them, the device application processes only the most recent update. The Synchronized option causes the device to queue and send all output network variable updates, and causes the device to queue and process all input network variable updates. The size of the input and output queues is limited to the size of the application buffer queues on the device, so you may need to allocate additional buffer space on the device if this option is selected.


*Polled* — This option is only available for output network variables. If this option is set, this network variable will not automatically send its value to the network when the value changes. It will only send the current value to a device when the device requests the update via a poll request.

## **Self-documentation**

Optional additional text to be appended to the self-documentation string for this network variable. Network variable members of functional blocks use a standard self-documentation format that is detailed in the *LONMARK Application Layer Interoperability Guidelines*. The Neuron C Compiler

automatically generates all required self-documentation information. This field can be used to provide additional notes, which can be accessed from a network tool. The total length of the self-documentation string can be up to 1024 characters, including the text that is automatically generated by the Neuron C Compiler.

### **Initializer**

An optional initializer value for the network variable. This is the value that will be set when the device is reset. If this network variable is a structure, union, float, signed 32-bit, or enum type, click  to open the Edit Initializer dialog to get more information about the network variable type (see *Editing the Initializer for Network Variables and Configuration Properties* for more information).

### **Advanced**

Click this button to open the Advanced Network Variable Properties dialog. This dialog allows you to view and set **advanced network variable properties**. Press F1 while this dialog is open for more information on the properties presented there.

3. Set your desired options, then click **OK** to add the network variable to the **Implementation-specific NVs** or **Network Variables** folder.

## **Adding Implementation-specific Configuration Properties**

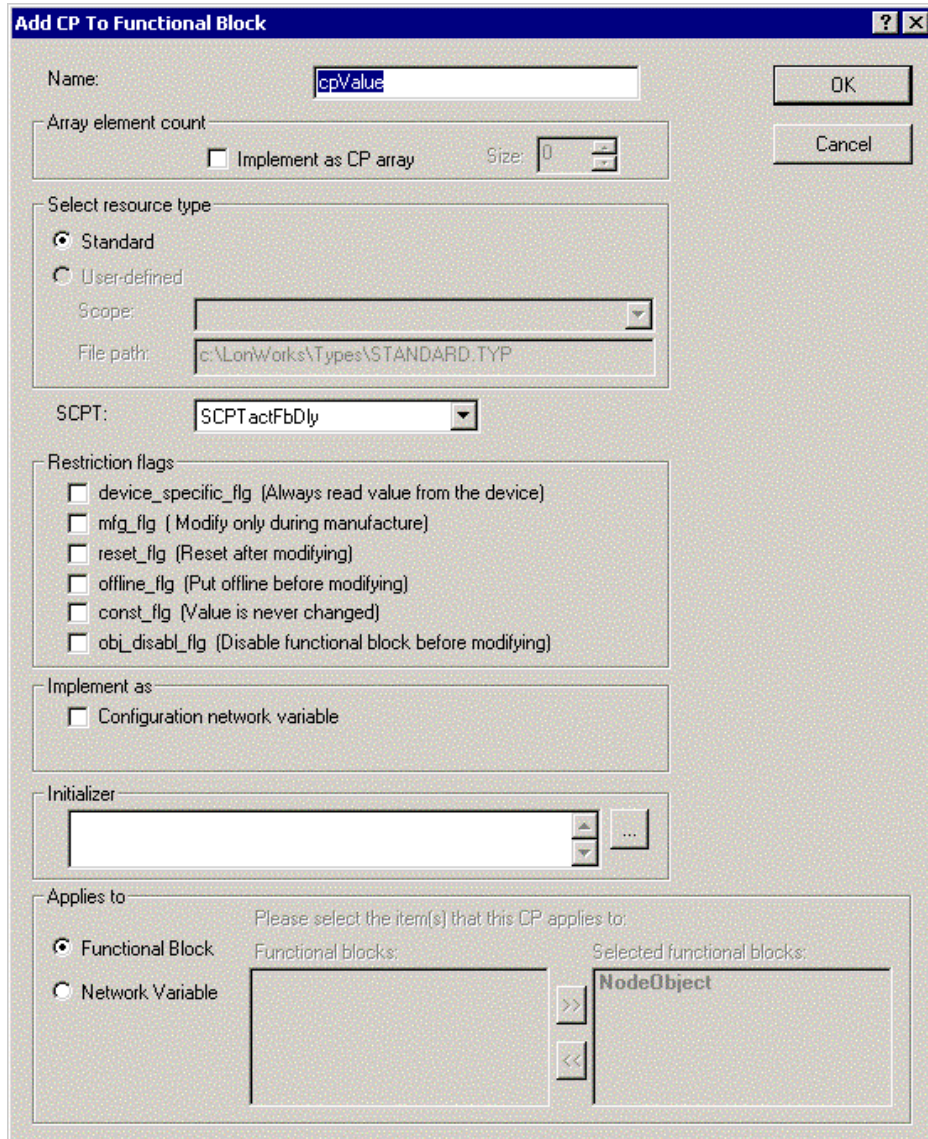
You can add a configuration property to a functional block that is not defined by the functional profile. This is called an *implementation-specific configuration property*. Implementation-specific configuration properties should be avoided as part of a device's interoperable interface since they are not documented by a functional profile. An alternate method to add members to a functional profile is to define a new functional profile that inherits from an existing standard profile. This method results in a new functional profile that you can easily reuse in new devices and is described in *Creating or Modifying a functional profile* in the *Editing Resource Files* chapter.

In order to add an implementation-specific configuration property to a functional block, the scope of the configuration property type must be less than or equal to the scope of the functional profile upon which the functional block is based. For example, you cannot add a UCPT to a standard functional profile (except by inheritance), but you may add a SCPT to a user functional profile.

To add implementation-specific configuration properties to a functional block, follow these steps:

1. Right-click the **Implementation-specific CPs** folder contained by a functional block in the **Interface** pane and then click **Add CP** on the shortcut menu. Alternatively, you can drag configuration properties from a **Configuration Properties** folder in the Resource pane to the

functional block's **Implementation-specific CPs** folder. A dialog appears that is similar to the following figure:



- Specify the type and other information about the new configuration property using this dialog. Enter the following information:

**Name**

The name of the configuration property as it will be used in the Neuron C program. This name must be unique within the device. This name may contain only letters, numerals, and the underscore character, and can be a maximum of 16 characters.

Once a configuration property has been added, you can rename it by right-clicking it and then clicking **Rename** on the shortcut menu.

**Array Element Count**

You may implement an implementation-specific configuration property as a single

element, or as an array. An array has a minimum size of 2 elements, and a maximum size of 65,500 bytes. The array size is limited by the amount of available persistent, modifyable, memory in the device. A linker error will occur if the specified array size exceeds the device's resources.

See the Neuron C documentation for more details about implementing configuration property arrays.

### Select Resource Types

The type of configuration property to implement. To use a standard configuration property type (SCPT) for this configuration property, select **Standard**. To use a user configuration property type (UCPT), first create or add the UCPT as described in *Editing Resource Files*, and then select **User-defined**.

### Scope

The scope of the resource file containing the UCPT definition. You can only change the scope if you select **User-defined**, specifies the scope of the resource file containing the UCPT definition. Each resource file has a scope and a program ID template. To select a UCPT, first select the scope of the resource file containing the UCPT definition. The program ID template of the resource file must match the program ID of the application to the degree specified by the scope (e.g. if the scope selector is 4, the manufacturer and device class components of the two program IDs must match). Possible scope selector values for user-defined resources are the following:

**3 – Manufacturer**

**4 – Manufacturer and Device Class**

**5 – Manufacturer, Device Class, and Usage**

**6 – Manufacturer, Device Class, Usage, and Device Model**

Scope selectors 0, 1, and 2 are reserved for use by the LONMARK association

Devices that use scope selector 6 should not use automatic program ID management (see *Device Template Wizard: Program ID* in the *Creating and Using Device Templates* chapter for more information).

### FilePath

The path of the resource file containing the

selected SCPT or UCPT definition.

## SCPT/UCPT

The label and contents of this field will change depending on the option selected in **Select Resource Types**. If you select **Standard**, this field will be labeled **SCPT** and the list displayed by clicking the arrow will contain all the SCPTs. If you select **User-defined**, this field will be labeled **UCPT** and the list displayed by clicking the arrow will contain all the UCPTs in resource files of the scope specified in **Scope** that match the program ID template to the degree specified by the scope. The added configuration property's type must have a scope that is numerically equal to or lower than the scope of the functional profile upon which the functional block is based.

## Restriction Flags

These options allow you to set configuration property flags. Network tools are responsible for checking these flags and handling configuration properties appropriately. You can set any combination of the following options:

These options allow you to set configuration property flags. Network tools are responsible for checking these flags and handling configuration properties appropriately.

See the *Neuron C Programmer's Guide* and *Neuron C Reference Guide* for more information about configuration property restriction flags.

## Implement As


Specifies implementation options for the configuration property. Specifies the following options:

*Configuration Network Variable* – If this option is set, the configuration property is implemented as a configuration network variable. This takes up network variable resources on the device but the configuration property can be read, written, and bound just like a network variable. If this option is cleared, the configuration property is implemented within a configuration file. Configuration properties implemented within configuration files do not use up any network variable resources, but they cannot be bound to output network variables on other devices.

*Static CP* – This option is available only if the configuration property is in a functional block array. Setting this option creates a single

configuration property to be shared among all functional blocks in the array. Modifying the value of the configuration property for any functional block on the array will modify all of them - there is actually only one variable allocated. If this option is cleared, a separate configuration property will be created for each functional block in the array.

### Initializer

An optional initializer value for the configuration property. This is the value that will be set when the device is reset. If this configuration property is a structure, union, float, signed 32-bit, or enum type, click  to open the Edit Initializer dialog to get more information about the network variable type (see *Editing the Initializer for Network Variables and Configuration Properties* for more information).

### Applies To

A configuration property can apply to an entire functional block or to any of the network variables on the functional block (in any of the **Mandatory NVs**, **Optional NVs**, or **Implementation-specific NVs** folders). You can share a configuration property between multiple functional blocks or network variables as described in *Sharing a Configuration Property*, later in this chapter.

3. Click **OK** to add the configuration property to the **Implementation-specific CPs** or **Configuration Properties** folder.

## Adding Device Network Variables



You can add a network variable to the device interface outside of any functional blocks. This allows you to create a non-LONMARK portion of your device interface for proprietary or legacy information. To add a network variable directly to the device interface, right-click the **Network Variables** folder contained by the device and select **Add NV** from the shortcut menu. Alternately, you can drag a network variable from a resource file set's **Network Variables** folder to the device's **Network Variables** folder. The Add Network Variable dialog opens. See *Adding Implementation-specific Network Variables*, earlier in this chapter, for more information on this dialog.

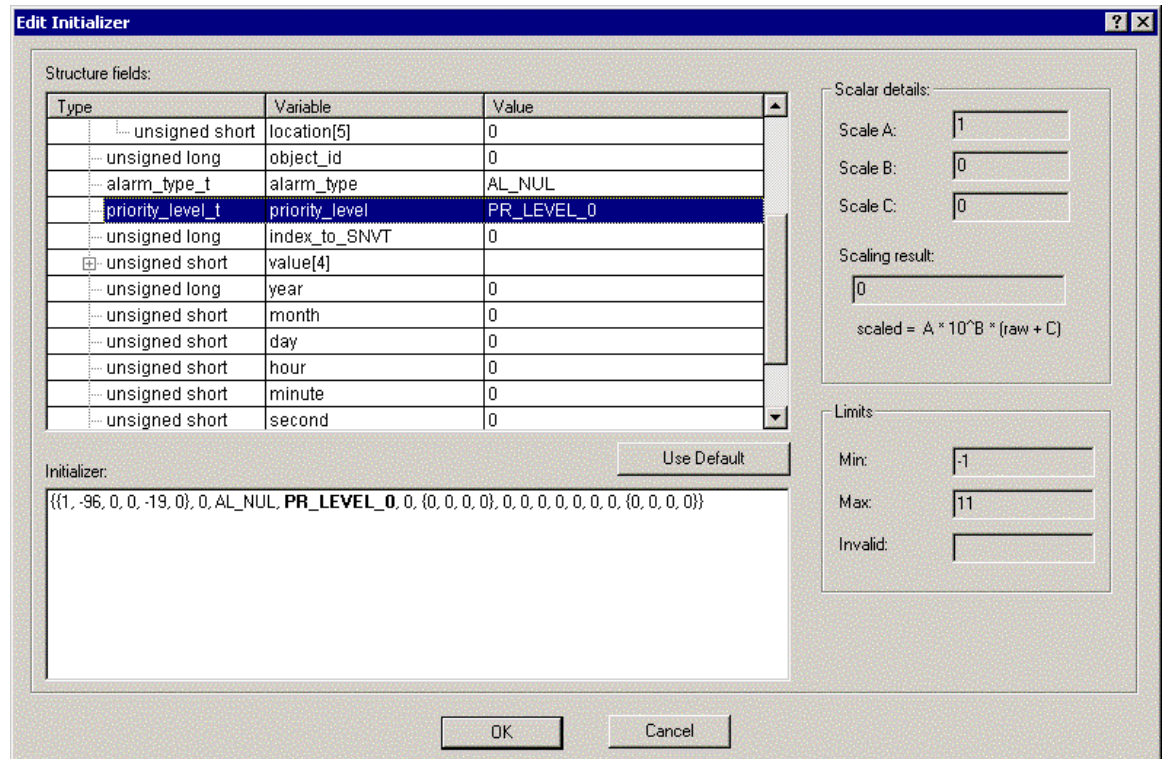
## Adding Device Configuration Properties

You can add a configuration property to the device interface outside of any functional blocks. This allows you to create a non-LONMARK portion of your device interface for proprietary or legacy information. To add a configuration property directly to the device interface, right-click the **Configuration Properties** folder contained by the device and select **Add CP**. Alternately, you can drag a configuration property from a resource file set's **Configuration Properties** folder to the device's **Configuration**

**Properties** folder. The Add Configuration Property dialog opens. See *Adding Implementation-specific Configuration Properties*, earlier in this chapter, for more information.

## Editing the Initializer for Network Variables and Configuration Properties

You can set the initial value for any network variable or configuration property. For network variables, this is the value that will be set when the device is reset. For configuration properties, the value will be stored in the LNS database, and will be set the first time the device is reset after the application has been loaded into the device. Each network variable and configuration property dialog has an Initializer field and a  button. You can enter the initial value in the field or click the button to get more information about the network variable or configuration property type. If you click the  button, a dialog appears that is similar to the following figure:



When this dialog opens, **Initializer** and **Structure Fields** will be empty if no initializer has been previously created for this network variable or configuration property. If either **Initializer** or **Structure Field** are updated, the other one will be automatically updated, and any fields not set explicitly will be set to the default values. Click **Use Default** to set **Initializer** and **Structure Fields** to default values. If it is available, the default value that is defined in the appropriate resource file will be used, otherwise it will be zero or the minimum value allowed if zero is out of range.

You can set or change the value of a field by either selecting the field and single-clicking **Value** in **Structure Fields** or directly editing the value in **Initializer**. You can add comments or arrange the initializer value to be displayed in a separate line by editing **Initializer** directly. If you select a

field in **Structure Fields**, the corresponding value in **Initializer** will be highlighted and vice versa.

You can expand the structure of the network variable or configuration property using the + and – buttons in the **Type** column. You can resize the columns under **Structure Fields** so you can see all relevant data.

To see the **Scalar Details** and **Limits** for a field on the right side of the dialog, select the field in **Structure Fields**.

If you change the value in **Structure Fields** and the selected value is an enumeration, a list of available enumeration values will be displayed.

Click **Use Default** to reset the entire structure to its default values. If it is available, the default value that is defined in the appropriate resource file will be used, otherwise it will be zero or the minimum value allowed if zero is out of range.

You can use a preprocessor **#define** statement to define a string that can be used as a structure initializer. For example:

```
#define myInit {FS_XFER_OK, 0, 0 {{{0},  
{0x00, 0x00, 0x00, 0x000}, 0}}}
```

If you do this, you can enter **myInit** directly in the **Initializer** field when creating the network variable or configuration property. The **Edit Initializer** dialog will not be aware of the **#define** statement, and it will not verify any data you enter.

Click **OK** to save the changes. The value specified in the **Initializer** will be transferred to the **Initializer** field of the network variable or configuration property dialog. The Code Wizard will not do any validation on the value that you specify in **Initializer**.

If you define an initializer for an array of configuration properties or network variables and the array size changes, you must redefine the initializer to reflect the new size. A warning will appear when you change the array size reminding you to do this.

## Sharing a Configuration Property

You can share a configuration property among multiple functional blocks or network variables. Sharing configuration properties can simplify device configuration by reducing the number of configuration properties that must be set by an integrator, and can also reduce the memory required for the device application. You can use one of the following two methods to share a configuration property: *static configuration property sharing* and *global configuration property sharing*.

### Using Static Configuration Property Sharing

You can share a single configuration property or configuration property array among all elements of a functional block or network variable array. This is called *static configuration property sharing*. To statically share a configuration property using the NodeBuilder Code Wizard, follow these steps:

1. Create a new configuration property in a functional block array or open an existing one.



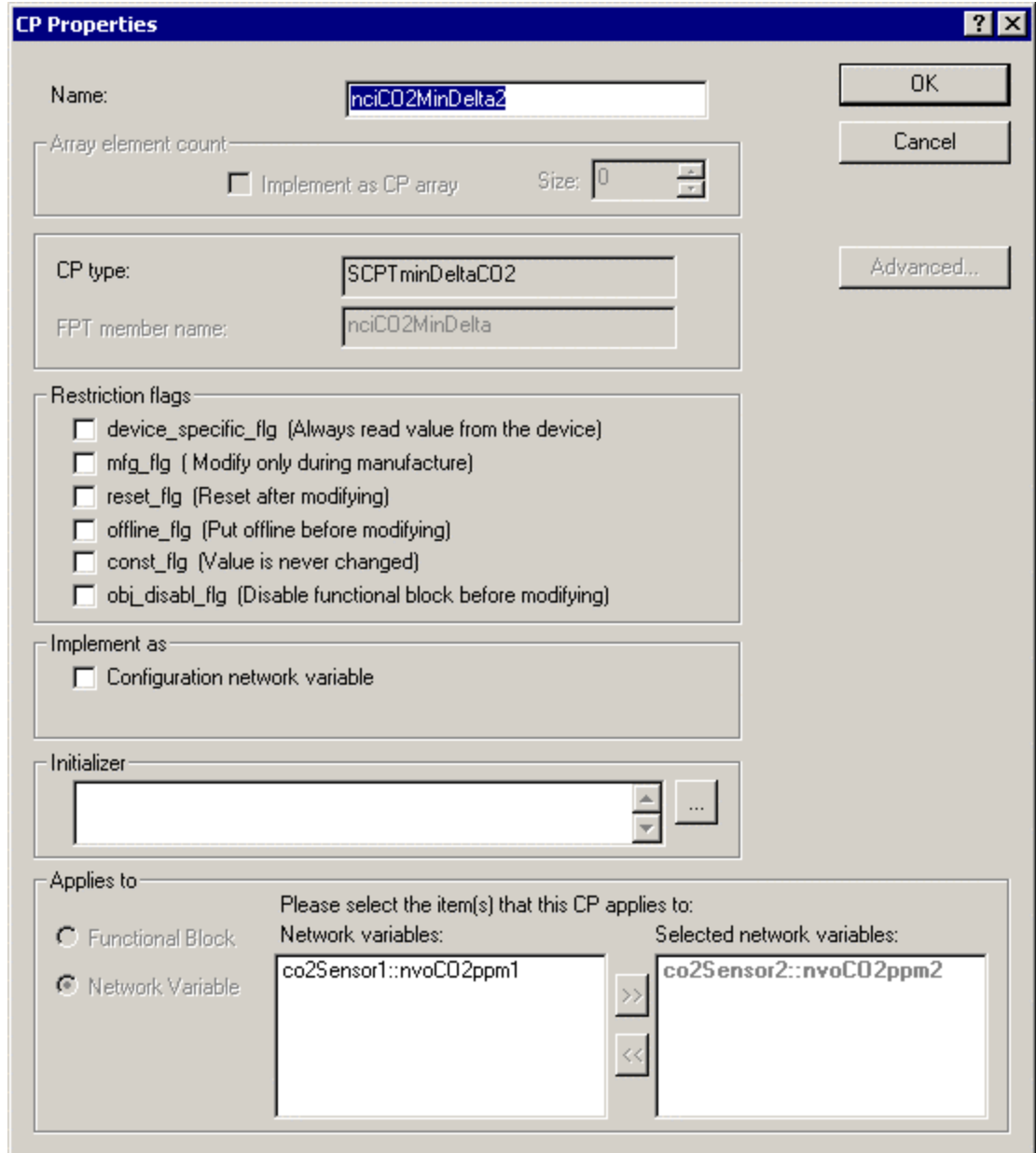
2. Set the **Static CP** checkbox. This checkbox is only available if the functional block is an array.

### *Using Global Configuration Property Sharing*


You can share a single configuration property or configuration property array among multiple network variables or functional blocks. This is called *global configuration property sharing*.

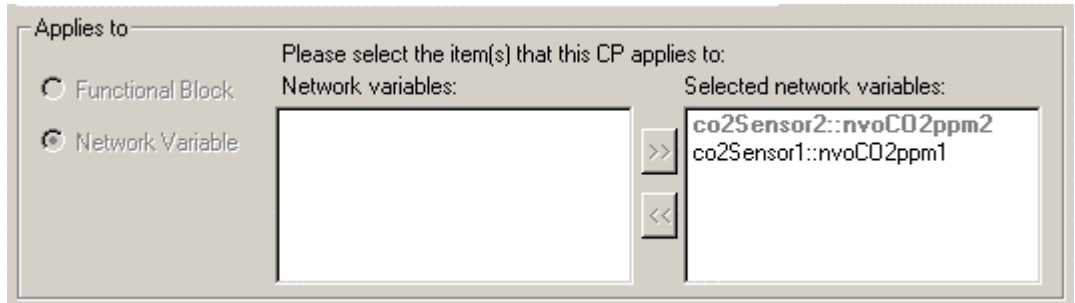
To globally share a configuration property using the NodeBuilder Code Wizard, follow these steps:

1. Create a new Configuration Property or open an existing one (if you are sharing optional configuration properties, you will need to implement the optional configuration property, then double-click it or right-click it and then click **Properties** on the shortcut menu). For example, if you implement two co2Sensor functional blocks (co2Sensor1 and co2Sensor2) and select the SCPTminDeltaC02 mandatory configuration property for the first one, the following dialog opens:



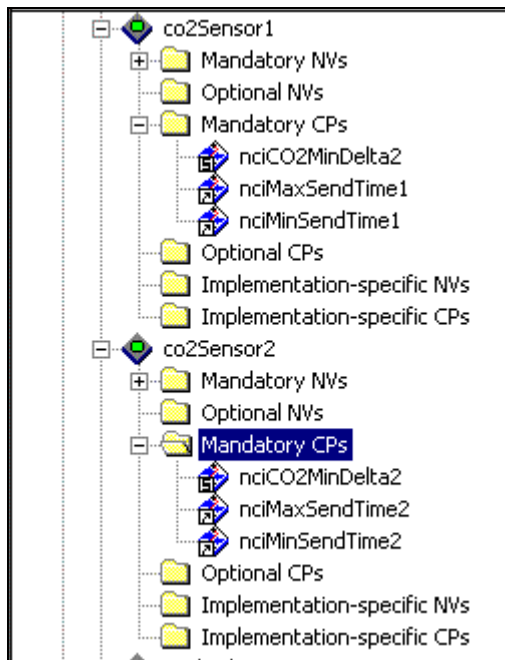
If another configuration property with the same type, array size, **Implement As** setting, and **Applies To** setting (i.e. **Functional Block** or **Network Variable**) exists on the device, it will appear in **Applies To** under **Network Variables** or **Functional Blocks**. In the example above, the device contains one other **SCPTminDeltaCO2** configuration property that applies to a network variable.

- To share the configuration property, select one or more of the items from the **Network Variables** or **Functional Blocks** list and click . You will see a warning that you are about to create a global configuration property. Click **OK** to move the selected functional block or network variable to the **Selected Network Variables** or **Selected Functional Blocks** list. In the example above, the bottom of the dialog will now appear as shown in the following figure:



The network variable that the originally selected configuration property applied to will appear in bold gray text to indicate that it is the root configuration property and cannot be removed from the list of shared configuration properties. You can remove any of the other configuration properties..

3. Click **OK**. If you have shared two mandatory or optional configuration properties or if you have shared two implementation-specific configuration properties from a different functional block, they will appear in the **Interface** pane of the Code Wizard with the same configuration property name in their respective folders, as shown in the following figure:




If you share an implementation-specific configuration property with an optional or mandatory configuration property within the same functional block, the implementation-specific configuration property will be removed from the Program Interface pane.

In this example, the configuration property being created will apply to both the **nvoCO2ppm1** network variable on the **co2Sensor1** functional block and the **nvoCO2ppm2** network variable on the **co2Sensor2** functional block; the Neuron C expression `co2Sensor1::nvoCO2ppm1::cpValue == co2Sensor2::nvoCO2ppm2::cpValue` will always be true, since these two expressions are two different names for the same configuration property.

**Note:** When using the LonMaker Browser or an LNS Plug-in to update a shared configuration property, the display may not automatically update the other shared configuration properties. You can force the Browser to update its display by opening the **Browse** menu and selecting **Refresh All**. Refreshing an LNS Plug-in display is plug-in specific.

## *Removing a Shared Configuration Property*

To remove a configuration property from a set of shared configuration properties, follow these steps:

1. Right-click one of the configuration properties that will remain and then click **Properties** on the short-cut menu. The **CP Properties** dialog appears.
2. Select the configuration property to be removed, and then click . The configuration property originally selected in the Code Wizard will be shown in bold gray text and cannot be removed through this dialog. To remove the configuration property that is shown in bold gray, close the **CP Properties** dialog and re-open it for one of the configuration properties that is to remain.

Each configuration property that is removed from the *Selected Network Variables* or *Selected Functional Blocks* list will be implemented as a separate, non-shared, configuration property.

## *Configuration Property Sharing Rules*

The following rules apply to configuration property sharing:

1. A configuration property can only be shared between multiple network variables, or between multiple functional blocks, but not between a combination of network variables and functional blocks at the same time.
2. All configuration property types can be shared.
3. A configuration property that applies to the entire device cannot be shared.
4. Multiple functional blocks or network variables can share a configuration property. A shared configuration property can apply to multiple singular functional blocks or network variables, a functional block or network variable array, a number of functional block or network variable arrays, or any combination thereof.
5. A configuration property that is shared among the members of a functional block or network variable array must always be shared among all members of that array.
6. A configuration property can be shared between network variables on different functional blocks.
7. A configuration property that inherits its type from a network variable can only be shared between network variables that are all of the same type. Therefore, all changeable type network variables that share an inheriting configuration property must also share an instantiation of `SCPTnvType`, so that the set of changeable network variables will always have the same, single, type and so that type changes occur at the same

time.

8. Two (or more) mandatory functional profile template configuration properties can be implemented using a single, shared, configuration property provided the shared configuration property meets the requirements of all individually listed FPT members (e.g. same type, same array size, etc.).
9. A single configuration property that inherits its type from a network variable cannot be shared simultaneously by both changeable and non-changeable network variables.
10. Configuration property arrays that are implemented as arrays of configuration network variables and that apply to a functional block array or to a network variable array must be shared.

## Using a Changeable-Type Network Variable

You can use changeable-type network variables to implement generic functional blocks that work with different types of inputs and outputs. For example, you can create a general-purpose device that can be used with a variety of sensors or actuators, and then create a functional block that allows the integrator to select the network variable type depending on the physical sensor or actuator attached to the device. Another example is a scheduler that can control a variety of device types by allowing the integrator to change the type of the output of the scheduler. The Code Wizard generates code that contains a framework for supporting changeable network variable types.

Starting with version 14, the Neuron firmware implements a new method for changing the size of a network variable. This new method uses an *NV length override system image extension* that is managed by the application.

Whenever the firmware needs the length of a network variable, it calls the NV length override system image extension to get it. This new method provides more reliable updates to network variable sizes, since the old method could cause a device to go applicationless if a power failure occurred in the middle of a network variable size update. The new system image extension method only works with version 14 firmware, or newer. Since the LTM-10A platform does not use version 14 firmware, you can develop an application that supports both methods, enabling only one of the methods for each type of platform.

For an in depth discussion of changeable-type network variables and the NV length override system image extension, as well as a commented source code example, illustrating all aspects of creating an application that uses changeable type network variables, see the *Neuron C Programmer's Guide*.

To implement a changeable-type network variable using the NodeBuilder tool and the Code Wizard, follow these steps (see the *Neuron C Programmer's Guide* for a more detailed discussion of each step):

1. Assure that the Program ID selected when the device template was created has the **Changeable Interface** option set. You can view this option by right clicking the device template in the NodeBuilder Project Manager and clicking **Settings** on the shortcut menu, and then clicking the **Calculator** button to open the Program ID Calculator.
2. Start the Code Wizard.

3. Create a new network variable or open an existing one.
4. Set the **Changeable Type** checkbox.
5. Click **OK** to close the network variable dialog and then click **Generate And Close** to close the Code Wizard and generate code.
6. Complete the implementation of the **nv\_length\_override** function. Code Wizard provides an empty implementation of this function in the device template's main source file. This function should return the length of any changeable type network variable in the device.

The Code Wizard in NodeBuilder 3.1 uses the **#pragma unknown\_system\_image\_extension\_isa\_warning** directive to generate Neuron C source code that will compile, for example, on both a LTM-10A target (debug platform) and a TP/FT-10F Flash Control Module (release platform). Code Wizard enables this directive in the device template's main header file. If you use a combination of Code Wizard-managed and handcrafted, you may want to override Code Wizard's preferences in that regard by editing the relevant portion of the main header file.

You should only use the older **nv\_len** method to support debugging of an application containing changeable type network variables on platforms that do not support the system image extension. For production release, the more robust system image extension method should be used, and both methods should not coexist in a production device.

7. Define the behavior of the application when a request to change the network variable type is received. The application must validate that the requested type change is supported. If it is not, it must reject the request (either by setting **invalid\_request** or by setting an application-specific error and putting the device offline) and set the network variable type back to the last valid type. If the type change is valid, it must implement the type and size change.

The Code Wizard does not provide framework code for this task, but a commented source code example is provided in the *Neuron C Programmer's Guide*.

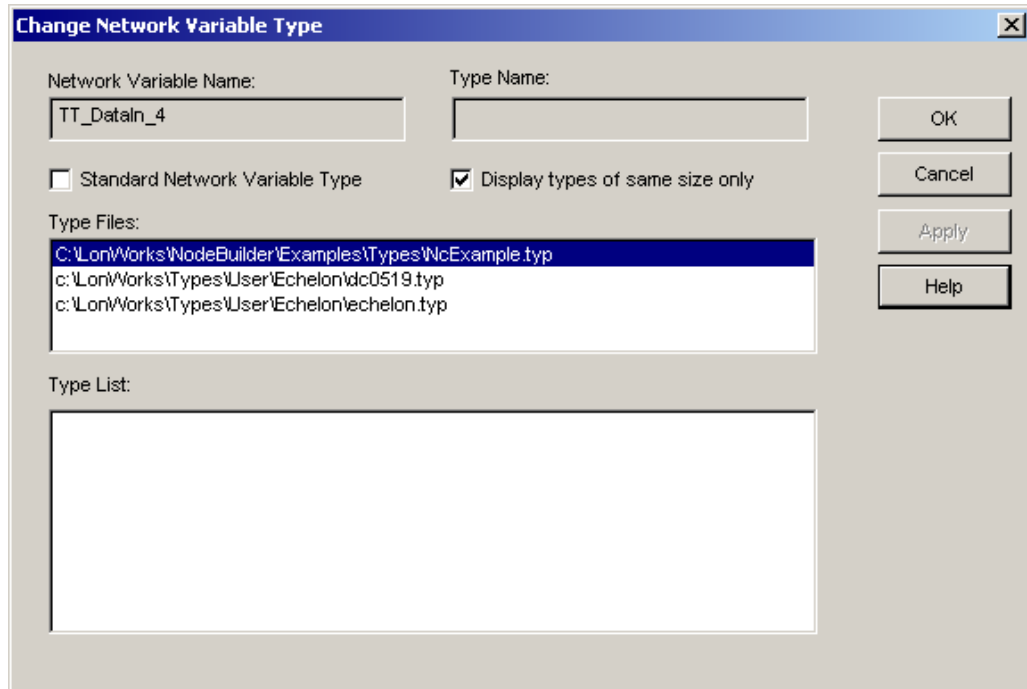
8. Define how the functional block behaves when sending or receiving values on changeable type network variables. For each valid type, the functional block must perform any necessary conversion before operating on the value.

The Code Wizard does not provide framework code for this task, but a commented source code example is provided in the *Neuron C Programmer's Guide*.

### ***Changing a Network Variable Type Using an LNS Tool***

See the LNS Plug-in Programmer's Guide for more information on changing the type of a changeable type network variable from an LNS plug-in.

To change the type of a changeable type network variable using the LonMaker Browser, right-click the network variable and then click **Change Type** on the shortcut menu. This menu item is only available for network variables with changeable types. The Change Network Variable Type dialog appears, as shown in the following figure:



Set **Standard Network Variable Type** to select the type from the SNVTs detailed in the *SNVT and SCPT Master List*. Clear this option to select a type from the available user-defined types.

Set **Display Types of Same Size Only** to view only network variable types of the same size as the selected network variable, or clear this option to view all available network variable types. Although it is possible to change a network variable to a type that is a different length than the original network variable, not all LonWorks devices support such a change, and some devices support such a change only through related LNS device plug-in software. Check the documentation for the functional block to verify that it supports changes to types with different lengths before changing the length. Failure to follow the correct procedures could result in unpredictable behavior. Requesting an Unsupported Type

The protocol to support changeable type network variables, described in the *Neuron C Programmer's Guide*, details how and when a device should respond to a request for a type change when the device does not support the desired type.

This validation and notification requires the device to be attached, commissioned, and accessible. When designing and configuring a large LonWorks network, this may not be the case. In this situation, a type change will be requested during the planning phase, but the device will only be able to validate and reject a type change when the device is commissioned.

Generic configuration tools like the LonMaker Browser therefore cannot inform the user of an invalid type request. Specialized tools such as LNS device plug-in software should be used, when available, and the affected devices should be validated after the commissioning is complete. The LonMaker tool's Manage and Test functions can be used to confirm correct operation or to detect errors.

---

## Editing Properties in the Code Wizard

You can view and modify properties for the *device* as a whole or for any *functional block*, *network variable* or *configuration property* on the device. To modify the properties for one of these objects, double-click it, or right-click it and select **Properties** from the short-cut menu. Any changes that you make to the properties are reflected the next time you generate code from the Code Wizard as described in *Generating Code with the Code Wizard*.

---

## Generating Code with the Code Wizard

You can generate Neuron C source code that implements a device interface that you specify. The device interface specifies the functional blocks, network variables, and configuration properties to be implemented by the device. You can also use the Code Wizard to modify code previously generated by the Code Wizard.

To generate new or modified Neuron C source code, follow these steps:

1. Start the NodeBuilder Code Wizard by creating a new device template or from the NodeBuilder Project Manager as described in *Starting the Code Wizard*, earlier in this chapter.
2. Define or modify the device interface as described in *Defining the Device Interface* earlier in this chapter.
3. Click the **Generate and Close** button to generate or modify the Neuron C source code that implements the device interface. The Code Wizard generates the Neuron C source code. If any read-only files will be overwritten, a confirmation dialog opens.

See *Files Created by the Code Wizard* for a description of the files generated by the Code Wizard. See *Code Generated by the Code Wizard* for a description of the features supported by the Code Wizard-generated code. See *Modifying Code Generated by the Code Wizard* for information about the code generated by the Code Wizard and how that code should be modified. Also see *Neuron C Version 2 Features Not Supported by the Code Wizard*.

The Code Wizard will perform limited validation on the device template interface. It will check the following:

- The device template must have a Node Object functional block with an index of 0.
- Each functional block, network variable, and configuration property name must not be longer than 16 characters and must be alphanumeric
- The Node Object functional block's mandatory `nvoStatus` network variable must have the synchronized option set. See *Viewing Network Variable Properties in the Code Wizard*, earlier in this chapter, for more information.
- The **Changeable Type** option must not be set for any network variables if the program ID doesn't have the **Changeable Type** option set (see *Using the Standard Program ID Calculator*).
- A member name must be defined for each implementation-specific network variable.
- All configuration property types, network variable types, and functional profiles must have defined resources when code is being generated.



- All network variables must have a specific type. Some functional profiles contain network variables with no pre-determined type (referred to as SNVT\_XXX); The Code Wizard forces a specific and valid type to be assigned to these network variables.

---

## Files Created by the Code Wizard

The Code Wizard generates the following files into the source file folder (see *New Device Template Wizard: New Device Template* in the *Creating and Using Device Template Wizard*).

**Main Source File:**                   The main source file for the Neuron C application. All other files generated by the Code Wizard are included in this one using `#include` statements.

<Device Template  
Name>.nc

This file also contains the declaration of globally shared CP families, global **when** tasks, such as the **when (reset)** task, and the **get\_nv\_length\_override()** system extension function.

**Main Header File:**  
<Device Template  
Name>.h

Header information and function declarations for the main source file. Defines a number of constants that are used in the application code.

This file also contains most global preferences and compiler directives. .

**Functional Block Source  
Files:** <Functional Block  
Name>.nc

Contains a Neuron C code framework for each network variable and configuration property defined in the functional block. A functional block source file is generated for every functional block defined in the device interface.

Once the Code Wizard has generated the framework, you will implement the functional blocks' algorithms. Most of this implementation will normally be done in the functional block source files.

**Functional Block Header  
Files:** <Functional Block  
Name>.h

Contains header information and function declarations for the corresponding functional block source file.

Once the Code Wizard has generated the framework, you will implement the functional blocks' algorithms. Most of the related function prototypes, type definitions or I/O object and timer declarations will be done in the functional block header files.

The following files are common files generated by the Code Wizard. You can move them to a shared folder so that they can be shared with other NodeBuilder projects or device templates. To share files that have been moved to a common share folder, you will need to specify the path in **Include Search Path** (see *Creating a NodeBuilder Project: Specify Project Default Settings* in the *Creating and Opening NodeBuilder Projects* chapter and *New*

*Device Template Wizard: New Device Template in the Creating and Using Device Templates Chapter.*

Each time you generate code using Code Wizard, it searches for each of the common files on the **Include Search Path**. If it exists, Code Wizard uses the one in the common folder; otherwise it creates the file in the source files folder.

**common.nc**

Common functional block functions such as enable, disable, and override. Several utility functions contained in this file may remain unused. You can remove these functions to regain code and data space on the device (you should not do this if you are sharing this file between multiple projects). You typically do not have to modify the functions in this file.

When compiling, the Neuron C compiler will issue several messages referring to symbols being defined in common.nc, but not used. The Code Wizard-generated framework does not include the **#pragma ignore\_notused** compiler directive to suppress these warnings. Review each of these efficiency warnings once your device's implementation nears completion; you may find many of these functions useful during your development but you may also find some of them are not required for your application.

**common.h**

Header information and function declarations for common.nc.

**filesys.nc**

Functions used to facilitate transfer of configuration properties implemented as configuration files. This file is generated only if the **File Transfer Protocol** configuration property transfer mode is selected. (see *Opening the Code Wizard*, earlier in this chapter).

The filesys.nc file implements a simple file system that targets the Neuron memory space. You may chose to implement a different filesystem to support, for example, storage of configuration property values in off-chip I2C EEPROM devices. See comments in filesys.nc and filesys.h files for more information.

**filesys.h**

Header information and function declarations for configuration properties implemented as configuration files.

This file contains type definitions and macros that are used to define the File Directory structure, and will be included irrespective of the chosen configuration property transfer mode.

<b>filexfer.nc</b>	Functions used to implement FTP transfer of configuration properties. This file is generated only if the <b>File Transfer Protocol</b> configuration property transfer mode is selected.  This file contains the implementation of the actual file transfer protocol, and does not normally require changes.
<b>filexfer.h</b>	Header information and function declarations for <code>filexfer.nc</code> . This file is generated only if the <b>File Transfer Protocol</b> configuration property transfer mode is selected, and does not normally require modification
<b>NodeObject.nc</b>	Implementation of the Node Object functional block. This file contains the core of the Code Wizard-generated framework; it contains code that responds to updates to the <code>nviRequest</code> network variable and routes requests to the individual functional block director functions. You will not normally edit this file, but you can change the feature set and functionality of the Code Wizard generated framework from here.
<b>NodeObject.h</b>	Header for Node Object declarations.

All files generated by the Code Wizard are added to the device template's **Source Files** folder in the Project pane of the NodeBuilder Project Manager.

---

## *Code Generated by the Code Wizard*

Once you have generated code with the Code Wizard, you can build the device without error immediately (see *Building a NodeBuilder Project*).

The Code Wizard generates a number of utility functions for your convenience. If you do not use these functions, they will unnecessarily increase the size of your application image and you will see warnings about unused functions when you compile your application. You can eliminate these warnings, reduce application image size, and decrease your application download time by commenting out or removing any unused functions. You can build your application and look for NCC#310 warnings to identify your unused functions.

Code Wizard-generated code contains the following features:

- *File directory structure.* If the application has any configuration properties implemented within configuration files, the Code Wizard creates code to reference the configuration property template and value files, for both direct memory read/write and FTP configuration property access. The two access methods cannot co-exist, and are thus treated as mutually exclusive.
- *FTP Server.* If FTP is used to access configuration property template and value files, and at least one configuration property within a configuration file has been implemented, the Code Wizard code also provides an implementation of the FTP server. The default implementation of the FTP server supports read and write access with both sequential and random

access. The FTP server supports configuration files with sizes up to the amount of available space on the Neuron Chip or Smart Transceiver. This space is equal to 64 KB minus any address space used for code, data, I/O, or other firmware features. The default implementation of the FTP server does not support local initiation or dynamic creation of files, but partially implements the framework for these operations. See `filexfer.h` for more details, if you wish to enhance or reduce the feature set provided by the default FTP server implementation.

- *File System.* If FTP is used to access configuration property template and value files and the default FTP server is used, a minimalistic file access system is implemented in `filesys.nc`. The FTP server uses this file system to read or write data to the local files. The default file system is used to access data located in the Neuron memory space, but can be changed or extended to support other data storage techniques. For example, a modified file system could support storage of CP value files in an offchip EEPROM that is accessed using a Neuron Touch I/O model.
- *Default directors for each functional block or functional block array.* The source code for each functional block or functional block or array contains a default implementation of a director function. When refining the Code Wizard-generated code for a less generic, but more resource-efficient implementation, you can change this scheme and implement a smaller number of more powerful director functions, shared between multiple functional blocks or functional block arrays. In some cases, the default framework results in multiple, almost identical, director functions. Code space can be saved in cases where a single director can serve multiple functional blocks or functional block arrays.
- *when() task for each functional block or functional block array.* This task provides notification upon incoming network variable updates for the functional block or array of functional blocks. If the functional block has no input network variables, no when task is generated. If the functional block implements multiple input network variables, a single **when** task is created to serve all associated input network variables. Implementing one **when** task to process updates to multiple input network variables may require extra code to determine which network variable received an update. On the other hand, the Neuron firmware's scheduler turnaround time benefits from a smaller number of **when** tasks. You will sometimes need to optimize the implementation to meet your application's needs in that regard, balancing runtime performance, device responsiveness, and memory requirements.
- *Code to handle device and functional block requests on the Node Object.* The Code Wizard generates code for the `nviRequest` and `nvoStatus` network variables on the Node Object functional block. This implementation routes requests to the functional block or blocks concerned by calling the relevant director functions, and provides a default implementation that allows for the following requests to be honored: `RQ_REPORT_MASK`, `RQ_UPDATE_STATUS`, `RQ_DISABLED`, and `RQ_ENABLE`. Handling for other requests is partially implemented but must be completed by the developer. See the comments in the director functions generated by the Code Wizard and the *LONMARK Application-Layer Interoperability Guidelines* for more information.
- *A set of utility functions to manage functional block state.* Code Wizard generates `common.h` and `common.nc`, which contain a number of utility functions. See these files for more information on these functions. You may

choose not to use some or all of these functions. Unused functions in the `common.nc` file will cause a compiler warning (NCC#310, Symbol defined but not used). You should remove unused functions to reduce code space and application download time.

- *Default implementation for system event handling.* These are events such as `when reset, online, offline`. These system events also get routed to the different director functions, allowing each functional block director function to respond to each event in an appropriate way.

---

## Modifying Code Generated by the Code Wizard

The code produced by the Code Wizard provides a framework for your application. It implements the device interface that you define, but does not include your application-specific code. Edit the source files generated by the Code Wizard to implement the functionality required by your device. See the *Editing Neuron C Source Code* chapter for more information on editing code using the NodeBuilder project manager.

Each file generated by the Code Wizard has sections that look like this:

```
//{{NodeBuilder Code Wizard Start
//{{NodeBuilder Code Wizard End
```

Neuron C code inside these comments will be modified by the Code Wizard every time you generate code for the device template. You can edit the Neuron C code outside these tags, and your changes will not be overwritten when you run the Code Wizard again.

Inside this Code Wizard generated code, there are commands used by the Code Wizard that look like this:

```
//<Command>
```

These commands indicate where the Code Wizard puts certain pieces of generated code. For example, a `//<Include Headers>` statement precedes the Code Wizard generated list of include statements. If you want to remove the Code Wizard statements from Code Wizard control, you can move them outside the Code Wizard generated code. Once you have moved code outside of the Code Wizard section indicated by the Code Wizard start

```
(//{{NodeBuilder Code Wizard Start) and end (//{{NodeBuilder
Code Wizard End) statements you can manage the code on your own
and the Code Wizard will not modify it.
```

For example, the Code Wizard automatically calculates the number of alias table entries based on the number of network variables that are currently implemented, with a maximum of 62. Due to limited resources on the target device, you may want to set a different different value than provided by the Code Wizard. The following example is taken from the NodeBuilder example (see Appendix A). When the Code Wizard generates code, `NcExample.h` contains the following statements:

```
//{{NodeBuilder Code Wizard Start
:
:
// <CP Access>
```

```

#define _USE_DIRECT_CPARAMS_ACCESS
//
//<Alias Entries>
#pragma num_alias_table_entries 15
//
//<Node Description String>
//
//<External Name>
#define USE_EXTERNAL_NAME
//
:
:
:
:
//}}NodeBuilder Code Wizard End

```

You can override the Code Wizard generated code by moving the `//<Alias Entries>` command and statements out of the Code Wizard section, as shown below:

```

//{{NodeBuilder Code Wizard Start
:
:
// <CP Access>
#define _USE_DIRECT_CPARAMS_ACCESS
//
//<Node Description String>
//
//<External Name>
#define USE_EXTERNAL_NAME
//
:
:
:
:
//}}NodeBuilder Code Wizard End

//<Alias Entries>
#pragma num_alias_table_entries 15
//

```

Once you take the `//<Alias Entries>` command out of the Code Wizard managed section of the code, the Code Wizard will no longer create the **num\_alias\_table\_entries** directive. If you add more network variables, you may need to change the number of alias table entries manually.

The following list suggests ways to modify the Code Wizard-generated code. The list is not comprehensive and the modifications you make will vary depending on the purpose of your device:

- Add I/O and timer declarations. Initialize global I/O, timers, and variables in the `when (reset)` task (in the main Neuron C file). Initialize functional block-specific I/O, timers, and variables in the relevant functional block's director function. Upon completion of the initialization for each functional block, release the `lockout` bit for each functional block and thus allow it to

operate. Here's an example, taken from the NcExample project (DigitalOutput.nc):

```
.....
else if ((TFblock_command)iCommand == FBC_WHEN_RESET)
// raised by when (reset) task
{
    // initialize output lines:
    GizmoSetLed(0,
        DigitalOutput[0]::cpDigitalDefault.state);
    GizmoSetLed(1,
        DigitalOutput[1]::cpDigitalDefault.state);
    setLockedOutBit(uFblockIndex, FALSE);
}
.....
```

- Add when() tasks to respond to I/O and timer-related events as needed. Add these event handlers to the main source file if they concern global I/O or timers, and add them to the individual functional block's source file if they concern functional block-specific items.
- The status of each functional block needs to be stored in non-volatile memory (one byte per functional block). By default, the Code Wizard locates this in onchip eeprom. If you require additional onchip eeprom and your hardware template has offchip eeprom available, you can relocate the PersistentFblockStatus array in far offchip eeprom by changing the statement

```
#define FBLOCK_PERSISTENT_STATUS_STORAGE onchip eeprom
```

to

```
#define FBLOCK_PERSISTENT_STATUS_STORAGE far offchip eeprom
```

The **FBLOCK\_PERSISTENT\_STATUS\_STORAGE** macro is defined in the main header file.

- For functional blocks that implement input network variables, add code to the Code Wizard-generated `<FbName>processNv()` function (with `<FbName>` being the name of the fblock or fblock array in question). The Code Wizard-generated framework will only call this function if the object is in an appropriate state. You can use the built-in Neuron C variables such as `nv_in_addr`, `nv_in_index`, or `nv_array_index`, to obtain more details about the update from within the `<FbName>processNv()` function.

For example, the Code Wizard-generated code implements one input network variable event handler for each functional block or functional block array in order to achieve short scheduler-cycles, and thus a responsive device. See step 11 in *NodeBuilder Example Task 8: Real Time Keeper* for an example and explanation of how to modify the Code Wizard-generated code in order to create a `when(nv_update_occurs)` event handler for each input network variable.

- When adding code that deals with application messages and contains unqualified `when(msg_arrives)` event handlers on a device that implements FTP with the sender-capability enabled, the sender routine already implements such an event handler. Since there can only be one such event handler and since this handler must be the last when-task in compilation order, you must share your code with the code provided in

filexfer.nc. The FTP server implementation uses the `#pragma scheduler_reset` directive if the sender-capability is enabled (this is the default). See `filexfer.nc` and `filexfer.h` for more details.

- The Code Wizard automatically calculates a recommended number of alias table entries. This number is a function of the total number of network variables being declared, and will always be larger than 0. Remove `{{NodeBuilder Code Wizard Start <Alias Entries> and }}NodeBuilder Code Wizard End` from the following Code Wizard generated lines to disable that feature and to provide your own number of alias table entries:

```
//{{NodeBuilder Code Wizard Start <Alias Entries>
#pragma num_alias_table_entries XXX
//}}NodeBuilder Code Wizard End
```

A number of alias table entries **must** be given. Set XXX to zero to disable support for LONWORKS network variable aliases. It is highly recommended to support aliases, resources allowing, due to the benefits when incorporating the device into a network.

- Consider adding the `#pragma enable_sd_name` directive. This directive causes network variables names to be stored in the device's self-documentation information. This is useful if the device gets installed when the device interface file is not available. See the *Neuron C Reference Guide* for more information about this directive.
- The code generated by the Code Wizard does not include code to support LonMark self test routines. See the *LONMARK Application-Layer Interoperability Guidelines* for more information.

---

## Neuron C Version 2 Features Not Supported by the Code Wizard

The following overview summarizes features of the Neuron C Version 2.1 language that are not used or not supported by the Code Wizard. See the *Neuron C Programmer's Guide* and *Neuron C Reference Guide* for more information about Neuron C Version 2.1.

### Message Tags

- The generation of declarations or the use of message tags is not supported with the exception of the automatically generated FTP server implementation that contains an `fx_explicit_tag` message tag. See also the unqualified `when()` clauses, described below.

### I/O models

- The Code Wizard does not generate code that declares or uses I/O objects.

### Network Variables

- *NV arrays*. The Code Wizard only generates declarations for an array of network variables if the network variable applies to an array of functional blocks. The sizes of the two arrays will be the same (i.e. one network



variable per functional block). The Code Wizard does not support declaring an array of network variables and distributing the elements of this array amongst multiple functional blocks or arrays of functional blocks.

- *polled network variable modifier*. The Code Wizard supports the `polled` network variable modifier for output network variables. It does not support the `polled` network variable modifier for input network variables. The latter is used by ShortStack Neuron C model files. This feature is not required or supported for development of Neuron Chip or Smart Transceiver hosted devices.

## Configuration Properties

- *Network variable class config*. The Code Wizard does not support the `config` network variable class, as this keyword is obsolete. The Code Wizard supports configuration network variables using the new Neuron C `cp` network variable class instead.
- *cp\_family re-use*. Code Wizard-generated code declares one `cp_family` of a given type for each instance of a configuration property, unless the configuration property it references is an array of functional blocks. That is, if the complete device requires two (or more) configuration properties of type `T`, the Code Wizard will generate two (or more) `cp_families` of type `T` even though only one may be required.

To put it another way, a CodeWizard-generated `cp_family` will always have a single member unless the configuration properties applies to an array of functional blocks, in which case the size of that array equals the size of the `cp_family`.

- The Code Wizard does not support declaring a configuration network variable as an array and distributing the elements of this array amongst multiple functional blocks or functional block arrays.
- The Code Wizard does not support sharing a configuration property amongst the members of a network variable array that applies to the entire device (i.e. it is not part of a functional block or functional block array). This restriction applies to both `static` and `global` configuration property sharing scopes.
- *range\_mode\_string*. The Code Wizard does not support the Neuron C Version 2.1 `range_mode_string` option to support setting the LONMARK range modification for a configuration property.

## when() clauses

- *Unqualified when(msg\_arrives)*. The Code Wizard generates an unqualified `when(msg_arrives)` task as part of the pre-defined FTP server implementation (see `filexfer.nc`). This code is only generated if you choose the FTP configuration property access method.

If your application processes incoming application messages and includes the pre-defined FTP server, you must use the existing implementation and branch out from there into your own handler code. See *Modifying Code Generated by the Code Wizard*, earlier in this chapter, for information about removing part of the Code Wizard-generated code from the Code Wizard's control.

- *when(nv\_update\_occurs(nv1..nvx))*. For functional block or functional block arrays that contain input network variables, the Code

Wizard always generates a single `when()` task to handle incoming network variable updates, using the Neuron C Version 2 `when(nv_update_occurs(nv1..nvX))` construct.

Code for multiple `when` tasks per functional block or functional block array (assuming each functional block has more than one input network variable) is not generated.

This implies that all input network variables that belong to a given functional block or functional block array are to be declared in subsequent order; refer to the *Neuron C Programmer's Guide* for more details about the use of network variable range specifications as arguments to the `nv_update_occurs()` function.

The Code Wizard does not generate code to handle the arrival of updates to configuration network variables.

- `#pragma scheduler_reset`. The Code Wizard implementation of the FTP server requires the presence of `#pragma scheduler_reset`. This is automatically inserted as needed by the Code Wizard (see `filexfer.nc`).
- Note additional comments on the use of the FTP server above.

## LONMARK style

- *NodeObject*. The Code Wizard is limited to generation of devices that have a valid Node Object functional block, and it enforces this functional block to be the first functional block in the device's list of functional blocks. Node Object functional blocks are identified by their functional profile number being zero at present scope so you can use your own functional profile for the Node Object by creating a functional profile with a functional profile number of zero that inherits from the scope 0 functional profile.

## Miscellaneous

- *Director Functions*. The Code Wizard always creates one director function per functional block or functional block array. It does not support functional blocks without director functions, nor does it support the sharing of one director function amongst multiple functional blocks (with the exception of functional block arrays). Modifying the director functions to perform these tasks can lead to significant gains in the application's memory footprint and runtime performance, while causing the application to become less generic. You can optimize the default director scheme if needed.





# 7

## Editing Resource Files

This chapter describes how to view, create, and edit resource files using the NodeBuilder Resource Editor.

---

## Introduction to Resource Files

Resource files provide definitions of functional profiles, type definitions, enumerations, and formats that can be used by network tools such as the LonMaker tool. The type definitions include definitions for network variable types and configuration property types.

Resource files are grouped into *resource file sets*, where each set applies to a specified range of program IDs. The program ID range is determined by a *program ID template* in the file, and a *scope* value for the resource file set that specifies the fields of the program ID template that are used when matching the program ID template to the program ID of a device. The program ID template has an identical structure to the program ID of a device, except that the applicable fields may be restricted by the scope. The scope value can be seen as a filter, indicating the relevant parts of the program ID. The scope may be one of the following:

**0** – Standard

**3** – Manufacturer

**4** – Manufacturer and Device Class

**5** – Manufacturer, Device Class, and Device Subclass

**6** – Manufacturer, Device Class, Device Subclass, and Device Model

For a device to use a resource file set, the program ID of the device must match the program ID template of the resource file set to the degree specified by the scope. This allows each LONWORKS manufacturer to create resource files that are unique to their devices.

For example, consider a resource file set with a program ID template of 81:23:45:01:02:05:04:00 and manufacturer and device class scope (scope 4). Any device with the manufacturer ID fields of the program ID set to 1:23:45 and the device class ID fields set to 01:02 would be able to use types defined in this resource file set, whereas resources on devices of the same class but by a different manufacturer could not access this resource file set.

A resource file set may also reference information in any resource file set with a numerically lower scope provided the relevant fields of their program ID templates match. For example, a scope 4 resource file set can reference resources in a scope 3 resource file set, provided the manufacturer ID components of the resource file sets' program ID templates match.

Scopes 0 through 2 are reserved for standard resource definitions published by Echelon and distributed by the LONMARK association. Scope 0 applies to all devices, and scopes 1 and 2 are reserved for future use. Since scope 0 applies to all devices, there is a single scope 0 resource file set called the *standard resource file set*. A standard resource file set is included with the NodeBuilder tool, but periodic updates are available from the LONMARK association at [www.lonmark.org](http://www.lonmark.org).

You can define your own functional profiles, types, and formats in scope 3 through 6 resource files.

Most LNS tools, including the LonMaker tool assume a default scope of 3 for all user resources. If you use scope 4, 5, or 6 resource files, you must explicitly set the scope in LNS so that LNS uses the appropriate scope. There are two ways to do this. The first method is to develop a plug-in as described in Chapter 13, *Introduction to LNS Device Plug-ins*. Your plug-in will automatically set the appropriate scope values when it is registered on a user's computer. The second method is to modify the device shape's **FbModes** user cell as described in *Additional Device User Cells* in the *LonMaker User's Guide* and help file.

Each resource file set may contain definitions for the following resources:

<b>Network Variable Types</b>	Type information for network variables. This information includes the size, units, scaling factors, and type category (float, integer, signed, etc) for each type. Network variables can contain a single value or they can contain a structure or union containing multiple fields (for example, the <code>SNVT_date_cal</code> network variable contains 3 fields for the year, month, and day). Network variables can also contain enumerated values which allow the network variable to be set to one of a discrete number of values. Network variable types are defined in a resource file with a <code>.typ</code> extension.
<b>Configuration Property Types</b>	Type information for configuration properties. This information includes the size, units, scaling factors, and type category (float, integer, signed, etc) for each type. Like network variables, configuration properties can contain structures, unions, and enumerated values. Configuration property types are defined in a resource file with a <code>.typ</code> extension (this is the same file used for network variable types).
<b>Functional Profiles</b>	Functional profiles define a template for functional blocks. A functional block is a collection of network variables and configuration properties designed to perform a single function on a device. Each functional profile can define mandatory and optional configuration properties and network variables. When a functional block implements a functional profile, it must implement all mandatory network variables and configuration properties defined by the functional profile, and it may implement some, all, or none of the optional network variables and configuration properties. Functional profiles are defined in a resource file with a <code>.fpt</code> extension. Functional profiles are also called <i>functional profile templates</i> .
<b>Enumerations</b>	An enumeration type is a list of numerical values, each associated with a mnemonic name. If a network variable or configuration property type

contains an enumeration, the definitions of the enumerated values are maintained separately as an enumeration type. Enumeration types are defined in a resource file with a .typ extension (along with network variable and configuration property types), and may also be defined in a separate C header file (.h extension).

## Language Files

Network variable types, configuration property types, functional profiles, and enumeration types can all reference text information used to describe their name, units, and function. This text information is contained in separate *language files*. There is one language file for every language your resource file set supports. When a language file is translated, the references contained in the network variable types, configuration property types, and functional profiles still point to the appropriate strings. The file extension of each language file depends on the language, and is one of the following:

Czech	"csy"
Danish	"dan"
Dutch (Belgian)	"nlb"
Dutch (default)	"nld"
English (UK)	"eng"
English (US)	"enu"
Finnish	"fin"
French (Belgian)	"frb"
French (Canadian)	"frc"
French (default)	"fra"
French (Swiss)	"frs"
German (Austrian)	"dea"
German (default)	"deu"
German (Swiss)	"des"
Greek	"ell"
Hungarian	"hun"
Icelandic	"isl"
Italian (default)	"ita"
Italian (Swiss)	"its"
Norwegian (Bokmal)	"nor"
Polish	"plk"
Portuguese (Brazilian)	"ptb"
Portuguese (default)	"ptg"
Russian	"rus"
Slovak	"sky"
Spanish (default)	"esp"
Spanish (Mexican)	"esm"
Swedish	"sve"
Turkish	"trk"

Resource files may support localization using these languages, but cannot support localization



that requires Unicode (or any other multibyte) encoding.

## Formats

Each network variable and configuration property type must have at least one format defined. This format describes how the value will be displayed to or entered by network integrators and network operators. It is possible to define multiple formats for a network variable type or configuration property type. Different formats can provide the information in a different order (if the value is a structure or union) or provide a different scaling factor (for example, the `SNVT_temp_f` network variable type has three formats, one for Fahrenheit, one for differential Fahrenheit, and one for Celsius). Formats are defined in format files with a `.fmt` extension.

You will use the *NodeBuilder Resource Editor* to create, modify, and view resource files. The resource editor is a standalone application that you can start from within the NodeBuilder Project Manager, or that you can start independently from the NodeBuilder program folder.

Since there may be many resource file sets installed on a user's computer, a *resource catalog file* is used to identify all the directories containing resource file sets. These directories are called *resource folders*. The resource catalog is stored in a file with a `.cat` extension. By default, the resource catalog is contained in `C:\LonWorks\Types\ldrf.cat`, but the directory and filename may be changed. You can use the resource editor to add and remove resource folders in the resource catalog.

The *SNVT and SCPT Master List* defines the SNVTs, SCPTs, and standard enumeration types. You can view this list by opening the Windows **Start** menu, pointing to **Echelon LNS Utilities**, and then clicking **LNS Utilities and LONMARK Reference Help**. Periodic updates to the standard definitions are published by the LONMARK Association at [www.lonmark.org](http://www.lonmark.org). Standard profile definitions are also available at [ww.lonmark.org](http://ww.lonmark.org). You can also view the current standard resourceSNVT, SCPT, and standaqrdr enumeration type definitions, defined by the LONMARK Association, at [types.lonmark.org](http://types.lonmark.org).

You can use the Resource Report Generator, as detailed in the *Resource Report Generator User's Guide* to generate documentation of standard and user-defined resource file sets.

---

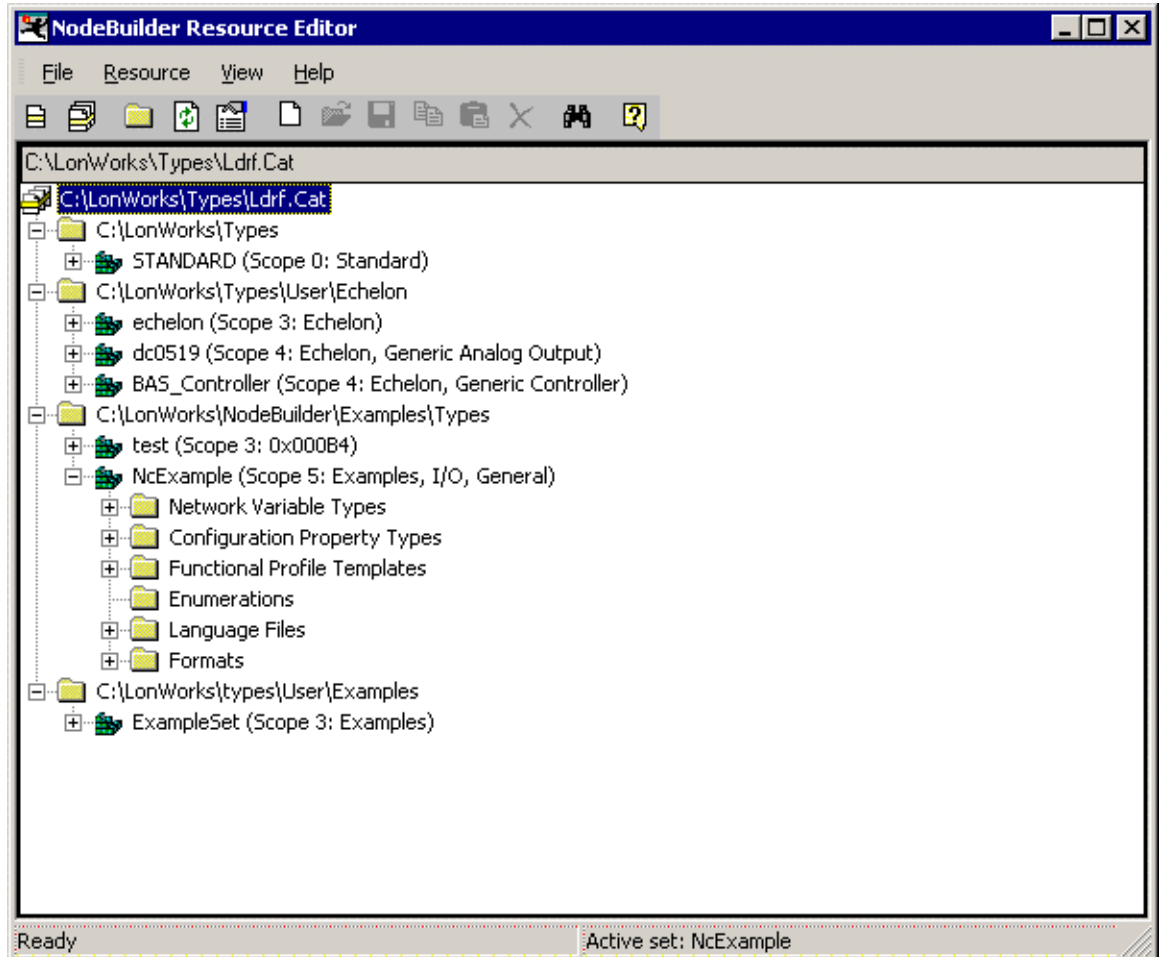
## Starting the Resource Editor

You will use the NodeBuilder Resource Editor to create, modify, and view resource files, and also to add user resource files to the resource catalog. You can use any of the following methods to start the resource editor:

- Click the Windows **Start** button, point to **Programs>Echelon NodeBuilder Software**, and then click **NodeBuilder Resource Editor**.
- If you are running the *NodeBuilder Project Manager*, open the **Tools** menu and then click **NodeBuilder Resource Editor**.

- If you are running the *NodeBuilder Code Wizard*, the Resource pane provides the full functionality of the resource editor.

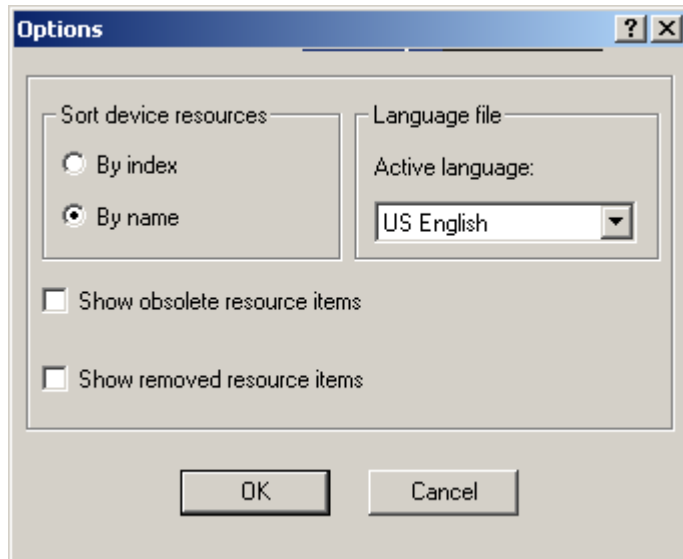
If you open the resource editor from the Start menu or from the NodeBuilder Project Manager, the NodeBuilder Resource Editor window appears, displaying a hierarchical view of your resource catalog, as shown in the following figure:



If you are using the Resource pane in the NodeBuilder Code Wizard, you will see the same hierarchical view of the resource catalog, but you will not see the resource editor menus and buttons. As described in this chapter, the full functionality of the resource editor is available from shortcut menus accessed from the resource catalog; so all resource editor functionality is available from within the code wizard Resource pane.

## Setting Resource Editor Options

You can set resource editor options that control how resources are displayed and specify the active language file. To view and modify resource editor options, click **Options** on the resource editor's View menu. The following dialog opens:



This dialog allows you to set the following options:

- |                                     |   |
|-------------------------------------|---|
| <b>Sort Device Resources</b>        | Displays resource items sorted by name or by index. If <b>By Name</b> is selected, resource items are sorted alphabetically. If <b>By Index</b> is selected, they are sorted by resource file index.  |
| <b>Active Language</b>              | Determines the language file that new strings will be placed in. You can create other language files and translate the strings as described in <i>Creating, Modifying, and Translating a Language File</i> . See <i>Creating and Editing a Language String</i> , later in this chapter, for more information. |
| <b>Show Obsolete Resource Items</b> | Displays resource items that have been marked obsolete. See <i>Removing and Obsoleting Resources</i> for more information.  |
| <b>Show Removed Resource Items</b>  | Displays resource items that have been removed. See <i>Removing and Obsoleting Resources</i> for more information.  |

---

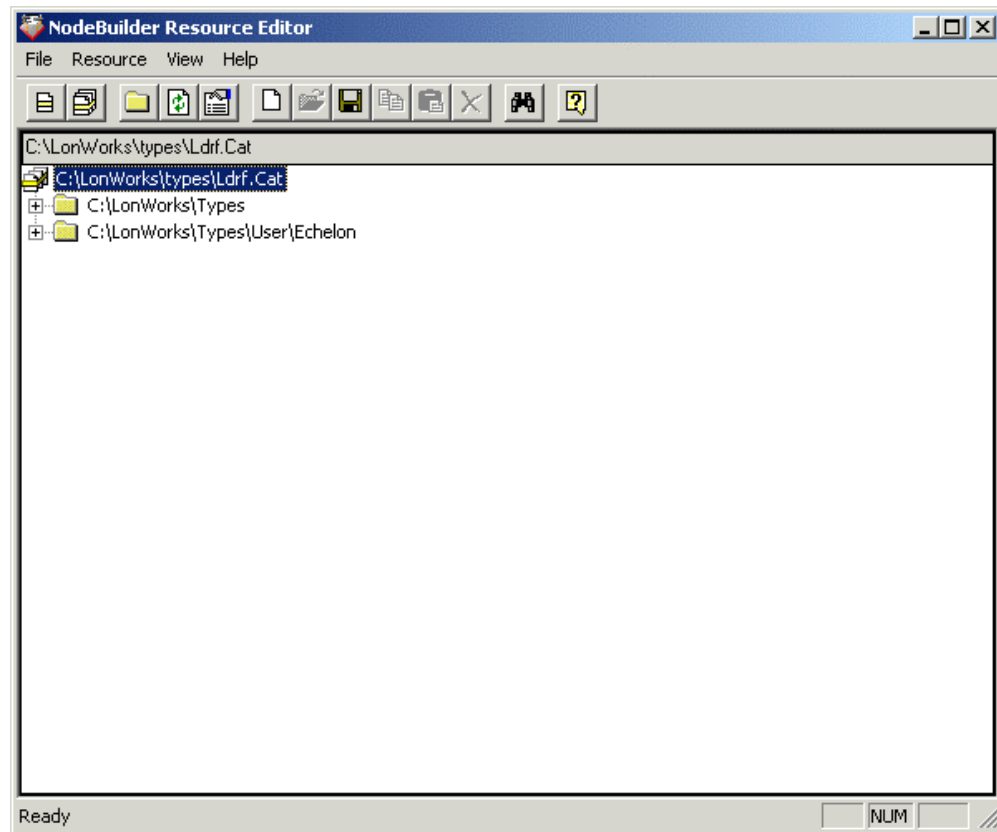
## Introduction to Resource Folders

A *resource folder* is a directory containing one or more resource file sets. You will typically create your resource file sets in a resource folder with your company name that is contained in the LONWORKS Types\User folder. For example, if you work for MyCo, you may create a resource folder in C:\LonWorks\Types\User\MyCo. If you anticipate creating many resource file sets, you may organize them in resource folders contained within your company resource folder.

A *resource catalog* is a file containing a list of resource folders. The resource catalog file is a file with a .cat extension. By default, the resource catalog file is contained in your LONWORKS Types folder and is named ldrf.cat (the full path is C:\LonWorks\Types\ldrf.cat by default, but you can change the folder and filename). Network tools use the resource catalog to find all the


resources that are defined on your computer. The resource editor also uses the resource catalog to display all of your available resources.

The resource editor displays a hierarchical view of your resource catalog, and all the resource folders and resource files that it contains. The resource catalog is the top of the hierarchy. The second level of the hierarchy below the resource catalog file contains entries for each of the *resource folders* contained in the resource catalog. In the following figure, the resource catalog file is C:\LonWorks\type\Ldrf.Cat and it contains two resource folders – c:\LonWorks\Types and c:\LonWorks\Types\User\Echelon. The C:\LonWorks\Types folder contains the standard resource file set, and the C:\LonWorks\Types\User\Echelon folder contains Echelon-specific resource file sets.

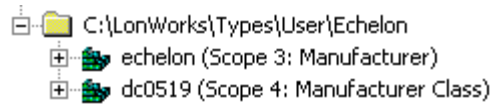


---

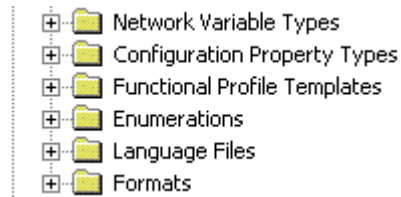
## Browsing the Resource Catalog

You can browse the resource catalog to view all the resource definitions contained within it. Click the  icon next to a resource folder to expand the hierarchy beneath it.

When you expand a resource folder you will see all resource file sets contained in that folder. Each folder can hold multiple resource file sets. Each resource file set in the folder is listed with its name and its scope. For example, the Echelon user resource folder may appear as shown in the following figure:



To view or modify the contents of a resource file set, expand it using the  $\square$  button. When you expand a resource file set you will see six folders containing resource file components, as shown in the following figure:




Each resource file set contains all of these folders, but some of them may be empty: Expand these folders to view or modify the following resources:

<b>Network Variable Types</b>	<p>Contains the network variable types defined by this resource file set.</p> <p>The network variable types defined here are used to implement network variables. They can also be referenced by other network variable or configuration property types, and may be referenced by functional profile templates.</p>
<b>Configuration Property Types</b>	<p>Contains the configuration property types defined by this resource file set.</p> <p>The configuration property types defined here are used to implement configuration properties. These types will typically be referenced by one or more functional profile templates.</p>
<b>Functional Profile Templates</b>	<p>Contains the functional profiles defined by this resource file set.</p>
<b>Enumerations</b>	<p>Contains the enumeration types defined by this resource file set. Network variable and configuration property types can use these enumeration types.</p>
<b>Language Files</b>	<p>Contains the language files defined by this resource file set. Each language file contains a set of strings translated into a specific language. Expand a language file to browse the individual strings in the language file.</p> <p>Language string resources are used to provide language-dependent details for all of the above resources.</p>
<b>Formats</b>	<p>Contains the formats defined by this resource file set. Each network variable type and configuration property type must have at least one format.</p>

---

## Adding a Resource Folder

You can add a new resource folder to the resource catalog. This makes all resource file sets contained within the folder available to network tools running on your computer, and also allows you to view and modify the resource files contained within the folder using the resource editor. To add a resource folder, follow these steps:

1. Right-click the resource catalog file at the top of the resource catalog, and then click **Add Folder** on the shortcut menu. You can also click the Add Folder () button on the toolbar, or open the **File** menu and then click **Add Folder**. An Add Folder window appears.
2. Browse to the folder to be added to the resource catalog, and then click **OK**. The folder should be located in a `Types\User\<Manufacturer Name>` folder within your LONWORKS folder (this is `C:\LonWorks\Types\User\<Manufacturer Name>` by default). The resource folder appears in the resource catalog.

---

## Removing a Resource Folder

You can remove a resource folder from the resource catalog. Removing a resource folder does not delete the resource files within the folder but will exclude the removed resources from the resource look-up mechanism.

To remove a resource folder, right-click the resource folder to be removed in the resource catalog, and then click **Remove** on the shortcut menu. The resource folder name is removed from the resource catalog.

Be careful not to remove resource folders that contain resource file sets that are referenced by your remaining resource file sets, or that contain resources that are used by any Neuron C application you may have in production or under development.

Use Windows Explorer to delete the related resource files, if you want to delete them.

---

## Moving a Resource Folder

You can move a resource folder to a different directory. To move a resource folder, follow these steps:

1. Use Windows Explorer to create the new folder.
2. Use Windows Explorer to move the resource file set to the new folder. A resource file set consists of a type file (".typ" extension), functional profile file (".fpt" extension), format file (".fmt" extension), and one or more language files with an extension dependent on the language (".enu" for English US strings). A resource file set may become unusable if one or more of these files are missing, so copy all of these files together if you copy or move the resource file set to another directory or computer.
3. Remove the old resource folder from the resource catalog as described in *Removing a Resource Folder*.

4. Add the new resource folder to the resource catalog as described in *Adding a Resource Folder*.

---

## Refreshing the Resource Catalog

The resource catalog may get out of synchronization with the resource files on your computer if you update resource files using a tool other than the resource editor, if you delete a resource file set using Windows Explorer, or if you copy new resource files into a resource folder using Windows Explorer. If this occurs, refresh the resource editor by right-clicking the resource catalog file at the top of the resource catalog and then clicking **Refresh Catalog** on the shortcut menu. Any empty resource folders are removed when you refresh the resource catalog.

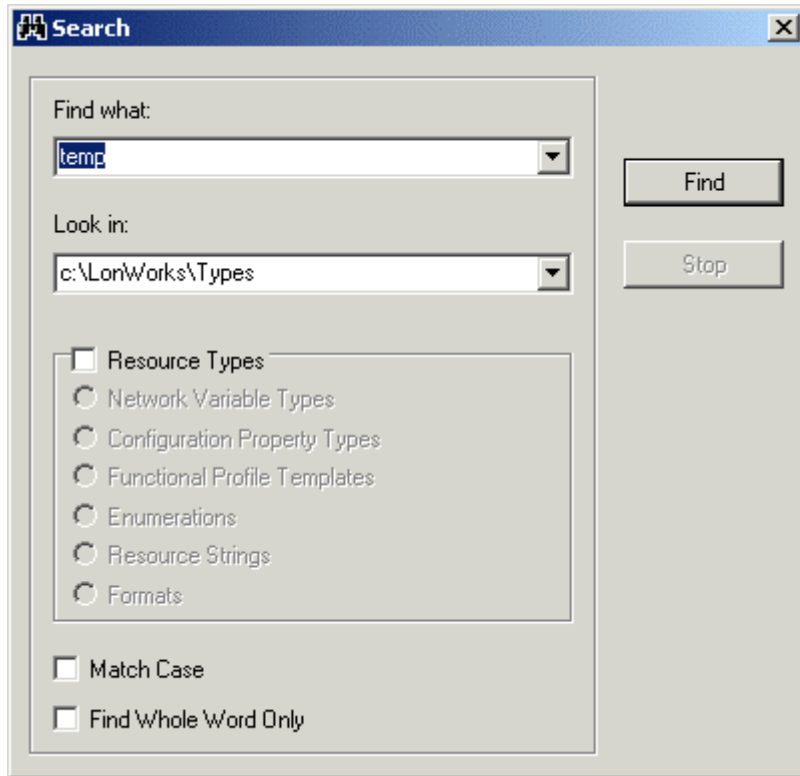
Any folders that are not present in your file system will automatically be removed from the resource catalog. This means that if you add a resource folder that is in a network or removable drive, and that folder becomes inaccessible, you will have to add the folder to the resource catalog again once the folder becomes available.

---

## Searching for a Resource

You can search for specific resources in the resource catalog. You can search for network variable types, configuration property types, functional profiles, enumeration types, language strings, or formats. You can search an individual resource file, a resource file set, a resource folder, or the entire resource catalog. To search for resources, follow these steps:

1. Right-click the folder to search and then click **Search** on the shortcut menu. The search will cover all resources within the folder that you select. The following dialog opens:



2. Enter the parameters for the search by filling in the following information:

**Find What**

The string to search for. You can enter all or part of a variable type name, configuration property type name, functional profile name, enumeration type name, language string, or format name. For example, you can enter “switch” to search for SNVT\_switch.

**Look in**

The resource catalog, resource folder, resource file set, or resource file to search. By default, this will contain the folder you selected to begin the search. Click the arrow to increase the scope of the search. A list of your original selection and all levels of the resource catalog above your selection appears.

**Resource Types**

The type of resource to search for. You can limit the search to **Network Variable Types**, **Configuration Property Types**, **Functional Profile Templates**, **Enumerations**, **Resource Strings**, or **Formats**. Clear **Resource types** to search all resource types.

**Match Case**

Searches for strings with the same case that you enter in **Find What**.

**Find Whole Word Only**

Searches for strings where the whole string matches what you enter in **Find What**. The search will not return results that contain the



string in **Find What** if it is part of a larger word.

3. Click **Find**. The first search result appears in the resource catalog. Close the Search window to operate on the result, or click **Find Next** to search for more results. To stop a search in progress, click **Stop**.

---

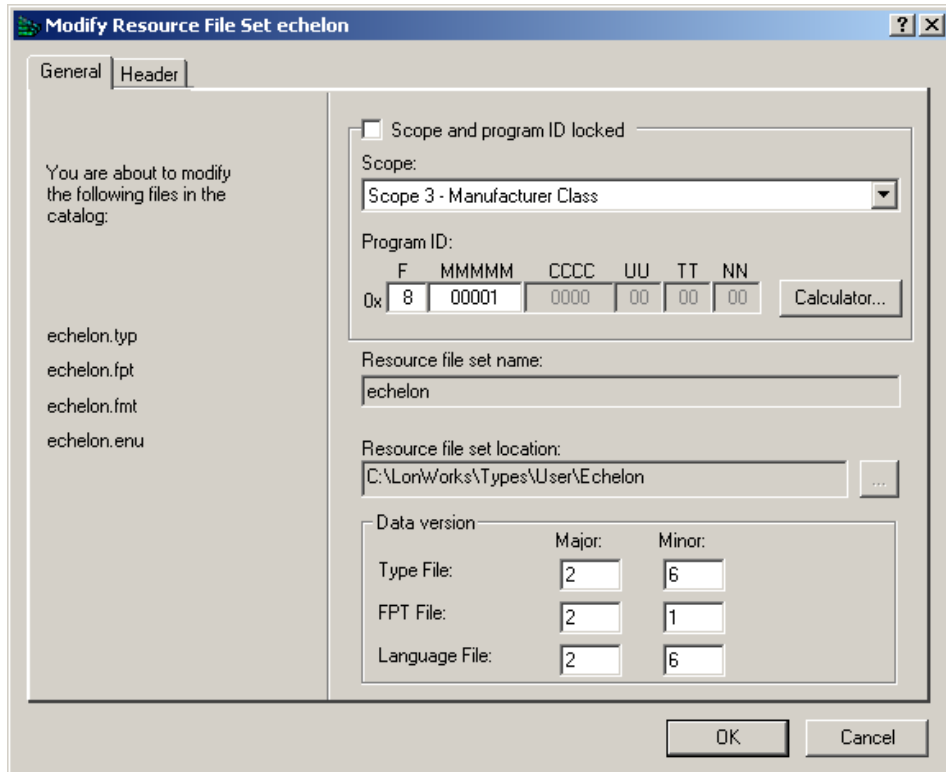
## Creating and Editing a Resource File Set

You can create a new scope 3, 4, 5, or 6 resource file set within any resource folder. Each resource file set has a number of properties that you will set when you create the set. You can edit these properties at any time.

The number of resource file sets that you will create will depend on the number of device types that you expect your company to develop, and the level of coordination that exists between your developers. For all but the largest enterprises, you may find it easiest to create a single manufacturer scope (scope 3) resource file set for your company, and use it to maintain and distribute all of your company's user types. Larger enterprises may find it easier to coordinate manufacturer and device class scope (scope 4) resource file sets, where a unique resource file set is created for each class of devices. Very large enterprises can request multiple manufacturer IDs from the LONMARK association, assign a different manufacturer ID to each major division that does LONWORKS development, and then maintain a separate scope 3 resource file set for each division. If your company has multiple LONWORKS developers, you may find it useful to initially create manufacturer, device class, and device subclass scope (scope 5) resource file sets during development, and then copy the definitions to the appropriate scope 3 or 4 resource file set when complete. You can create a manufacturer, device class, device subclass, and device model scope (scope 6) resource file set for any special-purpose types that you want to apply only to a single device type.

To create a new resource file set or edit an existing one, follow these steps:

1. Add a resource folder for your company if you do not have one already as described in *Adding, Moving, and Removing Resource Folders*.
2. To create a new resource file set, right-click the resource folder and then click **New Resource File Set** on the shortcut menu. To modify the properties for an existing resource file set, right-click the resource file set and then click **Open** on the shortcut menu. The following dialog opens:



3. Specify the following properties of the resource file set:

**Scope/Program ID Locked** Prevents modification of the scope or program ID template for this resource file set. This option is not displayed when you are creating a new resource file set because the scope and program ID template can be freely changed when you are creating a resource file set. This option appears when you are modifying a resource file set because modifying these options in an existing resource file set can break resources that reference this resource file set. You can modify the **Scope** and **Program ID** for an existing resource file set by clearing **Scope/Program ID Locked**. Be sure to fix any references to the resource file set and any references within the resources contained in this resource file set if you do this.

**Scope** The scope for the resource file set. See *Introduction to Resource Files* for more information.

**Program ID** The program ID template for the resource file set. Click **Calculator** to open the Standard Program ID Calculator (see *Using the Standard Program ID Calculator*). You only need to specify the program ID template fields that are required for the selected scope. Set

all other fields to 0. If you use the standard program ID calculator, verify that these fields are set to 0 after closing the calculator. The **Standard Development Program ID** setting is always ignored for resource files and should be cleared. The required values for each scope are as follows:

- Scope 3: **Manufacturer.**
- Scope 4: **Manufacturer and Device class.**
- Scope 5: **Manufacturer, Device Class, Usage, Channel Type, Has Changeable Interface, and Usage Field Values Defined by Functional Profile.**
- Scope 6: **Manufacturer, Device class, Usage, Channel Type, Has Changeable Interface, Usage Field Values Defined by Functional Profile, and Model Number.**

To change the program ID of an existing resource file set, create a new resource file set with the desired program ID and copy the resources from the old resource file set to the new one.

If you attempt to add or create a resource file set with a duplicate scope and program ID template to an existing resource file set, a warning is displayed when you generate the resource file set. You can resolve the conflict by removing one of the conflicting sets, or by changing the scope and program ID template of one of the sets.

**Resource File Set Name** The name of the resource file set as it will appear in the resource catalog. To change the name of an existing resource file set, you must copy it and remove the old resource file set, or edit the resource file names using Windows Explorer and restart the Resource Editor.

**Resource File Set Location** The resource folder containing the resource file set. Depending on which method you used to create the resource file set, you may be able to change the resource folder. If enabled, click  to create or select a new folder. If you select a folder that is not in the resource catalog, it will automatically be added. You cannot change the location for an existing resource file set. To change the location of an

existing resource file set, copy it to the new location, then remove the old resource file set.

### Data Version

The version number of the resource files. By default, **Major** is set to 1 and **Minor** is set to 0 for a new resource file set (i.e. version 1.0). Increment the major or minor version number whenever you publish new resource files (see *Generating Resource Files Using the Resource Editor*).

4. Set your desired options, and then click the **File Header** tab. The following dialog opens:

The screenshot shows a dialog box titled "New Resource File Set" with two tabs: "General" and "Header". The "Header" tab is active. On the left, there is a placeholder for "Summary information about this file." On the right, there are several input fields: "Creator" (Echelon), "Phone number" (555-1212), "Web ID" (empty), and "Email address" (jlon@abccorp.com). Below these are three radio buttons for "Type File", "Profile File", and "String File", with "Type File" selected. There are two text boxes with scrollbars: "File description" (containing "This file is a scope 3 type file. It contains user network variable types, configuration property types, and the") and "Creator description" (containing "This LonMark Device Resource TYP file was created by Echelon. Contact us at 555-1212 or at on the internet."). At the bottom right are "OK" and "Cancel" buttons.

5. Enter the following company information for the resource file set:

### Creator

The name of the company, and optionally the person, to contact about this resource file set. The default value is taken from the *NodeBuilder Registration Properties* tab.

### Phone Number

The phone number to contact for questions about this resource file set. The default value is taken from the *NodeBuilder Registration Properties* tab.

### Web ID

The Web address of the company that created this resource file set. The default value is taken from the *NodeBuilder Registration*

<b>Email Address</b>	<p>Properties tab.</p> <p>The email address to write to for more information about this resource file set. The default value is taken from the NodeBuilder Registration Properties tab. Enter a valid email address for general inquiries, such as <a href="mailto:lonworks@echelon.com">lonworks@echelon.com</a>, or the specific email address of the resource file set creator (e.g. <code>joe.block@acme.com</code>).</p>
<b>File Description</b>	<p>Text descriptions for the type file, functional profile, or language file (depending on what tab is selected). The default value is a string specifying the scope and file type.</p>
<b>Creator Description</b>	<p>Optional additional creator information (company name, contact information, etc) for the type file, functional profile, or string file (depending on what tab is selected). The default value is a string containing the information from the NodeBuilder Registration Properties tab. Changing the <b>Creator</b>, <b>Phone Number</b>, <b>Web ID</b>, and <b>Email</b> address fields in this tab will not update this string.</p>

6. Click **OK**. If you are creating a new resource file set, it is created and added to the resource catalog. If you are updating a resource file set, any changes are written to the resource file set. You can now add new network variable types, configuration property types, functional profiles, enumeration types, language strings, and formats to the resource file set. See *Creating and Editing Resources* for more information.

The Resource Editor can open older resource file formats. If you open a resource file set of an older resource file format version, the version will automatically be updated to the latest available format when you generate the resource files as described in *Generating Resource Files*, later in this chapter. To convert a resource file set to a older version, see *Converting a Resource File*, later in this chapter.

---

## Creating and Editing Resources

You can create and edit any resources within a scope 3, 4, 5, or 6 resource file set. If you are editing a resource file set that you have previously released, you must be careful to make changes that are backward-compatible with your released products. You can add a resource at any time and not affect your existing products. The following sections describe how to create and edit resources.

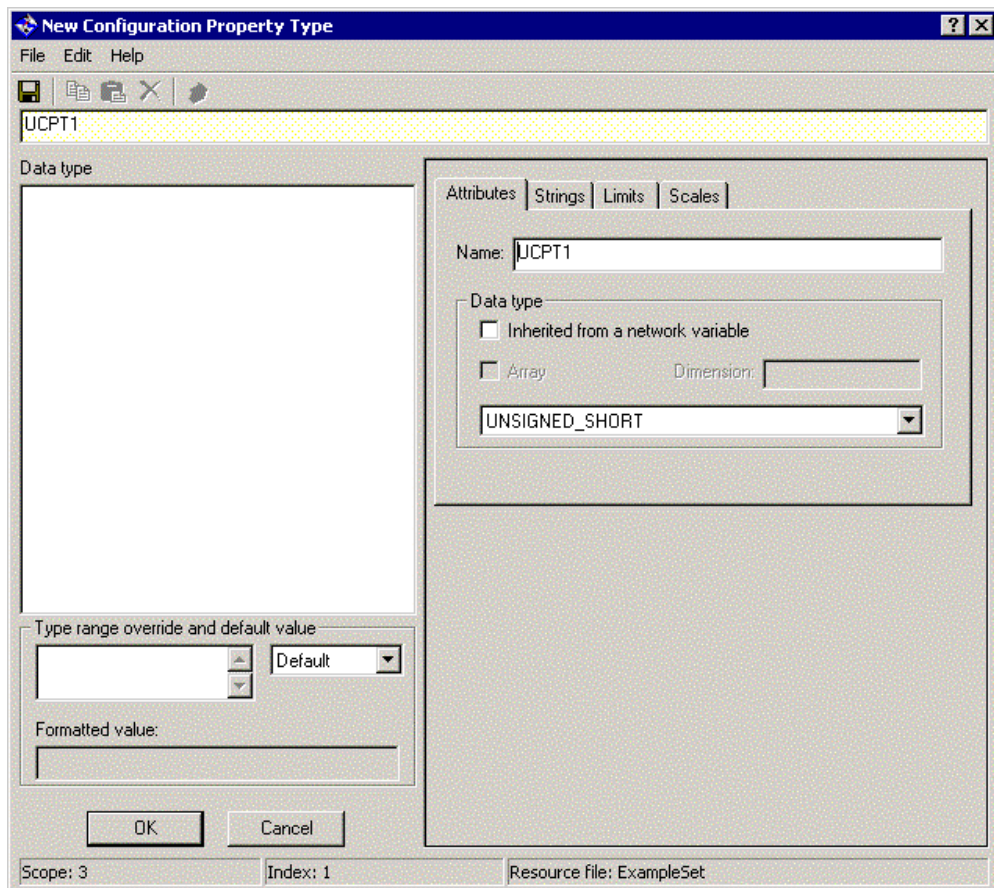
---

## Creating and Editing a Network Variable or Configuration Property Type

You can create and edit network variable types and configuration property types in any scope 3, 4, 5, or 6 resource file set. Do not attempt to do this in resource file sets that do not have your manufacturer ID or that you do not manage. If you create a configuration property types in a scope 6 resource file set, then create a configuration property in a device based on that type, note that the network integrator will be unable to preserve the value of the configuration property when the device is upgraded, since a new configuration property type (with a new Program ID) will have to be created, and the tool doing the upgrading will identify it as being a new configuration property.

To add a network variable type or configuration property type to a resource file set or to modify an existing network variable type or configuration property type, follow these steps:

1. To create a new type in a resource file set, right-click the **Network Variable Types** or **Configuration Property Types** folder in the resource file set and then click **New NVT** or **New CPT** on the shortcut menu. To modify an existing type, double-click the type. The following dialog opens:



The figure above shows the **New or Existing Configuration Property Type** dialog. The **New Network Variable Type** dialog is identical except it does not contain **Type Range Override and Default Value** or **Inherited**

**from a Network Variable.**

2. Enter a name for the new network variable type or configuration property type in **NV Name** or **CP Name**. Network variable type names must start with the letters “UNVT”. Configuration property type names must start with the letters “UCPT”. By default the name of the network variable or configuration property type is UNVT<NV Index> or UCPT<CP Index>, respectively. The network variable or configuration property index is shown in the status bar at the bottom of the window. Every new network variable and configuration property in a resource file set will be assigned a unique index (i.e. if you create a network variable with an index of 1 and then remove it, the next network variable you create will still have an index of 2. See *Removing and Obsoleting Resources* for more information).

By convention, user network variable type (UNVT) names have an underscore after “UNVT”; the first letter after the underscore is lower case; and the name uses all lower case with words separated by underscores (i.e. UNVT\_scale\_data). By convention, user configuration property type (UCPT) names do not have an underscore after the “UCPT” prefix, the first letter after UCPT is lower case, and the name uses mixed case (i.e. UCPTadcFilter). Network variable type names are limited to 64 characters, including the “UNVT\_” and “UCPT” prefixes. You can use upper and lower case alphanumeric characters and underscores. You cannot use spaces or other special characters in names.

3. Select the network variable or configuration property data type in **Data Type**.

If you are defining a configuration property type, and the type depends on the type of a network variable that the configuration property type applies to, set **Inherited From a Network Variable**. You can use this to create a configuration property type that will be used in multiple functional profiles that require different types. If this checkbox is set, the data type will be set to *INHERITED* and cannot be changed and **Type Range and Default Value** will be dimmed. When you implement a configuration property that uses an inherited configuration property type, the network variable the configuration property inherits from depends on how you implement the configuration property. If you associate the configuration property with a network variable, it will inherit its type from the network variable. If you associate the configuration property with a functional block, it will inherit its type from the principal network variable on that functional block. Configuration properties that inherit their type from a changeable type network variable will automatically change type when the network variable’s type is changed. See the *Neuron C Programmer’s Guide* for details about changeable type network variables. Configuration properties that inherit their type from a network variable cannot apply to the entire device.

You can create an array type if you are defining fields of a structure or union. To create a network variable or configuration property type array, first construct a structure, create a field within the structure, set **Array**, and then specify the size of the array in **Dimension**. See *Creating and Modifying a Structure or Union NV or CP Type*.

Also see *Adding a Configuration Property Member to a Functional Profile*, later in this chapter, for details about creating arrays of configuration properties as functional profile members.

If the type is not inherited, click the arrow to select from the following data types:

<b>BITFIELD</b>	A signed or unsigned bitfield, 1-8 bits wide. Only available for fields within a structure or union. See <i>Creating and Modifying a Bitfield</i> .
<b>ENUMERATED</b>	A signed 8-bit enumerated value. See <i>Creating and Modifying an Enumerated NV or CP Type</i> .
<b>FLOAT</b>	An IEEE 754 standard 32 bit floating point value.
<b>REFERENCE</b>	A reference to a network variable type. Uses the type definition of the referenced network variable type. If you are creating a structure or union, an individual field can reference a network variable type (see <i>Creating and Modifying a Structure or Union NV or CP Type</i> ). If the referenced network variable type changes in some way, the referencing type or field will automatically change as well. See <i>Creating and Modifying a Reference NV or CP Type</i> for more information.
<b>SIGNED_CHAR</b>	An 8-bit signed character value.
<b>SIGNED_LONG</b>	A 16-bit signed integer value.
<b>SIGNED_QUAD</b>	A 32-bit signed integer value.
<b>SIGNED_SHORT</b>	An 8-bit signed integer value.
<b>STRUCTURE</b>	A structure containing multiple fields. See <i>Creating and Modifying a Structure or Union NV or CP Type</i> .
<b>UNION</b>	A union containing multiple fields. See <i>Creating and Modifying a Structure or Union NV or CP Type</i> .
<b>UNSIGNED_CHAR</b>	An 8-bit unsigned character.
<b>UNSIGNED_LONG</b>	A 16-bit unsigned integer value.
<b>UNSIGNED_SHORT</b>	An 8-bit unsigned integer value.

4. If you have opened an existing type, **Make this Item Obsolete** will appear beneath **Data Type**. Set this checkbox to make this type obsolete as described in *Removing and Obsoleting Resources*, later in this chapter.
5. If you are creating a configuration property type, set the values in **Type Range Override and Default Value** (if you are creating a network variable type, skip to step 6). Set the default value, as well as override minimum and maximum values. If this is a reference configuration property, these limits will override the minimum and maximum values set in the **Limits** tab of the referenced network variable type.

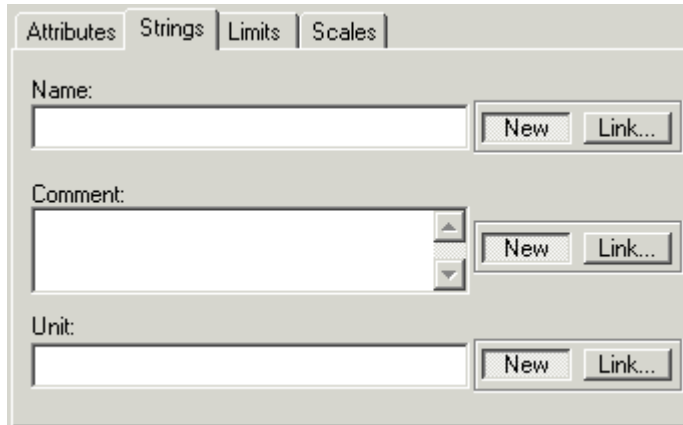
Use **Type Range Override and Default Value** to select **Minimum**, **Maximum**, or **Default**. Enter a hexadecimal value for each of these. You can only enter raw hexadecimal data, so when setting the default value for a structure, union, or floating point type, you must know the hexadecimal representation of the structure, union, or floating point type. See *Using the*



*LonMaker Browser to Calculate a Raw Value* for information on using the LonMaker browser to determine the hexadecimal representation of complex types.

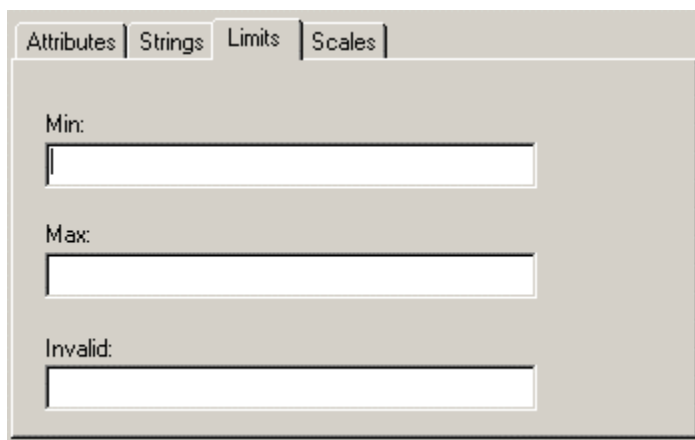
You can inspect the formatted version of the data entered into **Type Range Override and Default Value in Formatted Value**. If changes have been made to the type, this value may be based on outdated formatting rules; to ensure that the latest rules are applied, click **OK** to save changes and re-open the configuration property type by right-clicking it and clicking **Open** on the shortcut menu.

6. Click the **Strings** tab. This tab appears as shown in the following figure:



The screenshot shows a dialog box with four tabs: "Attributes", "Strings", "Limits", and "Scales". The "Strings" tab is selected. It contains three text input fields. The first is labeled "Name:" and has "New" and "Link..." buttons to its right. The second is labeled "Comment:" and has "New" and "Link..." buttons to its right. The third is labeled "Unit:" and has "New" and "Link..." buttons to its right.

7. Enter or link to text to provide a language-dependent name for the type, a language-dependent comment about the type, and a language-dependent name of the units for the type (for example, "Celsius" or "Kilometers-per-Hour"). If you enter a new string, it is created as a new language string in the currently active language file (see Setting Resource Editor Options). You can later translate this string to other languages as described in *Searching for a Language String*. See *Creating and Editing a Language String* for more information on creating and linking to language strings.
8. Click the **Limits** tab. This tab appears as shown in the following figure:

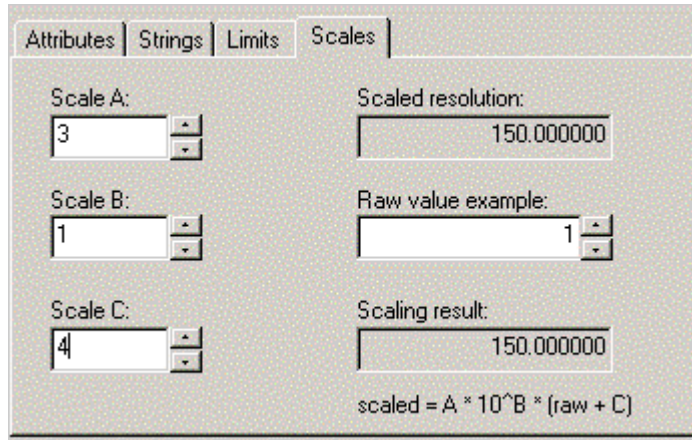


The screenshot shows the same dialog box with the "Limits" tab selected. It contains three text input fields. The first is labeled "Min:", the second is labeled "Max:", and the third is labeled "Invalid:".

9. For scalar types or scalar members of aggregates, set minimum and maximum allowable values for the type in the **Min** and **Max** fields, and the value to which the network variable or configuration property will be set when its value is unknown in the **Invalid** field. By default, **Min** and **Max** will contain the largest and smallest possible values for the Data Type in the

Attributes tab (see Step 3). You can set the limits for each field of a structure or union. You must create a format for the type for these limits to appear correctly in network tools such as the LonMaker tool. See *Creating and Editing a Resource File Format* for more information.

10. Click the **Scales** tab. This tab appears as shown in the following figure:



11. Set scaling factors for this type. This allows types to represent values outside of the limits of the base type. For example, an *UNSIGNED\_SHORT* data type can contain raw values between 0 and 255. By changing the scale, it could be used to represent values from 50-305, or from 0 to 510 (but contain only even values). The formula for converting the raw value to the scaled value is shown on the tab. By default, **Scale A** is 1, and **Scale B** and **Scale C** are 0, resulting in identical values for the raw value and scaled value (i.e. Scaled Value =  $1 * 10^0 * (\text{Unscaled Value} + 0)$ ). You can test your scaling factors by entering a value into **Raw Value Example** and observing the scaled value that appears in **Scaling Result**. You must create a format for the type for these limits to appear correctly in network tools such as the LonMaker tool. See *Creating and Modifying a Format* for more information.
12. Enter your desired values, and then click **OK**. The new network variable or configuration property type is added to the **Network Variable Types** or **Configuration Property Types** folder. A default format will be created for the network variable or configuration property type as described in *Creating and Editin a Resource File Format*.

## Using the LonMaker Browser to Calculate Raw Values

You can use the LonMaker browser to simplify setting default and minimum/maximum override configuration property values in the resource editor (see step 5 in *Creating and Editing a Network Variable or Configuration Property Type*). The LonMaker browser calculates the raw hexadecimal values that you must enter. This value can be difficult to determine without the LonMaker browser for configuration property types that use structure, union, or floating point types. To determine the raw hexadecimal value of a value, follow these steps:

1. Create the configuration property type using the resource editor as described in *Creating and Editing a Network Variable or Configuration Property Type*.
2. Add a configuration property based on the configuration property type to the device interface for a new device type using the Code Wizard (see *Adding a Network Variable to the Device Interface* in the *Generating Neuron C Code*

*Using the Code Wizard* chapter).

3. Generate code using the Code Wizard.
4. Build the device as described in *Building an Application Image*. You do not need to load the application into a device.
5. Add the device to the LonMaker network drawing.
6. Right-click the device and then click **Browse** on the shortcut menu. The LonMaker browser opens.
7. Find the configuration property you created. Right-click the configuration property and then click **Details** on the shortcut menu. The LonMaker browser's Details dialog opens.
8. Set the individual values of the fields in the structure, union, or the floating-point value to the desired default or override value and then click **OK** to return to the browser.
9. Right-click the configuration property and then click **Change Format** on the shortcut menu. The LonMaker browser's Change Format dialog appears.
10. Select **Built-in data types** from **Format Files**, then select **Raw (Hex)** from **Formats Available**, and then click **OK** to return to the browser. If the device you are browsing is not commissioned, you will get a warning message, which you can ignore.
11. In the browser, the value will now be displayed as a comma-separated list of values. Each value is a two-byte hexadecimal number (the browser removes leading zeroes).
12. Return to the configuration property type in the resource editor and enter the hexadecimal value for the default or override. Prepend a zero to any single digit values in the comma-separated list. For example, if the LonMaker browser shows a value of *0, 5, b, 36, c, 3, db*, enter *00050b360c03db*.

## Creating and Modifying a Structure or Union NV or CP Type

You can create network variable and configuration property types with multiple fields, each with their own data type, by defining a type as a structure or union. The fields in a structure are separate, whereas the fields in a union may overlap. For example, if a structure type contained an *UNSIGNED\_SHORT* and an *UNSIGNED\_LONG* field, the total size is 24 bits (8 bits for the short, and 16 bits for the long). If a union type contained these same two fields, the total size is 16 bits; the short shares the first 8 bits of the long. Unions and structures can both contain any data types, including other unions and structures, with the exception of bitfields, which can be used in structures, but not unions.

**Note:** While the display of the structure type bears a resemblance to Neuron C code, it is not in Neuron C syntax, and cannot be cut and pasted directly into a Neuron C file. To implement a variable using a network variable or configuration property type in Neuron C, simply declare this variable using the type name defined in the resource file. For example, `SNVT_count MyCount;` defines a variable, not a network variable, of type `SNVT_count`.

To create a union or structure type, follow these steps:

1. Create a new type as described in *Creating and Editing a Network Variable or Configuration Property Type*.

2. Set **Data Type** to *UNION* or *STRUCTURE*. The left pane displays `typedef struct { }` or `typedef union { }`.
3. Right-click the `typedef struct` or `typedef union` statement, and then click **Insert Field** on the shortcut menu. Enter attributes, strings, limits, and scales for the new field as described in *Creating and Editing a Network Variable or Configuration Property Type*. **Name** must be a valid name for an aggregate member in the Neuron C language, but is otherwise not limited (i.e. it does not have to start with “UNVT” or “UCPT” for fields). Each field contains its own data type, strings, limits, and scales. Repeat this step for each field in the structure.
4. Click **OK**. The new type appears in the **Network Variable Types** or **Configuration Property Types** folder.

To make changes to a field, click the field in the left pane and modify the field definition.

To remove a field, right-click the field in the left pane, and then click **Remove Field** on the shortcut menu.

## Creating and Modifying an Enumerated NV or CP Type

You can assign an enumerated type to a network variable type, configuration property type, or a field in a structure or union. To create an enumerated type, follow these steps:

1. If the enumeration type you will use does not already exist, create it as described in *Creating and Modifying an Enumeration Type*.
2. Create a new type as described in *Creating and Editing a Network Variable or Configuration Property Type*. In step 3, set **Data Type** to *ENUMERATED*. **Enum Information** appears in the dialog as shown in the following figure:

3. Set **Enum Scope** to the scope of the resource file set containing the enumeration type. You can select an enumeration type from the resource file set containing the type you are creating, or from any resource file set with a numerically lower scope and compatible program ID template. **File Path** is automatically updated to the path of the resource file set with the appropriate scope and program ID type. **Enum Set** is updated to contain all enumerations available in that resource file set.
4. Select the enumeration type to use in **Enum Set**. **Minimum** and

**Maximum** display the minimum and maximum values for the selected enumeration type. You can use these values to further restrict the range of available values for this type.

5. Continue creating the network variable or configuration property type as described in *Creating and Editing a Network Variable or Configuration Property Type*.

## Creating and Modifying a Bitfield

You can define a field of a structure as a bitfield. This data type is defined as a Neuron C bitfield. A bitfield is packed into a byte that can have from 1 to 8 bitfields that can each be 1 to 8 bits, for a total of no more than 8 bits per byte (for example, you can have one 8-bit field, two 4-bit fields, or 8 1-bit fields in a byte, or various combinations of these). Two subsequent bitfields whose total size exceeds the 8 bit boundary automatically cause a gap of unused data between the fields. To create a bitfield, follow these steps:

1. Create a new structure or union type as described in *Creating and Modifying a Structure or Union NV or CP Type*. Set **Data Type** to BITFIELD for the bitfield.
2. Enter the following information:

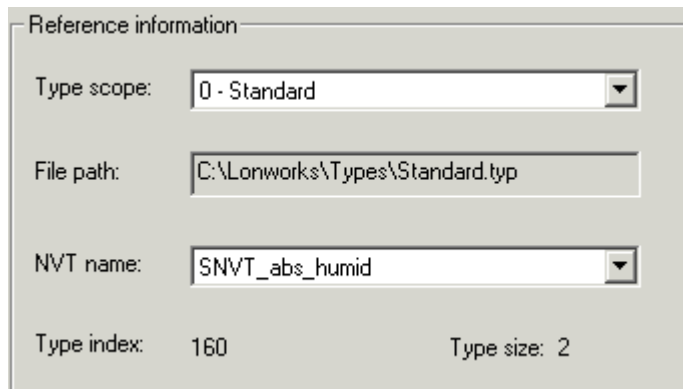
<b>Size</b>	The size of the bitfield, in bits, from 1 to 8. The bitfield size determines the maximum value the field can contain. An unsigned bitfield of $W$ bits width can accept values $0..2^W-1$ , a signed bitfield of width $W$ can hold values $-2^{W/2}..+2^{W/2}-1$
<b>Offset</b>	The offset of the bitfield within the byte. A full byte is always required regardless of how many bits in the bitfield are used. This value determines where in the byte the values will be set. This value can be from 0 to $(8 - \text{Size})$ .
<b>Signed/Unsigned</b>	Specifies a signed or unsigned value. An unsigned bitfield can only contain positive values. A signed bitfield can contain positive or negative values with negative numbers values stored using twos complement notation. Twos complement notation is created by converting the number to binary, complementing each bit, and adding 1 to the resultant binary number.  A signed bitfield with a width of 1 can contain only two values: -1 (minus one) and 0 (zero). This is often unwanted; the developer is likely to require an unsigned bitfield of the same width, accepting 0 (zero) and +1 (plus one). The use of unsigned bitfields is recommended due to a slightly better performance on the Neuron Chip, but also in order to avoid the common mistake of declaring bitfields with a width of one bit as signed.

Unlike the ANSI-C and Neuron C programming languages, device resource files do not support anonymous bitfields (bitfields that are declared with signedness and width, but without a name, e.g. `int : 3;`) or bitfields with a width of 0. To control the positioning of a bitfield within the compound byte, set the offset value accordingly. If the Resource Editor detects a gap between bitfields, it will raise a warning describing the situation and offer to leave the gap intact, or to close the gap by adjusting the offset preferences accordingly. Do not allow automatic adjustment if you have purposefully laid out the bitfields to match some specific requirement.

## Creating and Modifying a Reference NV or CP Type

Network variable types, configuration property types, and fields within structure or union types can be based on existing network variable types (but not configuration property types) that are defined within the same resource file, within the standard resource file, or within any other resource file that has a compatible program ID and scope selector. When this is done, if the referenced type changes in some way (type size, fields, etc), any configuration property and network variable types that reference it will automatically be changed as well. To create a type based on an existing type, follow these steps:

1. Create a new type as described in *Creating and Editing a Network Variable or Configuration Property Type*. In step 3, set **Data Type** to **REFERENCE**. **Reference Information** appears in the dialog as shown in the following figure:



The image shows a dialog box titled "Reference information". It contains four fields: "Type scope:" with a dropdown menu showing "0 - Standard"; "File path:" with a text box containing "C:\Lonworks\Types\Standard.typ"; "NVT name:" with a dropdown menu showing "SNVT\_abs\_humid"; and "Type index:" with the value "160" and "Type size:" with the value "2".

2. Set **Type Scope** to the scope of the resource file set containing the referenced network variable type. You can select a network variable type from the resource file set containing the network variable or configuration type you are creating, or from any resource file set with a numerically lower scope and matching program ID template. You can always choose a standard network variable type. **File Path** is automatically updated to the path of the resource file set with the appropriate scope and program ID template, and **NV Name** is updated to contain all network variable types available in that resource file set.
3. Set **NV Name** to the network variable type to use. **Type Index** and **Type Size** are automatically updated when you set this value. **Type Index** indicates the index of the network variable type within its resource file set. **Type Size** indicates the number of bytes in the selected network variable type.

4. Continue creating the network variable or configuration property type as described in *Creating and Editing a Network Variable or Configuration Property Type*.

---

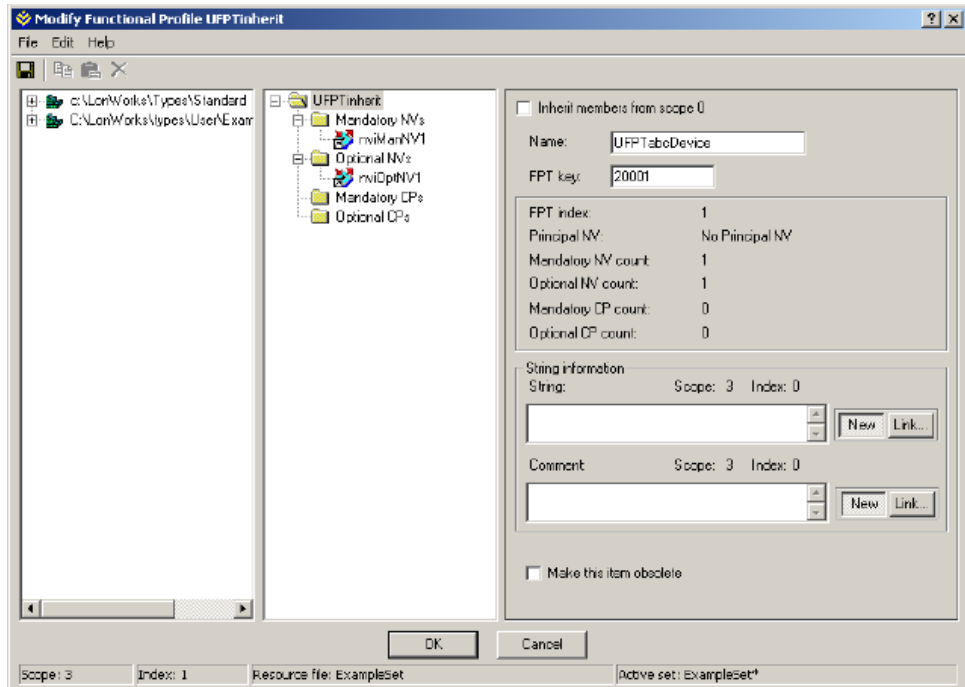
## Creating and Modifying a Functional Profile

You can create and edit a functional profile in any scope 3, 4, 5, or 6 resource file set. You can define a functional profile that inherits members from a scope 0 profile, or you can create a new functional profile. When you inherit from a scope 0 profile, you can add your own members to the scope 0 profile, or you can override members in the scope 0 profile.

Do not attempt to create or modify functional profiles in resource file sets that do not have your manufacturer ID or that you do not manage.

Functional profiles are used to create functional blocks. Each functional profile can contain mandatory and optional network variables and configuration properties. To create or modify a functional profile in a resource file set, follow these steps:

1. To create a new functional profile, right-click the **Functional Profile Templates** folder in the resource file set and then click **New FPT** on the shortcut menu. A new functional profile template will be added to the resource file set. To modify an existing functional profile, skip to step 3.
2. If this is a new functional profile, enter the name. The name must start with “UFPT” and may not contain spaces. By convention, there is no underscore following UFPT; the first letter after UFPT is lower case; and the name uses mixed case. Functional profile names are limited to 64 characters, including the “UFPT” prefix. You can use upper and lower case alphanumeric characters and underscores. You cannot use spaces or other special characters in names; a functional profile name must meet the requirements of a Neuron C variable name with the additional restriction that the dollar character is not permitted in a functional profile name. By convention, the functional profile name should indicate the application set of the profile, e.g. “SFPToccupancySensor”, or “UFPTturboCharger”.
3. Double-click the functional profile. The following dialog appears. This dialog contains three panes: a Resource pane showing all resources that may be referenced by this functional profile; a Profile pane that displays the network variables and configuration properties defined as part of this functional profile; and a Properties pane that displays functional profile properties or properties of member network variables and configuration properties.



4. Enter the following information about this functional profile in the Properties pane:

**Inherit Members from Scope 0**

Specifies that this functional profile inherits network variable and configuration property members from the scope 0 profile with the same key. You can use this option to add new members to an existing standard functional profile, to redefine existing members of an existing standard functional profile, or both.

**FPT key** must be set to a value less than 20000 to inherit from a scope 0 profile. When this checkbox is set, all network variables and configuration properties from the selected standard profile will be referenced by the functional profile (if there is no standard functional profile template with the same profile number, setting this option will have no effect). You cannot remove any of the inherited members, but you can add additional network variables and configuration properties to the functional profile as described below, and you can redefine any of the existing members. All inherited members will be displayed in pink, while new ones will be shown in light blue.

You cannot remove any of the inherited network variables and configuration properties, but you can override them. To override an inherited network variable or



configuration property, add a new configuration property or network variable with the same direction and name. The capitalization of the name must match. When you close the pane (e.g. by clicking **OK**), you will be prompted to confirm the override. Overridden network variables and configuration properties will inherit string information.

You must override all configuration properties that apply to the overridden network variable. A warning will list any configuration properties without overrides if you fail to meet this requirement.

Inheriting members from a standard functional profile allows you to extend and customize the standard functional profiles to suit your device, while maintaining the standard interface for increased interoperability.

**Name**

The name of the functional profile. This name must start with "UFPT" and may not contain spaces. By convention, there should be no underscore following "UFPT, the first letter after "UFPT should be lower case; and the name uses mixed case.

Devices that implement the profile can choose to publish the profile's name as part of the device's self-documentation, thereby promoting interoperability. It is therefore good practice to choose a profile name that indicates the basic function of the profile. Examples include UFPTtempSensor, SFPToccupancyDetector, etc.

**FPT Key**

A unique ID for this functional profile within this resource file set. Enter a key of 20000 or greater if you are creating a new functional profile. Enter the original key (which will be less than 20000) if you are redefining a standard functional profile using **Inherit Members from Scope 0**. This is the same as the functional profile number.

**FPT Index**

The index of the functional profile within the resource file set. The **FPT Key** (the profile number) is typically used to reference the functional profile, not the **FPT Index**.

**Principal NV**

The name of the principal network variable. The principal network variable is used to determine the type of configuration properties

assigned to the functional profile that use an inherited type, and may also be used to assist the network integrator by highlighting the prime network variable associated with this profile. Since a profile may not contain any mandatory network variables at all, this name can be empty. See *Adding a Network Variable or Configuration Property to a Functional Profile* for more information.

<b>Mandatory NV Count</b>	The number of mandatory network variables that have been created in this functional profile.
<b>Optional NV Count</b>	The number of optional network variables that have been created in this functional profile.
<b>Mandatory CP Count</b>	The number of mandatory configuration properties that have been created in this functional profile.
<b>Optional CP Count</b>	The number of optional configuration properties that have been created for in functional profile.
<b>String Information</b>	A language-specific name for the functional profile and a comment that describes the purpose of the functional profile. See <i>Adding Strings to a Language Resource File</i> , later in this chapter, for more information on creating and linking to language strings.

5. Add mandatory and optional network variable and configuration property types to the Profile pane as described in *Adding a Network Variable Member to a Functional Profile* and *Adding a Configuration Property Member to a Functional Profile*, later in this chapter.

## Adding a Network Variable Member to a Functional Profile

You can add mandatory and optional network variables to a functional profile. When you create a functional block from a functional profile, it must implement all the mandatory network variables defined by the functional profile, if any. It may implement some, all, or none of the optional network variables, and it may add implementation-specific network variables.

You can also add mandatory and optional network variables to a functional profile that inherits members from scope 0 (see *Creating and Modifying a Functional Profile*). The inherited profile initially contains all the members defined in the scope 0 profile. You can add new members, or you can redefine existing members. To redefine a member, define a new member with the same name as the scope 0 member to be redefined.

To add mandatory and optional network variables to a functional profile, follow these steps:

1. Drag a network variable type or configuration property type from the

Resource (leftmost) pane to the appropriate in the Profile (center) pane. The Profile pane contains the functional profile definition with **Mandatory NVs**, **Optional NVs**, **Mandatory CPs**, and **Optional CPs** folders for the network variables and configuration properties in the profile.

2. Select the new network variable to set options for it. The Properties (rightmost) pane appears as follows:

3. Enter the following information:

**Name**

The name of the network variable member within the functional profile. The name may contain only letters, numerals, and the underscore character, and it must not start with a digit. A prefix is not required, but you may start input network variable member names with "nvi" and output network variable member names with "nvo" to simplify identifying inputs and outputs in your functional profile. If you do not use one of these

	<p>prefixes, start the network variable name with an initial capital and use mixed case for the name.</p>
<b>Member</b>	<p>A member number for the network variable within the functional profile. Each network variable must have a unique member number. Member numbers may start with a “#” or “ ” character. The “#” prefix identifies members within a user functional profile. The “ ” prefix identifies member numbers in a scope 0 profile, both for scope 0 profiles as well as profiles with members that are inherited from a scope 0 profile.</p>
<b>Reference/Scope</b>	<p>The name and scope of the network variable type of this network variable. You can use <b>Reference</b> to change the network variable type. You can change which network variable types are available in <b>Reference</b> by changing <b>Scope</b>.</p>
<b>Principal NV</b>	<p>Designates this network variable as the principal network variable of this functional profile. Each functional profile may have one principal network variable. The principal network variable is used to determine the type of configuration properties with inherited types that apply to the functional profile.</p>
<b>Input/Output</b>	<p>Determines whether this network variable is an input or output.</p>
<b>Service Type</b>	<p>Sets the default communication service to use for an output network variable member. You can select <b>Acknowledged</b>, <b>Unacknowledged</b>, or <b>Repeated</b> to specify a specific service, or you can select <b>Unspecified</b> to leave the default up to the network tool used to install a device containing a functional block based on this profile. The service type may be changed by a network tool when the device is added to a network, or by the development tool when the network variable is being declared. The <b>Service Type</b> field should be seen as a recommendation, not a requirement.</p>
<b>Polled</b>	<p>Identifies an output network variable as a polled output. Polled outputs do not propagate their values on the network until they are polled by a receiving device. The <b>Polled</b> flag should be seen as a recommendation; the polling preference can be changed when this profile is implemented with a development tool.</p>
<b>String Information</b>	<p>A language-specific name for the network variable within the profile and a comment that describes the purpose of the network variable within this profile. See <i>Creating and Editing a Language String</i> for more information on creating and</p>

**Referenced Type Range  
Override Value**

linking to language strings.

Minimum and maximum values for the network variable member. A network variable member with a reference type can have a minimum and maximum range restriction that is more restrictive than the range restriction for the network variable type.

You can enter raw data into this field using any of the following methods:

- A continuous stream of hexadecimal digits. Example: 01010101ABCDABCD00
- Dash, colon, or dot separators for optional grouping. Example: 01010101-ABCDABCD-00
- A single asterisk character followed by a decimal multiplier, which repeats the preceding sequence of hexadecimal digits. Example: 01\*4-ABCD\*2-00. The raw data display may not match what you originally entered. For example, if you enter “00-00\*4,” it will later be shown as “00\*5.”

**Formatted Value**

Displays the translated value based on the raw value entered in **Referenced Type Range Override and Default Value**.

## Adding a Configuration Property Member to a Functional Profile

You can add mandatory and optional configuration properties to a functional profile. When you create a functional block from a functional profile, it must implement all the mandatory configuration properties defined by the functional profile. It may implement some, all, or none of the optional configuration properties, and it may add implementation-specific configuration properties.

You can also add mandatory and optional configuration properties to a functional profile that inherits members from scope 0 (see *Creating and Modifying a Functional Profile*). The inherited profile initially contains all the members defined in the scope 0 profile. You can add new members, or you can redefine existing members. To redefine a member, define a new member with the same name as the scope 0 member to be redefined.

To add mandatory and optional configuration properties to a functional profile, follow these steps:

1. Drag a configuration property type from the Resource (leftmost) pane to the appropriate folder in the Profile (center) pane. The Profile pane contains the functional profile definition with **Mandatory CPs** and **Optional CPs** folders for the configuration properties in the profile. Each type that you drag becomes a new member in the profile. You can drag the same type multiple times to create multiple members of the

same type, however, you cannot create more than one configuration property of the same type that applies to the same interface within a functional profile. For example, you can create multiple **SCPTmaxSendTime** configuration properties, but you cannot create two **SCPTmaxSendTime** configuration properties that apply to the same network variable. You can, however, create an array of **SCPTmaxSendTime** configuration properties that apply to a single network variable.

2. Select the new configuration property to set options for it. The Properties (rightmost) pane appears as follows:

The screenshot shows a configuration dialog box with the following sections:

- Inherit members from scope 0:** A checkbox that is unchecked.
- Name:** A text field containing "nciManCP1".
- Member:** A dropdown menu showing "#1".
- Reference:** A dropdown menu showing "SCPT airTemp1Alrm".
- Scope:** A dropdown menu showing "0".
- CP settings:**
  - Array implementation:** A dropdown menu showing "Prevent".
  - Min. array size:** A spinner box set to "0".
  - Max. array size:** A spinner box set to "0".
  - Flags:** Four unchecked checkboxes:
    - const\_flg (Value is never changed)
    - device\_specific\_flg (Always read value from the device)
    - mfg\_flg (Modify only during manufacture)
    - obj\_disabl\_flg (Disable functional block before modifying)
  - Applies to:** A dropdown menu showing "Functional block".
- String information:**
  - String:** A text field containing "Air temperature 1 percent alarm".
  - Scope:** "0".
  - Index:** "790".
  - Buttons:** "New" and "Link..."
  - Comment:** A text field containing "The weighting of the air temp 1 sensor when calculating the air temp alarm".
  - Scope:** "0".
  - Index:** "791".
  - Buttons:** "New" and "Link..."
- Type range override and default value:**
  - Value:** A text field containing "4E-20".
  - Default:** A dropdown menu showing "Default".
  - Formatted value:** A text field containing "100.000".

3. Enter the following information:

**Name**

The name of the configuration property member within the functional profile. This name may contain only letters, numerals, and the underscore character, and it must not start with a digit. A prefix is not required, but "nci" and "cp" are commonly used. If you do not use this prefix,

start the configuration property name with an initial capital and use mixed case for the name.

**Member**

A member number of the configuration property within the functional profile. Each configuration property must have a unique member number. Member numbers may start with a “#” or “|” character. The “#” prefix identifies members within a user functional profile. The “|” prefix identifies member numbers in a scope 0 profile, both for scope 0 profiles as well as profiles with members that are inherited from a scope 0 profile.

**Reference/Scope**

The name and scope of the configuration property type of this configuration property. You can use **Reference** to change the configuration property type. You can change which configuration property types are available in **Reference** by changing **Scope**.

**Array  
Implementation/Min  
Array Size/Max Array  
Size**

Specifies whether the configuration property within the profile may be implemented as an array, must be implemented as an array, or may not be implemented as an array. Select from the following values:

*Prevent* — Functional blocks created using this functional profile template cannot implement this configuration property as an array. **Min Array Size** and **Max Array Size** will be deactivated. This is the default setting, and also applies to all functional profiles created prior to NodeBuilder 3.1.

*Permit* — Functional blocks created using this functional profile template can implement this configuration property as an array at the discretion of the implementer. Use **Max Array Size** to limit the maximum size of this array. **Min Array Size** will be deactivated.

*Require* — Functional blocks created using this functional profile template must implement this configuration property as an array. Use **Max Array Size** and **Min Array Size** to limit the minimum and maximum size of this array.

**CP Settings**

Options that control how network tools update the configuration property. See the *Neuron C Programmer’s Guide* and the *Neuron C Reference Guide* for details about the configuration property restriction flags.

Configuration property restriction flags are requirements. When a functional block implements a profile, each of the implemented member configuration properties must specify at

least those restriction flags that are set in the profile. Restriction flags that are not set in the profile may be set by the implementing property, unless this would cause an ambiguous restriction flag set.

**Applies To**

Specifies whether the configuration property applies to a network variable or the entire functional block. If the configuration property applies to a network variable, **Applies To** also specifies the specific network variable.

If the configuration property applies to the entire functional block but itself implements an inheriting type, the property will derive its type from the principal network variable. A principal network variable must be defined in this case.

**String Information**

A language-specific name for the configuration property within the profile and a comment that describes the purpose of the configuration property within this profile. See *Creating and Editing a Language String* for more information on creating and linking to language strings.

**Type Range Override and Default Value**

Minimum, maximum, and default values for the configuration property member. A configuration property member can have a minimum and maximum range restriction that is more restrictive than the range restriction for the configuration property type it is based upon, and can have a different default value than the base type.

You can enter raw data into this field using any of the following methods:

- A continuous stream of hexadecimal digits. Example: 01010101ABCDABCD00
- Dash, colon, or dot separators for optional grouping. Example: 01010101-ABCDABCD-00
- A single asterisk character followed by a decimal multiplier, which repeats the preceding sequence of hexadecimal digits. Example: 01\*4-ABCD\*2-00. The raw data display may not match what you originally entered. For example, if you enter “00-00\*4,” it will later be shown as “00\*5.”

**Formatted Value**

Displays the translated value based on the raw value entered in **Type Range Override and Default Value**.



## Using Cascading Resource File Sets

When adding network variable or configuration property members to a functional profile, you can add resources that are defined at the profile's scope, at the standard scope (i.e. scope selector 0), or at any matching scope that is numerically less than the profile's scope.

For example, your company may maintain several resource file sets at different scopes: a corporation wide set with general-purpose definitions at manufacturer scope (scope selector 3), and a second set with more specific resources at device class scope (scope selector 4). This is known as *cascading resource file sets*.

It is possible that both sets will contain resources using the same name. For example, both resource file sets might contain a configuration property type named UCPTsetpoint.

From a functional profile editing point of view, it is possible to add both UCPTsetpoint (scope selector 3) and UCPTsetpoint (scope selector 4) to the same functional profile.

However, the Neuron C language is commonly used to implement functional profiles. The hypothetical example can cause the Neuron C compiler to implement the UCPTsetpoint that is defined at the highest scope level (device class scope, scope selector 4) in both cases.

The Neuron C language requires a configuration property to be implemented using a declaration that relies on the property type name (UCPTsetpoint). When searching the entire contents of the resource file catalog for a matching resource, the Neuron C Compiler will be satisfied with the one found at device class scope.

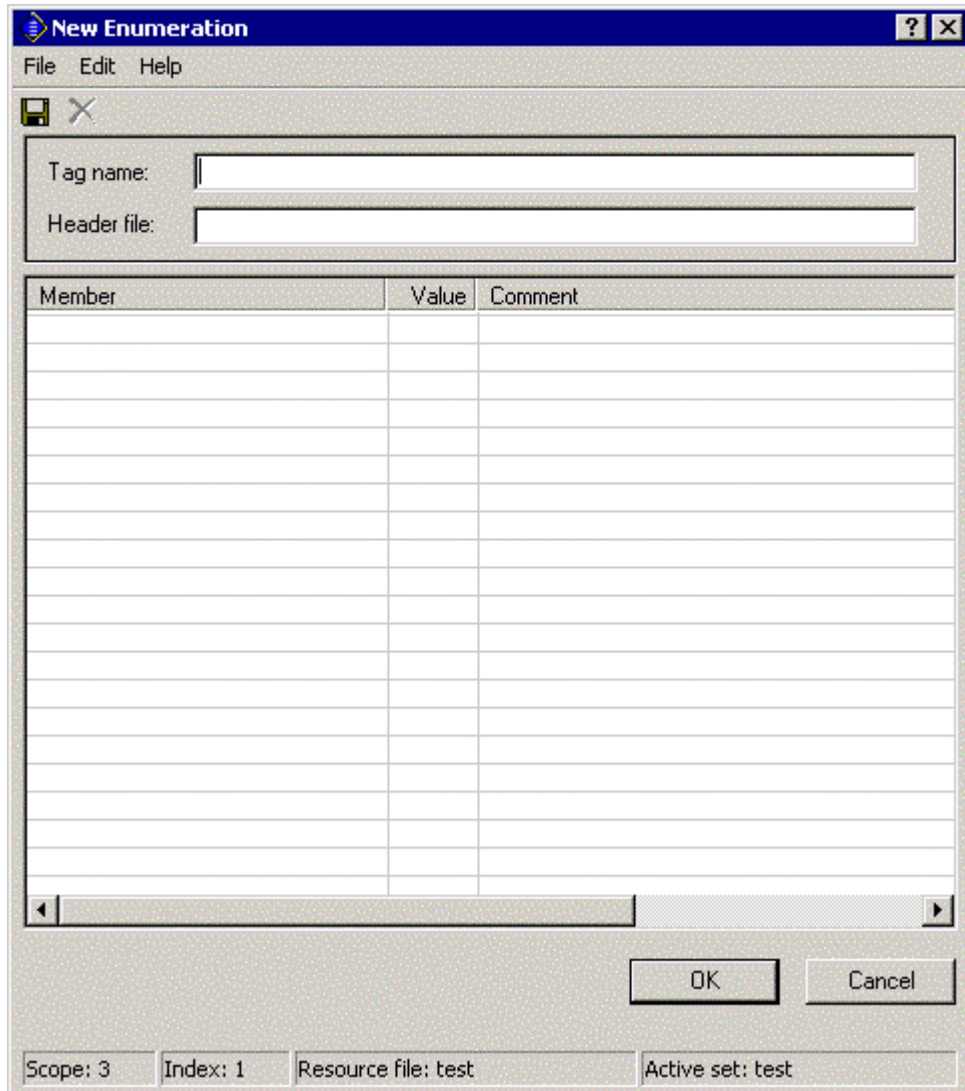
Therefore, you should avoid reusing names of network variable or configuration property types in cascading resource file sets.

---

## *Creating and Modifying an Enumeration Type*

You can create and edit enumeration types in any scope 3, 4, 5, or 6 resource file set. Do not attempt to do this in resource file sets that do not have your manufacturer ID or that you do not manage. An enumeration type is a list of enumerators that may be assigned to an enumeration network variable, configuration property, or field of one of these. Each enumerator consists of a name, value, and associated language string or strings. Network variable and configuration property types can reference enumeration types as described in *Creating and Editing Enumerated Network Variable and Configuration Property Types*, earlier in this chapter. To create or modify enumeration types in a resource file set, follow these steps:

1. To create a new enumeration type in a resource file set, right-click the **Enumerations** folder in the resource file set and then click **New Enum** on the shortcut menu. To modify an existing enumeration type, double-click the enumeration type. The following enumeration type editor appears:



2. Enter or change the name of the enumeration type in **Tag Name**. This name is called the *tag name*. By convention, the tag name is all lower case, with each word in the name separated by an underscore, and ending with “\_t” (for example: count\_control\_t). Tag names are limited to 64 characters, including the “\_t” suffix. You can use upper and lower case alphanumeric characters (though upper case is typically not used for tag names) and underscores. You cannot use spaces or other special characters in tag names. When you enter a name in **Tag Name**, **Header File** is automatically set to <Tag Name>.h. **Header File** contains the name of the C header file (.h extension) that will store the enumeration definition. Each enumeration type is stored in its own header file, which is placed in the resource folder. When resource files are generated (see *Generating Resource Files*, later in this chapter), the enumeration types are stored in the type file (.typ extension). In order to use the enumeration types in a NodeBuilder project, you must add the directory containing the header files to the **Include Search Path** in the *Project* tab of the NodeBuilder Project Properties dialog.
3. Enter or change the enumerators in the table. For each enumerator, enter the following information:

**Member** The name of the enumerator. The name must be unique for all enumeration types that may be used in an application. To ensure uniqueness, select a unique prefix for each enumeration type. By convention, enumerator names use all upper case, with words separated by underscores (for example: DSIT\_DRY\_CONTACT). Enumerator names are limited to 64 characters, including the unique prefix. You can use upper and lower case alphanumeric characters (though lower case is typically not used) and underscores. You cannot use spaces or other special characters in names.

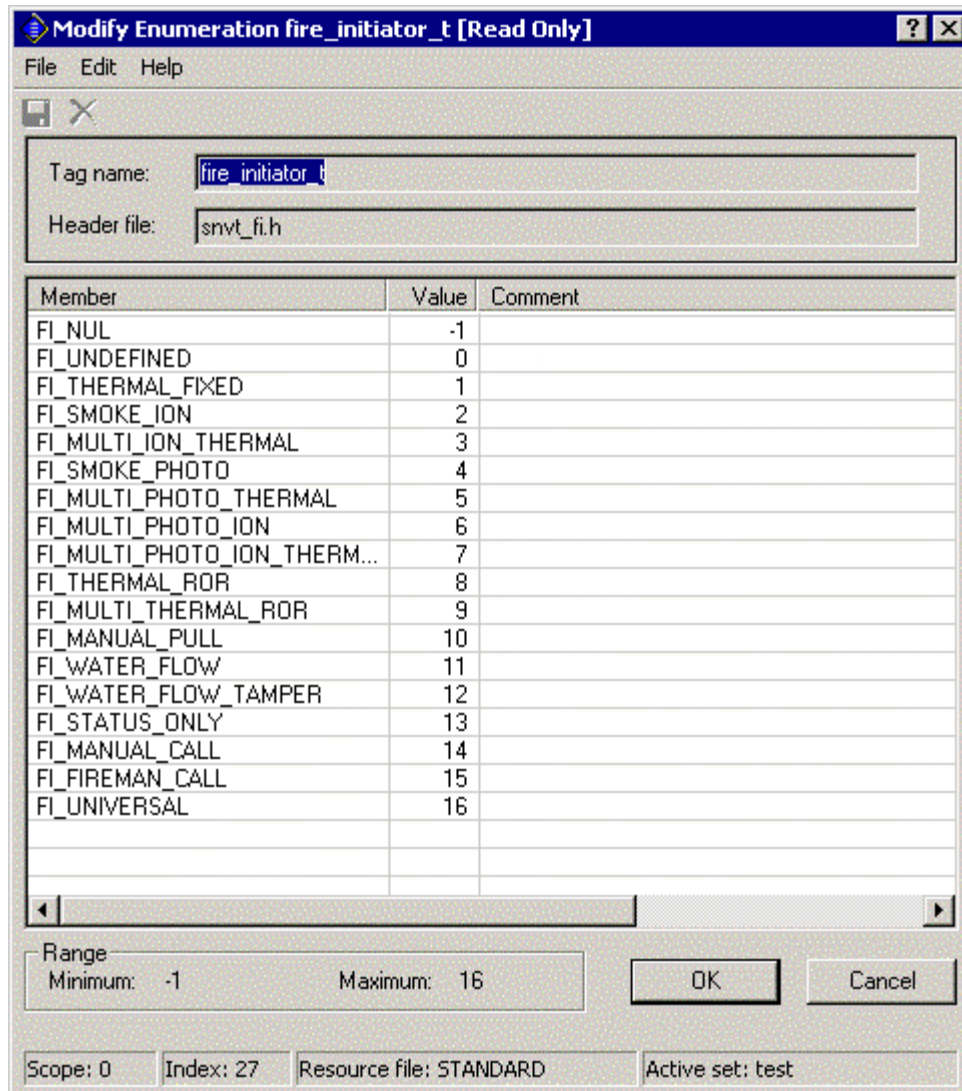
**Value** The value associated with the enumerator. This value must be between -128 and 127 (inclusive). A value of -1 is used to indicate an invalid value, and must be included in every enumeration type.

**Comment** The language string associated with the enumerator. You can enter a new string or select an existing one. When a network integrator or network operator uses the enumeration, they will see this value for the enumerator.

To enter a new string, double-click **Comment** and enter a new value. The new value will be added to the language file for this resource file set (see *Using Resource Strings*, later in this chapter).

To reference an existing string, click **Comment** and then click the  button that appears when **Comment** is selected. The Link dialog appears that allows you to browse to the desired string as described in *Adding Strings to a Language Resource File*.

**Example:** The following diagram shows the `fire_initiator_t` enumeration type from the standard resource file set:



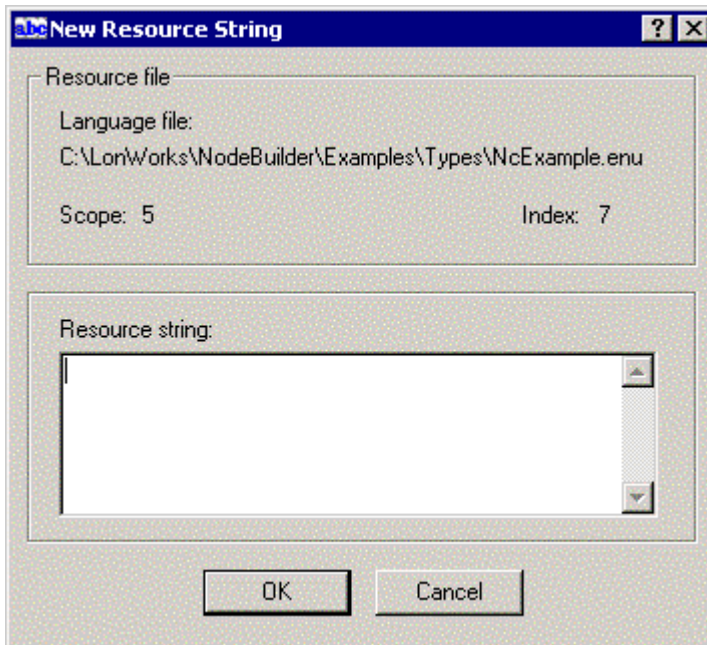
## Creating and Editing a Language String

You can create and edit language strings that may be referenced by network variable types, configuration property types, enumeration types, functional profiles, and resource files. These strings are contained in a *language file*. Each resource file set contains a language file for each language it supports. You can create new language strings directly (see *Adding a String to a Language File*), or create them as you define the types that will use them (see *Adding a String to a Language File While Defining a Resource*). Once you have a language file created in one language, you can create other language files and translate the strings as described in *Creating, Modifying, and Translating a Language File*.

### Adding a String to a Language File

You can add a language string directly to a language file. You can then reference the string from any resource that requires a string reference. To add a string to a language file, follow these steps:

1. Expand the **Language Files** folder in the resource file set.
2. Right-click a language file, and then click **New Resource String** on the shortcut menu. This command will only be available if the active language is the same as the selected language resource file. See *Setting Resource Editor Options* for information about setting the active language. The following dialog appears. This dialog shows the name and scope of the selected language file and the index of the new string within that file.



3. Enter the text of the new language string into **Resource String**, and then click **OK**.

You can also create new strings by copying existing strings. To copy a string, follow these steps:

4. Right-click the source string and then click **Copy** on the shortcut menu.
5. Right-click the destination language file and then click **Paste** on the shortcut menu.

You can only paste language strings into a language file of the same language as the file from which the string was copied (i.e. a string copied from a USA English resource file can only be pasted into a USA English resource file).

You can create new language strings as you define the network variable types, configuration property types, functional profiles, and enumeration types that reference them. See *Adding a String While Defining a Resource*, later in this chapter, for more information.

## Adding a String While Defining a Resource

You can create language strings as you create network variable types, configuration property types, functional profiles, and enumeration types (see *Creating and Editing a Network Variable or Configuration Property Type*, *Creating and Editing a Functional Profile*, and *Creating and Editing an Enumeration Type*, earlier in this chapter).

The functional profile dialog and the **Strings** tab of the network variable type and configuration property type dialogs each contain fields that appear similar to the following figure:

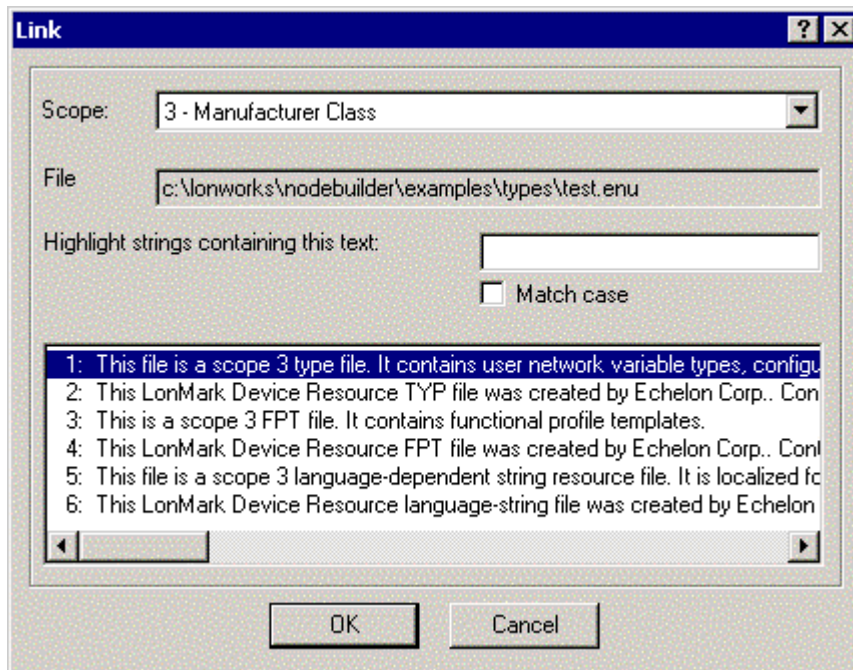


The title of the field may vary (this one is intended for entering comments about the resource file element being created), but there is always a text field, with **New** and **Link** buttons.

To create a new language string, click **New** to create a new language string. The text string will automatically be saved in the active language file.

To link to an existing language string, follow these steps:

1. Click **Link**. The following dialog opens. This dialog allows you to choose a string from the current resource file set, or from any other applicable resource file set.

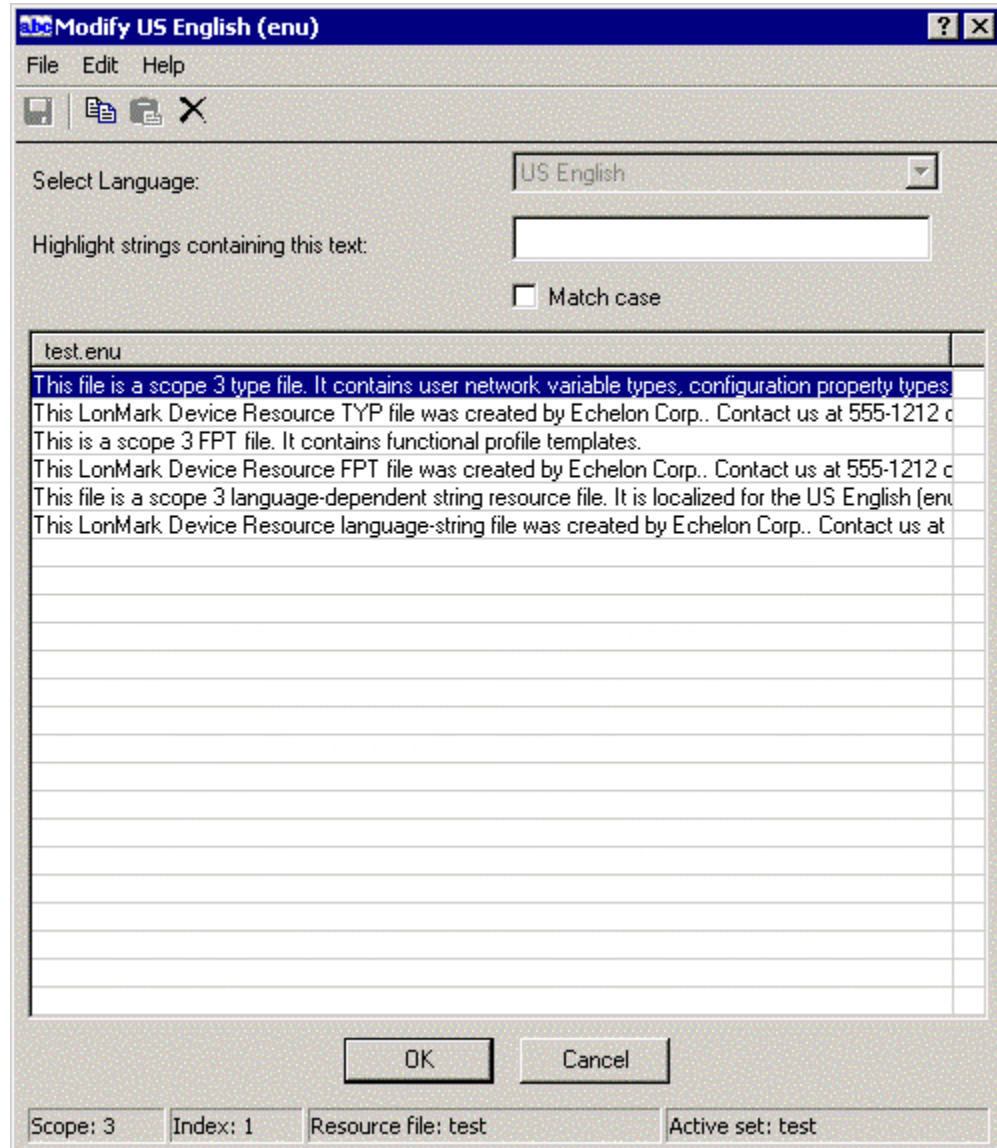


2. Set **Scope** to the scope of the resource file set containing the language string. You can only select a scope value equal to or less than the scope of the current resource file set. To select a string from the current resource file set, set **Scope** to the current resource file set's scope.
3. Set **Index** to the language string index. The selected string is displayed in **Resource String**.
4. Click **OK**.

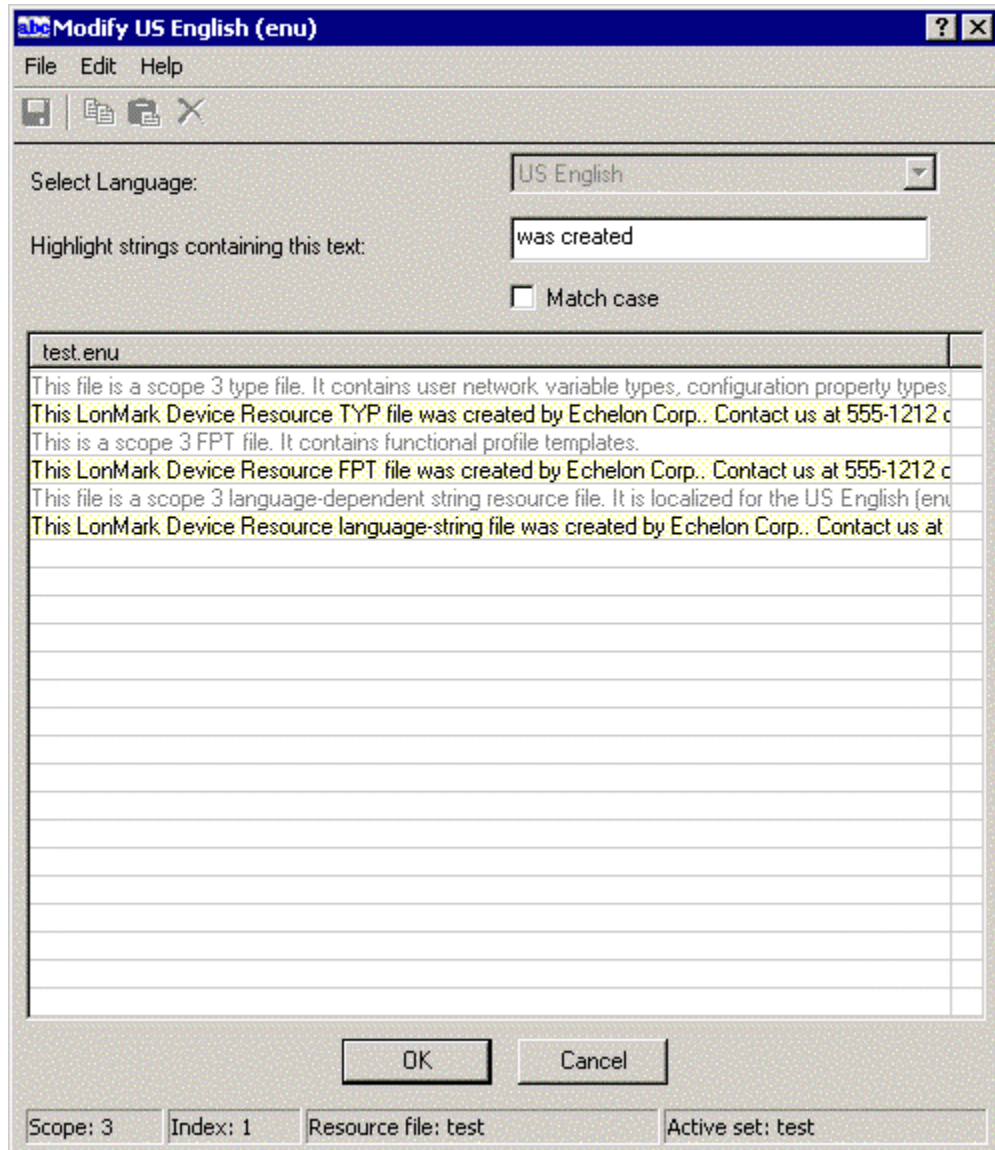
## Searching for a Language String

You can search for a language string within a language file. To search for a language string, follow these steps:

1. In the Resource Editor, right-click a language file and then click **Open** on the shortcut menu. The **Modify** dialog appears as shown in the following figure:

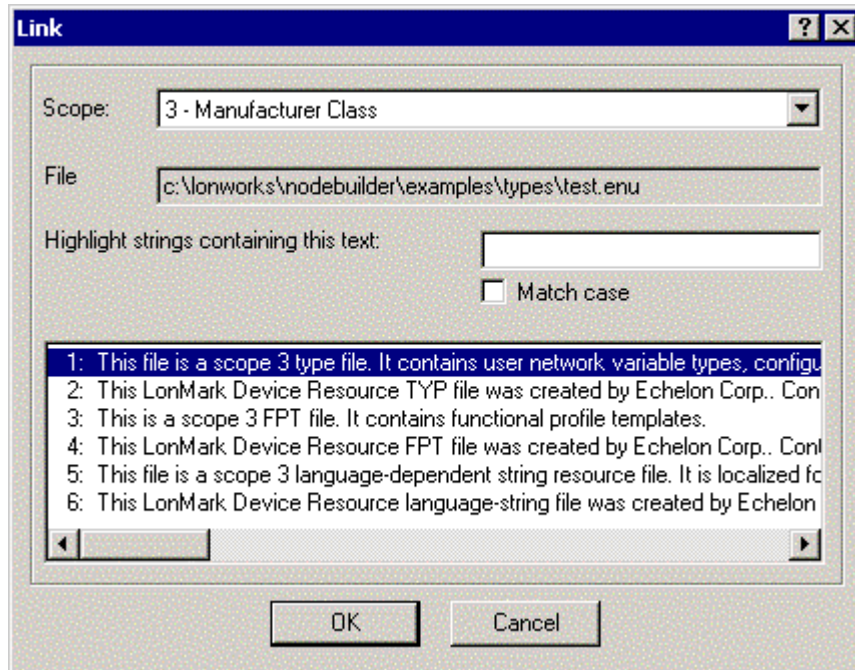


2. Type a string into **Highlight Strings Containing This Text** to have all strings containing the specified string highlighted. Set the **Match Case** checkbox to make the search case sensitive, for example:



You can also search for a language string when using the **Link** button in the **Strings** tab when creating or modifying a network variable or configuration property type, as shown in the following figure:





## Creating, Modifying, and Translating a Language File

You can create a new language file to hold language strings in a new language, you can edit language strings in a language file, and you can translate language strings in a language file to new language strings in a second language file.

To create a new language file in a resource file set, follow these steps:

1. Right-click the **Language Files** folder and then click **New Language File** on the shortcut menu. The **New Language File** dialog appears.
2. Select the language for the new **language** file under **Select Language**.
3. To translate a string as you create the file, double-click the string and enter the translated version based on the selected language. Repeat this step for each string to translate. You can translate strings after you create the file as described later in this section.

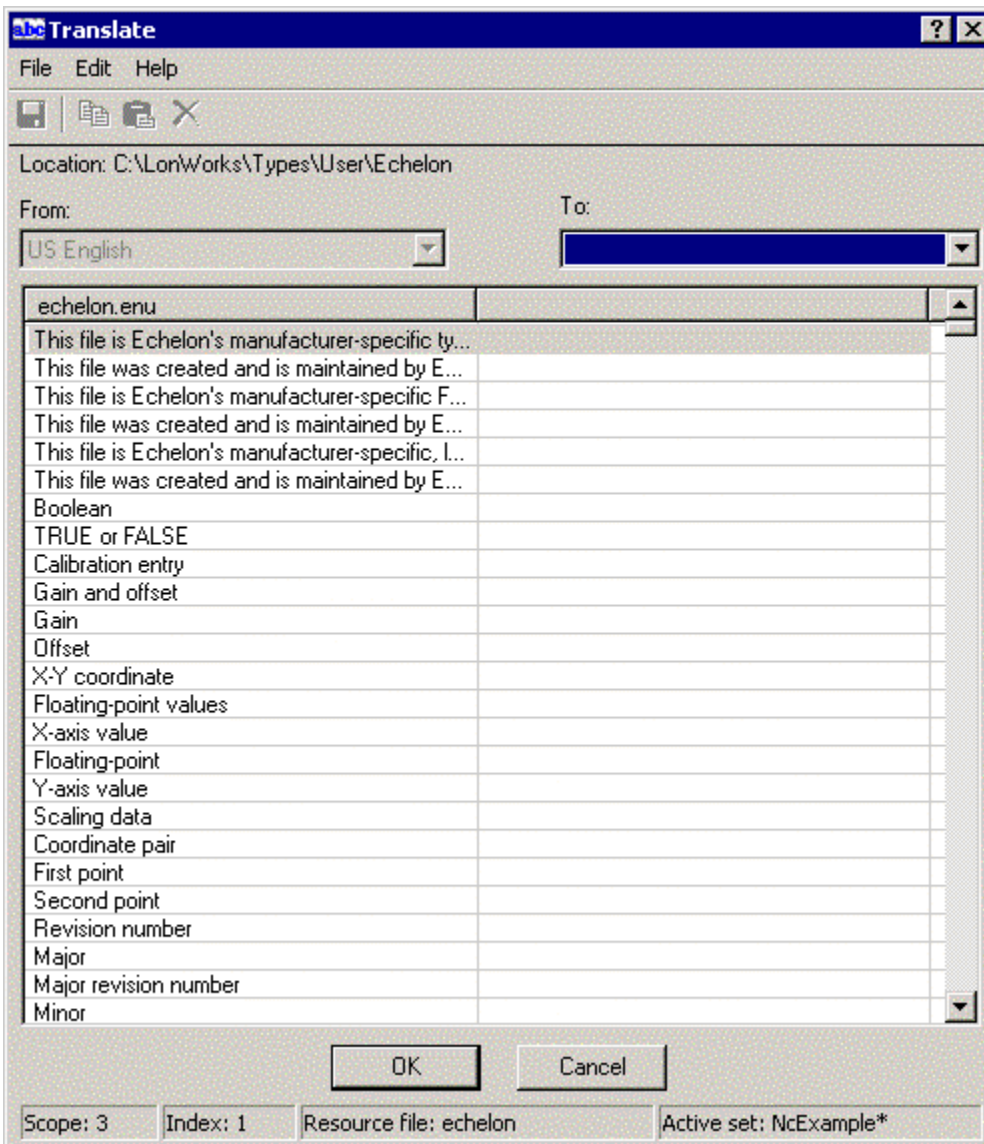
To modify a language file in a resource file set, follow these steps:

1. Set the active language to the language of the language file as described in *Setting Resource Editor Options*.
2. Double-click the language file in the Language Files folder. A dialog appears listing all the strings in the language file. The dialog does not open if you double-click a language file that is not active. To set the active language, see *Setting Resource Editor Options*, earlier in this chapter.
3. To modify a string as you create the file, double-click the string and modify the contents. Repeat this step for each string to modify.

You can view language strings from two language files side-by-side so that you can translate the strings in one file to the other. To translate language files in a resource file set, follow these steps:

1. Expand the **Resource Strings** folder. This folder contains all language files for the resource file set.

- Right-click the active **language** file and then click **Translate** on the shortcut menu. You can change the active language by opening the resource editor **View** menu, clicking **Options**, and changing the value of **Active Language**. See *Setting Resource Editor Options* for information on setting the active language file. The following dialog opens:



This figure shows the American English language resource file for the Echelon resource file set.

- Set **To** to the language you want to translate to.  
 When selecting a language that does not yet have a language file within the current resource file set, the Resource Editor will create generic strings for each language string resource that is listed in the Resource pane. For those strings that are deleted but still listed in the Resource pane, or those that were previously purged from the source language shown in the Resource pane (indicated by a "Purged Record~" placeholder), the Resource Editor will generate a string that is marked deleted (e.g. "String7~"). You can undelete such a string by removing the tilde character, and you can overwrite a

previously purged string (shown with a "Purged Record~" placeholder) by overwriting the placeholder. This feature simplifies synchronization between multiple translations of the same string resources.

4. For each string on the left pane, provide a translation in the selected language in the right pane. Some strings may not have a translation in all languages, in that the text from the source language translates to an empty string. In those cases, delete all text from the translated string. A language string resource may be an empty string. The Resource Editor issues a warning when you save your changes containing an empty string.

You can export the selected language files to text files for use by translation services. To export the selected language files to text files, open the **File** menu and then click **Export to Text**.

5. Open the **File** menu and then click **Save** to save the changes.
6. Open the **File** menu and then click **Exit** to close the Translate dialog.

## Creating and Modifying a Format

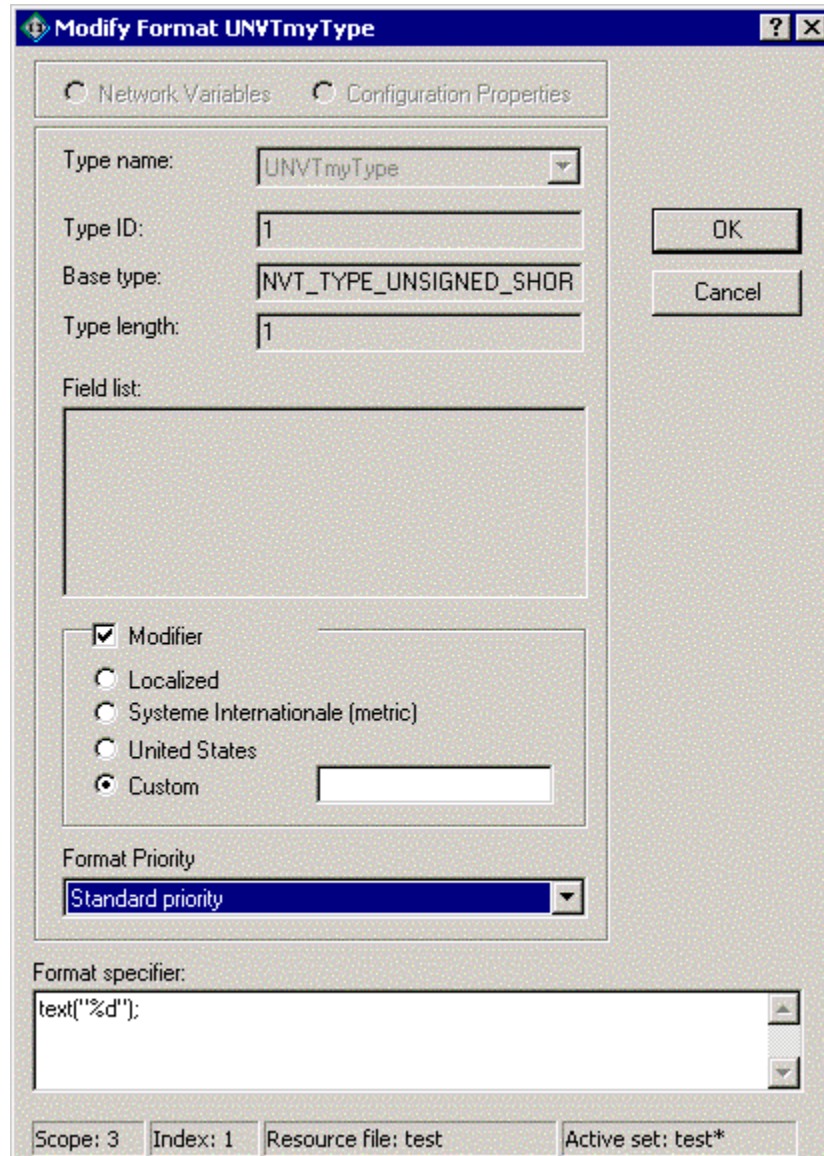
You can create and modify formats for each network variable type or configuration property type. A *format* specifies how a value is to be displayed, printed, or entered. Formats allow the physical representation of data to be independent of how users view the data. This is especially important for any type of measurement data since most measurement types have at least two display formats — one for United States (US) units and one for Système Internationale (SI or metric) units. Formats are also important for data that is viewed differently in different locales. For example, times and dates are displayed differently in different regions of the world. Formats may include locale-specific interpretation of times and dates, using locale information from the user's operating system.

If a format is not available for a network variable or configuration property, most network tools will display the value as raw hex bytes. Formats allow you to customize how network integrators and network operators see the values. When you create a network variable or configuration property type, a default format is created. The default format uses the text format specifier (see *Using The Text Format Specifier*, later in this chapter). In the case of character, short, long, enumeration, float, or quad types, this format will display the raw value. In the case of an array, structure, union, bitfield, or reference type, the format will be set to `Missing format for <TYPENAME>`, where `<TYPENAME>` is replaced by the name of the network variable or configuration property type. If the network variable or configuration property type is a structure or union type that contains more than 127 fields, the resource editor will create a placeholder default format that contains the text `>>Note: This item can not be displayed due to a large number of fields.<<`. You can modify this format to display up to 127 fields of the network variable or configuration property type.

Each format is named with a type name followed by an optional modifier. For example, if you create a network variable type named `UNVT_my_type`, you will have a default format also named `UNVT_my_type`. You can create multiple formats for a type by appending a *modifier* to the additional formats. A modifier is a string that is appended to the format name, following a “#” character. Standard modifiers are defined for SI, US, and localized formats, and you can also create your own modifiers. For example, you can create

UNVT\_my\_type#SI and UNVT\_my\_type#US if you want your type to be formatted differently when displayed in US or SI units.

To create or modify a format in a resource file, expand the **Formats** folder in the resource file set. All formats defined for the resource file set are displayed. There will be a minimum of one format per network variable and configuration property type, but there may be more than one format for some (or all) types. To modify a format, right-click it and then click **Open** on the shortcut menu. To create a new format, right-click the **Formats** folder and then click **New Format** on the shortcut menu. The following dialog opens:



If you are modifying a format, this dialog will be titled **Modify Format** and will contain the selected format.

Enter the following information:

**Network Variables**                      Creates a network variable format.

**Configuration Properties**          Creates a configuration property format.

<b>Type Specification Name</b>	Specifies a type in the resource file set.
<b>Base Type</b>	Displays the base type of the network variable or configuration property type.
<b>Type Id</b>	Displays the index of the selected network variable or configuration property type within the resource file set.
<b>Base Type</b>	Displays the data type of the selected network variable or configuration property type.
<b>Type Length</b>	Displays the length in bytes of the selected network variable or configuration property type, in bytes.
<b>Field List</b>	<p>If the selected network variable or configuration property type contains a structure or union, this box displays the type and name of each field. These names are for use within the <b>Format Specifier</b>.</p> <p>The field list presents a close approximation to a C-language equivalent <code>typedef</code> of the respective network variable or configuration property type, but may not always provide correct ANSI C language syntax.</p>
<b>Modifier</b>	<p>Specifies that one of the following modifiers is used:</p> <p><i>Localized</i> — Time and date formats are determined by the settings on the user's computer. If you are creating a format for a network variable type or configuration property type that contains the time or date, or requires a localized list separator, select this option.</p> <p><i>Système Internationale (metric)</i> — Indicates that this format displays SI units. If a format file contains SI formats, these formats will automatically be selected if the user's computer is configured to use SI units.</p> <p><i>United States</i> — Indicates that this format displays US units. If a format file contains US formats, these formats will automatically be selected if the user's computer is configured to use US units.</p> <p><i>Custom</i> — Specifies a custom format modifier. If this modifier is selected, enter the name in the accompanying field. This modifier can only contain letters, numbers, and the underscore character.</p>

## Format Priority

Specifies whether the format is the default for the type, or the default for the type within a specified measuring system. Select one of the following:

*Standard Priority* — No special priority. This is the default setting.

*Default For This Data Type* — The format is the default for network variables or configuration properties created from this data type. The format name will be preceded with an asterisk (\*) in the format file (.fmt extension).

*Default For This Data Type and Measurement System* — The format is the default for the currently selected measurement system (i.e. SI or US). The format name will be preceded with a plus (+) in the format file.

## Format Specifier

The formatting instructions for this type. There are 4 format specifiers:

*real* — The value will be displayed as a single-precision, 32-bit, IEEE 754 floating point number.

*int* — The value will be displayed as a signed, 32-bit integer.

*discrete* — The value will be displayed as an 8-bit value that contains either 0 or 1 for each bit.

*text(...)* — The text format specifier can be used for data that is not a simple number (enumerations, strings, characters, and structures); for data that must be localized, scaled, or conditionally formatted; or where data formatted as text is preferred. The standard formats defined in `STANDARD.FMT` are all text format specifications, since most network tools are adept at handling text-formatted data, and text-formatted data may be specified for every data type. See *Using the Text Format Specifier*, later in this chapter, for more details.

## Using the Text Format Specifier

The text format specifier has the following syntax: `text (<text format list>)`. The text format list is similar to the ANSI C `printf()` arguments, with some simplifications and extensions. The text format list is a comma-separated list of text formats. Each text format consists of one of the following:

- A quoted string called a *format string*. The format string consists of characters to be included in the formatted output, and may include *conversion specifications* that specify how a corresponding field data argument is formatted. A conversion specification may apply to the entire value to be formatted, or may apply to fields within the value by adding the

field names to the text format list. You can also include localized list separators in format strings. See *Using Conversion Specifications* and *Using Localized List Separators* for more information.

- A field name from the value being formatted. The value must be a structure or union type. Field names are applied to conversion specifications in format specifications that precede the field name in the text format list, applied from left to right. A format can display up to a maximum of 127 fields of a structure or array type. See *Using Conversion Specifications* for more information.
- A conditional format to specify one of two different formats, where one format is selected when a value is formatted based on a conditional value. See *Using Conditional Formats* for more information.
- A scaling factor to specify a multiplier and adder, and an optional unit string suffix, that are used to scale the value to be formatted. A scaling factor may be applied to the entire value, or to an individual field of a structure or union. See *Using Scaling Factors* for more information.
- A localized time or date function. These functions format a time or date according to the user's operating system's locale settings. See *Using Localized Time and Date Formats* for more information.

Following are a few examples from the standard format file (standard.fmt). See the standard format file for more examples.

**Example 1:** A simple integer that does not require localization, with a “%d” conversion specification:

```
SNVT_count: text("%d");
```

**Example 2:** A simple floating point value that does not require localization, with a “%f” conversion specification:

```
SNVT_count_f: text("%f");
```

**Example 3:** A temperature value that must be displayed differently in US, SI, and US differential units, with a “%f” conversion specification and scaling factors:

```
SNVT_temp#SI: text("%f", *1+0(0:854)); ! degrees C  
SNVT_temp#US: text("%f", *1.8+32(0:855)); ! degrees F  
SNVT_temp#US_diff: text("%f", *1.8+0(0:855));
```

**Example 4:** A time that must be localized, with an LO modifier and time() localization function:

```
SNVT_date_time#LO: text(time(hour, minute, second));
```

**Example 5:** A refrigeration type that requires a string, floating-point values, and locale-specific list-separators:

```
SCPTrefrigType#LO:text("%s %f|%f|%f", refrigerant, A, B, C);
```

This format definition displays the refrigerant field as a string, and A, B, and C as floating point values.

**Example 6:** A geographic position that includes conditional text:

```
SNVT_earth_pos#SI: text( ("%d %d ",
    latitude_direction, longitude_direction),
    ( (latitude_direction == 0) ? ("S") : ("N") ),
    (" %d %d ", latitude_deg, latitude_min),
    ( (longitude_direction == 0) ? ("E") : ("W") ),
    (" %d %d %f", longitude_deg, longitude_min,
    height_above_sea) );
```

Following is a formal definition of the text format:

```
<text format group>      = '(' <text format list> ')'  
                        = <text format>  
  
<text format list>     = <text format list> ',' <text format>  
                        = <text format>  
  
<text format>          = '(' <condition> '?' <text format  
                        group> ':' <text format group> ')'  
                        = '(' <text format string> ',' <field  
                        spec list> ')'  
                        = 'time' '(' < field spec string > ','  
                        < field spec string > [',' < field  
                        spec string >]  
                        [',' < field spec string >] ')'  
                        = 'date' '(' <field spec string> ','  
                        <field spec string> ',' <field spec  
                        string> ')'  
  
<condition>           = '(' <field spec string> <conditional  
                        operator> <decimal const> ')'  
  
<conditional operator> = '=='  
                        = '!='  
  
<field spec list>     = <field spec list> ',' <field spec  
                        with modifiers>  
                        = <field spec with modifiers>  
  
<field spec with modifiers> = <field spec with multiplier  
                        and adder> <string resource reference>  
                        = <field spec with multiplier and  
                        adder>  
  
<field spec with multiplier and adder> = <field spec  
                        string> <multiplier> <adder>  
                        = <multiplier><adder>  
                        = <field spec string>  
  
<field spec string>   = <field spec string> '.' <field name>  
                        = <field name>  
  
<string resource reference> = '(' <mode> ':' <index> ')'
```

## Using Conversion Specifications

You can use a *conversion specification* within a format string to specify how the value of a field should be formatted. To format a field, append the field name in the text format list after the format string. Include one field name for each conversion specification in the list. The conversion specifications are



applied to the field names from left to right. You can specify the following conversion specifications

- %c** A single character. The base type in Neuron C must be char, int, or enum.
- %d** A signed or unsigned decimal number (based on the signedness defined in the type file). The base type must be a Neuron C char, int, enum, or long or a structure or array. If it is a structure or an array of at least four bytes in length, it is assumed to be a Neuron C signed 32-bit number of s32\_type.
- %f** A floating point number. The base type must be a structure, an array, or a fixed point Neuron C int or long. If it is a structure or array of at least four bytes in length, it is assumed to be a Neuron C floating-point number of float\_type or SNVT\_xxx\_f type.
- %m** An enumeration. The base type must be an enumerated list. If an enumeration does not exist for the value, the format string is processed as if it were %d.
- %s** A null-terminated string. The base type must be an array of 8-bit data. String data must be null terminated.
- %x** An unsigned hexadecimal integer. The size is determined from the type file. The data are always treated as unsigned. The base type must be char, int, or long. If it is a structure, or an array of at least four bytes in length, it is assumed to be a Neuron C signed 32-bit number of s32\_type.

You can use a backslash (“\”) character as an escape character to include other format characters as text. For example, the following characters can be included in a format string:

- `\%` The % character.
- `\\` The \ character.
- `\"` The " character.
- `\\|` The | character.

## Using a Conditional Format

You can use a conditional format to specify one of two different formats, where one format is selected when a value is formatted based on a conditional value. The syntax for a conditional format is similar to the ANSI C “?:” conditional expression. The syntax is as follows:

```
<condition> ? <format if condition is true> : <format if condition is false>
```

The condition is limited to expressions with the equal to (“==”) and is not equal to (“!=”) comparison operators.

The field that appears in the conditional statement must appear in a text format list *before* it appears in the conditional statement. Formats are processed in left-to-right order.

Following is an example of a format definition with conditional format specifiers extracted from the SNVT\_earth\_pos#SI format definition (much of the format definition has been deleted for simplification):

```
UNVT_DM_Command: text( ("%m ", cmd),  
    ((cmd == 1) ? ("%d", cmdData.databaseId) :
```

```

    ((cmd == 2) ? (" ") :
    ((cmd == 3) ? ("%d", cmdData.  deviceIndex) :
    <additional conditions deleted>
) ) ) );

```

## Using Scaling Factors

You can use a scaling factor within a format string to specify a multiplier and adder, and an optional unit string suffix, that are used to scale the value to be formatted. You can scale any simple data type, and you can also scale any field in a structure or union that is a simple data type. The scaling factors are applied as a multiplication and an addition when data is converted for output, and they are applied in the reverse order, as a subtraction and a division when data is input.

You can also specify a scope and language string index that specifies a language string to use as the unit description. This string overrides the unit description string found in the type file.

Alternate formats with scaling factors can be used for converting units to the United States (US) or other measurement systems.

The syntax for a scaling factor is as follows:

```
*<Multiplier>+<Adder>[(<Unit Description Scope>:<Unit
Description Index>)]
```

Following are example formats using scaling factors.

**Example 1:** The following lines define SI and US formats for the SNVT\_temp\_f standard network variable type:

```
SNVT_temp_f#SI:    text("%f", *1+0(0:854));    ! degrees C
SNVT_temp_f#US:    text("%f", *1.8+32(0:855)); ! degrees F
```

The SI format multiplies the value by 1 and adds 0 (i.e. shows the raw value) and appends “degrees C” (scope 0, string index 854). The US format multiplies the value by 1.8 and adds 32 and appends “degrees F” (scope 0, string index 855).

**Example 2:** The following lines define the SI and US formats for the SCPTsetPnts standard configuration property type:

```
SCPTsetPnts#SI:    text("%f,%f,%f,%f,%f,%f",
                    occupied_cool, standby_cool, unoccupied_cool,
                    occupied_heat, standby_heat, unoccupied_heat);
SCPTsetPnts#US:    text("%f,%f,%f,%f,%f,%f",          ! degrees F
                    occupied_cool*1.8+32(0:855),
                    standby_cool*1.8+32(0:855),
                    unoccupied_cool*1.8+32(0:855),
                    occupied_heat*1.8+32(0:855),
                    standby_heat*1.8+32(0:855),
                    unoccupied_heat*1.8+32(0:855));
```

## Using Localized List Separators

You can include a locale-specific list-separator character in a format string. To do this, specify a localized (“#LO”) modifier and include a vertical bar (“|”) where you want the list separator in the format string. The vertical bar is translated to the operating system list-separator character for the current operating system default locale. The current setting of the Windows list-

separator character may be found in the List Separator setting on the Number tab of the Regional Options in the Windows Control Panel. The list-separator character can only be used with localized alternate formats, as described under *Using Localized Time and Date Formats*.

## ***Using Localized Time and Date Formats***

You can include time and date localization functions to format a time or date value as specified by the operating system default locale method. The date format specifier requires three parameters, which specify the data fields where it will find the year, month, and day values to be formatted. The time format specifier requires two to four parameters, specifying hour and minute values to be formatted, and optionally, second and millisecond values.

For the Windows operating system, the current setting of the date format may be found under Short Date Style on the Date tab of Regional Settings in the Windows Control Panel. The current setting of the time format may be found under Time Style on the Time tab of the Regional Settings, with the following exceptions:

1. The time format specifier does not support AM/PM time formats, so this type of time format will be converted to a 24-hour format.
2. The time format specifier supports display of milliseconds, which is not defined in Windows time styles. If supplied, the milliseconds field will be appended to the seconds field, and separated from the seconds field by the Decimal Symbol character from the Number tab of the Regional Settings.

The time and date format specifiers may only be used in localized formats (formats with the “#LO” modifier).

Following are examples of the time and date localization functions.

**Example 1:** A time format specifier from the SCPTmaxSntT#LO format definition:

```
SCPTmaxSndT#LO:  text((" %d ", day),  
                  time(hour, minute, second, millisecond));
```

**Example 2:** A date format specifier from the SNVT\_date\_cal#LO format definition:

```
SNVT_date_cal#LO: text(date(year, month, day));
```

---

## **Copying Resources**

You can copy any resources in the resource catalog. You can copy resources to a new resource file set, or copy them within the same resource file set if you created the original resource file set. To copy a resource, follow these steps:

1. Right-click the resource to be copied and then click Copy on the shortcut menu.
2. Right-click the folder that will contain the copied resource and then click Paste on the shortcut menu.
3. If you are copying a functional profile, the resource editor will attempt to use the same functional profile number (key) for the new profile. If the profile number is already in use, you are given the option of overwriting the existing profile or using a different profile number.

Resources may reference other resources in the same resource file set or in resource file sets with numerically lower scopes and compatible program ID templates. If you copy a resource to a new resource file set, some of the references may become invalid. If the resource editor determines that a reference may be invalid, it removes the index number from the reference. This gives you an opportunity to find the invalid references and correct them. To fix an invalid reference, first ensure that the referenced item is available within the new resource file set or within a resource file set with a numerically lower scope and compatible program ID template. Then change the invalid reference to the new resource.

If you are making additions to a standard functional profile, create a new functional profile that inherits from the standard profile instead of copying and modifying the profile. This enables your new profile to stay consistent with any changes to the original profile. If you create a new profile by copying and pasting an existing profile, any changes to the original profile that are made after you make the copy will not be reflected in your new profile. See *Creating and Editing a Functional Profile* for more information.

---

## Removing and Obsoleting Resources

Resources may reference other resources in the same resource file set or other resource file sets, so deleting a resource could impact other resources, even in resource file sets that you may no longer have loaded on your computer. Applications can also reference resources, and it is important that an invalid resource not be passed to an application because the original resource was replaced by another. To help prevent references to non-existent resources or invalid resources (due to reuse of a deleted index), the resource editor does not allow you to delete an individual resource. It instead provides two alternatives: you can either mark a resource as deleted (and later purge it from the resource file), or mark it as obsolete.

During development, you can delete resources that you have defined, but not yet released to production. To mark a resource as deleted, right-click it and then click **Remove** on the shortcut menu. Alternatively, you can select the resource and then press **DEL**. This does not physically delete the resource from the resource file. Instead, a tilde (“~”) character is appended to the name to indicate that the resource has been removed. By default, the resource editor does not display resources with a tilde as the last character, so the resource will appear to be deleted. You can see any resources that you have removed by opening the **View** menu, clicking **Options**, and then setting **Show Removed Resource Items**. You can undelete a removed resource by first showing removed resource items, and then deleting the tilde from the resource name. To remove a deleted resource, you can purge removed resources from the resource file set as described in *Purging a Resource File Set*, later in this chapter.

You should not delete any resources that you have released to production. You may have users that have created devices based on those resources, created resource files that reference those resources, or created applications that use those resources. However, you may decide that a resource that you have created should no longer be used for new designs, even if it is used in existing designs. In this case you can mark the resource as *obsolete*. This tells your users that they should no longer use the resource in new designs,

but allows the resource to continue to be used in existing designs. For example, the SNVT\_lev\_disc type is marked as obsolete in the standard resource file set because it has been replaced by SNVT\_switch. The SNVT\_lev\_disc type continues to be used in many devices, but newly designed devices should use SNVT\_switch instead.

To mark a resource as obsolete, double-click the resource and then set **Mark this item obsolete**. You can specify whether you want to see obsolete resources by opening the **View** menu, clicking **Options**, and then setting or clearing **Show Obsolete Resource Items**. You can remove the obsolete flag from a resource by clearing **Mark this Item Obsolete** for the resource.

---

## *Purging a Resource File Set*

You can purge a resource file set. When you *delete* a resource in the Resource Editor, it is marked as deleted, but it is still physically in the resource file. This helps prevent serious problems that could occur if you had other resources referencing the deleted resource. This is most important once you have started shipping a device. You should never delete resources used by devices in the field, though you can mark them as obsolete. However, during development, you may create resources that you decide never to ship. In this case, you may prefer that these deleted resources not remain in your resource file. In this case, you can *purge* the resource file. Purging physically removes all deleted resources from the resource file. You must be careful not to purge a resource file that contains deleted resources that are in use by devices that you have shipped, and you must also be careful not to purge a resource file set that contains deleted resources that are referenced by other resources.

To purge a resource file set, follow these steps:

1. Close all applications that may be using the resource files to be purged. This includes the NodeBuilder and LonMaker tools.
2. Click the Windows **Start** button, point to **Programs**, point to **Echelon NodeBuilder Software**, and then click **Resource Converter Utility**. The Resource Converter opens.
3. Select the **Resource File Set to be Converted**.
4. Select which files in the resource file set you want to purge by setting the **Convert** option for each resource file type to be purged.
5. Set **New File Version** to the latest version for each selected resource file type.
6. Set **Purge Deleted Items** for each selected resource file type.
7. Select an output folder for the purged resource file set, or set the **Replace** checkbox. If you replace the resource file set, the utility will automatically create a backup folder and a backup copy of the original, un-purged, resource file set, for you. This folder is named "Backups" and is a subfolder to the location that contains the original resource files.
8. Click **Convert**.

## Converting a Resource File Set

You can convert the file format of a resource file. This may be necessary to generate a resource file using an older file format for compatibility with a tool or device that does not support the current resource file formats. For example, the resource files generated by NodeBuilder 3.1 can only be read by tools or devices that are based on version 2.3 or newer of the Resource File API, or by tools or devices that have a compatible API. LNS tools such as the LonMaker tool can be upgraded by installing the latest version of the Resource File API. Updates may be available from tool or device manufacturers that install the new API, as well as on the LONMARK website ([www.lonmark.org](http://www.lonmark.org)). You can convert the file format of a resource file set to provide compatibility with older tools or devices that have not been upgraded. The following table lists the file formats that have been defined for each of the types of resource files:

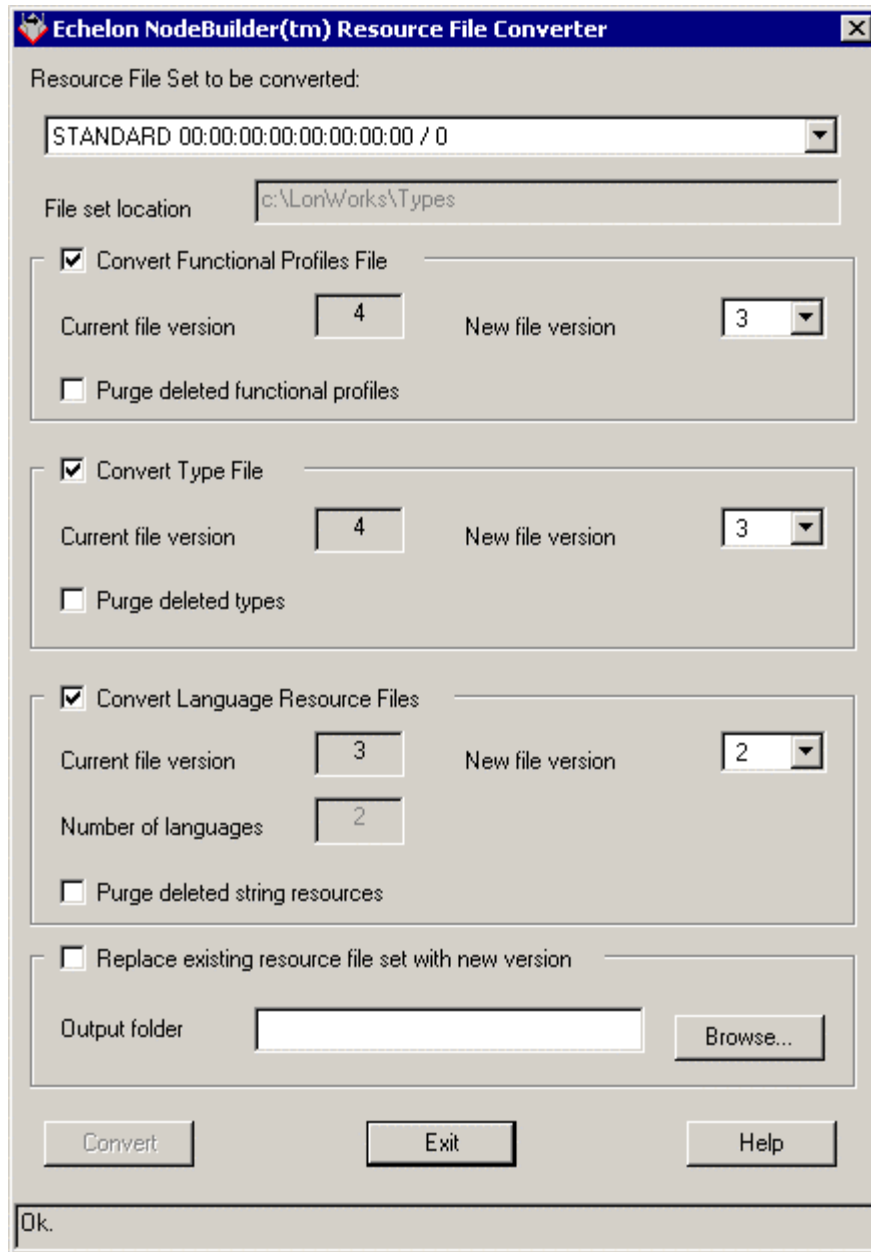
<b>File Type</b>	<b>Format Version</b>	<b>Format Changes</b>	<b>Minimum Required Resource File API</b>
<b>Functional Profile</b>	1	Initial release.	1.0
	2	Added support for larger profiles and for marking profiles as obsolete.	2.0
	3	Added support for inheriting profiles and for non-contiguous member numbers.	2.1
	4	Added support for configuration property arrays and for deleting profiles.	2.2
<b>Format File</b>	1	Initial release.	N/A
	2	Added support for scale factors.	N/A
	3	Added support for language localization.	N/A
<b>Language File</b>	1	Initial release.	1.0
	2	Added support for larger language files.	2.0
	3	Added support for deleting language strings.	2.2
<b>Type File</b>	1	Initial release. Included NVTs only.	N/A

<i>File Type</i>	<i>Format Version</i>	<i>Format Changes</i>	<i>Minimum Required Resource File API</i>
	2	Added CPTs and enumeration types.	1.0
	3	Added support for invalid values and for marking types as obsolete.	2.1
	4	Added support for configuration property arrays and for deleting types.	2.2
	5	Added support for unsigned quad and double float	2.3

Converting a resource file set can result in a loss of data that was introduced in later resource file formats. Always save a copy of your resource file set prior to converting it to an older file format.

To convert the format of a resource file, follow these steps:

1. Click the Windows **Start** button, point to **Programs**, point to **Echelon NodeBuilder Software**, and select **Resource Converter Utility**. The Resource Converter opens, as shown in the following figure:



2. Select the **Resource File Set to be Converted**.
3. Determine which files in the resource file set you want to convert. Set **Convert Functional Profiles File**, **Convert Type File**, and **Convert Language Resource Files** as desired.
4. For each file type you choose to convert, set **New File Version**. See the table above for a summary of the characteristics of older file format versions.
5. If you wish to replace the existing resource file set, set **Replace Existing Resource File Set With New Version** (save a backup copy of the resource file set before using this option). If you wish to save the converted resource file set to a different location, clear this option and enter a folder name in **Output Folder** (click the **Browse** button to



browse to a location). If you choose to replace the existing set, a sub-folder named **Backup.ResConv** will be created and the old version of the resource file set will be saved there. The backed up files will get a `.v $x$`  suffix, where  $x$  is the format version number (e.g. a version 2 file would get a `.v2` suffix).

#### 6. Click **Convert**.

You may choose to convert and purge a resource file set at the same time. In this case, the source file will be purged, and then converted. You may also choose to convert a resource file without changing the format version (e.g., converting a type file from version 4 to version 4). Such a conversion enables some housekeeping work and error-checking to be performed within the resource file, and may result in a resource file of a slightly different size.

When converting a resource file set to an older format, advanced features and related data will be removed from the set. This includes previously purged resources; although purging of deleted resources actually removes the resource from the file set, a non-continuous sequence of resource indices results. Only the more recent resource file formats support this case; for older file formats, the gaps in the sequence of consecutive indices must be filled. The Resource Converter does this by creating “dummy” resources as needed, and marking them as deleted at the same time. Thus, file format conversion might seem to “unpurge” previously purged resources, however, this is not the case. The resources that are created in the empty indices during the conversion will be of a simple type, and will not have any any properties of the original, purged, resource.

---

## Viewing Resource File Properties

You can see a summary of many of the items in the resource catalog by right-clicking them and then clicking **Properties** on the shortcut menu. A window appears showing information about the resource that was selected. You can view the following properties:

### **Catalog**

Right-click the resource catalog file at the top of the resource catalog, and then click **Catalog Properties** on the shortcut menu to display a window showing the number of directories, number of type files, number of functional profile files, number of format files, and number of language files contained in the resource catalog.

### **Resource File Set**

Right-click a resource file set and then click **Properties** on the shortcut menu to display a window showing the header information for the resource file set, as well as the header information for each type file, functional profile file, format file, and language file contained in the resource file set.

### **Network Variable Types, Configuration Property Types, Functional Profile Templates, Enumerations,**

Right-click a resource file folder and then click **Properties** on the shortcut menu to display a window showing the header of the resource file set and the header or headers of the files that contain

**Language Files, and Format Files** the definitions displayed in the selected folder.

---

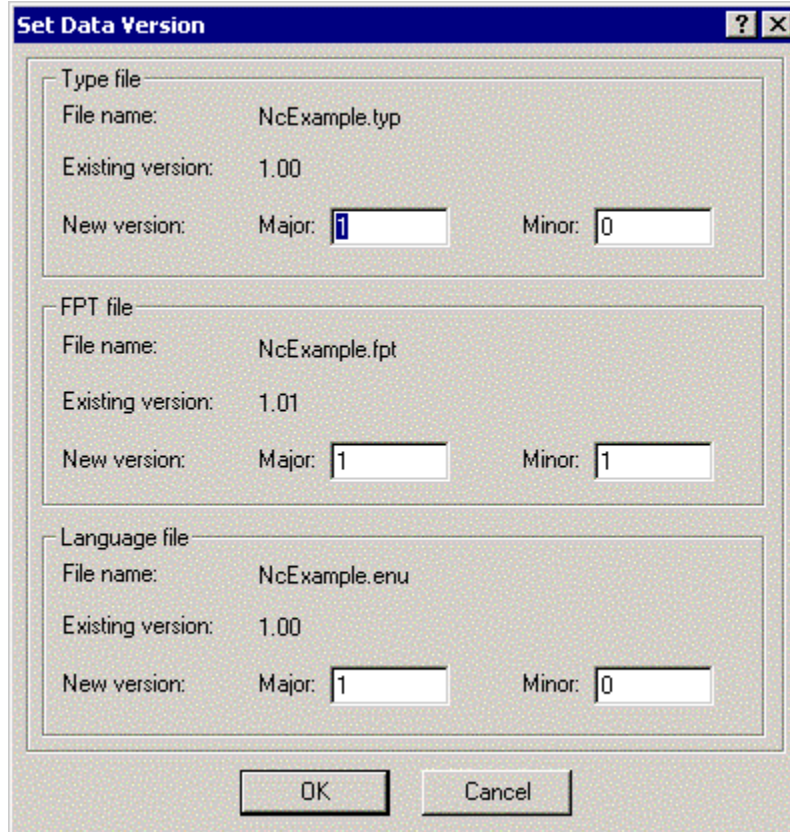
## Generating Resource Files

You can generate a resource file set at any time while editing the resource file set. If you have made any changes to a resource file set, you must generate the new resource file set before exiting the resource editor, otherwise your changes may be lost.

You can only make changes to one resource file set at a time. Once you have made any changes to a resource file set, it becomes the *active set*. The active set is shown in the status bar at the bottom of the Resource Editor window. If you have made any changes to the active set, the name of the active set is followed by an asterisk (\*) to indicate that you need to generate resource files. Once you have made any changes to a resource file set, it becomes the active set and you can only make changes to the active set. If you attempt to make changes to another resource file set, you will be given the option to either generate the resource file set for the active set, or cancel changes.

To generate a resource file set, follow these steps:

1. Right-click the active set in the resource catalog and then click **Generate Resource Files** on the shortcut menu. A dialog opens listing the files that will be generated (only files that have had changes made will be generated).
2. Click **Set Version** to set the version of the resource files to be generated. The following dialog opens:



This dialog displays the current versions of the type file, functional profile file, and the language files (**File Name** for the language file will contain the name of the language file for the language currently selected as the active language; see *Setting Resource Editor Options*, earlier in this chapter, for more information). The format file does not contain versioning information and is not listed.

3. Set the major and minor version numbers for each of these files. The files may have differing minor versions, but use the same major version number for all files in a resource file set to simplify configuration management. Once you have set the version information, click **OK**. You will be returned to the resource file generation confirmation dialog.
4. Click **Yes** to generate resource files. The new resource files will be placed in the directory indicated by the resource file catalog. If you do not wish to generate resource files at this time click **Cancel**. If you wish to revert to the most recently generated version of the resource files, click **No** (this will cancel all changes you have made using the resource editor since the last time resource files were generated).

---

## Resource Reports

You can create a resource report that contains a summary of all the resources in a resource file set, or in multiple resource file sets. You can use a resource report during development as a reference guide for your resource definitions. You can also define supplementary documentation that is automatically included in your resource report. See [types.lonmark.org](http://types.lonmark.org) and <http://types.echelon.com> for two examples of resource reports.

**WARNING:** The resource report generator is included as an unsupported component of the NodeBuilder 3.1 product. It has not undergone the same level of testing as the remainder of the NodeBuilder tool. However, you may find it to be a useful aid to your product development.

To start the *Resource Report Generator*, right-click the resource you wish to report and then click **Report** on the shortcut menu.

See the *Resource Report Generator User's Guide* for more information on creating resource reports, available from the Windows **Start** menu under Echelon NodeBuilder Software.



# 8

## Editing Neuron C Source Code

This chapter describes how to use the NodeBuilder editor to edit, search, and bookmark Neuron C code.

---

## Introduction to Editing

You can display and edit source and text files using the NodeBuilder Project Manager; this includes Neuron C files (.nc extension), header files (.h extension), C files (.c extension), and text files (.txt extension). You can open any file in a device template folder or device template **Source Files** folder by double-clicking it. You can open multiple files in the Edit pane. You can switch between open files using the **Window** menu.

You can cut, copy, and paste text using standard Windows commands. For example, you can cut selected text using CTRL+X, the **Cut** button on the toolbar, or by clicking **Cut** on the Edit menu. You can search for a text string in a single source files or in all source files in the project as described in *Searching Source Files*, later in this chapter. The Edit pane automatically highlights source code based on Neuron C syntax as described in *Using Syntax Highlighting*. You can return to frequently used parts of your code as described in *Using Bookmarks*.

The editor includes a command that allows you to find a matching bracket, parenthesis, or brace. Place your cursor in front of or select any of the following characters: [ ] { } ( ), and press Ctrl-]. Your cursor will jump to the matching bracket, parenthesis, or brace, if it exists.

---

## Using Syntax Highlighting

If you are editing a Neuron C (.nc extension), header file (.h extension), or C (.c extension) file, the Edit pane automatically color-codes text based on Neuron C syntax. This color-coding is designed to make your Neuron C code easier to read. You can change these colors using the editor options (see *Setting Editor Options*, later in this chapter). The default colors and their significance are shown below:

<b>Green</b>	Green indicates that the text is part of a Neuron C comment. Commented text is not compiled during a build.
<b>Blue</b>	Blue indicates that the text is a Neuron C language specific keyword or function.
<b>Pink</b>	Pink indicates that the text is a string or number. This includes the arguments to #include statements and numerical values assigned to variables.
<b>Dark Blue</b>	Dark blue indicates that the text is a constant or preprocessor directive.
<b>Grey</b>	Grey indicates that the code is generated and updated by the NodeBuilder Code Wizard. See <i>Modifying Code Generated by the Code Wizard</i> in the <i>Generating Neuron C Code Using the Code Wizard</i> chapter for more information.
<b>Black</b>	All text that does not fit one of the above classifications is black.

---

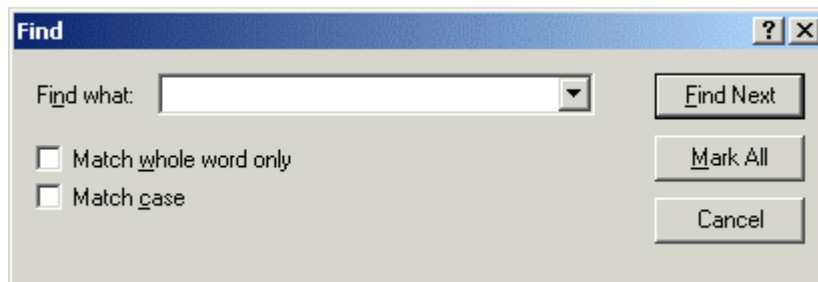
## Searching Source Files

You can search for a string in a single source file or multiple source files, or you can search for a string and replace it with another.

### Searching a Single File for a String

You can search a single file for a text string. To search for a text string, follow these steps:

1. Open the file that you want to search in the NodeBuilder Project Manager. Click anywhere in the file.
2. Type CTRL+F, or open the Edit menu and then click **Find**. The following dialog opens:

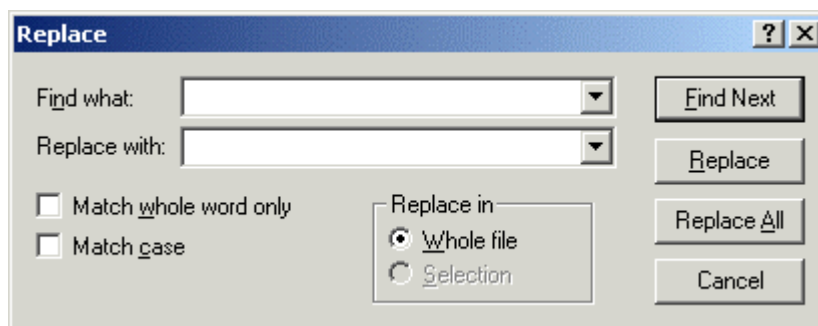


3. Enter the text string to search for in **Find what**.
4. Set **Match whole word only** to find only whole words that match the string. Set **Match case** to make the search case sensitive.
5. Click **Find Next** to find the next occurrence of the string, starting from the current cursor position and moving down. Click **Mark All** to add a bookmark to every line in the file containing the string (see *Using Bookmarks*, later in this chapter, for more information).

### Replacing Text

You can search for a string and automatically replace it with another string. To search and replace, follow these steps:

1. Open the file that you want to search in the NodeBuilder Project Manager. Click anywhere in the file.
2. Type CTRL+H, or open the Edit menu and then click **Replace**. The following dialog opens:



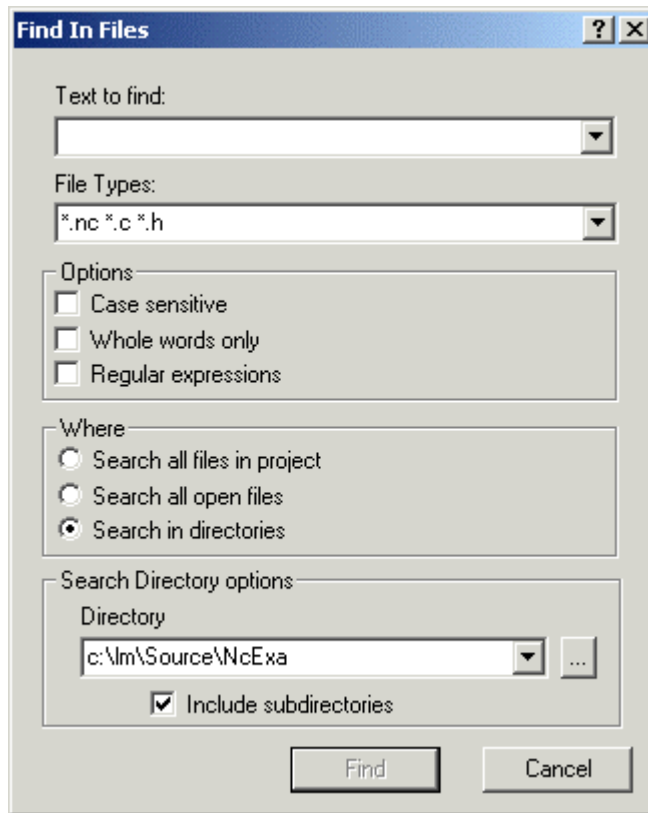
3. Enter the text string to search for in **Find what**.
4. Enter the text string that you want to replace it with in **Replace with**.

5. Set **Match whole word only** to find only whole words that match the string. Set **Match case** to make the search case sensitive.
6. If you selected text prior to opening this dialog, set **Selection** to search only the selected text for the string. Set **Whole file** to search and replace in the entire file.
7. Click **Find Next** to find the first instance of the string. It will be selected and this dialog will remain open.
8. Click **Replace** to replace the selected string with the string in **Replace with**. Click **Replace All** to automatically replace all the selected strings without confirmation.

## Searching Multiple Files for a String

You can search for a string in multiple source files at once. You can use this capability to find all calls of a certain function or uses of a certain variable in an entire project. To search for a string in one or more files, follow these steps:

1. Type CTRL+SHIFT+F, or open the Edit menu and then click **Find in Files**. The following dialog opens:



This dialog contains the following fields:

<b>Text to find</b>	The text string to search for.
<b>File Types</b>	The file types to be searched. The type of a file is determined by its file extension. By default, the search will look in Neuron C files (.nc extension), C files (.c extension), and header files (.h extension). You can remove a



file type from the search by removing the corresponding \*.<file type extension> entry. You can add additional file types by adding \*.<file type extension> to this field.

**Case sensitive**

Performs a case-sensitive search.

**Whole words only**

Limits the search to whole words that match the search string.

**Regular expressions**

Enables regular expression syntax in the search string. If this option is set, you can use the following expressions in your search string:

\* – An asterisk in the search string replaces zero or more characters. An asterisk must be accompanied by at least two other characters (i.e. you could search for zo\*, which would find instances of zo, zoo, zoom, zoot, but not z\*). Use \\* to represent an asterisk character.

+ – The plus sign behaves just like the asterisk, but it must replace at least one character (i.e., if you search for zoo+, it will return zoot and zoom, but not zoo. Use \+ to represent a plus character.

? – The question mark replaces one or zero characters. The search must contain at least two other characters. Use \? to represent a question mark character.

. – The period replaces exactly one character. The search must contain at least two other characters. Use \. to represent a period character.

(pattern) – Matches the pattern and remembers the match. The matched substring can be retrieved by using '\0'-'\9' later in the regular expression, where '0'-'9' are the number of the pattern. Example: regular expression (re).\*\0s+ion will match regular expression. First the search matches re string and stores that pattern with index 0. .\* will match gular exp in **regular expression**. The \0 expression retrieves the pattern with index 0 (i.e. re). This re matches the re in **expression**. Finally the s+ion expression matches ssion.

`x|y` – Matches either character `x` or `y`. You can combine more than two characters like `x|y|z`.

`{n}` – The preceding character must match exactly `n` times. For example `bo{2}k{2}e{2}per` would match `bookkeeper`. `n` must be a positive integer.

`{n,}` – The preceding character must match `n` or more times (i.e. `bo{2,}k{2,}e{2,}per` would find instances of `bookkeeper`, `boookkeeeper`, etc.). `n` must be a positive integer.

`{n,m}` – The preceding character must match between `n` and `m` times. `n` and `m` must be positive integers, and `m` must be greater than `n`.

`[xyz]` – Matches any of the enclosed characters. `[xyz]` produces identical results to `x|y|z`.

`[^xyz]` – Matches any character other than the enclosed characters.

`\b` – Matches a word boundary.

`\B` – Matches anything other than a word boundary.

`\d` – Matches any numerical digit (0-9).

`\D` – Matches any non-digit.

`\f` – Matches a formfeed.

`\n` – Matches a new line character.

`\s` – Matches any white space character.

`\S` – Matches any non-white space character.


`\t` – Matches any tab character.

`\v` – Matches any vertical tab character.

`\w` – Matches any letter, number, or underscore.

`\W` – Matches anything other than a letter, number or underscore.

`\<num>` - Where `<num>` a number from 0-9. Matches indexed pattern (see, `(pattern)`, above).

	/n/ - Where n is any number from 1-255. Matches the character with the ASCII value n.
<b>Where</b>	Determines which files to search. Choose one of the following options:  <i>Search all files in project</i> –Searches all files in the current NodeBuilder project.  <i>Search all open files</i> –Searches all currently open files. Open the Window menu to see which files are currently open.  <i>Search in directories</i> –Search all files in a specific directory.
<b>Directory</b>	Specifies the directory to search if <i>Search in directories</i> is selected. Choose a directory to search. The NodeBuilder project directory will be selected by default. Click the  button to browse to a different directory.
<b>Include subdirectories</b>	Searches all subdirectories of the selected directory if <i>Search in directories</i> is selected.


2. Set your desired search parameters, and then click **Find**. The **Search Results** tab of the Results pane displays the results of the search. Each instance of the string results in a line in the Results pane that lists the file, line number, and line text where the string was found. Double-click a line in the Results pane to open the specified file and go to the specified line.

---

## Using Bookmarks

You can flag lines of code in you source and text files using *bookmarks*. You can use bookmarks to easily return to important sections of your source or text files. You can set bookmarks manually or as a result of a search (see *Searching Source Files*, earlier in this chapter).

To manually set or remove a bookmark, follow these steps:

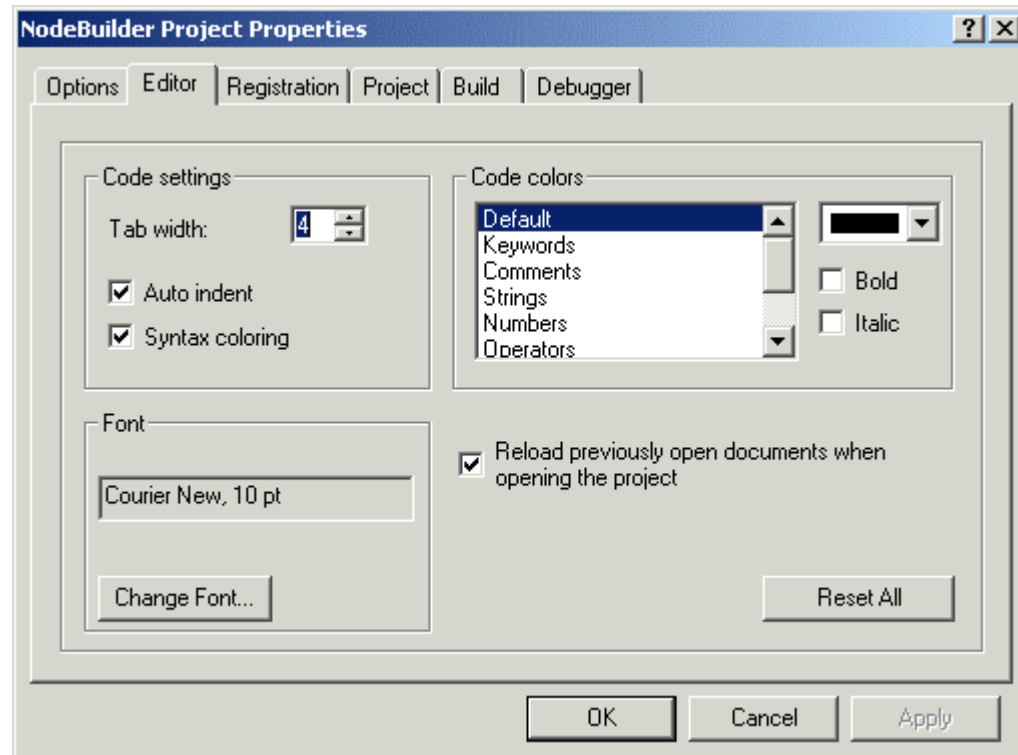
1. Open the file that you want to search in the NodeBuilder Project Manager.
2. Place the cursor on the line you want to bookmark, or on the line containing the bookmark you want to remove.
3. Open the **Edit** menu, point to **Bookmarks**, and then click **Toggle Bookmark**. If the line does not contain a bookmark, a bookmark symbol () appears to the left of the line. If the line already contains a bookmark, it is removed.


Once you have set any bookmarks in a file, you can go to the next bookmark in the file. To go to the next or previous bookmark, open the **Edit** menu, point to **Bookmarks**, and then click **Next Bookmark** or **Previous Bookmark**.

To remove all bookmarks from the current source file, open the **Edit** menu, point to **Bookmarks**, and then click **Clear All Bookmarks**.

## Setting Editor Options

You can set editor options that control syntax highlighting, tab settings, auto indent, font settings, and automatic loading. To set editor options for the current NodeBuilder project, open the **Tools** menu and then click **Options**. The **NodeBuilder Project Properties** dialog opens with the **Editor** tab selected. This tab appears as shown in the following figure:



You can also open this dialog by selecting **Settings** from the **Project** menu or by clicking the project settings button () on the toolbar.

This tab contains the following fields:

- |                        |  |
|------------------------|--|
| <b>Tab Width</b>       | Determines the tab size. By default, the tab size is 4.  |
| <b>Auto Indent</b>     | Automatically indents code inside a function or conditional statement.   |
| <b>Syntax Coloring</b> | Enables syntax highlighting. You can specify colors in <b>Code Colors</b> .  |
| <b>Font</b>            | Sets the font and font size used to display text in the editor. Click <b>Change Font</b> to choose a new font or font size. You may only choose from fixed width fonts.                      |
| <b>Code Colors</b>     | Sets the colors used by the editor when <b>Syntax Coloring</b> is set. You can choose different colors for keywords, comments, strings, numbers, operators, code wizard maintained code, and |

preprocessor statements, as well as the default color for code that doesn't fit into any of these categories. Select one of these categories and then choose a color using the color picker. You can also make the specified text bold or italic by setting **Bold** or **Italic**.

**Reload previously open documents when opening the project** Opens all documents that were open the last time you closed the project when you open a project.

**Reset All** Resets all options on this tab to their defaults.



# 9

## **Compiling, Building, and Loading Applications**

This chapter describes how to use the NodeBuilder tool to compile Neuron C source code, build an application image, and load that image into a device.

---

## Building an Application Image

You can build an application image for a target, each target in a device template, or for all targets with qualifying device templates in a NodeBuilder project. When you build an application image, the NodeBuilder tool compiles the source code specified by the device template, links the compiled code with the standard libraries and any libraries that you specify in the device template, optionally creates a loadable application image, optionally creates a ROM image, and also creates device interface files required by network tools such as the LonMaker tool. The next section describes the files that are produced when you build a device template. These files are placed in the Development or Release folder in the device template's Output folder (see the *New Device Template* window of the Device Template Wizard).

To build to all targets in a NodeBuilder project, follow these steps:


1. Close the LonMaker browser if it is open.
2. Close any other LNS applications that are monitoring the device, if any.
3. Open the project in the NodeBuilder Project Manager.
4. Open the **Project** menu and then click **Build All**. This builds all non-excluded targets in the project (see *Excluding Targets from a Build*, later in this chapter, for information on excluding targets). Or, click **Build All Unconditionally** to automatically clean each target before it is built (see *Cleaning Build Output Files*, later in this chapter, for more information).

To build an individual device template, follow these steps:

1. Close the LonMaker browser if it is open.
2. Open the project in the NodeBuilder Project Manager.
3. Right-click a device, device template, Development folder, or Release folder in the Project pane and then select **Build** from the shortcut menu.

The NodeBuilder tool automatically saves all unsaved project files when you start a build. If **Prompt before saving files** is set in the *Options* tab of the NodeBuilder Project properties, you will be asked if you want to save changes or cancel the build when there are unsaved changes.

When a device is being built, the results of the build are displayed in the **Messages** tab of the Results pane. This pane displays Neuron C errors, linker errors, warnings, and build status. You can double-click an error or warning to be taken to the line of code that generated the message. The information displayed during a build is also saved in a log file (.log extension) in the **Development** or **Release** target subfolder of the device template's output directory. To stop a build in progress, open the **Project** menu and then select **Stop Build**.

If **Load After Build** () is set in the NodeBuilder toolbar or if **Load after Build** in the Build tab of NodeBuilder Project Properties is set, all commissioned devices that use one of the applications produced by the build will be automatically loaded. If there are uncommissioned devices associated with the NodeBuilder project, the devices will be commissioned with the LonMaker tool when the build is complete (see the LonMaker documentation



for more information). The status of this operation will be shown in the NodeBuilder Results pane.

When a device is loaded, it will be assigned the LNS Device Template specified by **LNS device template name** in the device template's *Program ID* settings. If you change a device's program ID, the device template name must also be changed. This is handled automatically if you enable program ID management for the NodeBuilder device template. If you are unable to load a device due to a program ID conflict, you can set the device applicationless by right-clicking the device in the Project pane and selecting **Force Applicationless** from the shortcut menu.

---

## *Files Created When You Build An Application Image*

The following files are generated by the NodeBuilder tool when you build a NodeBuilder device template:

### **Downloadable Application Image Files (.nxe and .apb)**

These files contain an application image that is used by a network tool such as the LonMaker tool to download the compiled application image to a device. There are two types of downloadable application image files. They are the text application image file (.nxe extension) and the binary application image file (.apb extension).

### **Programmable Application Image Files (.nri, .nei, and .nfi)**

These files contain an application image that is used by a programming tool such as a PROM programmer, a Neuron 3120 programmer, or a FT 3120 programmer to program a PROM or 3120 chip prior to assembly into a LONWORKS device. There are three types of programmable application image files. They are the ROM application image file (.nri extension), the EEPROM application image file (.nei extension), and the flash application image file (.nfi extension).

The ROM application image file contains a read-only application image that is used for programming a PROM or flash memory for use in a device based on a Neuron 3150 Chip or FT 3150 Smart Transceiver. The first 16Kbytes of the ROM application image file contains the Neuron Chip firmware, and optionally contains a copy of some or the entire on-chip EEPROM image, as selected by the Exporter Reboot Options for the device template target.

The EEPROM application image file contains a EEPROM application image that is used for programming an external EEPROM, NVRAM, or flash memory, or on-chip EEPROM. If the application image was built for a Neuron 3150 Chip or an FT 3150 Smart Transceiver, the EEPROM application image file contains the

application code and data that resides in off-chip EEPROM, flash, or NVRAM (if any). For these devices, this file is used with a PROM programmer to program the external memory chips. If the application image was built for a Neuron 3120 Chip, this file contains some or all of the on-chip EEPROM image in a special format for use only with a Neuron 3120 programmer.

The flash application image file contains an EEPROM application image that is used for programming the on-chip EEPROM of a Neuron 3120E4 Chip or an FT 3120 Smart Transceiver. It contains the same information as the EEPROM application image file for the Neuron 3120 Chip, but uses a different format because of the different programming requirements of the 3120E4 and FT 3120 chips.

#### **Device Interface Files (.xif, .xfb, and .xfo)**

These files contain a definition of the device interface that is used by network tools to learn the interface to a device, without requiring the device to be physically available. There are three types of device interface files. They are the text device interface file (.xif extension), the binary device interface file (.xfb extension), and the optimized device interface file (.xfo extension). The optimized device interface file is optional and will be generated automatically as needed.

The text device interface file is a text description of the device interface. The format of this file is detailed in the *LONMARK External Interface File Reference Guide* available in **LNS Utilities and LONMARK Reference Help** in the Echelon LNS Utilities program folder.

The binary device interface file and optimized device interface file are used by LNS tools such as the LonMaker tool to create LNS device templates, which define the device interface to LNS tools.

---

## ***Excluding Targets from a Build***

You can exclude a target or a device template from project builds, and you can exclude a target from a device template build. To exclude a target or device template from a build, right-click the device template or the **Release** or **Development** target folder and then click **Build Exclude** on the shortcut menu. The selected device template or target folder will be grayed out. To include the device template or target in the build after you have excluded it, right-click it and select **Build Exclude** again.

You can also choose to build to only development or release targets in the entire project. To do this, select **Development Targets** or **Release Targets**

in **Build Type** in the NodeBuilder toolbar. To build all targets, select **All Targets**.

---

## *Cleaning Build Output Files*

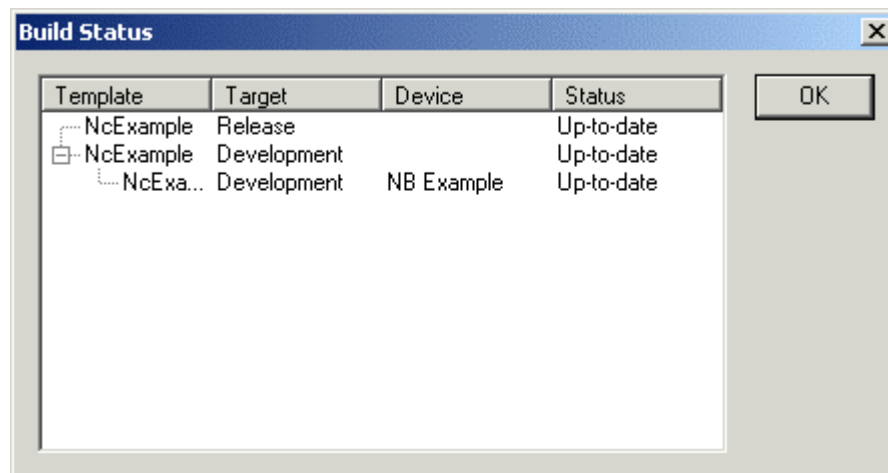
You can remove all files and folders produced by a build from the device template's output folder. To remove all build outputs in the project, right-click the Device Templates folder and then select **Clean** from the shortcut menu. To clean all build outputs from a specific device template or target, right-click the device template or target folder and then select **Clean** from the shortcut menu.

The Clean command only removes files and folders produced by the NodeBuilder tool. It will not remove any files that you have produced.

---

## *Viewing Build Status*

You can view the build status of all NodeBuilder device templates and targets. The build status shows whether the latest version of the source files have been compiled and built and whether all known devices have had the latest version of the application loaded. You can see the build status for the entire project, a specific device template, a specific device template target, or a specific target. To see the build status for the entire project, right-click the **Device Templates** folder and then click **Status** on the shortcut menu. To see the build status for a specific device template, target, or device, right-click it and then click **Status** on the shortcut menu. The status window appears as shown in the following figure:



Each line in this window represents a device template target or a target. Targets are listed beneath their associated device template target. The status window has the following columns:

- Template** The NodeBuilder device template.
- Target** The target type (Release or Development).
- Device** If this row contains the status for a target, this column displays the target name. If this column

## Status

contains status for a device template target, this column is empty.

The target status. This is one of the following values:

*Up-to-date* – For device template targets, this indicates that the application image is consistent with the source code. For targets, this indicates that the target has been loaded with the latest application image.

*Compile required* – Applies to device template targets only. Indicates that the source code or a property that would change the compiled version of the application has been modified since the application was last compiled.

*Assembly required* – Applies to device template targets only. Indicates that the assembly file has been modified since it was last assembled or a property that would modify the assembled version of the application has changed. This status is unlikely to occur.

*Link required* – Applies to device template targets only. Indicates that one of the libraries or the system image has been modified since the application image was last built or that a property has been changed that requires the project be re-linked.

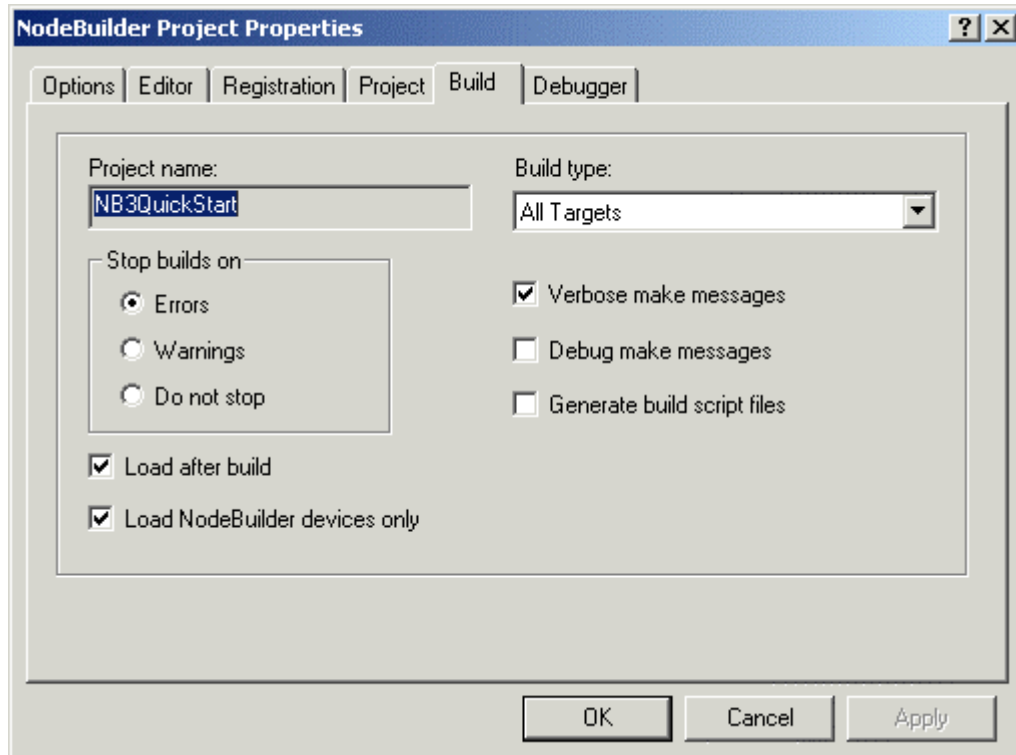
*Export required* – Applies to device template targets only. Indicates that a property has been changed that requires the device to be exported.


*Load required* – Applies to targets only. Indicates that the application image has been modified since the target was last loaded. The NodeBuilder tool will only be aware of loads performed by the NodeBuilder or LonMaker tools. If you load the application with another tool, the NodeBuilder tool will not update the status until the application is built and loaded using the NodeBuilder or LonMaker tools.

---

## Setting Build Options

You can set build properties that control the build process. To set build properties, open the **Project** menu and then click **Settings**. The **NodeBuilder Project Properties** dialog appears with the **Build** tab selected. This tab appears as shown in the following figure:



Alternatively, open the **Tools** menu and then click **Options**, or click the Project Settings button ()

This tab contains the following options:

**Stop builds on**

Determines when a build is stopped. A build may be stopped when an error is returned, when a warning is returned, or upon completion. The default is **Errors**. If **Do not stop** is selected and an error occurs, the build process will move on to the next target, rather than aborting the build.

**Load after build**

Loads the application into a device immediately after the application image is built. The devices must be commissioned with the LonMaker tool and the LonMaker drawing containing the device must be open and attached to the network. The



**Load After Build** button on the NodeBuilder toolbar reflects changes to this option and vice versa.

**Load NodeBuilder devices only**

Limits loads to the targets listed in the *Devices* folder in the Project pane.

**Build Type**

Determines whether **All Targets**, **Development Targets**, or **Release Targets** will be built when you build a project or device template. The build type is displayed on, and can be changed from, the NodeBuilder toolbar.

- Verbose make messages**      Displays more descriptive messages in the Results pane when you build a device template.
- Debug make messages**      Displays debugging messages in the Results pane when you build a device template. This output may be used by Echelon Support to help you diagnose problems.
- Generate Build Script Files** Generates build script files when you build a device template. Build scripts are described in Appendix C.

## Loading an Application Image

You can load an application image that you have built with the NodeBuilder tool into a LONWORKS device. The device may be a development platform such as an LTM-10A Platform or LonBuilder emulator, or it may be a custom device that you have manufactured or purchased from a third-party. You will typically do your initial debugging on a development platform before building a custom device, but you can create and load a custom device at any time.

Development platforms such as the LTM-10A Platform and the LonBuilder Emulator include Neuron firmware that is preloaded into the device (or downloaded by the LonBuilder software for the LonBuilder Emulator). The Neuron firmware allows these devices to be downloaded over a LONWORKS network so that you do not have to use any special memory device programming tools. If you are using a development platform, you will automatically load the platform when you add a NodeBuilder target as described in the next section, *Adding Targets*.

If you are using a custom device, you must load your application image into device before using the device as a target. Once you have completed development, you will also load your application image into the device as part of your manufacturing process. The files containing the application image are described in *Files Created When You Build an Application Image* earlier in this chapter. The following table summarizes the processor/memory combinations that you can use, and the files that you will use to program each.

<b>Processor</b>	<b>Neuron Firmware Memory Type</b>	<b>Application Memory Type</b>	<b>Application Image File Extension</b>	<b>Application Image Programming Tool</b>
Neuron 3120xx Chip (except Neuron 3120E4 Chip)	On-chip EEPROM	On-chip EEPROM	.nei	Neuron 3120 Programmer
			.nxe and .apb (TP/XF-1250 devices only)	Network Tool
Neuron 3120E4 Chip and FT 3120 Smart	On-chip EEPROM	On-chip EEPROM	.nfi	Compatible PROM Programmer

Transceiver			.nxe and .apb (TP/XF-1250 devices only)	Network tool
Neuron 3150 Chip or FT 3150 Smart Transceiver	Off-chip ROM, PROM, or flash	On-chip EEPROM	.nri and .nei	PROM programmer (.nri) and network tool (.nei)
Neuron 3150 Chip or FT 3150 Smart Transceiver	Off-chip ROM or PROM	Off-chip ROM or PROM	.nri	PROM programmer
Neuron 3150 Chip or FT 3150 Smart Transceiver	Off-chip ROM or PROM	Off-chip flash, EEPROM, or NVRAM	.nri and .nei	PROM programmer
Neuron 3150 Chip or FT 3150 Smart Transceiver	Off-chip flash	Off-chip flash	.nei	PROM programmer

The procedure that you will use to program the application image depends on whether you are programming on-chip or off-chip memory for a Neuron 3150 Chip or FT 3150 Smart Transceiver, or if you are programming on-chip memory for a Neuron 3120 Chip or FT 3120 Smart Transceiver. These three procedures are described in the following sections. See the Smart Transceiver databook for more information

---

## ***Programming 3150 Off-chip Memory***

A device based on a Neuron 3150 Chip or FT 3150 Smart Transceiver will always have off-chip ROM or flash memory, and may also have off-chip EEPROM or flash, and RAM. The Neuron firmware must reside in the ROM or flash. The application code may reside in any combination of the off-chip memory types, and the on-chip EEPROM. See *Using Memory* in *The Neuron C Programmer's Guide*, for information on the placement of application code in the various memory types.

You can program the Neuron firmware and your application image into a PROM or flash memory device using a PROM programmer. You will use the ROM application image file (.nri extension) if your device uses off-chip PROM, or the EEPROM application image file (.nei extension) if your devices uses off-chip flash, EEPROM, or NVRAM. You will use both types of image files if your device uses both types of memory. These files are described in *Files Created When You Build an Application Image* earlier in this chapter. All off-chip memory devices containing Neuron firmware or an application image must be programmed before loading them in the device. You can load an initial blank application if you plan on downloading a new application over the network to your device.

When using flash memory, always enable the flash programmer's software data protect, SDP, feature if possible. You must have at least 0x5600 bytes, mapped for flash or else the SDP algorithm will not work.

You can define sections of application code that will reside in EEPROM, flash memory, or NVRAM, coexisting with the Neuron firmware and other application code in ROM. The portion of the code that will reside in EEPROM, flash, or NVRAM is contained in the EEPROM image file (.nei extension). You must program this memory before installation, just like the ROM, since the application must be completely present when the device is powered-up.

---

## Programming 3150 On-chip Memory

The Neuron firmware automatically initializes the on-chip EEPROM for a Neuron 3150 Chip or FT 3150 Smart Transceiver by copying a block of memory from off-chip memory called the boot image. The boot image is contained in the system area (the first 16Kbytes). It contains a copy of some or all of the on-chip EEPROM memory. Its contents depend on which firmware state you select when you build the application image. If you select the unconfigured state (the default), the boot image contains application code and data and a default network image with no network addressing information. If you select the configured state, the boot image contains a complete copy of on-chip EEPROM, including network configuration complete with network addressing information. When a Neuron 3150 Chip or FT 3150 Smart Transceiver is powered up and the firmware determines that EEPROM should be initialized (see below), the data from the boot image will be copied to on-chip EEPROM, and the appropriate firmware state will be set. If the firmware state is unconfigured, the remaining EEPROM data must then be loaded over the network. If the firmware state is configured, the chip will be fully programmed at this point, though no network connections will be defined.

The boot image is used to initialize the on-chip EEPROM of a Neuron 3150 Chip or FT 3150 Smart Transceiver when the chip is powered up and the firmware detects that EEPROM has not yet been initialized by the current Neuron firmware or if the Neuron firmware detects an error and reboot options are specified as described in *Setting Reboot Options* later in this section. To accomplish this, there is a special value, or *boot ID*, placed in the application image file when it is exported. This 16-bit value normally changes each time you build the application image. On power-up, the Neuron firmware compares the boot ID in the firmware image with the boot ID copy in the on-chip EEPROM. If they don't match, the Neuron firmware initializes the on-chip EEPROM from the boot image. It also copies the boot ID to EEPROM, so the initialization will not happen again until a new firmware image with a different boot ID is installed. Additional EEPROM boot recovery options are available as described under *Setting Reboot Options*.

Since the boot ID normally changes each time an application image file is exported, exporting, programming, and inserting a new memory chip will normally result in the EEPROM initialization taking place, even if no changes have been made to the application or configuration. While a device normally only does this initialization once for a given firmware image, it is



possible to force this process to occur again with the same firmware image by resetting the Neuron 3150 Chip or FT 3150 Smart Transceiver to the blank state (the initial state of EEPROM on a newly manufactured Neuron Chip or Smart Transceiver) using a special application image. This image is shipped with the NodeBuilder software in a file named EEBLANK.NRI, and is located in the LONWORKS Images folder (c:\LonWorks\Images by default), where x is 12 or higher. To reset a 3150 chip's state, program this image into a memory chip and power up the device with this memory chip in place of the normal firmware. For a short period, the service LED will flash, then it will change to full on, indicating that the chip has been returned to the blank state. The next time any memory created from an exported firmware file is placed in the device, the on-chip EEPROM will again be initialized from the special data area in the firmware.

In addition to the boot ID, external EEPROM, RAM, and flash memory areas coexisting with ROM will each have a 16-bit signature value, or *memory signature*, calculated over any application code or data (but not user variables) that resides in the area. These values are kept in the respective memory areas, as well as in on-chip EEPROM. Whenever the Neuron Chip or Smart Transceiver is reset, the Neuron firmware compares the on-chip and off-chip signatures, and if there is a mismatch, the Neuron firmware changes the device state to applicationless. If the device copies the boot image to on-chip EEPROM, this check will follow that process, and will override the firmware state selection if the signatures do not match.

## Setting Reboot Options

Some hardware designs, such as designs with inadequate power supply noise decoupling for the Neuron Chip or Smart Transceiver, can cause corruption of the contents of the Neuron on-chip EEPROM. The Neuron firmware can detect this because it maintains several 8-bit checksums, one for the configuration image, one for the application image, and one for the system image. You can set reboot options that specify the action to be taken by the Neuron firmware when it detects a checksum error. To set the reboot options, follow these steps:

1. Right-click the development or release target in the device template folder in the NodeBuilder Project Pane and then click **Settings** on the shortcut menu. The NodeBuilder Device Template Target Properties window appears.
2. Select *Configuration* in **Category**. A set of options appear that specify when the Neuron firmware should copy the network configuration from the off-chip boot image to on-chip EEPROM. The reboot options are only available if the target hardware is a Neuron 3150 Chip, with version 6 or later firmware, or a custom system image based on version 6 or later firmware, such as the LTM-10A firmware. This includes the address assignment and binding information in the device, but does not include the communications parameters. Select from the following options:

**Checksum Error**

Reboot the network configuration whenever there is a configuration checksum error.

**Fatal Application Error**

Reboot the network configuration whenever there is a fatal application error such as an application checksum error, illegal device state, memory

allocation failure, or application image inconsistency.

**Always**

Reboot the network configuration every time the Neuron Chip or Smart Transceiver is reset.

3. Select *Application* in **Category**. A set of options appear that specify when the Neuron firmware should copy the on-chip portion of the application image from the off-chip boot image to on-chip EEPROM. All applications have at least part of their image in on-chip EEPROM in the Read-Only data structure. Corruption of any part of the application image resident in off-chip memory cannot be recovered by this mechanism. If off-chip memory gets corrupted, recovery will fail and the device will go applicationless. If this happens, you must reprogram the memory chip or re-download the application over the network. Select from the following options:

**Fatal Application Error**

Reboot the application image whenever there is a fatal application error such as an application checksum error, illegal device state, memory allocation failure, or application image inconsistency.

**Always**

Reboot the application image every time the Neuron Chip or Smart Transceiver is reset.

**App'less is Fatal**

Specifies that detection of the applicationless state is to be treated as a fatal application error. If you specify this option, you should also specify that the application be rebooted on occurrence of a fatal application error, otherwise no recovery will occur. DO NOT SELECT this option if you wish to download an application over the network to the device, since the device must be in the applicationless state for a download to occur.

**Reboot EE Vars**

Specifies that on-chip EEPROM variables, including configuration network variables, will also be rebooted any time the application image is rebooted. This will undo any changes to the initial state of the EEPROM variables located in the on-chip EEPROM by a network tool, the application, or another device.

4. Select *Communication Parameters* in **Category**. A set of options appear that specify when the Neuron firmware should copy the communication parameters from the off-chip boot image to on-chip EEPROM. Re-initializing the communications parameters will cause the device to lose its priority assignment if it previously had one. Select from the following options:

**Checksum Error**

Reboot the communication parameters whenever there is a checksum error.

**Type/rate Mismatch**

Reboot the communication parameters whenever the transceiver type (differential, single-ended, or special purpose mode) or the interface bit rate of the on-chip communications parameters do not match the values in the ROM copy. This usually indicates corrupted communications parameters, although this might not be the case if your transceiver supports multiple bit rates.

**Always**

Reboot the communication parameters every time the Neuron Chip or Smart Transceiver is reset.

---

## Programming 3120 On-chip Memory

Since a Neuron 3120xx Chip or FT 3120 Smart Transceiver does not support external memory, the only memory to program is on-chip EEPROM, and this must be programmed over the network or with a 3120 programmer.

A blank Neuron 3120xx Chip or FT 3120 Smart Transceiver comes up with its communications interface initialized to 1.25Mbps differential mode with a 10MHz input clock (TP/XF-1250 twisted-pair compatible), and a Neuron firmware state of applicationless. If your custom device has a compatible transceiver and clock, you can load all of the application and network configuration over the network, using the LonMaker tool.

To pre-program a Neuron 3120xx Chip or FT 3120 Smart Transceiver with an application or network configuration other than the default, you must program it in a Neuron 3120 Chip programmer. Refer to the documentation supplied with the particular programmer for details.

---

## Adding Targets

A *target* is a LONWORKS device whose application is built by the NodeBuilder tool. There are two types of targets, *development targets* and *release targets*. Development targets are used during development; release targets are used when development is complete and the device will be released to production. Each NodeBuilder device template specifies the definition for a development target and a release target. Both target definitions use the same source code, resource files, and must have identical device interfaces, but can use different hardware templates and compiler, linker, and exporter options. The source code may include code that is conditionally compiled based on the type of target.

Each target is defined by a LonMaker Target shape and its corresponding LNS device, a NodeBuilder device template and its corresponding LNS device template, and a NodeBuilder hardware template. You can add a target to a NodeBuilder project using the LonMaker tool or the NodeBuilder Project Manager. Using the LonMaker tool is typically the fastest and easiest method. See *Adding a Target with the LonMaker Tool*.

---

## Adding a Target with the LonMaker Tool

You can add a target to a NodeBuilder project using the LonMaker tool. To add a target with the LonMaker tool, follow these steps:

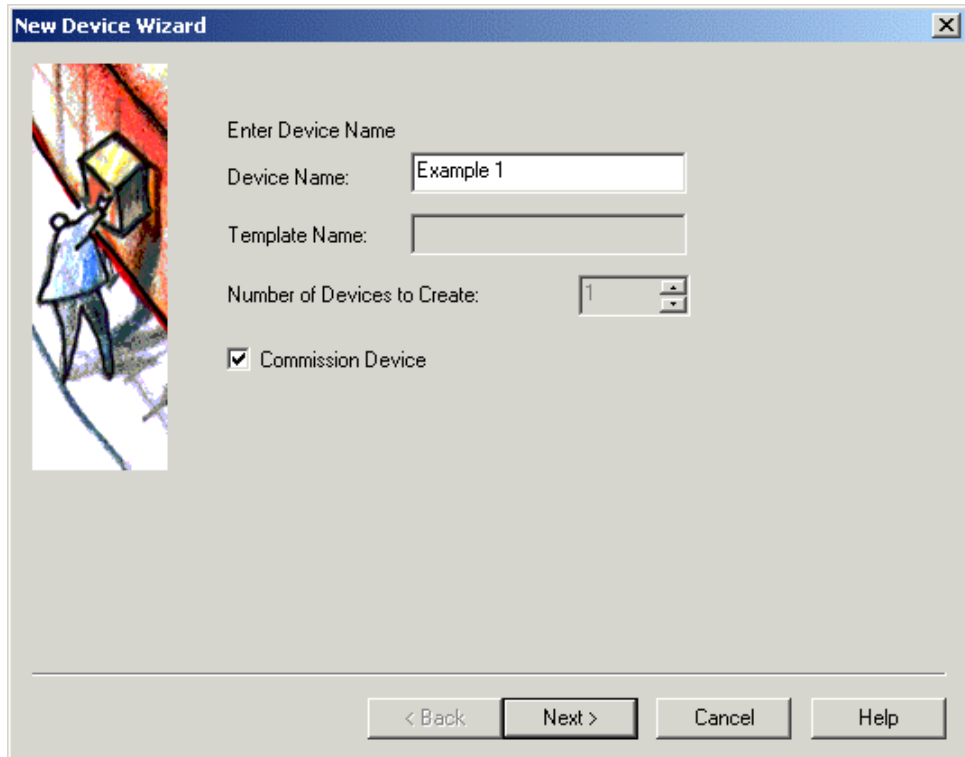
1. Build the application image for the target as described in *Building an Application Image* earlier in this chapter.
2. Correct any build errors before proceeding.
3. Click the LonMaker button in the Windows taskbar to switch to the LonMaker window.

You will use the LonMaker Integration Tool to install, bind, configure, and test the targets in your project. The LonMaker tool displays a network drawing that shows the devices, functional blocks, and connections in your network.

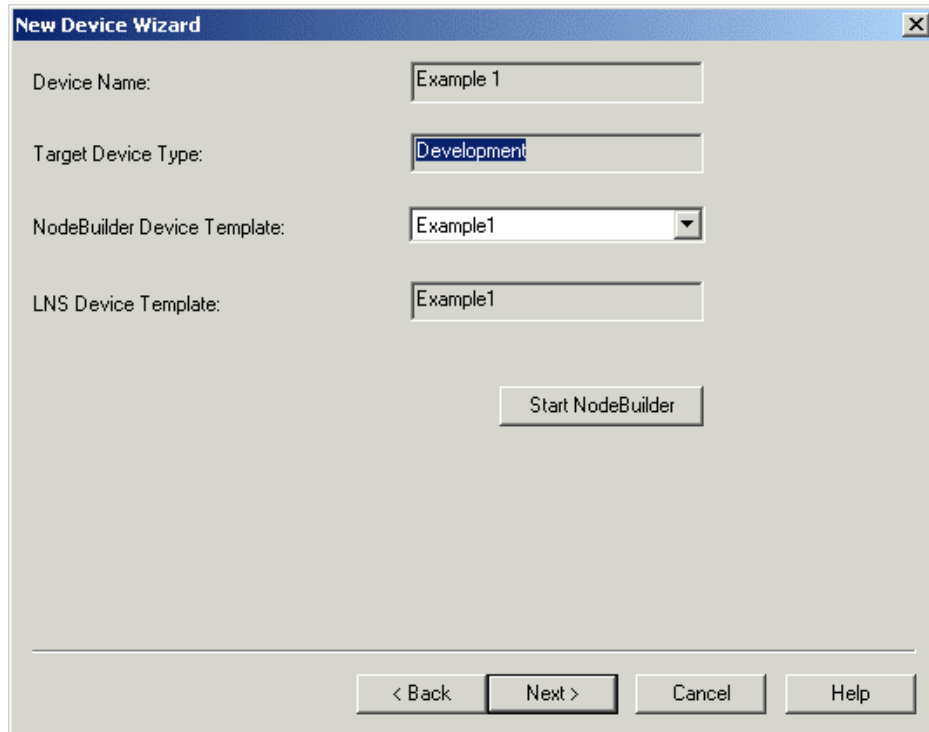
The LonMaker tool also displays *stencils* that contain shapes that you can drag to your LonMaker drawing. The LonMaker tool includes a NodeBuilder Basic Shapes stencil with shapes that you will use to add new devices, functional blocks, and connections to your network drawing. The NodeBuilder Basic Shapes stencil contains shapes that can be used with any device. You can also create custom stencils with shapes customized for your devices and networks.

The NodeBuilder Basic Shapes stencil contains two shapes that you will use to define your targets during development. They are the Development Target Device shape and the Release Target Device shape. These special device types help distinguish between other devices on the network and the target devices used by the NodeBuilder tool. The NodeBuilder tool allows you to create a mixed network of development hardware (such as LTM-10A Platforms and LonBuilder Emulators), release hardware (your own hardware), and other devices.

4. To create a development target, drag a **Development Target Device** shape from the **NodeBuilder Basic Shapes** stencil to your network drawing. To create a release target, drag a **Release Target Device** shape from the **NodeBuilder Basic Shapes** stencil to your network drawing. You can drop the shape anywhere. The New Device Wizard starts, as shown in the following figure:



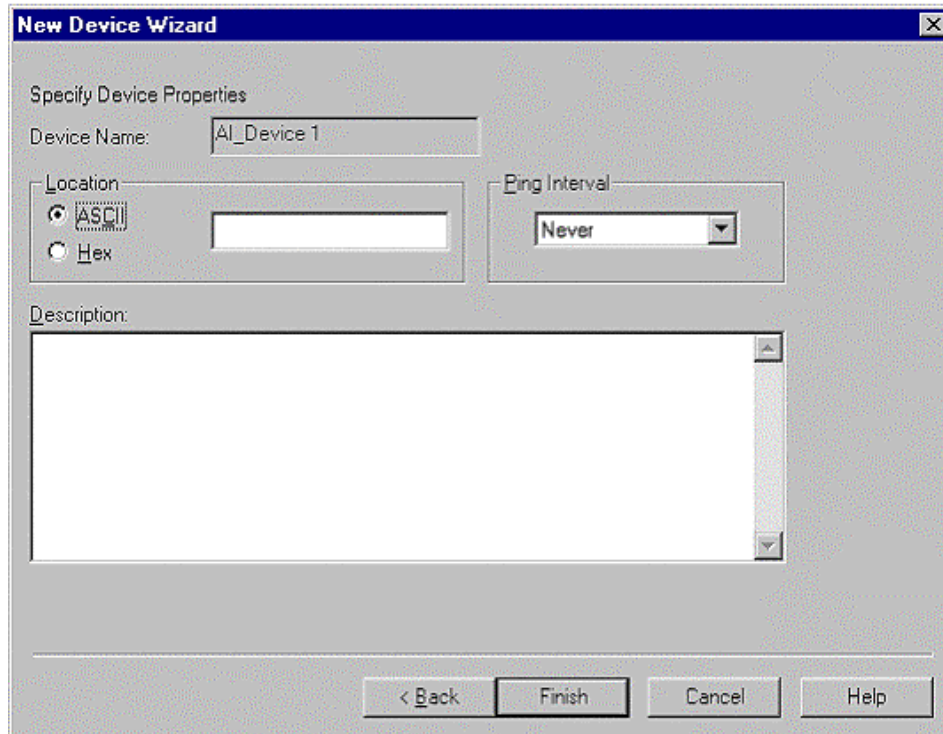
5. Enter a name for the target. This name must be unique for all the devices and targets within the current page (subsystem). The default name is the “Device” followed by an integer (e.g. Device 1). The target name may be up to 85 characters, may include embedded spaces, and may not include period, backslash, colon, forward slash, or double quote characters.
6. If you have a development platform for the target, set **Commission Device**, and then click **Next**. A window opens that allows you to select the NodeBuilder device template to use for this target device, as shown in the following figure:



7. Click the arrow, and then select a NodeBuilder device template from a list of all NodeBuilder device templates that you have built for this project.
8. Click **Next**. The Channel Selection window appears.
9. Specify the channel for the new device. If you select *Auto-Detect*, the LonMaker tool will automatically determine what channel the device is on. To use this option, you must ensure that the LonMaker tool is attached to the network and all routers between the LonMaker network interface and the device being defined have been installed and commissioned and are online. Otherwise an error will be returned during commissioning.

Do not use **Auto-Detect** if you are using routers configured as repeaters or bridges or if the LonMaker tool is not attached to the network.

If *Auto-Detect* is not selected you must explicitly select the channel to which the device is attached. Use **Xcvr Type** to list only the channels for a specific transceiver type (some device shapes will limit the types of transceivers you can select). Click **Next**. The following window appears:



10. Enter the following information:

**Location**

Location information for this device. This information may be entered as an ASCII string (up to 6 characters) or a hex string (up to 12 hex digits). It is used to document the device's location within the network. This information is not used by the LonMaker tool, but may be useful for network recovery if you lose the LonMaker drawing and database. For example, you can put an abbreviation of the subsystem name or a subsystem number in the Location field.

**Ping Interval**

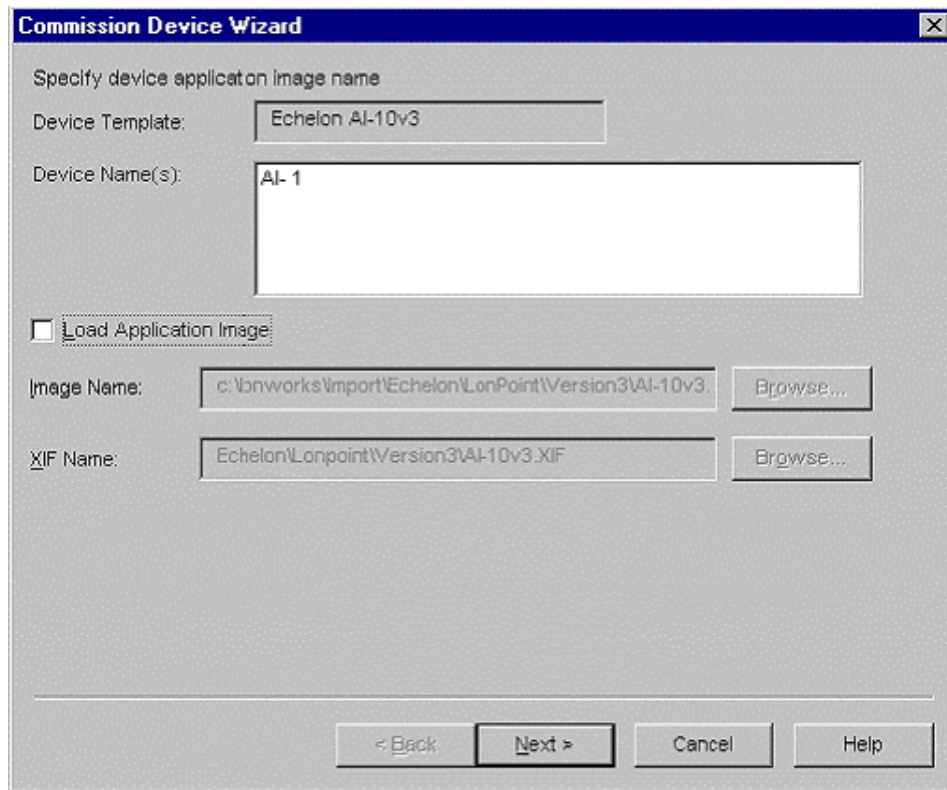
Ping interval for this device. The ping interval determines how often a device is pinged by the LNS Server to ensure it is still operating and in communication with the network. You can leave the default value of **Never** for targets.

**Description**

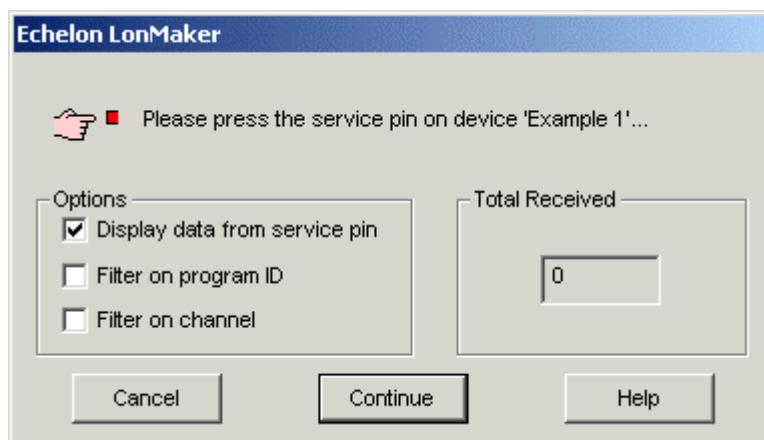
A description of the device. This description has no effect on network operation so you can use this field for any target-specific documentation that you would like to save with the LonMaker drawing.

11. Click **Finish** to create the Target shape and complete the target definition. If a shape representing the target's channel does not exist on the page in which the target is being created, the appropriate channel shape will be created automatically. If you cleared **Commission Device**,

the remaining steps are skipped, and you will have to commission the target later. If you set **Commission Device**, the first commissioning window, similar to the following figure, appears:



12. Set **Load Application Image**, and then click **Next**. The final window of the New Device Wizard appears.
13. Select the **Online** device state option to start your target online, and then click **Finish**. The Press Service Pin window appears:



14. Press the service pin on the target you wish to load and commission. The LonMaker tool loads the application image for your application to the target and makes it operational. You can now test and debug the application as described in the *Using the NodeBuilder Debugger* and *Testing a NodeBuilder Device Using the LonMaker Tool* chapters.

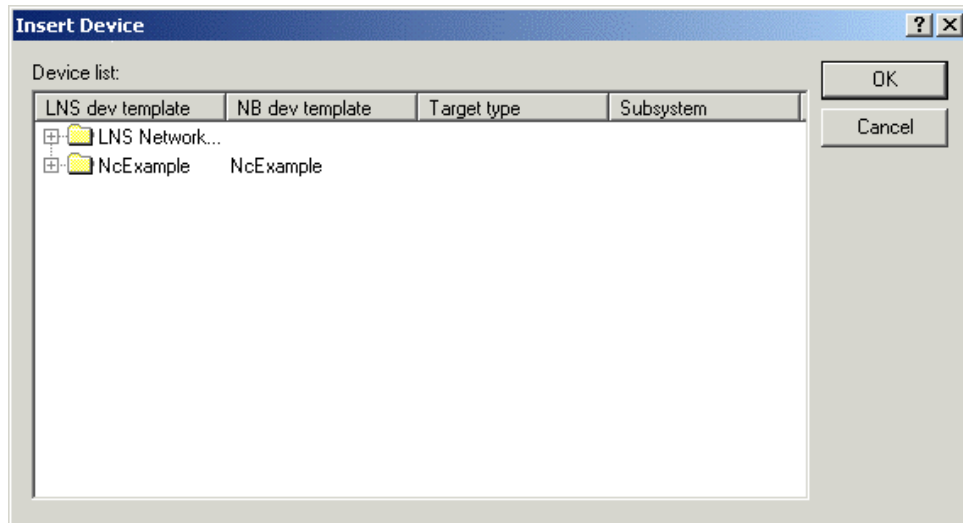


---

## Adding a Target with the Project Manager

You can add a target to a NodeBuilder project using the NodeBuilder Project Manager. To add a target with the project manager, follow these steps:

1. Right-click the **Devices** folder in the Project pane and click **Insert** on the shortcut menu. The following dialog opens:



This dialog organizes the devices in currently open LonMaker networks by LNS device template name. If the LNS device template is based on a NodeBuilder device template, the Nodebuilder device template name will be shown in the **NB Dev Template** column.

2. Browse to and select the device you want to add using the  $\mathbb{F}$  button next to the appropriate device template name.
3. Click the arrow under the **NB Dev Template** column and then select a NodeBuilder device template if the LNS device template does not already have an associated NodeBuilder device template. When you click the arrow, the list will contain all NodeBuilder device templates in the current project. Select a NodeBuilder device template for this target.
4. Click the **Target Type** field, and then click the arrow that appears to display the target types. Select either a development or release target type.
5. Click **OK** to add the target to the **Devices** folder in the NodeBuilder Project pane. If this device is commissioned, the NodeBuilder tool will load the device the next time you build the application image for the device.

---

## Using Targets in the Project Manager

You can view, define, and build targets using the NodeBuilder Project Manager. You can also define targets using the LonMaker tool as described in *Adding a Target with the LonMaker Tool*, earlier in this chapter.

The **Devices** folder in the Project pane contains all the devices that you have defined as release or development targets in the NodeBuilder project for the current LonMaker network. Right-click the **Devices** folder and then select **Insert** from the shortcut menu to browse all open LonMaker networks for additional devices to add to the project. See *Inserting a Device into a NodeBuilder Project*, later in this chapter, for more information.

Right-click a device to open a shortcut menu with the following commands:

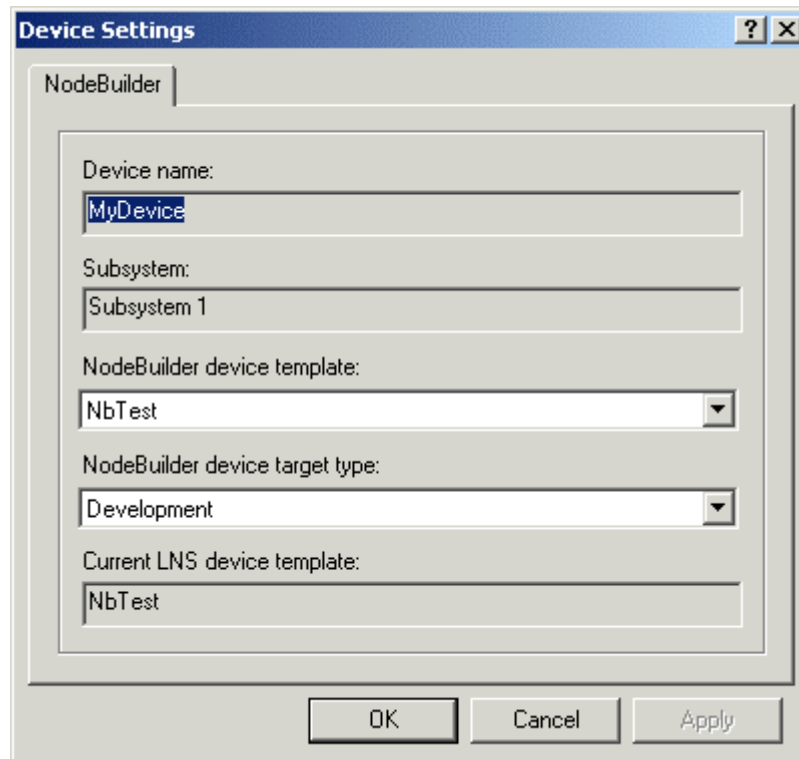
<b>Settings</b>	Displays device settings including the NodeBuilder device template and target type. These settings are described in <i>Setting Build Options</i> , earlier in this chapter.
<b>Remove</b>	Removes the device as a target for future builds. The device will be removed from the project but the device is not removed from the LonMaker drawing or network, and none of the device files are deleted. To change the LonMaker shape, replace the Development or Release Target shape with the LonMaker Device shape after removing the target in the project manager. You can do this by dragging the Device shape on top of the Target shape.
<b>Build</b>	Builds the application image for the device template assigned to this device. See <i>Building an Application Image</i> , earlier in this chapter, for more information.
<b>Debug</b>	Debugs the device. See the <i>Using the NodeBuilder Debugger</i> chapter for more information. This command will be dimmed if the application image has not been built; it is not shown if the device is already being debugged.
<b>Stop Debugging</b>	Stops debugging the device. This command is not shown if the device is not being debugged.
<b>Force Applicationless</b>	Forces the selected device to the applicationless state by clearing its program ID. You must then reload the application, or load a new application, to use the device.
<b>Status</b>	Displays the build status for this device and its device template.
<b>Go to LonMaker</b>	Brings the LonMaker drawing containing the device to the foreground with the device's shape selected.

---

## *Editing Target Device Settings*

You can edit the device settings for a target. The device settings include the NodeBuilder device template and NodeBuilder target type for the target. To edit the target device settings, right-click the target in the **Devices** folder in

the Project pane and then click **Settings** on the shortcut menu. The following dialog appears:



This dialog contains the following fields:

<b>Device Name</b>	The name of the device as contained in the LonMaker drawing.
<b>Subsystem</b>	The LonMaker subsystem that contains the device.
<b>NodeBuilder device template</b>	<p>The name of the NodeBuilder device template. Click the arrow to display a list of all the NodeBuilder device templates in the project. If you change the template, the change does not have any affect until the next time you build the device template and load the target.</p> <p>When you load the target using a new device template, the LonMaker tool will preserve any functional blocks and connections that have remained unchanged between the old device template and the new device templates. Functional blocks and connections that have been modified will be deleted.</p> <p>The device shape in the LonMaker drawing will not change when you change the NodeBuilder device template. If there is a different device shape associated with the new LNS device template, drop the new shape on top of the old shape to replace it.</p>

**NodeBuilder device target type**

The type of the device target. The target type may be a development target or a release target. Changing this value will not change the shape in the LonMaker drawing. If you change the target type, you can replace the shape by dropping the appropriate device shape on top of the old one.

**Current LNS device template**

The name of the LNS device template associated with the target. This field is automatically updated when you build the target with a new NodeBuilder device template.

# 10

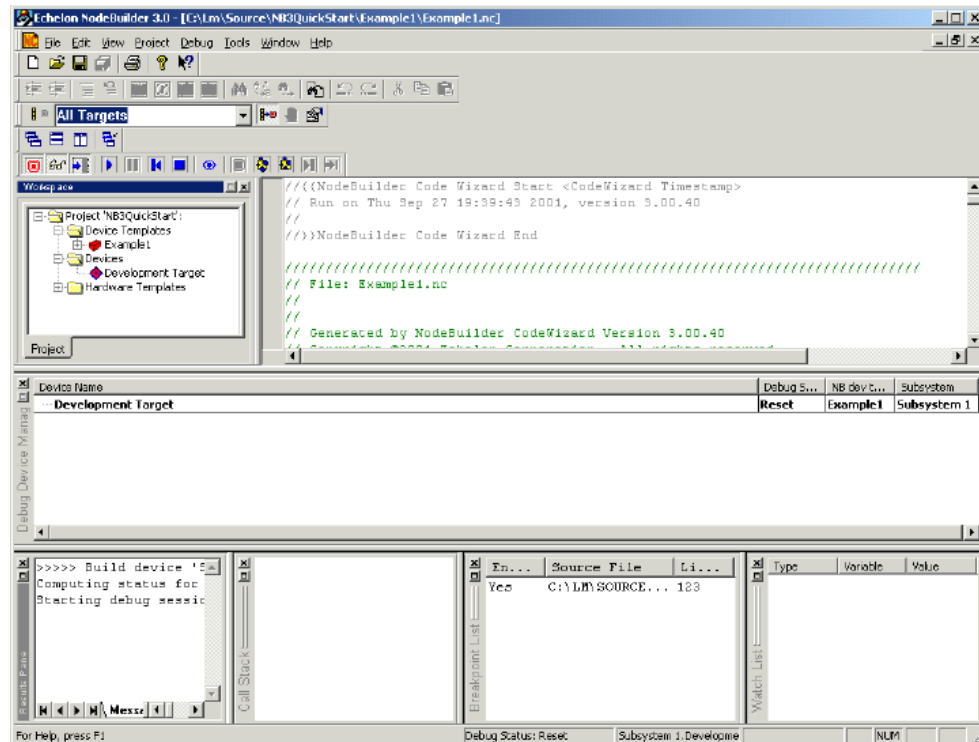
## Using the NodeBuilder Debugger

This chapter describes how to use the NodeBuilder debugger to troubleshoot your Neuron C application.

## Using the Debugger

The NodeBuilder debugger allows you to control and observe your application's behavior to facilitate debugging. The debugger allows you to set breakpoints, monitor variables, halt the application, step through the application, view the call stack, and peek and poke memory. You can make changes to the code as you debug and debug multiple devices simultaneously.

To start debugging, you must have built the application you want to debug and loaded it into an attached device, and the device should be online (if the device is not online, a dialog will open when debugging starts that allows you to place the device online). Once this is done, start the debugger by right-clicking the device to debug in the Project Pane and selecting **Debug** from the shortcut menu. You can also start a debugging session from the LonMaker drawing by right-clicking the device to be debugged and selecting **Debug** from the shortcut menu. In debugging mode, the NodeBuilder Project Manager is reconfigured to display a Debug menu, a Debugger toolbar, and additional debugging windows as shown in the following figure:



Only targets that contain the debug kernel can be debugged; to ensure that the debug kernel is included in a target, right-click the **Development** or **Release** target folder and select **Settings** from the shortcut menu. Ensure that the **Use debug kernel** option is set. By default, this option is set for development targets but not for release targets.

When a debugging session is started, the following four new panes appear in the NodeBuilder Project Manager: the *Debug Device Manager* pane, the *Breakpoint List* pane, the *Call Stack* pane, and the *Watch List* pane.

The Debug Device Manager pane shows which devices are currently being debugged, and allows you to pause and resume the application on each device. See *Using the Debug Device Manager Pane*, later in this chapter, for more information.

The Breakpoint List pane contains all breakpoints you have set. See *Setting and Using Breakpoints*, later in this chapter, for more information.

The Call Stack pane displays a list of active function calls when the debugger is halted in application source code. You can use this information to trace program execution logic. See *Using the Call Stack*, later in this chapter, for more information.

The Watch List pane contains the names and values of monitored variables. See *Using the Watch List*, later in this chapter, for more information.

By default, these panes, with the exception of the Debug Device Manager pane, are docked into the NodeBuilder Project Manager main window. The Debug Device Manager pane appears as a floating window by default, but you can dock it into the NodeBuilder Project Manager. To enable independent dragging and resizing for a pane, right-click the pane and clear **Allow docking** on the shortcut menu.

If at least one debug session is in progress, the status bar indicates the device currently being debugged and its current state (Running, Halted, Reset, etc).

If at least one debug session is in progress, the Results pane contains the Debug Error Log tab, which lists device errors. You can use this pane to display trace information while debugging.

The Debugger toolbar appears as shown in the following figure:



These buttons provide the following functions:



**View Breakpoint List** Toggles the Breakpoint List pane. See *Setting and Using Breakpoints* for more information.



**View Watch List** Toggles the Watch List pane. See *Using the Watch List* for more information.



**View Call Stack** Toggles the Call Stack pane. See *Using the Call Stack* for more information.



**Resume** Resumes execution of a halted application. See *Starting and Stopping an Application Using the Debugger* for more information.









**Halt** Halts the application running on the current device. See *Starting and Stopping an Application Using the Debugger* for more information.



**Reset** Resets the current device.



**Stop** Stops debugging the current device.

 <b>Watch Variable</b>	Opens the Watch Variable dialog. See <i>Using the Watch List</i> for more information.
 <b>Toggle Breakpoint</b>	Toggles whether the current line of code has a breakpoint. See <i>Setting and Using Breakpoints</i> for more information.
 <b>Step Over</b>	Executes the current line of the application. If the current line contains a function, the function executes in its entirety. See <i>Stepping Through Applications</i> for more information.
 <b>Step Into</b>	Executes the current line of the application. If the current line contains a function, the application halts at the first line of the function. See <i>Stepping Through Applications</i> for more information.
 <b>Run to Cursor</b>	Sets an implicit breakpoint at the line containing the cursor. The application resumes if it is currently halted and continues to execute if it is already running. The application halts when it reaches this implicit breakpoint. The breakpoint is cleared once it is encountered.
 <b>Current Instruction Source Code</b>	Jumps to the line of code on which the application has halted if the application has halted.



You can debug multiple devices simultaneously by repeating the procedure described at the beginning of this section..

To stop debugging a single device, right-click the device and select **Stop Debugging** from the shortcut menu. Alternatively, you can open the **Debug** menu, point to **Stop Debugging**, and then select **Current Device** while the appropriate device is displayed in the status bar. To stop debugging all devices, open the **Debug** menu, point to **Stop Debugging**, and then select **All Devices**. You can also stop debugging by right-clicking a device in the Debug Device Manager pane and selecting **Stop Debugging** on the shortcut menu.


---

## Starting and Stopping an Application


There are three ways that you can stop an application while it is running in debug mode. They are halting the application, running to the cursor, and setting breakpoints. See *Setting and Using Breakpoints* for more information on setting breakpoints.

To halt an application, click the halt button (). Alternatively you can open the **Debug** menu, point to **Halt**, and then select **Current Device** or **All Devices**. If the device halts in application code, the Edit pane shows the line of code where the application was halted using an arrow () in the left margin. If the device halts in system code, no arrow will appear and the Call Stack pane displays the following message: Call stack not available.



To run to the current cursor location, place the cursor in the line where you want the application to halt and then click the Run to Cursor button () . Alternatively, you can open the Debug menu and then select **Run to Cursor** while the cursor is at the line where you want the application to halt. The application automatically halts when it reaches the cursor.



Once you halt an application, you can observe the values of application variables in the watch list (see *Using the Watch List*), observe the condition of the call stack (see *Using the Call Stack*), and step through the application one statement at a time (see *Stepping Through Applications*).

To resume execution of an application that has halted, click the Resume button () , press <F5>, or select **Go** from the **Debug** menu. The application continues running until it hits another breakpoint (or the same one again). You can also click another statement and then click the Run to Cursor button to have the application resume execution until it gets to the line containing the cursor.

---


## Setting and Using Breakpoints


Breakpoints allow you to set lines in your source code where the application will stop so you can check variable values, device hardware status, etc. This allows you to pinpoint the line of code that is causing an error or unexpected behavior.

To set a breakpoint, place your cursor in the line of code in which you want to set a breakpoint and then click the Toggle Breakpoint button () . Alternatively you can right-click the line of code and select **Toggle Breakpoint** from the shortcut menu, press <F9>, or open the **Debug** menu, point to **Breakpoints**, and select **Toggle Current Line**. When you set a breakpoint, the following icon appears to the left of the line of code: .

You can only set breakpoints on lines that contain underlying executable code. Examples of such lines include function calls, variable assignments, `if` statements, and macros. Examples of source lines that cannot contain breakpoints include comments, `when` clauses, pre-processor directives, and variable declarations.

You can cause breakpoints to get out of synch with the application when you edit the source file. You can prevent this by not editing code while debugging. If you suspect that breakpoints have gotten out of synch, stop debugging, recompile and load, and then restart debugging.


When the application reaches a line with a breakpoint, the application halts, as described in *Starting and Stopping an Application Using the Debugger*. When an application halts on a breakpoint, the arrow icon appears on top of the breakpoint icon, resulting in an icon that looks like this: .


If you place a breakpoint in a reset clause and perform a software reset, you may have to force the application to continue using the resume () button in order to hit your breakpoint.


---

## Stepping Through Applications

You can step through the code in your application one source statement at a time. You must first halt the application as described in *Starting and Stopping an Application*. You can then use one of two methods to step through your code; you can either step into or step over a source line. The two methods are identical for all statements except for function calls. When you step over a function call, the function executes and you step to the line of code after the function call. When you step into a function, you step to the first executable line of the function.

As described above, when you halt an application, an arrow () appears in the left margin indicating the current line of code. When you step to the next statement, the arrow moves to indicate the current line of source code at which the application is stopped.

To step over the current statement, click the Step Over button () , press F10, or select **Step Over** from the **Debug** menu.

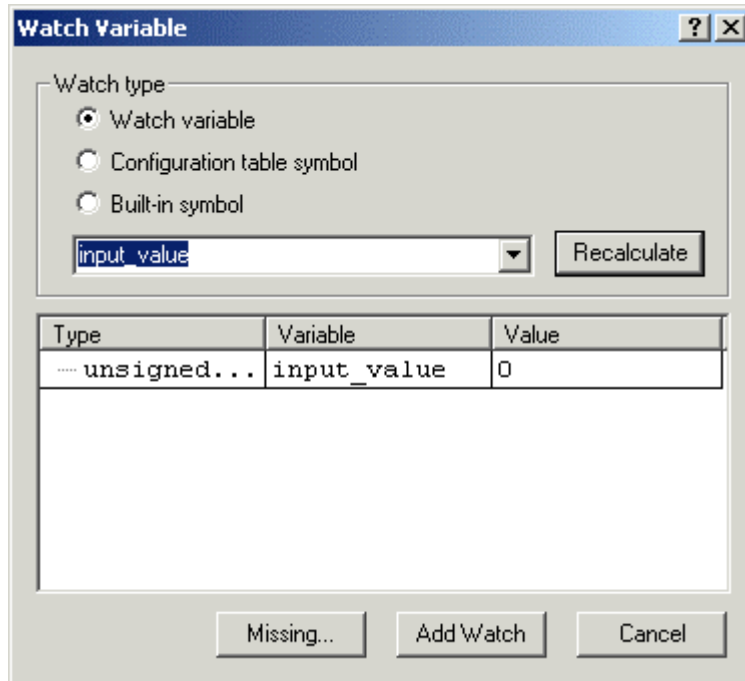
To step into the current statement, click the Step Into button () , press F11, or select **Step Into** from the **Debug** menu.

---

## Using the Watch List

You can monitor a list of variables from your application using the watch list displayed in the Watch List pane. The watch list displays a list of variable names and their current values. You can watch Local variables when the application is halted in a context where the variables are available. You can watch Global variables and network variables at any time, either while the application is running or if it is halted. You can also modify the values of global variables and input network variables while the application is running or halted. You can only modify output network variables when the application is halted in the debugger. You cannot watch the `msg_in`, `msg_out`, `resp_in`, and `resp_out` built-in variables from the debugger.

To add a variable to the watch list, right-click any statement within the source code, and then select **Watch Variable** from the shortcut menu. The following dialog opens:



If you right-click a variable name, the selected variable name appears as the variable to watch in **Watch type**. Otherwise, this field will be blank. Select one of the following to choose which type of variable to watch:

#### Watch variable

You can watch a network variable using its global network variable name or using its functional block member name (i.e. using the scope operator “:.”). Similarly, you can watch a configuration network variable using its global network variable name or using the corresponding configuration property syntax. Please see the *Neuron C Programmer’s Guide* and *Neuron C Reference Guide* for more information on referencing configuration network variables.

To watch a configuration property that is implemented within a configuration file (i.e. it is not implemented as a configuration network variable), specify the configuration property to be watched as follows:

```
[<FB or NV name>][[<FBNVindex>]]::<CP name>[[<CPindex>]]
```

If the configuration property applies to a functional block or network variable, enter <FB or NV name>; if the property applies to the entire device start the name with the scope operator (for example: ::cpValue). If the functional block or network variable is part of an array, enter the <FBNVindex> value to specify the array member. <CP name> can be a configuration property variable or array. If the configuration property is part of an array, enter the <CP index>

to specify the member of the array to watch. In addition, the following rules apply:

- You cannot watch an entire configuration network variable array. You must specify a single element to be watched using the <CPindex> field.
- You can only watch an entire cp\_family array. In this case, do not specify a <CPindex>; the entire array will be displayed in a tree structure in the watch list.

See the *Neuron C Programmer's Guide* and the *Neuron C Reference Guide* for more information on the syntax used for accessing configuration properties.

**Configuration table symbol** Watch a configuration table value . Click the arrow to select from a list of all available configuration table symbols.

**Built-in symbol** Watch a built-in symbol. Click the arrow to select from a list of all available system symbols.

Click **Recalculate** to search for the currently selected watch variable. If the selected variable is a structure type, the pane at the bottom of the dialog allows you to browse the variable structure. If the variable does not exist, a **Symbol Not Found** dialog is displayed.

Click **Missing** to list any header files not used in this application that contain other system variables. If you want to watch one of these system symbols, you will need to include the header file and rebuild.

Click **Add Watch** to add the selected variable to the watch list. If the variable is a structure or union, you will be able to browse all fields of the structure. For each variable or field in a structure, the watch list will display the type, name, and value. Non-structure network variables will contain a single field that contains their value. If the variable does not exist, a **Symbol Not Found** dialog is displayed. If the variable is a valid symbol that cannot be watched within the debugger, the message “**This type of symbol cannot be placed on the watch list. Only variables and network variables may be watched.**”, is displayed. For example, this error is returned when an attempt is made to watch an entire network variable array or a configuration property network variable array. Only individual elements within such arrays can be watched.

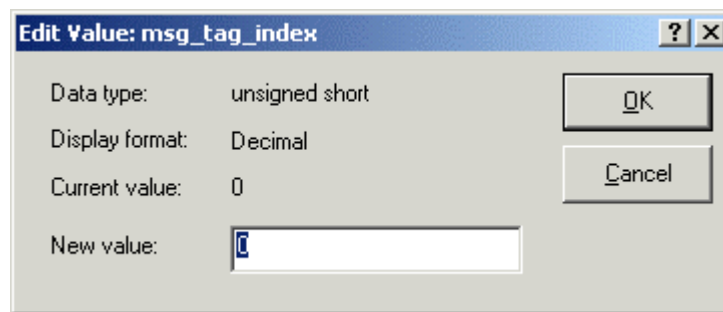
To change the value of a variable in the watch list, right-click it and select **Edit Value** from the shortcut menu. To edit a non-structure network variable value, you must edit the field contained by the network variable. See *Editing a Watch List Value* for more information.

To remove a variable from the watch list, right-click the variable in the watch list and select **Delete** from the shortcut menu. To remove all variables from the watch list, right-click anywhere in the Watch List pane and select **Delete all Watches** from the shortcut menu.

You can display the values in the watch list in either decimal or hexadecimal format by default. Set the **Default Display Radix** option in the *Debugger Options* to determine this behavior. To override this setting for individual entries in the watch list, right-click the entry, click **Display Format** on the shortcut menu, and then select the desired format. You can also display individual entries within each of the variables using string, signed 32-bit, and floating-point format where applicable.

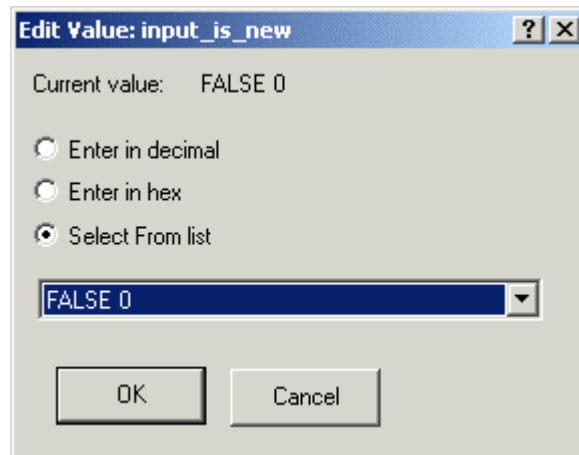
## Editing a Watch List Value

To edit the value of a variable or field within a structure on the watch list, right-click the variable or field and then click **Edit Value** on the shortcut menu. A dialog appears that allows you to edit the value. What dialog appears is dependent on what sort of value is being edited. If you edit an integer value, the following dialog appears:



Enter the new integer value for the variable and then click **OK** to set it.

If you edit an enumerated value, the following dialog appears:



To select from the possible enumerations for the variable, click **Select From list**, click the arrow, and choose a value. You can also choose **Enter in decimal** or **Enter in hex** to enter the value as a decimal or hexadecimal number.

---

## Using the Call Stack

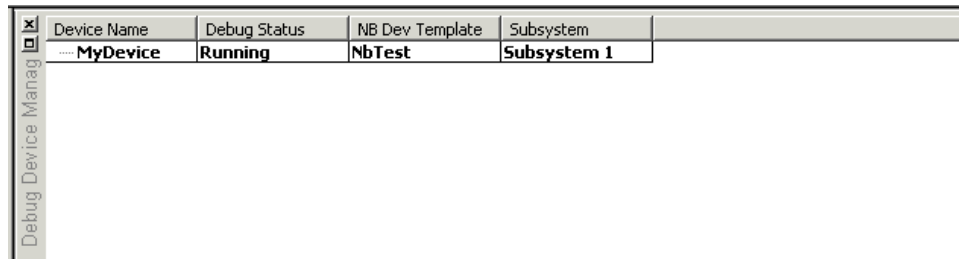
The Call Stack pane shows what functions have been called when the application is halted. If you are halted within a function, this allows you to determine if that function was called from within another function, and if so,

which one. If you are within multiply nested functions, the most recently called one will be on the top of the call stack list. Double-click any entry on the call stack list to be taken to the function call for the selected call.

---

## Using the Debug Device Manager Pane

You can see the status of all devices that are currently being debugged using the Debug Device Manager pane. The following figure illustrates a typical Debug Device Manager pane:



Device Name	Debug Status	NB Dev Template	Subsystem
MyDevice	Running	NbTest	Subsystem 1

This pane contains a list of all devices that are currently being debugged, and whether their applications are running, halted, or reset. The **Reset** status will only be displayed if the device is reset while halted. Right-click a device to see a shortcut menu with the following commands:

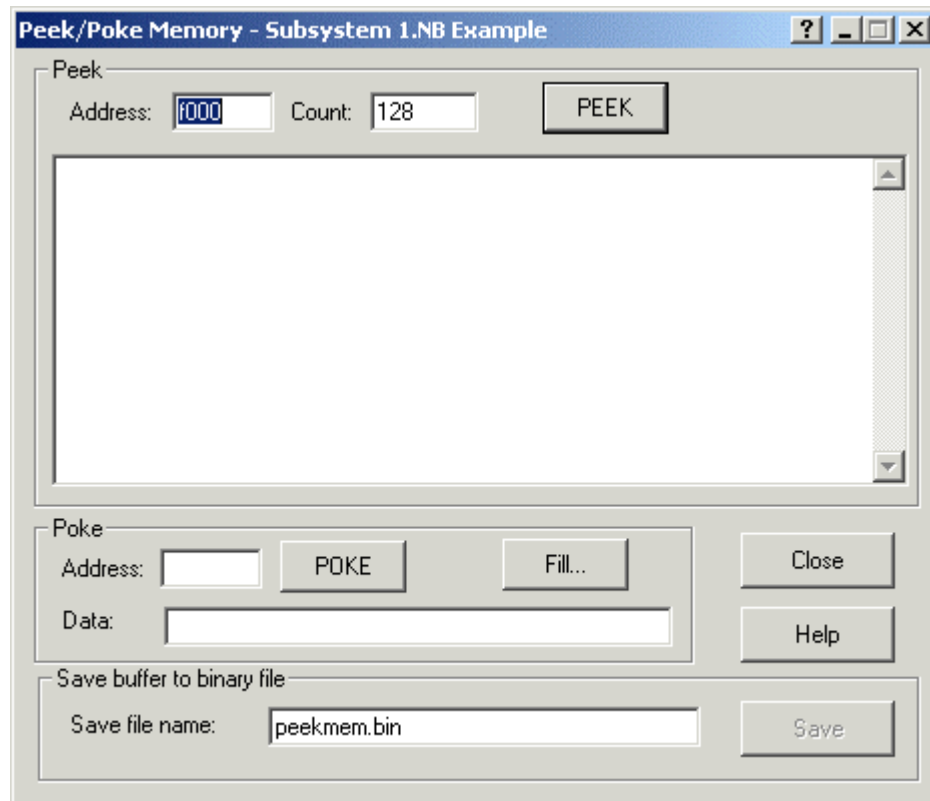
- Make Current** Makes the selected device the current device. This affects operations that are performed on the **Current Device** from the **Debug** menu.
- Stop** Stops debugging the selected device. The device is removed from the list. To restart debugging this device, right-click the device in the Project Pane and select **Debug** from the shortcut menu as described in *Using the Debugger*, earlier in this chapter.
- Halt** Halts the application in the selected device. See *Starting and Stopping an Application Using the Debugger*, earlier in this chapter, for more information about stopping and starting device applications.
- Resume** Resumes running a halted application in the current device.
- Stop All** Stops all debugging. All devices are removed from the dialog and the debug panes are closed. You can restart debugging as described in *Using the Debugger*, earlier in this chapter.
- Allow Docking** Enables docking for the Debug Device Manager pane. By default, docking is enabled. Select this option to toggle docking for this window.
- Hide** Hides the Debug Device Manager pane. To view it again, open the **View** menu, point to **Debug**

**Windows**, and then select **Debug Device Manager**.

---

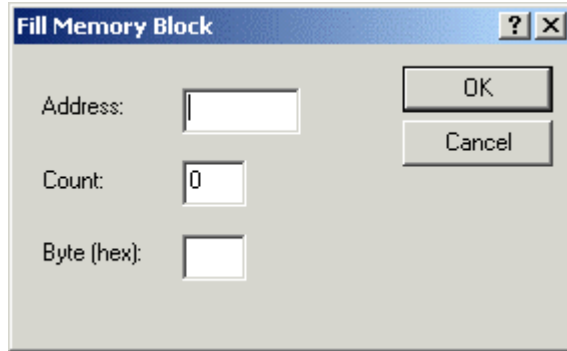
## Peeking and Poking Memory

You can view (peek) and modify (poke) the memory contents of a device that you are debugging. You must be very careful when modifying memory contents, since a device can be rendered inoperable by changing an inappropriate memory location. To view and modify memory, open the **Debug menu** and then select **Peek/Poke Memory** from the **Debug menu**. The following dialog opens:



To peek memory, set **Address** and **Count** at the top of the dialog and then click **PEEK**. A Peek window appears that displays **Count** bytes starting at **Address**. The data is displayed in both hexadecimal and ASCII format. To save the results, set a file name in **Save file name**, and then click **Save**; this opens a dialog allowing you to choose the location of the file to be saved.

To poke memory, set **Address** and **Data** in the **Poke** frame and then click **POKE**. Enter Hexadecimal values in **Data**. The **Data** values are written to the device starting at **Address**. To write multiple bytes, separate each byte with a space, comma, tab, newline, hyphen, or colon. To fill multiple bytes of memory with the same value click **Fill** to open the following dialog:



Enter the starting address in **Address**. Enter the number of bytes to write in **Count**. Enter a two digit hexadecimal value to write to all bytes in the memory block in **Byte**. Click **OK** to write the **Byte** value a number of times equal to **Count** starting at **Address**.

---

## *Executing Code in Development Targets Only*

You can designate code for execution in development targets only. This allows you to build simultaneously to development and release targets and include debugging or test code that executes on the development targets only. To have one or more lines of code execute on development targets only, put a `#ifdef _DEBUG` directive before the code, and a `#endif` directive after the code, as shown below:

```
#ifdef _DEBUG
    //Test code.  Executes on development targets only
    <test code>
#endif
```

You must not define network variables or configuration properties or make any changes to the device interface inside the `#ifdef` clause, since both release and development targets have the same program ID.

The `_DEBUG` macro is predefined for the development target, but not for the release target. To edit the predefined macros, right-click **Development** in the NodeBuilder program manager, and select **Settings** from the shortcut menu. Press F1 for help in this dialog.

---

## *Using the Debug Error Log Tab*

When you are debugging, a **Debug Error Log** tab is added to the Results pane (see *Introduction to the NodeBuilder Project Manager* in the *Creating and Opening NodeBuilder Projects* chapter) This tab allows you to trace program execution without stopping your application and helps you debug timing-related problems. To display an event or error code in the Debug Error Log tab, call the Neuron C `error_log()` function from your application. See the *Neuron C Programmer's Guide* and *Neuron C Reference Guide* for more information about the `error_log()` function.



---

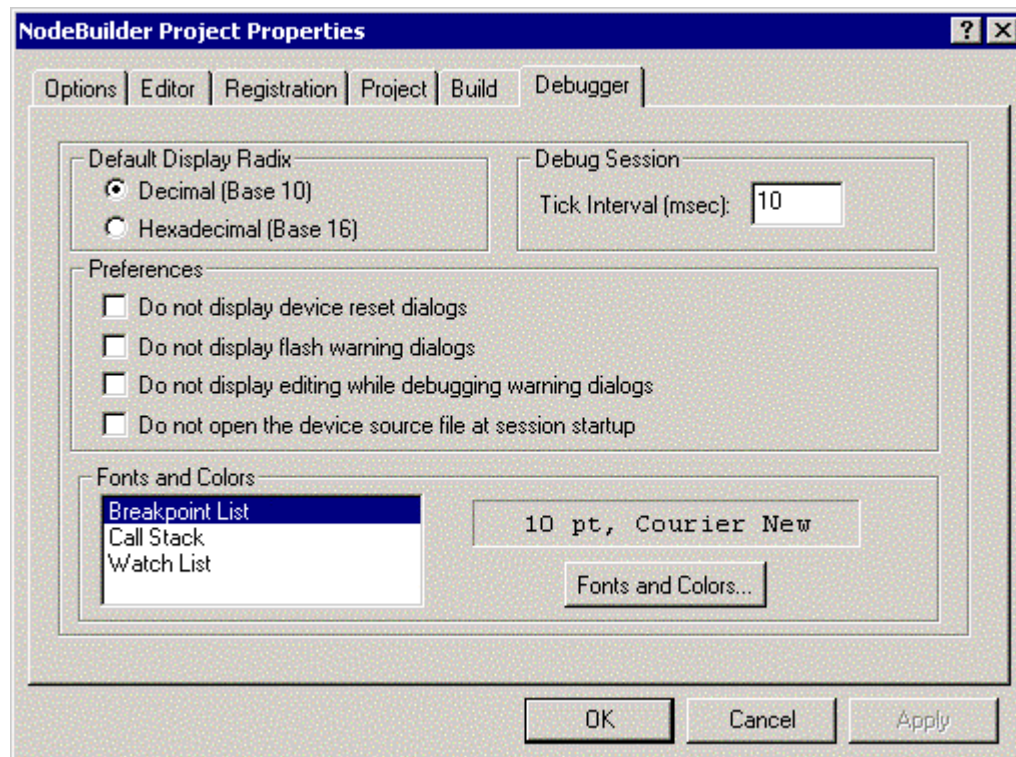
## Editing Source Code While Debugging

You can modify source code while debugging. However, doing this may cause a mismatch between the modified copy of the source file and the source code in the device. You can prevent this by not editing code while debugging. If you suspect that breakpoints have gotten out of synch, stop debugging, recompile and load, and then restart debugging.

---

## Setting Debugger Options

To set debugger options, open the **Project** menu and select **Settings**. The **Settings** dialog appears. Select the **Debugger** tab to display the following dialog:



This tab allows you to set the following options:

<b>Default Display Radix</b>	Displays watch list data in decimal or hexadecimal format by default.
<b>Tick Interval</b>	Controls the interval (in milliseconds) at which incoming debug messages coming from the device are processed by the debugger.
<b>Do Not Display Device Reset Dialogs</b>	Suppresses warning messages that are displayed when a device in the project encounters a hardware, software, or watchdog timer reset. A message confirming the reset will still appear in the Results pane.
<b>Do Not Display Flash</b>	Suppresses warning messages that are displayed

<b>Warning Dialogs</b>	when you set a breakpoint in application code that resides in flash memory.
<b>Do Not Display Editing While Debugging Warning Dialogs</b>	Suppresses warning messages that are displayed when you edit code while in a debugging session. Editing code in a debugging session can cause unpredictable debugger behavior and is not recommended.
<b>Do Not Open The Device Source File At Session Startup</b>	Prevents the source file (<template name>.nc) automatically opening when you start the debugger. This may prevent unnecessary windows from being opened if you are debugging other source files. If a breakpoint is hit in this file (or any file), that file will be opened, regardless of this option.
<b>Fonts and Colors</b>	Select <b>Breakpoint List</b> , <b>Call Stack</b> , or <b>Watch List</b> in the <b>Font and Colors</b> pane to see the currently selected font, font size, and color. To change the font, font size, and color, click <b>Fonts and Colors</b> to open a font-editing dialog.

# 11

## Testing a NodeBuilder Device Using the LonMaker Tool

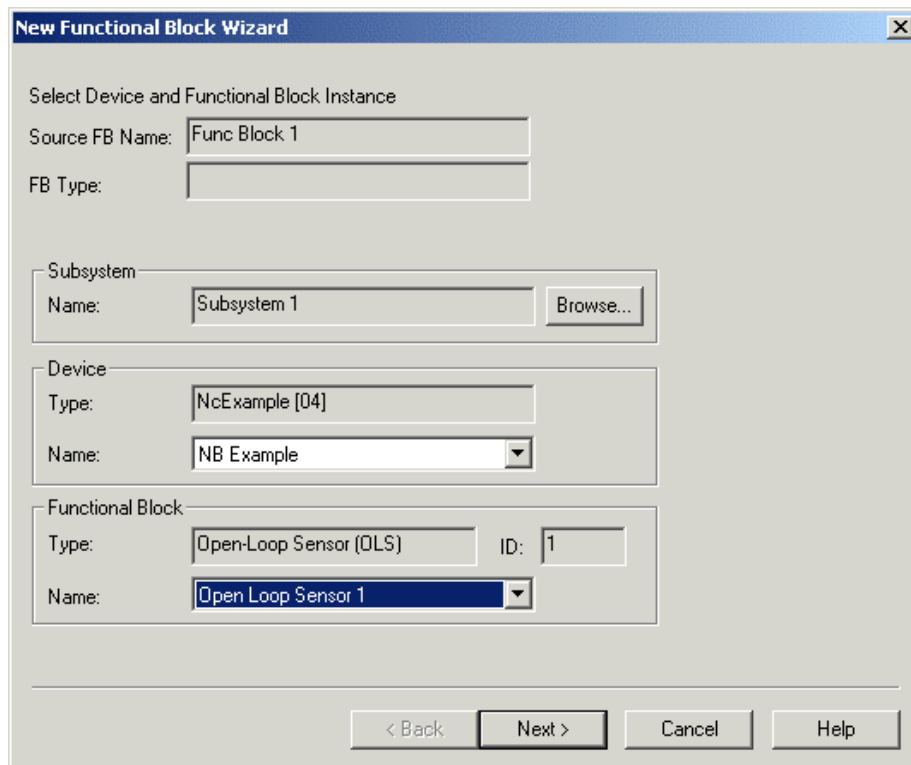
This chapter describes how to use the LonMaker tool to verify that your device has all of its functional blocks, network variables, and configuration properties, and to verify that all these components are functioning correctly.

---

## Testing a NodeBuilder Device

Once you have built your NodeBuilder project, you can use the LonMaker tool to verify that it is functioning correctly. To test a NodeBuilder project, follow these steps:

1. Open the LonMaker drawing that contains the NodeBuilder device. The device must be built and it must be associated with the appropriate LNS device template.
2. Open the LonMaker drawing that contains the device.
3. If the LonMaker drawing does not contain a device shape for the device, drag a Development or Release Target shape from the NodeBuilder Basic Shapes stencil to your drawing and assign it the LNS device template associated with your device.
4. Drag a functional block shape from the NodeBuilder Basic Shapes stencil to your drawing. The New Functional Block Wizard appears, as shown in the following figure:



The screenshot shows the 'New Functional Block Wizard' dialog box. It is divided into several sections for configuration:

- Select Device and Functional Block Instance:** Contains 'Source FB Name' (Func Block 1) and 'FB Type' (empty).
- Subsystem:** Contains 'Name' (Subsystem 1) and a 'Browse...' button.
- Device:** Contains 'Type' (NcExample [04]) and 'Name' (NB Example).
- Functional Block:** Contains 'Type' (Open-Loop Sensor [OLS]), 'ID' (1), and 'Name' (Open Loop Sensor 1).

At the bottom, there are four buttons: '< Back', 'Next >', 'Cancel', and 'Help'.

5. Select the NodeBuilder device in the **Device Name** field and one of the functional blocks in the **Functional Block Name** field. Continue through the *New Functional Block Wizard* as described in the LonMaker User's Guide. Set **Create shapes for all network variables**. A new functional block shape appears in the LonMaker drawing.
6. Repeat steps 4 and 5 for each functional block in your device. If the device contains any network variables or configuration properties that are not associated with a specific functional block, the device will contain a functional block named **Virtual Functional Block**. Create this functional block as well. Verify that all functional blocks defined in the Code Wizard can be



10. Right-click each network variable and configuration property and select **Properties** from the shortcut menu. The Network Variable Properties or Configuration Property Properties dialog opens. Use this dialog to confirm that the network variable or configuration property has the correct type and size.
11. Change input network variable and configuration property values and confirm that the device hardware and application work as expected.
12. Switch to the LonMaker drawing and connect the functional blocks to other functional blocks on this device or other devices and test their functionality. See *Testing Your Device Interface* in the NodeBuilder Quick-Start tutorial for an example of this. The device must be attached to the network and in the Online state to perform this step.

If any of these steps do not give the expected result, return to the NodeBuilder tool and check your code. Use the *NodeBuilder debugger* to help you pinpoint problems.

# 12

## Creating Custom LonMaker Shapes



This chapter describes how to create a LonMaker stencil containing custom LonMaker shapes for your devices and functional blocks. Custom LonMaker shapes make your devices easier to install and configure. You can provide your custom stencils to network integrators with to allow them to quickly integrate your devices into their LONWORKS networks using the LonMaker tool.

---

## Creating a New LonMaker Stencil

You can create a LonMaker stencil containing custom LonMaker shapes for your NodeBuilder device and functional blocks. Custom LonMaker shapes allow you to create LonMaker shapes designed to be used with your devices and their functional blocks. You can provide these custom shapes to network integrators to allow them to quickly integrate your device into their LONWORKS network using the LonMaker tool.

To create custom LonMaker shapes, you must first create a new LonMaker stencil to contain those shapes. To create a new LonMaker stencil, follow these steps:


1. Open the LonMaker drawing containing the NodeBuilder device for which you want to make custom shapes.
2. Open the LonMaker tool's **File** menu, point to **Stencils**, and select **New Stencil**. A blank LonMaker stencil named **Stencil1** appears.
3. Click the document icon  on the new stencil's title bar and then click **Save As**.
4. Select a folder for the new stencil. You can choose any folder, but if you save the stencil in the LONWORKS LonMaker\Visio folder (c:\LonWorks\LonMaker\Visio by default), it will appear with the built-in LonMaker stencils when you click the Visio **Open Stencil**  button.
5. Enter a name for the stencil file.
6. Click **Save**. Visio creates the stencil file with a .vss extension.

See *Creating a Custom Shape for a NodeBuilder Device*, *Creating Custom Shapes for Functional Blocks* for information on creating, and *Creating Complex Custom LonMaker Shapes* for information on adding shapes to the stencil.

---


## Creating a Custom Shape for a Device

You can create a custom LonMaker shape for your device. When an integrator drags this custom device shape to a LonMaker drawing, the LonMaker **New Device Wizard** automatically selects the appropriate device template and application for the new device. The new device will have the same properties as the device used to create the template. To create a custom LonMaker device shape for a device, follow these steps:

1. Create a new stencil as described in *Creating a New LonMaker Stencil*, or open an existing stencil by clicking the Visio **Open Stencil**  button and selecting an existing stencil.
2. Create a new device using the LonMaker Device shape as described in the *LonMaker User's Guide*. Assign the LNS device template that you created for your device to this new device. Assign the name that you want to appear on your custom LonMaker device shape to this device. Set **Location** and **Ping Interval** to the values you want saved with the custom device shape in the LonMaker stencil. Any changes made to **Description** will not be saved with the custom shape.
3. Right-click the Development Target or Release Target Device shape in the



LonMaker drawing and select **Properties** from the shortcut menu. The **Device Properties** dialog appears.

4. Select the **Advanced Properties** tab.
5. Set **Non-group Receive Timer** to the value you want saved with the custom device shape.
6. Click **OK** to save the device properties.
7. Hold the CTRL key and drag the device shape to the stencil that you created or opened in step 1. A new custom LonMaker shape appears in the stencil with the same name as the NodeBuilder device.
8. Click the document icon () on the stencil's title bar and then click **Save**.


---


## Creating Custom Shapes for Functional Blocks

You can create custom functional block shapes for each functional block in your device. The standard LonMaker Functional Block shape has no network variable shapes; if it is used to create a functional block in a LonMaker drawing, a network integrator must manually add network variable shapes or choose to add all network variables to the new functional block. You can create custom functional block shapes so that network integrators have quick and easy access to the appropriate network variables for your functional blocks.

You can set configuration property values on the custom functional block. This means you can create several versions of the same functional block for different configurations of the functional block. This can further reduce the network design time for your devices.

To create a LonMaker custom functional block shape, follow these steps:

1. Create a new stencil as described in *Creating a New LonMaker Stencil*, or open an existing stencil by clicking the Visio **Open Stencil** () button and selecting an existing stencil.
2. Create a device as described in *Creating a Custom Shape for a Device*.
3. Create functional block shapes for the device as described in the *LonMaker User's Guide*. Assign the name that you want to appear on each of the custom LonMaker functional block shapes to each of the functional blocks. Click **Create shapes for all network variables** to add a network variable shape for every network variable on a functional block.
4. If you did not create shapes for all network variables, add input and output network variable shapes to the functional blocks as described in the *LonMaker User's Guide*.
5. You need not add all network variable shapes to the functional block shape. For example, you can create multiple custom functional block shapes, each of which exposes a different set of network variables, depending on the function.
6. Right-click each functional block and select **Browse** from the shortcut menu. The LonMaker Browser opens. If you have already designed and registered an LNS Device Plug-in to configure the functional block you can select **Configure** instead to start your plug-in.
7. Use the LonMaker Browser or your plug-in to configure the functional block. You can modify network variable and configuration property values. Once you have finished, click **OK**.

8. Hold the CTRL key and drag the new functional block shape to the stencil that you created or opened in step 1. A new custom LonMaker functional block shape appears in the stencil and may be used in any LonMaker drawing containing the appropriate device.
9. Repeat steps 3 through 7 for each functional block on the device.
10. Click the document icon () on the stencil's title bar and then click **Save**.

---

## Creating Complex Custom LonMaker Shapes

custom LonMaker shapes with multiple functional blocks, devices, and connections. You can create custom LonMaker shapes for multiple connected functional blocks, for a device and all of its configured functional blocks, or for supernodes that consist of many devices, functional blocks, and connections. To accomplish this, select multiple shapes or supernodes and drag them to a custom stencil. See the *LonMaker User's Guide* for more information regarding complex custom LonMaker shapes.

# 13

## Creating an LNS Device Plug-in for a NodeBuilder Device

This chapter describes how to start the LNS Device Plug-in Wizard from the NodeBuilder tool. See the *LNS Plug-in Programmer's Guide* for more information on the LNS Device Plug-in Wizard and creating plug-ins.

---

## Introduction to LNS Device Plug-ins

An LNS device plug-in is an LNS application that provides a simple device-specific configuration tool that is designed to be launched from another LNS application such as the LonMaker tool. Plug-ins make your devices easier to install and configure since you can use them to provide an application-specific view of your devices and functional blocks.

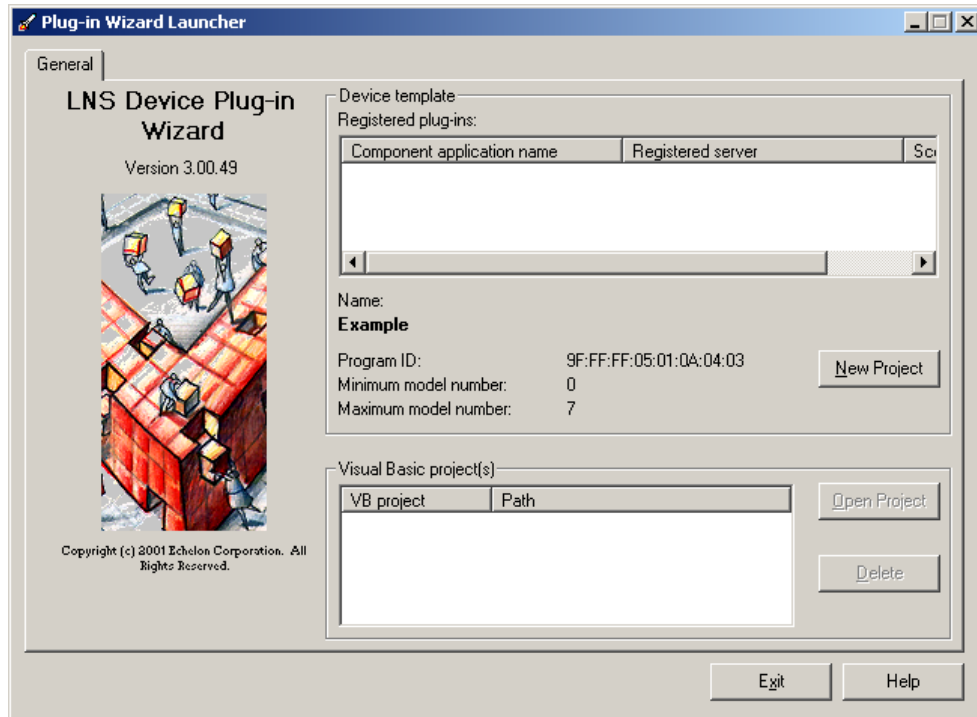
You can use Microsoft Visual Basic or Visual C++ to create an LNS device plug-in. If you are using Visual Basic, you use the LNS Device Plug-in Wizard to greatly reduce the time required to develop a plug-in. Microsoft Visual Basic 6.0 (with service pack 5 or better) must be installed on your PC before installing the NodeBuilder tool in order to use the LNS Device Plug-in Wizard. You can reinstall the NodeBuilder software if you install Visual Basic after installing the NodeBuilder software.

---

## *Starting the LNS Device Plug-in Wizard*

You can start the LNS Device Plug-in Wizard from the NodeBuilder Project Manager. This provides the fastest way to start developing plug-ins. You can also start the LNS Device Plug-in Wizard from Visual Basic as described in the *LNS Plug-in Programmer's Guide*. To start the LNS Device Plug-in Wizard from the NodeBuilder Project Manager, follow these steps:

1. Open the NodeBuilder project containing the device template for your plug-in.
2. Right-click the device template and then select **Plug-in Wizard** from the shortcut menu. The following dialog appears listing the selected device template, device template properties, and the registered plug-ins for the device template:



3. To create a new plug-in, click the **New Project** button. The **Program ID**, **Minimum Model Number**, and **Maximum Model Number** will automatically be added to the project configuration. To work on an existing project, select the project and then click **Open Project**. The LNS Device Plug-in Wizard appears. See *Using the LNS Device Plug-in Wizard* in the *LNS Plug-in Programmer's Guide* for details on running the plug-in wizard.

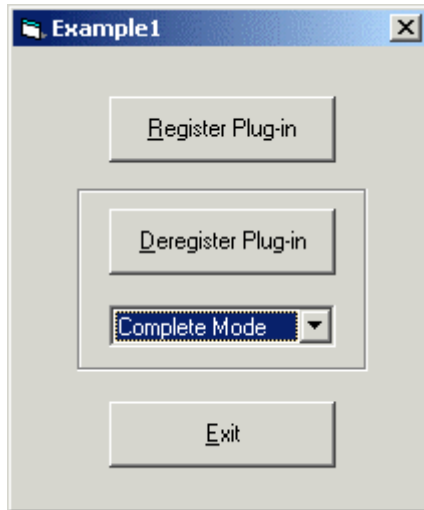
**Note:** If you start the LNS Device Plug-in Wizard from the NodeBuilder tool and your Visual Basic project uses the Source Safe add on, the LNS Device Plug-in Wizard starting process may time-out while you are dealing with Source Safe dialogs. You can ignore (Windows 2000, ME) or close (Windows 98) the Source Safe dialog and allow the LNS Device Plug-in Wizard to launch.

---

## Registering and Running your LNS Device Plug-in

Once you have created an LNS device plug-in for your NodeBuilder device, you must register the plug-in with Windows and with the LonMaker network containing the device. To accomplish this, follow these steps:

1. In Visual Basic, open the **File** menu and select **Make <Project Name>**. A dialog opens allowing you to choose the name and location for the executable file (.exe extension). Choose a name and location and click **OK**. Wait for the project to compile.
2. Once the project has compiled, use Windows Explorer to browse to the plug-in executable (the .exe file created in the previous step). Run this file by double-clicking it. The following dialog opens:



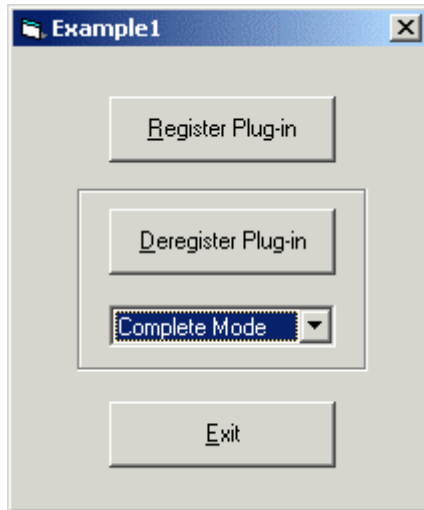
3. Click the **Register Plug-in** button to register the plug-in with Windows.
4. Click **OK** after the plug-in is registered, and then click **Exit** to exit the registration window.
5. If the LonMaker network containing the device is not already open, open it. When you get to the plug-in registration screen verify that the new plug-in appears in the **To Be Registered** field. If the network is already open, open the **LonMaker** menu, select **Network Properties**, and then select the **Plug-in Registration** tab. Verify that the new plug-in appears in the **To Be Registered** field.
6. If you are opening a LonMaker network, finish the start-up wizard. If the network is already open, click **OK** in the **Network Properties** dialog. The plug-in will be registered with LNS and you will be able to use your plug-in. See the *LNS Plug-in Programmer's Guide* for more information on plug-in capabilities.

---

## Deregistering your LNS Device Plug-in

You can deregister a plug-in to remove it from the Windows registry and the LNS database. To accomplish this, follow these steps:

1. Run your plug-in as described in *Registering and Running your LNS Device Plug-in*. The following dialog appears:



2. Click the arrow beneath the **Deregister Plug-in** button and select one of the following:

**Minimal Mode**

Removes the entries for this plug-in from the Windows registry.

**Complete Mode**

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

**Exhaustive Mode**

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

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

Do not try to deregister a plug-in when you have started it stand-alone from the Visual Basic debugger. This will cause the deregistration to fail.

3. Click **Deregister Plug-in**.





# 14

## Creating a Human-Machine Interface

This chapter describes how to create a human-machine interface (HMI) for your NodeBuilder device.

---

## Human-Machine Interfaces

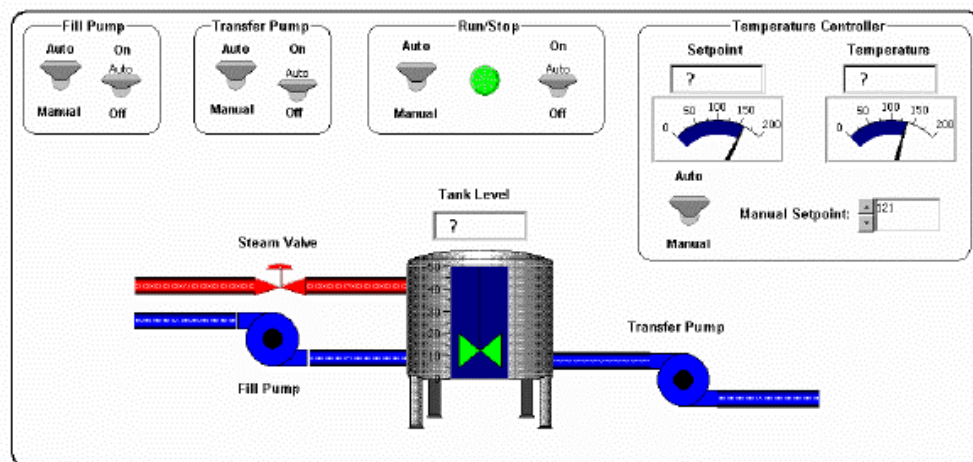
A human-machine interface (HMI) is an application supplied to system operators and end users that allows them to monitor and control devices in a LONWORKS network using an intuitive graphical interface. It differs from an LNS device plug-in in that while LNS device plug-ins are intended to be used by network installers to configure devices in a network, an HMI is intended to be used by operators and end-users on installed and active networks. For example, in a network designed to control the heating and cooling in a building, an HMI could be designed allowing the user to view temperatures of each room using a floor plan of the building and set the temperature in each room using graphical sliders or dials.

---

### *LonMaker Integration Tool*

You can use the LonMaker Integration Tool to build a simple HMI. The LonMaker tool's HMI capability provides a low-cost platform for delivering simple operator interfaces. It is not designed to replace high-end HMI tools such as Wonderware InTouch or Intellution FIX. The LonMaker tool's HMI application is sufficient when you want to, for example, monitor and control states of values or graphically represent interactions in the network. The high-end HMI tools are best for representing more complex types of network interactions. These tools are developed with a scripting language tuned to specifically address HMI tasks. In addition, these tools offer components that provide reporting and analysis, history, alarm logging, event handling, and Internet-enabling.

The following figure is an example HMI developed with the LonMaker tool and third-party ActiveX controls from National Instruments.



See *Creating HMI Applications* in the *LonMaker User's Guide* for more information about using the LonMaker Integration Tool to monitor and control LONWORKS networks.

---

## Third-Party HMIs and the LNS DDE Server

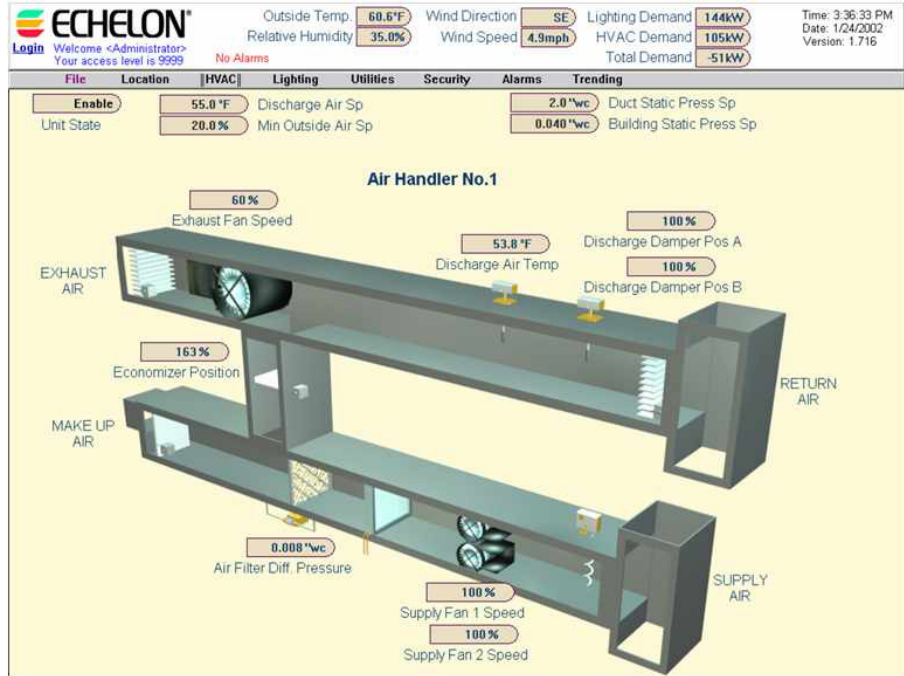
If your HMI requirements are more complex than can be provided by the LonMaker Integration Tool, you can use a third-party HMI tool. If you use the LonMaker tool to install your networks, it is important to use an HMI tool that either includes an LNS 3 (or newer) driver, or is compatible with an LNS 3 (or newer) driver such as the LNS DDE Server. You can find a list of both types of tools at [www.echelon.com/lns](http://www.echelon.com/lns). Select *LNS Tools* from the navigation box to display a list of LNS tools and LNS-compatible tools from many manufacturers. Scroll down to *LNS-Enabled HMI/SCADA Tools* to see a list of HMI tools.

If you are using an HMI tool that does not include an LNS driver, you will need to use a stand-alone LNS I/O driver that is compatible with the tool. HMI tools typically support open protocols for interfacing with I/O drivers. The two most typical protocols used for I/O servers are OLE for Process Control (OPC) and Dynamic Data Exchange (DDE). Many HMI tool vendors also support proprietary interfaces for I/O drivers. For example, Wonderware supports the SuiteLink protocol for many of their products including the InTouch HMI tool. High-performance I/O drivers can be written for any of these protocols as long as they are based on LNS 3 or newer. This is because the bottleneck for I/O server performance tends to be the network interface used to attach the HMI computer to the LONWORKS network. Network interfaces that do not support the LNS 3 Fast Network Interface protocol are limited to approximately 200 updates per second maximum for bound connections and approximately 50 updates per second for polled values. Network interfaces that support the LNS 3 Fast Network Interface protocol offer substantially better performance. For example, an LNS 3 I/O driver used on a LONWORKS/IP channel can achieve performance of over 1,000 updates per second for both bound and polled updates. This is possible using either the OPC or DDE protocol, or a proprietary protocol such as SuiteLink.

A critical requirement for any LONWORKS I/O driver is that it must provide access to all the points that will be required to create an HMI. Points may be network variables, network variable fields, configuration properties, configuration property fields, and functional block controls. Before selecting a LONWORKS I/O driver, verify that it provides access to all of these types of points.

The NodeBuilder Development Tool includes the LNS DDE Server. This is an I/O driver based on LNS 3 that provides DDE and SuiteLink interfaces to a LONWORKS network. It provides access to all type of points including network variables, network variable fields, configuration properties, configuration property fields, and functional block controls. With the LNS DDE Server, any Windows application that can act as a DDE or SuiteLink client can monitor and control one or more LONWORKS networks. The LNS DDE Server uses the naming, addressing, and timing information stored in an LNS Server by a network tool such as the LonMaker Integration Tool.

The LNS DDE Server is compatible with many popular HMI applications such as Wonderware InTouch, Intellution FIX, and National Instruments BridgeView and LabView. The following figure is a sample HMI developed with Wonderware InTouch and the LNS DDE Server:



See the *LNS DDE Server User's Guide* for more information about using the LNS DDE Server to monitor and control LONWORKS networks with third-party HMI tools.

# 15

## Creating a Software Installation

This chapter describes how to create a software installation for your device.

---

## Creating a Software Installation

You can create a software installation program that installs all the files required by integrators to easily design and install your devices in LONWORKS networks. These files include your plug-in, resource files, and other support files. You will typically create the software installation program using a third-party installation application such as the InstallShield® product.

In order for customer to use your device, they must have already installed an LNS tool such as the LonMaker tool, or they must have installed an LNS developer's kit. When an LNS tool or an LNS developer's kit is installed on a computer, a LONWORKS folder is created. By default, this folder is created in C:\LonWorks, but the user can change the name and location of this folder when it is first created. You can find the location of the LONWORKS folder in the Windows registry at the following location:

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

Your installation program should install the following files:

**Application Image Files** If your device supports field downloading of an application image, you can supply application image files to support this feature. Application image files are used by a network tool such as the LonMaker tool to download a compiled application image to your device. During development, these files are located in the **Development** or **Release** target folder in the device template folder. See *Files Created When you Build an Application Image* for a description of these files. Be sure to include the binary device interface file (.xib extension) and optimized device interface file (.xfo extension) to reduce installation time with LNS tools.

Install these files in a folder with your company's name within the folder specified by the following registry key, if present:

```
HKEY_LOCAL_MACHINE\SOFTWARE\LonWorks\LonMaker for Windows\NxeSearchPath
```

If this registry key is not present, install these files in a folder with your company's name within the LONWORKS Import folder (c:\LonWorks\Import\*<Company Name>* by default). If a folder with your company's name is not found in the specified directory, create one.

**Device Interface Files** These files contain a definition of the device interface that is used by network tools to learn the interface to a device, without requiring the device to be physically available. During development, these files are located in the **Development** or **Release** target folder in the device template folder. See *Files Created When you Build an*

*Application Image* for a description of these files. Be sure to include the binary device interface file (.xfb extension) and optimized device interface file (.xfo extension) to reduce installation time with LNS tools.

Install these files in a folder with your company's name within the folder specified by the following registry key, if present:

```
HKEY_LOCAL_MACHINE\SOFTWARE\LonWorks\LonMaker for Windows\XifSearchPath
```

If this registry key is not present, install these files in a folder with your company's name within the LONWORKS Import folder

(**c:\LonWorks\Import**\<Company Name> by default). If a folder with your company's name is not found in the specified directory, create one.

## Resource Files

These files define the functional profiles, network variable types, and configuration property types for the functional blocks, network variables, and configuration properties implemented by your device. You do not have to install resource files for standard resources, since these are installed by both LNS tools and LNS developer's kits, but you do have to install all the user resource files required by your device. You create user resource files with the NodeBuilder Resource Editor as described in *Using the Resource Editor*. During development, these files are located in the folder specified within the NodeBuilder Resource Editor. For each resource file set, you must install the type file (.typ extension), the format file (.fmt extension), the functional profile file (.fpt extension), and any language resource files (language resource file extensions vary by language as described in *Generating Resource Files* in the *Using the Resource Editor* chapter. Do not remove any of these files when uninstalling your device files since these may be used by other devices developed by other parts of your organization.

Install these files in a folder with your company's name within the LONWORKS Types folder (**c:\LonWorks\Types**\<Company Name> by default).

Use the MKCAT catalog utility to register your resource folder with the resource catalog. You must execute the MKCAT utility from the folder that contains the resource catalog (c:\LonWorks\Types by default).

### **LNS Device Plug-in**

If you created a plug-in for your device as described in *Creating an LNS Device Plug-in* install and register it. See *Installing Your Plug-in* in the *LNS Plug-in Programmer's Guide* for more information.

Install your plug-in in a folder with your company's name within the LONWORKS Apps folder (**c:\LonWorks\Apps\<Company Name>** by default).

### **LonMaker Stencil**

If you created a LonMaker stencil containing custom shapes for your device and its functional blocks (see *Creating a New LonMaker Stencil* in the *Creating Custom Shapes* chapter), copy it to a folder with your company's name within a Visio folder within the folder specified by the following registry key, if present:

```
HKEY_LOCAL_MACHINE\SOFTWARE\LonWorks\LonMaker for Windows
```

This will be  
**c:\LonWorks\LonMaker\Visio\<Company Name>** by default.

### **HMI Application**

If you have created an HMI application for your device as described in the *Human-Machine Interfaces* chapter, install and register it. See the documentation for your installation creation software and your HMI development tool for more information on what this entails.



# Appendix A

## NodeBuilder Example

This Appendix gives a step-by-step description of how the NodeBuilder example was created using the NodeBuilder and LonMaker tools.

---

## Introduction to the NodeBuilder Example

The NodeBuilder example provides an example of a device application developed using the NodeBuilder tool. The example includes a LonMaker backup file containing the example device and its functional blocks and an LNS device plug-in designed to configure the application.

The example application is designed to run on a Gizmo 4 I/O board attached to an LTM-10A Platform. These are included with the NodeBuilder 3 tool, but are not included with the NodeBuilder 3 upgrade. If you do not have an LTM-10A Platform or a Gizmo 4 I/O Board, you can still use the NodeBuilder tool to create and compile the application, but you cannot see the effects of the device application.

The NodeBuilder example is in the LONWORKS NodeBuilder\Examples folder (C:\LonWorks\NodeBuilder\Examples by default). This folder contains the following folders:

<b>Database</b>	Contains a LonMaker backup file (NcExa.zip) for a LonMaker network and NodeBuilder project for the example device.
<b>NcExample</b>	Contains a copy of the example NodeBuilder project
<b>PlugIn</b>	Contains the LNS device plug-in used to configure the example device application.
<b>Types</b>	Contains the resource files developed in the course of this example.

You can either follow the procedure in the rest of this appendix to create the LonMaker network and NodeBuilder project, or you can restore a completed LonMaker network and NodeBuilder project from the LonMaker backup file. To install the LonMaker network and NodeBuilder project from the LonMaker backup file, follow these steps:

1. Open the Windows **Start** menu, point to **Programs**, and then click **LonMaker for Windows**. The LonMaker Design Manager appears.
2. Click **Restore**. The Select Backup File dialog appears.
3. Select the NcExa.zip file in the NodeBuilder\Examples\Database folder and then click **Open**. The Confirm Restore dialog appears.
4. Click **OK** to restore the network and project. The LonMaker network and NodeBuilder project are copied to your computer. The NodeBuilder project is copied to the c:\Lm\Source folder and the Restore Complete dialog appears.
5. Click **Yes** to complete the restore.
6. Open the Windows Start menu, point to **Programs>Echelon NodeBuilder Software**, and then click **NodeBuilder Resource Editor**.
7. Right-click the LONWORKS NodeBuilder\Examples\Types folder and then click **Remove** on the shortcut menu.
8. Right-click the resource catalog file at the top of the resource catalog and then click **Add Folder** on the shortcut menu.
9. Select the Lm\Source\NcExa\Types folder and then click **OK**.

10. Click the Echelon LonMaker button in the Windows taskbar to switch to the LonMaker Design Manager.
11. Select the NcExa drawing directory and then click **Open Network**. Complete the LonMaker start-up wizard as described in the *LonMaker User's Guide*.
12. Right-click the NcExa device and then click **Replace** on the shortcut menu. Replace the device with your own device.

This appendix describes how each part of the example was developed using the NodeBuilder tool and the LonMaker tool. The example is divided into tasks. Each task introduces different parts of the device development process, and assumes the previous task has been completed.

This appendix includes a number of code examples. Many of these examples show code that is generated by the Code Wizard as well as the code to be added. In these examples, code that is generated by the Code Wizard is shown in *italics* and code which has been added is shown in **bold**. See the *Neuron C Programmer's Guide* and *Neuron C Reference Guide* for more information on programming in Neuron C.

---

## NodeBuilder Example Task 1: Setting Up The Project

The purpose of this task is to create a LonMaker network that contains the device to be developed and to create the NodeBuilder project. To accomplish this, follow these steps:

1. Start the LonMaker tool and create a new LonMaker network named **NcExa**. Ensure that the LonMaker tool is attached to the network and in the OnNet management mode. See the *LonMaker User's Guide* for more information on creating and opening a LonMaker network.
2. Drag the Development Target Device shape from the NodeBuilder Basic Shapes stencil to the LonMaker drawing. The LonMaker New Device Wizard opens as described in *Device Template Wizard New Device Template*.
3. Choose a name for the new device, set **Commission Device**, and then click **Next**. The second window of the New Device Wizard opens.
4. Click the **Start NodeBuilder** button. The NodeBuilder Project Manager appears. When prompted, indicate that you want to create a new NodeBuilder project. The New NodeBuilder Project Wizard opens as described in *Device Template Wizard New Device Template*.
5. Name the new NodeBuilder project **NcExa** (this will be the default name if you named the LonMaker network this) and click **Next**. The Project Default Setting window opens.
6. In the **Project Default Settings** window, add the location of the Gizmo 4 utility files (i.e., Gizmo4.h) to **Include Search Path** (see *Creating a NodeBuilder Project: Creating a New Project* for more information). By default, the Gizmo 4 utility files are located in the LONWORKS NodeBuilder\Gizmo4 folder (C:\LonWorks\NodeBulder\Gizmo 4 by default). Set **Run NodeBuilder device template wizard** and click **Next**. The Device Template Wizard opens (see *Device Template Wizard New Device Template* for more information about the Device Template Wizard).

7. In the first window of the Device Template Wizard, name the new Device Template `NcExample`. Click **Next**. The *Program ID* window opens.
8. Leave automatic Program ID management enabled and use the *Standard Program ID Calculator* to generate a Program ID. The example provided with the NodeBuilder project uses `9F:FF:FF:05:00:8A:04:00`, but the Program ID you use should use your company's manufacturer ID. See *Creating a NodeBuilder Project: Creating a New Project* for more information about the Standard Program ID Calculator. If you are using a non-FT-10 channel transceiver, when you build you will get a warning that you have a mismatch between the Program ID and the transceiver type. For purposes of the example, you can ignore this warning. If you want to change the Program ID to the appropriate transceiver value, you must set the scope of the resource file created in *NodeBuilder Example Task 5* to 4, so the Program ID of the resource file set will match the Program ID of the device. Click **Next**. The Target Platforms window opens.
9. Set the development target hardware to `LTM-10A RAM`, and the release target hardware to `LTM-10A FLASH`. See *Setting Hardware Template Properties: Hardware* for more information about hardware templates. To proceed immediately to *Task 2*, set **Run NodeBuilder Code Wizard**. Click **Finish**.

---

## NodeBuilder Example Task 2: Configuring the Node Object

The purpose of this task is to create an empty but fully functioning LONWORKS device with a Node Object functional block. The Node Object functional block is used by network tools to manage all the functional blocks on a device. This task uses the Code Wizard (described in *Using the NodeBuilder Code Wizard*) to configure the device's Node Object functional block. This task also adds code to initialize the Gizmo 4 I/O Board.

1. Open the Code Wizard. If you have just used the Device Template Wizard (e.g. in Task 1), you can set **Run NodeBuilder Code Wizard** before exiting the Device Template Wizard. Otherwise, right-click the device template in the Project pane (described in *The Project Pane Device Templates Folder*) and select **Code Wizard** from the shortcut menu. The Code Wizard opens.
2. Click **Generate and Close**. The Code Wizard generates code and returns you to the NodeBuilder Project Manager (see *Overview of the NodeBuilder Project Manager*).
3. Double-click the `common.h` file contained in the `NcExample` device template in the Project pane to open it. Add the following line at the top of the list of include files:

```
#include "Gizmo4.h"
```

This statement makes the Gizmo 4 utility functions and I/O declarations available to all components of the example application. You must have included the folder containing the Gizmo 4 header files in **Include Search Path**. If you did not, right-click the device template, select **Settings** from the shortcut menu, open the **Paths** tab, and update **Include Search Path**.

4. In `NxExample.nc`, in the "when ( reset )" task, add the following lines shown

in bold:

```
when (reset)
{
    GizmoReset();
    GizmoBuzz(TRUE);
    GizmoDisplayString(2,0, "Echelon NEURON C");
    GizmoDisplayString(0,1, "Example Application");
    initAllFblockData();
    executeOnEachFblock(FBC_WHEN_RESET);
    GizmoBuzz(FALSE);
}
```

This code change initializes the Gizmo 4 I/O Board.

5. Save the file by selecting **Save** from the **File** menu.
6. Right-click the **Development** folder and then select **Build** from the shortcut menu. This builds only the development target which is all that is necessary for this example.
7. Once the build has completed, click the LonMaker icon in the Windows taskbar to return to the LonMaker tool. The New Device Wizard opened in Task 1 will still be open.
8. Click the **NodeBuilder Device Template** arrow and select the newly created **NcExample** NodeBuilder device template.
9. Continue through the New Device Wizard in the LonMaker tool (see the LonMaker User's Guide for more information). Set **Load Application Image**, and set **State** to **Online**.

When prompted, press the service pin on the LTM-10A Platform. The application, including the Node Object functional block and the Gizmo utilities, is loaded into the device. The application download takes up to 30 seconds.

Now that you have added the device to the LonMaker drawing and loaded the device with its application, the NodeBuilder Project Manager and the LonMaker tool will automatically load new builds of the application into the device.

10. The Gizmo display shows an Echelon NEURON C Example Application message after loading and commissioning has been completed.
11. Use the LonMaker tool to test the device as described in *Testing a NodeBuilder Device Using the LonMaker Tool*. For example, you can add a Node Object functional block and confirm that it has the appropriate network variables and configuration properties.

---

## NodeBuilder Example Task 3: Adding Digital I/O

The purpose of this task is to add digital input and output functionality to the device. We add a pair of digital actuators and sensor functional blocks to the device, and connect them using the LonMaker tool. Once this task is completed the buttons on the Gizmo 4 can be used to turn on the LEDs.

After completing this task, you will have a fully functioning LONWORKS device. This task and the functional blocks added in this task are kept simple in order to focus on the essential steps. More sophisticated examples follow in the upcoming tasks. To perform this task, follow these steps.

1. Click the NodeBuilder icon in the Windows taskbar to return to the NodeBuilder Project Manager.
2. Right-click the `NcExample` device template and select **Code Wizard** from the shortcut menu. The Code Wizard opens (see *Using the NodeBuilder Code Wizard* for more information on using the Code Wizard).
3. Right-click the **Functional Blocks** folder and select **Add Functional Block** from the shortcut menu. The **Add Functional Block** dialog opens.
4. Add an array of 2 `SFPTOpenLoopSensor` functional blocks to the device (see *Adding Functional Blocks to the Device Template with the Code Wizard*). Name the functional blocks **DigitalInput**. These two functional blocks will be used to control the two push-buttons on the Gizmo 4.
5. Open the `DigitalInput` functional block's Mandatory NVs folder, right-click the network variable contained in the folder, and select Properties from the shortcut menu. The **Network Variable Properties** dialog opens.
6. Rename the network variables to **nvoDigitalInput**, and set the type to `SNVT_switch`. The network variable is implemented as an array of size 2. These network variables will be used to send the value of the digital input on the network (i.e. whether the button is being pressed).
7. Repeat steps 3 and 4, but add an array of 2 `SFPTOpenLoopActuator` functional blocks, and name them **DigitalOutput**. These functional blocks will be used to control the two LEDs on the Gizmo 4.
8. Repeat steps 5 and 6 but rename the **DigitalOutput** functional block's mandatory network variable to **nviDigitalOutput**, and change the type to `SNVT_switch`. These two network variables will be used to receive values from the network to drive the LEDs (i.e. turn them on and off).
9. Right-click the **DigitalOutput** functional block's **Optional CPs** folder and select **Implement Optional CP** from the shortcut menu. The **Implement Optional Configuration Property** dialog opens.
10. Select `nciDefault` from the **FPT Member Name** dialog to implement this configuration property. Name the configuration property **cpDigitalDefault**. A single configuration property will be created for each member of the functional block array (the **Static CP** option is used to create a single configuration property that applies to all functional blocks in the array; this option is discussed in Task 4).  

This configuration property will be used to control the initial state of the physical output lines after power-up or reset. Since it is applied to an actuator, the type of this configuration property is the same as the type of the primary input network variable of the functional block (**nviDigitalOutput**).
11. Click **Generate and Close**.
12. Open the device template's **Source Files** folder and open `ncexample.h` by double clicking it. Add the following lines of code shown in bold:

```

#ifndef _NcExample_H_
#define _NcExample_H_

#define SWITCH_ON 0x01
#define SWITCH_OFF 0x00

```

This code defines enumerations to use for on and off values for the buttons and LEDs.

- Open `DigitalOutput.nc` from the **Source Files** folder and add the following lines of code shown in bold to `DigitalOutputProcessNV()`:

```
void DigitalOutputprocessNV(void)
{
    // drive the LED as appropriate:
    GizmoSetLed(deviceState.nvArrayIndex,
nviDigitalOutput[deviceState.nvArrayIndex].state);
}
```

This code causes the LED to be updated whenever the input network variable on the associated **DigitalOutput** functional block receives an update.

- Open the `DigitalInput.nc` file from the **Source Files** folder and add the following lines of code:

```
void setDOutValue(unsigned uIndex) {
    // set the nvo to reflect the input line state.
    if (fblockNormalNotLockedOut(DigitalInput[uIndex]::global_index)) {
        nvoDigitalInput[uIndex].state
            = input_value ? SWITCH_OFF : SWITCH_ON;
    }
}

when (io_changes(ioButton1)) {
    setDOutValue(0);
}

when (io_changes(ioButton2)) {
    setDOutValue(1);
}
```

This code causes the output network variables on the **DigitalInput** functional blocks to be updated whenever the value from the hardware (i.e. the push-buttons) changes.

The `fblockNormalNotLockedOut()` function ensures that the functional block is enabled. Alternatively, the following clause can also be used for the argument of the `fblockNormalNotLockedOut()` function to retrieve the current functional block index:

```
fblock_index_map[nv_table_index(nvoDigitalInput[uIndex])]
```

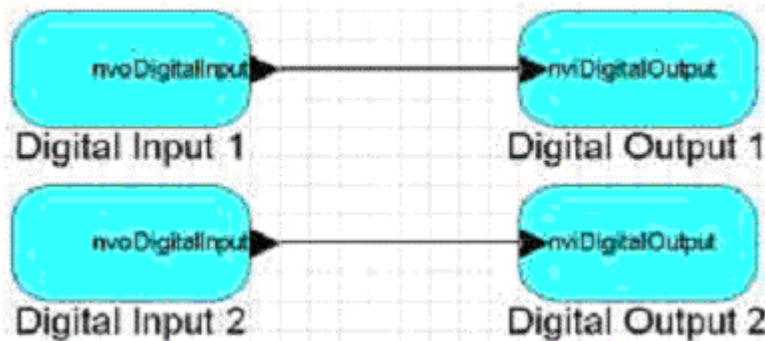
The `DigitalInput[uIndex]::global_index` clause is used to demonstrate the scope operator (`::`), and because this clause is more efficient.

- Open `DigitalOutput.nc` from the **Source Files** folder. Add the following code in bold to the `FBC_WHEN_RESET` else-if statement:

```
else if ((TFblock_command)iCommand == FBC_WHEN_RESET)
{
    // initialize output lines:
    GizmoSetLed(0, DigitalOutput[0]::cpDigitalDefault.state);
    GizmoSetLed(1, DigitalOutput[1]::cpDigitalDefault.state);
    setLockedOutBit(uFblockIndex, FALSE);
}
else if ((TFblock_command)iCommand == FBC_DISABLED)
```

This code causes LEDs to be set to the value specified in the **cpDigitalDefault** configuration property when the device is reset.

16. Build the development target as described in *Building a NodeBuilder Project*. The LonMaker tool automatically reloads the application into the device.
17. Click the LonMaker button in the Windows taskbar to return to the LonMaker tool.
18. Drag 4 functional block shapes to your drawing, one for each of the **DigitalOutput** and **DigitalInput** functional blocks. Set **Create shapes for all network variables** for each functional block.
19. Connect each input network variable to the corresponding output network variable. See the *LonMaker User's Guide* for more information on performing these operations. When you are done, your LonMaker drawing should look something like this:



20. Press the SW1 and SW2 buttons on the Gizmo 4 Board to verify that these now control the LEDs.
21. Use the LonMaker tool to verify the functional blocks behave as expected.

---

## NodeBuilder Example Task 4: Analog Input and Output

The purpose of this task is to add a pair of analog input and output functional blocks to the device. This task shows an implementation-specific configuration property being added to a functional block. To accomplish this task, follow these steps:

1. Click the NodeBuilder button in the Windows taskbar to return to the NodeBuilder tool.
2. Right-click on the device template and select **Code Wizard** from the shortcut menu. The Code Wizard opens.
3. Right-click the device template's **Functional Blocks** folder and select **Add Functional Block** from the shortcut menu. The **Add Functional Block** dialog opens.
4. Add an array of 2 `SFPTanalogInput` functional blocks. Name the functional block array **AnalogInput**.
5. Open the **AnalogInput** functional block's **Mandatory NVs** folder, right-click the **nvoAnalog** network variable in the folder, and select **Properties** from the shortcut menu. The **Network Variable Properties** dialog opens.
6. Rename the network variable to **nvoAnalogInput**.
7. Repeat steps 2 and 3, but add an array of 2 `SFPTanalogOutput` functional blocks, and name the array **AnalogOutput**.



8. Repeat steps 3 and 4, but rename the **nviAnalog** network variable to **nviAnalogOutput**.
9. Right-click the **AnalogInput** functional block's **Implementation-specific CPs** folder and select **Add CP** from the shortcut menu. The **Add Configuration Property** dialog opens.
10. Add an implementation-specific **SCPTupdateRate** configuration property. Name the new configuration property **cpUpdateRate**. Set **Static CP** for this configuration property; this will cause a single configuration property to be added that applies to all functional blocks in the **AnalogInput** functional block array. Set **Initial Value** to **5**. This configuration property will be used to specify how often each AnalogInput functional block will read the analog-to-digital converter (ADC) hardware inputs.  
 Setting the **InitialValue** field to **5** will cause the value of this configuration property to be set to **5** when the application is loaded into the device. The value of "5" is the unscaled value, representing 500ms or 0.5s.
11. Click **OK**. A dialog opens asking you to confirm code generation.
12. Click **Generate and Close**.
13. Open the `AnalogOutput.nc` file from the **Source File** folder and add the following code in bold to the `FBC_WHEN_RESET` else-if statement in the `AnalogOutputDirector()` function:

```

else if ((TFblock_command)iCommand == FBC_WHEN_RESET)
    // init output signals to 0
    GizmoWriteAnalog(0, 0L);
    GizmoWriteAnalog(1, 0L);
    // get going:
    setLockedOutBit(uFblockIndex, FALSE);

```

This code causes the analog output signals to be set to 0 when the device is reset. You could add a default value implementation specific configuration property for a more flexible solution than a hard-coded 0V output after power-up and reset. Since we've discussed and shown implementation of such a configuration property in *Task 3*, this has not been implemented here.

14. Still in the `AnalogOutput.nc` file, add the following code in bold to the `AnalogOutputprocessNV()` function:

```

void AnalogOutputprocessNV(void)
{
    signed long s1OutputValue;

    s1OutputValue = nviAnalogOutput[deviceState.nvArrayIndex];
    s1OutputValue /= 20L;

    GizmoWriteAnalog(deviceState.nvArrayIndex, abs(s1OutputValue));
}

```

This code computes the output value. The `SNVT_lev_percent` network variable type has a valid range of -163.84% to 163.83% in steps of 0.005%. The value expected by the `GizmoWriteAnalog()` function, however, has a value range of 0.0 to 100.0% in steps of 0.1%.

The `s1OutputValue` variable has the correct value but is still a signed variable, and could have the correct absolute value but the incorrect sign. This example uses the `abs()` function to ignore the sign.

15. Open the `AnalogInput.nc` file from the **Source Files** folder. Add the following declarations in bold at the top of the file:

```

#ifndef _AnalogInput_NC_
#define _AnalogInput_NC_

#include "common.h"
#include "AnalogInput.h"

#define AI_FILTERSIZE 4
#define AI_CHANNELS AnalogInput_FBLOCK_COUNT

mtimer ai_timer;
// the buffer for the averaging filter:
unsigned long ai_rawdata[AI_CHANNELS][AI_FILTERSIZE];
// recent value (required to detect changes for minimum NV updates)
unsigned long ai_rawrecent[AI_CHANNELS];

//{{NodeBuilder Code Wizard Start

```

The Gizmo 4's PIC controller does not provide an interrupt upon the availability of new analog data. Therefore, this example reads both channels every **cpUpdateRate** interval, which defaults to 1 minute (the PIC converts every 100ms). The minimum sample rate is 0.1s, which matches the PIC controller's real sample rate. This example averages the last `AI_FILTERSIZE` values obtained for an improved signal quality, where the filter size defaults to 4 and should not be less than two. This implementation will only update the output network variable if the value has been changed.

16. Still in `AnalogInput.nc`, add the following code in bold to the `FBC_WHEN_RESET` else/if statement in the `AnalogInputDirector()` function:

```

else if ((TFblock_command)iCommand == FBC_WHEN_RESET)
    // reset filter and start sampling timer
    memset(ai_rawdata, 0, sizeof(ai_rawdata));
    memset(ai_rawrecent, 0, sizeof(ai_rawrecent));
    ai_timer = AnalogInput[0]::cpUpdateRate * 100L;
    // get going:
    setLockedOutBit(uFblockIndex, FALSE);

```

This code clears out the filter and starts sampling the hardware input when the device is reset.

17. Still in `AnalogInput.nc`, add the following code in bold to the `FBC_WHEN_ONLINE` else-if statement in the `AnalogInputDirector()` function:

```

else if ((TFblock_command)iCommand == FBC_WHEN_ONLINE)
    // start sampling timer:
    ai_timer = AnalogInput[0]::cpUpdateRate * 100L;

```

This code starts the sampling the hardware input when the device is set online.

18. Still in `AnalogInput.nc`, add the following code in bold to the `FBC_WHEN_OFFLINE` else/if statement in the `AnalogInputDirector()` function:

```

else if ((TFblock_command)iCommand == FBC_WHEN_OFFLINE)
    // stop sampling timer:
    ai_timer = 0L;

```

This code stops sampling the hardware input when the device is set offline.

19. Still in `AnalogInput.nc`, add the following code in bold to process expiry of the sampling timer:

```
#endif // _HAS_INPUT_NV_

when (timer_expires(ai_timer)) {
  int iIndex;
  int iChannel;
  unsigned long ulValue;

  // are we in business?
  if (fblockNormalNotLockedOut(AnalogInput[0]::global_index)) {
    // yes we are. Repeat for each channel:

    for (iChannel = 0; iChannel < AI_CHANNELS; ++iChannel) {
      // Move historic data:
      for (iIndex = 0; iIndex < AI_FILTERSIZE-1; ++iIndex) {
        ai_rawdata[iChannel][iIndex] =
          ai_rawdata[iChannel][iIndex + 1];
      }

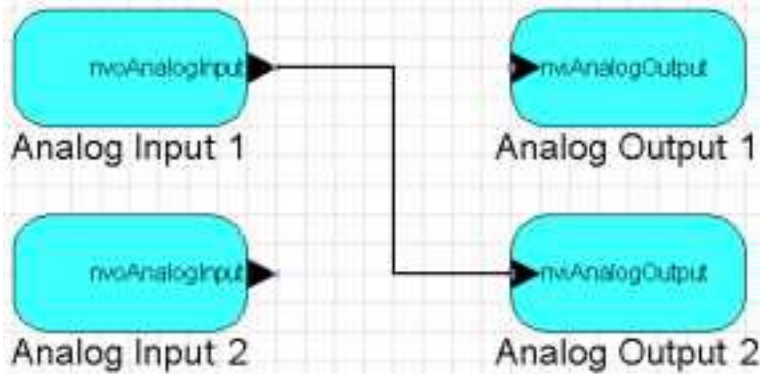
      // fetch current value (store in filter history and also
      // use current value to initialize current result
      ulValue = ai_rawdata[iChannel][AI_FILTERSIZE-1] =
        GizmoReadAnalog(iChannel);

      // compute average over averaging window:
      for (iIndex = 0; iIndex < AI_FILTERSIZE-1; ++iIndex) {
        ulValue += ai_rawdata[iChannel][iIndex];
      }
      // now we've got the sum, let's divide in a reasonable
      // way. That is, we divide and round if appropriate:
      if ((ulValue % AI_FILTERSIZE) >= (AI_FILTERSIZE / 2)) {
        ulValue = ulValue / AI_FILTERSIZE + 1L;
      } else {
        ulValue /= AI_FILTERSIZE;
      }

      // has it changed?
      if (ulValue != ai_rawrecent[iChannel]) {
        // it has indeed. Update history and network variable
        ai_rawrecent[iChannel] = ulValue;
        nvoAnalogInput[iChannel] =
          ((SNVT_lev_percent)ulValue) * 20L;
      }
    } // next channel
  } // not in business

  // re-load timer. We do not use auto-reloading ("mtimer
  // repeating...") because we want the update frequency to be
  // adjustable through cpUpdateRate.
  ai_timer = AnalogInput[0]::cpUpdateRate * 100L;
}
```

20. Build the development target. The `LonMaker` tool automatically loads the new application into the device hardware.
21. Drag 4 new functional blocks to your drawing, one for each **AnalogInput** and **AnalogOutput** functional block. Set **Create shapes for all network variables** for each functional block.
22. Connect `AnalogInput[0]` to `AnalogOutput[1]`. See the *LonMaker User's Guide* for more information. When you are done, your `LonMaker` drawing should look something like this:



23. Insert jumpers between pins 1 and 2 of JP7 and JP8 in the lower right-hand corner of the Gizmo 4 board. These jumpers connect the AOUT1 output to the AIN1 input, and the AOUT2 output to the AIN2 input.
24. Browse the **Analog Output 1** and **Analog Input 2** functional blocks using the LonMaker Browser. Verify that an update to the **nviAnalogOutput** network variable on **Analog Output 1** gets reflected in the **nvoAnalogInput** network variable on **Analog Input 2**. Be sure to allow for a generous conversion error - the Gizmo 4 has a 10 bit ADC and an 8 bit DAC converter; being daisy-chained like this causes conversion errors to be multiplied. Also verify correct operation by changing the **cpUpdateRate** configuration property value, disabling one or more functional blocks in the loop, etc.

---

## NodeBuilder Example Task 5: Simple Translator

The purpose of this task is to create a simple user-defined functional profile, `UFPTtranslator`, using the NodeBuilder Resource Editor. This functional profile translates an input network variable of type `SNVT_temp_p` into an output network variable of type `SNVT_lev_percent`, allowing a temperature sensor functional block to be connected to the analog output functional blocks on this example device. For more information about the resource editor, see *Editing Resource Files*.

This task covers the basic implementation of a very simple user functional profile. Task 6 implements a number of improvements in the design. To accomplish this task, follow these steps:

1. Click the NodeBuilder button in the Windows taskbar to return to the NodeBuilder tool. Right-click the device template and select **Code Wizard** from the shortcut menu. The Code Wizard opens.
2. Right-click the LONWORKS NodeBuilder\Examples\Types folder (`c:\LonWorks\NodeBuilder\Examples\Types` by default) and then click **Remove** on the shortcut menu.
3. Right-click the resource catalog file at the top of the resource catalog (the file with the `.cat` extension), and then select **Add Directory** from the shortcut menu. Browse to an existing folder or create a new one.
4. Right-click the folder added in step 3 and select **New Resource File Set** from the shortcut menu. The **New Resource File Set** dialog opens.
5. Set **Scope** to **Scope 3 - Manufacturer Class** and set **Program ID** to the Program ID that you chose for the device in *Task 1* (if the manufacturer ID

does not match, you will not be able to access this resource file set for use with your device). Name the new resource file set **NcExample**.

6. Right-click the **Functional Profile Templates** folder in the new resource file set and then select **New FPT** from the shortcut menu. A new functional profile will be created with the default name of UFPT1.
7. Rename the functional profile **UFPTtranslator**.
8. Right-click the resource catalog file and select **Options** from the shortcut menu. The Resource Editor Options dialog appears.
9. Set **Show obsolete resource items**. The network variable to be added in step 11 is of an obsolete type. Click **OK**.
10. Right-click the **UFPTtranslator** functional profile and select **Open** from the shortcut menu. The **Modify Functional Profile Template** dialog opens.
11. In the left-hand pane of this dialog, open the C:\LONWORKS\Types\Standard resource file set folder, and browse to the SNVT\_lev\_percent network variable type. Drag this network variable type to the **UFPTtranslator** functional profile's **Mandatory NVs** folder in the right-hand pane. The network variable will be added to the **Mandatory NVs** folder with the name **nviManNV1**.
12. Repeat step 11 but add a SNVT\_temp\_p network variable to the **UFPTtranslator** functional profile's **Mandatory NVs** folder. The new network variable will be named **nviManNV2**.
13. Click **nviManNV1** (the SNVT\_lev\_percent type network variable added in step 11). The right-hand portion of the dialog displays the network variable properties. Change **Name** to **nvoPercentage** and set **Output** to indicate that it is an output network variable.
14. Repeat step 13 for **nviManNV2** (the SNVT\_temp\_p type network variable added in step 12). Change **Name** to **nviTempP**, set **Input** to indicate that it is an input network variable, and set **Principal NV** to make this the functional profile's principal network variable.
15. Click **OK** to close the **Modify Functional Profile Template** dialog.
16. In the Interface pane of the Code Wizard, right-click the device template's **Functional Blocks** folder and add a single **UFPTtranslator** functional block to the device. Set **User-defined** and set **Scope** to 3 in the **Add Functional Block** dialog to access the new resource file set.
17. Click **OK**. Click **Generate and Close** to generate code and exit the Code Wizard.
18. Build and load the application.
19. Add the **Translator** functional block to the LonMaker drawing.
20. Use the LonMaker Browser to browse the translator. Enable monitoring for **nvoPercentage**, and force **nviTempP** to several values within and outside the supported range of 0-+30°C.
21. Connect the **nvoPercentage** output network variable to the input network variable of one of the analog output functional block blocks, connect a multimeter to the relevant analog output.
22. Use the LonMaker Browser to change the **nviTempP** value, and observe the results.

---

## NodeBuilder Example Task 6: Enhancing the Translator

The purpose of this task is to refine and enhance the **UFPTtranslator** functional profile defined in Task 5. The **UFPTtranslator** functional profile, with hardcoded input and output limits, is very specialized for our task. This task adds 2 configuration properties for the input range (replacing the hardcoded minimum and maximum of 0 and 30 degrees Celsius) and two configuration properties to define the minimum and the maximum output signal values.

The configuration properties used to set the minimum and maximum output will use the `SCPTminRnge` and `SCPTmaxRnge` types. The *SNVT and SCPT Master List* states (quoted from `SCPTminRnge`): *This configuration property is used to limit the minimum value of the primary output network variable for the object.*

There are no existing configuration property types appropriate for limiting the input signal range (`SCPTHighTemp` looks like a promising candidate, however *The SNVT and SCPT Master List* states “this configuration property indicates the high alarm set point for the `nvoAlarmAirTemp`,” so this type is not appropriate). Therefore, this task will create 2 UCPTs, one for the minimum input and one for the maximum input.

To accomplish this task, follow these steps:

1. Click the NodeBuilder button in the Windows taskbar to return to the NodeBuilder tool.
2. Right-click the device template and select **Code Wizard** from the shortcut menu. The Code Wizard opens.
3. In the Resource pane, browse to the **UFPTtranslator** functional profile created in *Task 5*. Right-click the functional profile and select **Open** from the shortcut menu. The **Modify Functional Profile Template** dialog opens.
4. Select the **nviTempP** network variable in the **Mandatory NVs** folder and clear **Principal NV**.
5. Select the **nvoPercentage** network variable in the **Mandatory NVs** folder and set **Principal NV**. This must be done because the `SCPTminRange` and `SCPTmaxRange` standard configuration property types should apply to the principal network variable, as stated in the SCPT description above.
6. In the left-hand pane of this dialog, expand the LONWORKS Types\Standard (c:\LonWorks\Types\Standard by default) resource file set folder, and browse to the `SCPTminRnge` configuration property type. Drag the configuration property to the **UFPTtranslator** functional profile's **Mandatory CPs** folder in the right-hand pane. The configuration property will be added to the **Mandatory CPs** folder with the name **nciManCP1**.
7. Repeat step 6 but add a `SCPTmaxRnge` configuration property to the **UFPTtranslator** functional profile's **Mandatory CPs** folder. The new configuration property will be named **nciManCP2**.
8. Click **nciManCP1** (the `SCPTminRnge` configuration property added in step6). The right-hand portion of the dialog displays the configuration property

- properties. Change **Name** to **cpTransOutMin**.
9. Repeat step 8 for **nviManCP2** (the `SCPTmaxRnge` network variable added in step 7). Change **Name** to **cpTransOutMax**.
  10. Click OK to close the **Modify Functional Profile Template** dialog.
  11. Right-click NcExample resource file set's **Configuration Property Types** folder and select **New CPT** from the shortcut menu. The **New Configuration Property Type** dialog opens.
  12. Set **CP Name** to **UCPTminTemp**, set **Inherited from a network variable**.
  13. Repeat steps 11 and 12, but set **CP Name** to **UCPTmaxTemp**.
  14. Right-click the **UFPTtranslator** functional profile and select **Open**. The **Modify Functional Profile Template** dialog opens.
  15. Add one configuration property of each of the new types to the **Mandatory CPs** folder. Name them **cpTransInMin** and **cpTransInMax**, respectively. Click **OK** to close the **Modify Functional Profile Template** dialog.
  16. Change the "Applies To" setting so that the **cpTransInMin/cpTransInMax** properties apply to the input network variable, and **cpTransOutMin/cpTransOutMax** to the output network variable. Click **OK** to close the **Modify Functional Profile Template** dialog.
  17. In the right-hand pane of the Code Wizard, right-click the **Translator** functional block and select **Refresh** from the shortcut menu. The functional block will be refreshed to include the new configuration properties that you added to the functional profile.
  18. Assign default values to each new configuration property on the **Translator** functional block. The following values will cause the functional block to behave just as it did after Task 5.

<b>cpTransInMin</b>	0
<b>cpTransInMax</b>	3000
<b>cpTransOutMin</b>	0
<b>cpTransOutMax</b>	10000

This step sets defaults for the configuration properties on this device only. This is different than setting the defaults in the functional profile, which will set the defaults for all functional blocks created from that functional profile unless they are otherwise specified.

19. Click **Generate and Close**.
20. Click **Yes** to generate resource files.
21. Open `Translator.nc` from the Source Files folder and add the following code in bold to the `TranslatorprocessNV()` function:

```
void TranslatorprocessNV(void)
{
long lValue;

    // get scaled value:
    lValue = Translator::nviTempP;

    // limit temperature to supported range 0-30.00 Celcius
    lValue = max(Translator::nviTempP::cpTransInMin,

```

```

    min(lValue, Translator::nviTempP::cpTransInMax);
Translator::nvoPercentage = (short)muldiv(2L*lValue,
Translator::nvoPercentage::cpTransOutMax,
Translator::nviTempP::cpTransInMax);

```

This code takes a `SNVT_temp_p` value, which has a range of -273.17 to 327.66 in steps of 0.01, and converts it into a `SNVT_lev_percent` value, which has a range of 163.84% to +163.83% in steps of 0.005%. See the *SNVT and SCPT Master List* for more information.

This particular application limits the output signal range to between 0 and 100%. It also limits the range of the valid input values from 0 to 30° Celsius for room temperature values (i.e. 30° C or more results in a 100% output signal; 0° C or less results in a 0% output signal).

In this task, all the above limits are hardcoded. Task 6 shows how to make these limits changeable.

With hard-coded factors, and using unscaled network variable values, the formula for conversion is:

$$\text{percentage} = (\text{tempP} * 2) * (100 / 30) = \text{tempP} * 20 / 3$$

The  $(\text{tempP} * 2)$  term transforms an unscaled `SNVT_temp_p` value into an equivalent unscaled `SNVT_lev_percent` value, and the second  $100/30$  term adjusts so that 30° C converts to 100% of the output signal range. Both terms could be combined in a single factor, but this example uses both for double-precision intermediate results

22. Build the development target. The NodeBuilder and LonMaker tools automatically load the new application into the device hardware.
23. Add the **Translator** functional block to the LonMaker drawing.
24. Use the LonMaker Browser to browse the translator. Set the new configuration properties to various values, enable monitoring for **nvoPercentage**, and force **nviTempP** to several values within and outside the set range of 0-+30°C. Connect **nvoPercentage** to the input network variable of one of the analog output functional block blocks, connect a voltmeter to the relevant analog output, use the LonMaker Browser to change the **nviTempP** value, and observe the results.
25. Build and load the device.

---

## NodeBuilder Example Task 7: Temperature Sensor

The purpose of this task is to implement a standard temperature sensor profile (SFPT #1040, see the LONMARK website for more information: [www.lonmark.org](http://www.lonmark.org)) to provide a temperature sensor implementation for the Gizmo 4 Board's temperature sensor hardware. This task demonstrates the use of floating-point vs. fixed-point arithmetic in Neuron C. To perform this task, follow these steps:

1. Click the NodeBuilder button in the Windows taskbar to return to the NodeBuilder tool. Right-click on the device template and select **Code Wizard** from the shortcut menu. The Code Wizard opens.
2. Right-click the device template's **Functional Blocks** folder and select **Add Functional Block** from the shortcut menu. The **Add Functional Block**



dialog appears.

3. Add a single SFPThvacTempSensor functional block. Name the new functional block **TempSensor**. Click **OK**.
4. Right-click the TempSensor functional block's **Optional NVs** folder and select **Implement Optional NV** from the shortcut menu. The **Implement Optional Network Variable** dialog appears.
5. Implement the **nvoFloatTemp** network variable. This network variable has the SNVT\_temp\_f type. Click **OK**.
6. Change the names of the three mandatory configuration properties to **cpMaxSendTime**, **cpMinDelta**, and **cpMinSendTime**, respectively.
7. Click **Generate and Close**.
8. Open TempSensor.h from the **Source Files** folder and add the following code in bold:

```
#include "common.h"

SNVT_temp_pHVACTempOld; // most recent value, used for heartbeats
#define HVAC_CORETICK 500UL // internal sampling rate
//and minimum heartbeat interval
mtimer repeating hvac_coretick = HVAC_CORETICK;

unsigned long HvacMinSendTimer;
unsigned long HvacMaxSendTimer;
float_type f100 = {0, 0x42, 0x01, 0x48, 0 }; // 100.0 - see NXT.EXE
//utility for initializer

//{{NodeBuilder Code Wizard Start
```

9. Open TempSensor.nc from the **Source Files** folder add the following code in bold to the FBC\_WHEN\_RESET else-if statement in the TempSensorDirector() function:

```
else if ((TFblock_command)iCommand == FBC_WHEN_RESET)
HVACTempOld = 0;
UpdateTemperature();
// get going:
setLockedOutBit(uFblockIndex, FALSE);
break;
```

10. Still in TempSensor.nc, add the following functions to the code:

```
#endif // _HAS_INPUT_NV_

int cmptime ( const SNVT_elapsed_tm * const a, const unsigned long b ) {
    unsigned long ulA;
    int iResult;

    // convert SNVT_elapsed_tm_a into a value of type(b).
    ulA = a->millisecond;
    ulA += (1000uL / HVAC_CORETICK)
        * ( a->second + 60UL * (a->minute + 60ul
            * (a->hour + 24ul * a->day)));

    if (b > ulA) {
        iResult = -1;
    } else if (b < ulA) {
        iResult = +1;
    } else {
        iResult = 0;
    }
    return iResult;
}
```

```

void PropagateTemp(const SNVT_temp_p Value) {
    float_type fTemp, fResult;

    // set the temp_p type nvo:
    TempSensor::nvoHVACTemp = Value;

    // convert to float.

    // Get the float_type representation of the scaled
    //temp_p value:
    fl_from_slong(Value, &fTemp);
    // Get it right by correcting the fixed decimal point:
    fl_div(&fTemp, &f100, &fResult);
    // That's it!
    TempSensor::nvoFloatTemp = fResult;

    // restart the minsend/maxsend timers
    HvacMinSendTimer = HvacMaxSendTimer = 0L;
}

void UpdateTemperature(void) {
    // Get new value
    SNVT_temp_p NewValue;
    NewValue = GizmoReadTemperature(FALSE, TRUE);
    // Transmit if new value varies by more than nciMinDelta from
    // old value:
    if ((NewValue < (HVACTempOld - TempSensor::nvoHVACTemp::cpMinDelta))
        || (NewValue > (HVACTempOld
            + TempSensor::nvoHVACTemp::cpMinDelta))) {
        // Even so, only transmit if nciMinSendTimer allows:
        if (cmptime((const SNVT_elapsed_tm * const)
            &(TempSensor::cpMinSendTime), HvacMinSendTimer) <= 0 ) {
            // min send time has expired, really send data now:
            PropagateTemp(NewValue);
        }
    }
    // In either case, make sure to keep record of the latest value:
    HVACTempOld = NewValue;
}

when (timer_expires(hvac_coretick)) {
    // advance the timers:
    HvacMinSendTimer += HVAC_CORETICK;
    HvacMaxSendTimer += HVAC_CORETICK;

    // get new value and re-transmit if needed
    UpdateTemperature ();

    // transmit most recent value if needed due to heartbeat timer:
    if (cmptime((const SNVT_elapsed_tm * const)
        &(TempSensor::cpMaxSendTime), HvacMaxSendTimer) <= 0 ) {
        PropagateTemp( HVACTempOld );
    }
}

void TempSensorDirector(unsigned uFblockIndex, int iCommand )

```

The `cmptime(a,b)` function compares the **a** and **b** values. It returns `sign(a-b)`, i.e. - 1 if `b > a`, +1 if `b < a`, and 0 if `b == a`. The value of **b** is assumed to tick at the rate defined by the `HVAC_CORETICK` value in milliseconds.

The `PropagateTemp()` function is used to propagate the output network variable. It includes code to perform the necessary conversion to maintain the `FLOAT` type network variable, and to administrate the functional block's timers. This function mostly operates on fixed-point values and converts to floating-point values when needed. This minimizes the number of floating-

point operations and thus maximizes the performance of the LONWORKS device.

The `UpdateTemperature()` function is used to obtain new temperature readings from the Gizmo 4 temperature sensor hardware. It includes logic to decide whether this new value should be made available to the network immediately or at a later time, based on the minimum update interval defined in the `nciMinSendTime` configuration property.

The `when` statement uses the `HVAC_CORETICK` value to maintain the min/max send timers, looks after regular conversions, and assures that updates are sent no further apart than the time specified by the `nciMaxSendTime` configuration property (the heartbeat).

11. Build the development target. The `NodeBuilder` and `LonMaker` tools automatically load the new application into the device hardware.
12. Add the new functional block and network variables to the `LonMaker` drawing and use the `LonMaker` tool and `LonMaker Browser` to verify correct operation.

---

## NodeBuilder Example Task 8: Real Time Keeper

This task implements the standard real time keeper functional profile (SFP #3300, see the LONMARK website for more information: [www.lonmark.org](http://www.lonmark.org)). Implementation-specific configuration properties and network variables are added to this functional block, and we demonstrate processing of network variable updates for a functional block with multiple input network variables.

1. Click the `NodeBuilder` button in the Windows taskbar to return to the `NodeBuilder` tool.
2. Right-click the device template and select **Code Wizard** from the shortcut menu. The `Code Wizard` opens.
3. Right-click the device template's **Functional Blocks** folder and select **Add Functional Block** from the shortcut menu. The **Add Functional Block** dialog appears.
4. Add a single `SFPRealTimeKeeper` functional block. Name the new functional block **RealTimeKeeper**.
5. Right-click the **RealTimeKeeper** functional block's **Optional NVs** folder and select **Implement Optional NV** from the shortcut menu. The **Implement Optional Network Variable** dialog appears.
6. Implement the `nviTimeSet` network variable.
7. Right-click the **RealTimeKeeper** functional block's **Optional CPs** folder and select **Implement Optional CP** from the shortcut menu. The **Implement Optional Configuration Property** dialog appears.
8. Implement the `nciUpdateRate` configuration property. Name the new configuration property `cpRtcUpdRate`. Set **Initial Value** to 3L.
9. Right-click the **RealTimeKeeper** functional block's **Implementation-specific NVs** folder and select **Add NV** from the shortcut menu. The **Add Implementation-specific Network Variable** dialog appears.
10. Add a `SNVT_time_stamp` network variable. Set the direction to **Input**. Name the new network variable `nviAlarmTime`.

11. Right-click the **RealTimeKeeper** functional block's **Implementation-specific NVs** folder and select **Add Implementation-specific NV** from the shortcut menu. The **Add NV** dialog appears.
12. Add a `SNVT_switch` network variable. Set the direction to **Output**. Name the new network variable **nvoAlarmState**.
13. Right-click the **RealTimeKeeper** functional block's **Implementation-specific NVs** folder and select **Add NV** from the shortcut menu. The **Add Implementation-specific Network Variable** dialog appears.
14. Add a `SNVT_switch` network variable. Set the direction to **Input**. Name the new network variable **nviAlarmAck**.
15. Click **Generate and Close**.
16. Open the `RealTimeKeeper.nc` file from the Source Files folder and add the following code in bold:

```
#define RTC_CORETICK 250L
mtimer rtc_coretick;
enum {
    rtc_alarm_idle, rtc_alarm_armed, rtc_alarm_alarm
}; eeprom rtc_alarmstate = rtc_alarm_idle;

//{{NodeBuilder Code Wizard Start
```

This code adds a core timer to the device, which is used to poll the Gizmo 4's real-time clock hardware on a regular interval. The `RTC_CORETICK` enumeration is used to control the state engine within the alarm clock. The states are: *alarm disabled*, *waiting for alarm condition*, and *alarm currently on* (awaiting acknowledgement).

Sending a value to the **nviAlarmTime** network variable specifies the alarm time. The `second`, `minute` and `hour` fields of the network variable are used to input the alarm time. The `date`, `month`, and `year` fields can be set to 0 to disable the alarm, or to any non-zero value to arm the alarm clock.

17. Still in `RealTimeKeeper.nc`, add the following code in bold:

```
#endif // _HAS_INPUT_NV_

when (timer_expires(rtc_coretick)) {
    SNVT_time_stamp current;
    if (fblockNormalNotLockedOut(RealTimeKeeper::global_index)) {
        GizmoGetTime(&current);
        RealTimeKeeper::nvoTimeDate = current;

        switch(rtc_alarmstate) {
            case rtc_alarm_idle:
                // alarm is off
                break;
            case rtc_alarm_armed:
                // waiting for alarm condition to occur
                if ((current.second
                    == RealTimeKeeper::nviAlarmTime.second)
                    && (current.minute
                    == RealTimeKeeper::nviAlarmTime.minute)
                    && (current.hour
                    == RealTimeKeeper::nviAlarmTime.hour) ) {
                    // raise alarm
                    rtc_alarmstate = rtc_alarm_alarm;
                    RealTimeKeeper::nviAlarmState.state
                        = SWITCH_ON;
                }
                break;
            }

```

```

        case    rtc_alarm_alarm:
            // alarm currently visible/audible,
            // awaiting acknowledgement
            break;
    }
}
rtc_coretick
    = RealTimeKeeper::nvoTimeDate::cpRtcUpdRate * 100UL;
}

void RealTimeKeeperDirector(unsigned uFblockIndex, int iCommand)

```

This code controls timer processing.

18. Still in RealTimeKeeper.nc, add the following code in bold:

```

void RealTimeKeeperprocessNV(void)
{
    if (deviceState.nvIndex
        == nv_table_index(RealTimeKeeper::nviAlarmAck)) {
        // alarm acknowledgement:
        if (rtc_alarmstate == rtc_alarm_alarm) {
            rtc_alarmstate = rtc_alarm_armed;
            RealTimeKeeper::nvoAlarmState.state = SWITCH_OFF;
        }
    } else if (deviceState.nvIndex
                == nv_table_index(RealTimeKeeper::nviAlarmTime)) {
        // alarm spec:
        if ((RealTimeKeeper::nviAlarmTime.year == 0)
            && (RealTimeKeeper::nviAlarmTime.month == 0)
            && (RealTimeKeeper::nviAlarmTime.day == 0)) {
            // stop the nonsense!
            rtc_alarmstate = rtc_alarm_idle;
        } else {
            // start/restart the nonsense
            rtc_alarmstate = rtc_alarm_armed;
        }
    } else if (deviceState.nvIndex
                == nv_table_index(RealTimeKeeper::nviTimeSet)) {
        // set time:
        GizmoSetTime(&RealTimeKeeper::nviTimeSet);
    }
}

#endif // _HAS_INPUT_NV_

```

This code processes input network variable updates to the network variable handler function. This can be implemented in a number of ways. The solution presented here minimizes the number of when statements and thereby limits the scheduler latency. This is at the expense of extra processing time when the event occurs, since the function has to find out what network variable generated the event.

Other ways to approach the problem would be processing both input network variables at all times (i.e., whenever either one of them has been updated, or implementing one when `(nv_update_occurs(...))` task for each input network variable. To implement the latter solution, you would have to remove or comment out the Code Wizard start/end tags around the existing when statement. The following code shows how this would be implemented:

```

/--{NodeBuilder Code Wizard Start
// disabled the above to prevent CodeWizard from re-generating
// the associated code
// the NodeBuilder Code Wizard will add and remove code here.
// DO NOT EDIT the NodeBuilder Code Wizard generated code in these blocks!

//<Input NV Define>
#ifdef _HAS_INP_NV_6

```

```

//
//<Fblock NV When>
when(nv_update_occurs(nviTimeSet)) {
if (fblockNormalNotLockedOut( fblock_index_map[nv_in_index] ) )
{
    updateDeviceState(nv_in_index, nv_array_index,
        fblock_index_map[nv_in_index]);
    // TODO: process nviTimeSet event here
}
}

when(nv_update_occurs(nviAlarmAck))
//
/--}}NodeBuilder Code Wizard End
// disabled the above to prevent CodeWizard from re-generating
// associated code
{
    if (fblockNormalNotLockedOut(fblock_index_map[nv_in_index] ) ) {
        updateDeviceState(nv_in_index, nv_array_index,
            fblock_index_map[nv_in_index]);
        // TODO: process nviAlarmAck event here
    }
}

```

19. Still in `RealTimeKeeper.nc`, add the following code in bold to the `FBC_WHEN_RESET` else-if clause in the `realtimekeeperdirector()` function:

```

else if ((TFblock_command)iCommand == FBC_WHEN_RESET)
    rtc_coretick = nvoTimeDate::cpRtcUpdRate * 100UL;
    if (rtc_alarmstate == rtc_alarm_alarm) {
        RealTimeKeeper::nviAlarmState.state
        = SWITCH_ON;
    }
    setLockedOutBit(uFblockIndex, FALSE);

```

20. Build the development target. The NodeBuilder and LonMaker tools automatically load the new application into the device hardware.
21. Add the new functional block and network variables to the LonMaker drawing and use the LonMaker tool and LonMaker Browser to verify correct operation.

---

## NodeBuilder Example Task 9: Wheel Input

The purpose of this task is to complete the device application by implementing an open loop sensor where the principal network variable reports the status of the quadrature hardware input. This task shows a more comprehensive implementation of a functional profile by supporting more of the functional profile's optional features. To perform this task, follow these steps:

1. Click the NodeBuilder button in the Windows taskbar to return to the NodeBuilder tool. Right-click the device template and select **Code Wizard** from the shortcut menu. The Code Wizard opens.
2. Right-click the device template's **Functional Blocks** folder and select **Add Functional Block** from the shortcut menu. The **Add Functional Block** dialog appears.
3. Add a single `SFPTopenLoopSensor` functional block. Name the new functional block **Wheel**. Click **OK**.

4. When prompted, indicate that you do not want to create the new functional block as part of an array.
5. Open the Wheel functional block's **Mandatory NVs** folder. Right-click the **nvoValue** network variable and select **Properties** from the shortcut menu. The **Network Variable Properties** dialog opens.
6. Set **NV Type** to **SNVT\_lev\_percent** and **Name** to **nvoWheel**.
7. Right-click the **Wheel** functional block's **Optional CPs** folder and select **Implement Optional CP** from the shortcut menu. The **Implement Optional Configuration Property** dialog appears.
8. Implement the **nciGain** configuration property. Name the new configuration property **cpWhGain**. Set **Initializer** to  $\{1, 1\}$ . This configuration property holds the gain value between the physical input and the **nvoWheel** network variable.
9. Right-click the **Wheel** functional block's **Optional CPs** folder and select **Implement Optional CP** from the shortcut menu. The **Implement Optional Configuration Property** dialog appears.
10. Implement the **nciLocation** configuration property. Name the new configuration property **cpWhLocation**. This configuration property holds the location of the sensor device.
11. Right-click the **Wheel** functional block's **Optional CPs** folder and select **Implement Optional CP** from the shortcut menu. The **Implement Optional Configuration Property** dialog to appears.
12. Implement the **nciOverBehave** configuration property. Name the new configuration property **cpWhOvrBehave**. Set the **Initial Value** field to **OV\_RETAIN**. This configuration property determines the override behavior of the device. See the **SCPTovrBehave** configuration property in the *LONMARK SNVT and SCPT Master List* for more information.
13. Right-click the **Wheel** functional block's **Optional CPs** folder and select **Implement Optional CP** from the shortcut menu. The **Implement Optional Configuration Property** dialog appears.
14. Implement the **nciOvrValue** configuration property. Name the new configuration property **cpWhOvrValue**. This configuration property determines the override value of the device. See the **SCPTovrValue** configuration property in the *LONMARK SNVT and SCPT Master List* for more information.
15. Right-click the **Wheel** functional block's **Optional CPs** folder and select **Implement Optional CP** from the shortcut menu. The **Implement Optional Configuration Property** dialog appears
16. Implement the **nciMaxSendT** configuration property. Name the new configuration property **cpWhMaxSendT**. Set **Initializer** to  $\{0, 0, 0, 0, 0\}$ . This configuration property determines the maximum time between network variable updates for the functional block (the heartbeat).
17. Right-click the **Wheel** functional block's **Optional CPs** folder and select **Implement Optional CP** from the shortcut menu. The **Implement Optional Configuration Property** dialog appears.
18. Implement the **nciMinSendT** configuration property. Name the new configuration property **cpWhMinSendT** and set **Initializer** to  $\{0, 0, 0, 0, 0\}$ . This configuration property determines the minimum time

between network variable updates for the functional block (the throttle).

19. Right-click the **Wheel** functional block's **Optional CPs** folder and select **Implement Optional CP** from the shortcut menu. The **Implement Optional Configuration Property** dialog appears.
20. Implement the **nciOverValue** configuration property. Name the new configuration property **cpWhOvrValue**. This configuration property determines the override value for the **nvoWheel** network variable.
21. Click **Generate and Close**.
22. Open the `Wheel.nc` file from the Source Files folder. Add the `Cp2Tick()` utility function and a `when` statement to handle I/O processing as shown below:

```
unsigned long Cp2Tick (const SNVT_elapsed_tm *const pSnvt) {
    unsigned long ulResult;
    ulResult = ((pSnvt->minute * 60UL) + pSnvt->second)
        * (1000UL/WHEEL_HBCORE)
        + (pSnvt->millisecond / WHEEL_HBCORE);
    return ulResult;
}

priority when (io_changes(ioWheel)) {
    if (fblockNormalNotLockedOut( Wheel::global_index)) {
        if (Wheel::cpWhGain.divisor) {
            // No division by zero. Use gain factor and send new
            // incremental value to heartbeat/throttle handler
            WheelIncrValue(muldivs(input_value,
                Wheel::cpWhGain.multiplier,
                Wheel::cpWhGain.divisor));
        }
    }
}
```

The `Cp2Tick()` function converts a `SNVT_elapsed_tm` value into a tick count (a tick occurs each `WHEEL_HBCORE` milliseconds). It uses the `seconds`, `milliseconds`, and `minutes` fields of the `SNVT_elapsed_tm` value but ignores the `hours` and `days` fields. The function is only used in conjunction with heartbeat and throttle intervals, which typically do not have values as large as an hour, and therefore this partial implementation is used for performance reasons.

A `priority when` statement is used due to the nature of this physical input, which is to provide an incremental reading. You should normally avoid the use of `priority` statements except for critical processing. This `when` clause is critical since this function must not miss any hardware events as they occur.

23. Open the `wheel.h` file from the **Source Files** folder, and then add the following declarations for heartbeat/throttle handling:

```
#ifndef USE_QUADRATURE
#error "You must enable the use of quadrature input in the gizmo4.h header file!"
#endif

#define WHEEL_HBCORE    100L    // heartbeat/throttle timer ticks
                                //with 100ms period

mtimer repeating WheelTimer = WHEEL_HBCORE;

long lWheelValue = 0L;    // last known real value,
                            //see the .nc file for details on
```



```

// heartbeat/throttle implementation
long lWheelPhysical = 0L; // same as lWheelValue,
// but limited to data coming from
// physical sensor. See .nc file for
// details on override and rmv_override
unsigned long ulMinSendT = 0L; // number of WHEEL_HBCORE ticks
// expired on cpMinSendT
unsigned long ulMaxSendT = 0L; // number of WHEEL_HBCORE ticks
// expired on cpMaxSendT

// forward declaration:
void WheelIncrValue (long Value);

```

24. Open the `wheel.nc` file from the **Source Files** folder and add the following function to handle the network variable throttle:

```

void WheelIncrValue (long Value) {
// maintain internal mirror of most recent value - remember the
// quadrature input provides incremental data. We cannot lose
// a single signal update.
lWheelValue += Value;

// also maintain a copy of the last known value coming from the
// physical sensor (as opposed to override values). This data
// is used when leaving override mode, see director function
// for more details.
lWheelPhysical = lWheelValue;

// Manage the throttle preferences. Note the throttle tick
// counter is maintained by the WheelTimer routine.
if (ulMinSendT >= Cp2Tick(&Wheel::nvoValue::cpWhMinSendT)) {
    ulMinSendT = 0;
    ulMaxSendT = 0;
    nvoWheel = lWheelValue;
}
}

```

25. Still in `wheel.nc`, add the following function to handle the network variable heartbeat:

```

when (timer_expires(WheelTimer)) {
// update throttle timer:
if (ulMinSendT < Cp2Tick(&Wheel::nvoValue::cpWhMinSendT)) {
    ++ulMinSendT;
}

// manage heartbeats:
if (ulMaxSendT < Cp2Tick(&Wheel::nvoValue::cpWhMaxSendT)) {
    ++ulMaxSendT;
} else {
// propagate the latest value. Note that we should not use
// the propagate() function here, as propagate() would only
// re-propagate the last NV value. There could have been
// value updates internally since then, which have not made
// it into the NV value due to the throttle. We do
// therefore use an internal mirror of the truly most
// recent value:
Wheel::nvoValue = lWheelValue;
ulMaxSendT = 0L;
}
}

```

26. Still in `wheel.nc`, implement the override behavior by adding the following code in bold to the `wheelDirector()` function's `FBC_OVERRIDE` `else/if` statement:

```

else if ((TFblock_command)iCommand == FBC_OVERRIDE)

```

```

setFblockOverride( uFblockIndex, TRUE );
switch (Wheel::cpWhOvrBehave) {
    case OV_RETAIN:    // do nothing, keep last value
        break;
    case OV_SPECIFIED:
        // override with specified override value. We
        // still must honor heartbeats, but we must
        // switch to override value immediately
        // (ignoring throttle preferences)
        nvoWheel = lWheelValue =
            Wheel::nvoValue::cpWhOvrValue;
        break;
    case OV_DEFAULT:
        // override with sensor-specific default value
        // (zero). We must continue to honour
        // heartbeats, but we must switch to override
        // value immediately (ignoring throttle
        // preferences)
        nvoWheel = lWheelValue = 0L;
        ulMinSendT = ulMaxSendT = 0L;
        break;
}

```

27. Complete the implementation of the override behavior by adding the following code in bold to the `wheelDirector()` function's `FBC_RMV_OVERRIDE` else-if statement:

```

else if ((TFblock_command)iCommand == FBC_RMV_OVERRIDE)
    setFblockOverride(uFblockIndex, FALSE);
    nvoWheel = lWheelValue = lWheelPhysical;
    // ignore throttle but re-start heartbeat:
    ulMinSendT = ulMaxSendT = 0L;

```

This code updates the output network variable with recent physical data to wipe out the override value. This implementation ignores any value updates received during the override period, but re-sets the output to the last known value when the device entered the override state. This allows the device to be set into override while the sensor unit is replaced or while diagnosing the network. The interpretation of correct override behavior is device-dependent and subject to the device implementation. Different, but equally acceptable implementations would be to await new readings from the sensor, or to save value changes during the override state.

28. Build the development target. The NodeBuilder and LonMaker tools automatically load the new application into the device hardware.
29. Add the new functional block and network variables to the LonMaker drawing and use the LonMaker tool and LonMaker Browser to verify correct operation.

---

## Continuing with the NodeBuilder Example

You have completed the implementation of the Neuron C portion of the example development project. See *Creating an LNS Device Plug-in* and the *LNS Plug-in Programmer's Guide* for more information on creating a plug-in for this example device.

For additional practice with the NodeBuilder tool, here are a few ideas how you could improve the Neuron C application:

- Most functional profiles in these tasks are not fully implemented. Override features, self-test features, and many of the traffic-control configuration

properties (heartbeat, throttle, and heartbeat control on the input side) are not implemented for simplicity.

- You could implement a `UFPTdisplay` functional profile. This functional block would display data received from input network variables in a configurable manner.

You could implement a more generic translator object, using network variables of a changeable type for input or output. See the *Neuron C Programmer's Guide* for more details.



# Appendix B

## Converting a NodeBuilder 1.5 Project to a NodeBuilder 3.1 Project

This appendix describes how to convert a NodeBuilder 1.5 project into a NodeBuilder 3.1 project, including how to update your Neuron C Version 1 source code for Neuron C Version 2.1.

---

## Converting a NodeBuilder 1.5 Project to a NodeBuilder 3.1 Project

NodeBuilder 1.5 hardware templates and device templates with the NodeBuilder 3.1 Development Tool. You must use the NodeBuilder 3.1 tool to convert the templates to the new NodeBuilder 3.1 format as described in this section.

The NodeBuilder 1.5 and NodeBuilder 3.1 tools will run side-by-side. You may continue to use the NodeBuilder 1.5 tool, but the device file sets that it produces will not be identical to NodeBuilder 3.1 device file sets. The two tools use different debug file formats. You cannot use the output files from one tool to debug on the other. See *Running NodeBuilder 1.5 and NodeBuilder 3.1 Concurrently* for more information.

In order to create a NodeBuilder 3.1 project with Neuron C code from a NodeBuilder 1.5 project, follow these steps:

1. Create or open a LonMaker network to be used for development.
2. Create a NodeBuilder project as described in *Creating a NodeBuilder Project: Creating a New Project*. Do not create a new device template.
3. Right-click the **Device Templates** folder and select **Insert Copy** from the shortcut menu. The Open File dialog appears.
4. Open the NodeBuilder 1.5 device file (.dev extension). A new NodeBuilder 3 device template (".NbDt" extension) will be created, using information from the old one.
5. Right-click the **Hardware Templates** folder and select **Insert Copy** from the shortcut menu. The Open File dialog appears.
6. Browse to and select the NodeBuilder 1.5 device template file (.dtm extension) to create a new NodeBuilder 3 hardware template (.NbHwt extension) based on the information in this file.
7. The NodeBuilder 3.1 tool expects UNVTs and UCPTs to be defined in user resource files, and SNVT and SCPT types to be defined in standard resource files. If your code uses typedefs or Neuron C declarations to declare them, you need to modify them. The "SNVT," "SCPT," "UCPT," and "UNVT" prefixes are reserved; you will get compiler errors regarding this. You can either replace the names with new names—for example by changing the upper-case prefixes to lower case—or you can specify the new `#pragma names_compatible` compiler directive, within your source code. See the *Neuron C Reference Guide* for more information about this compiler directive.  
The most maintainable solution is to move the definitions into resource files and modify the code accordingly (see *Converting a Neuron C Version 1 Application to a Neuron C Version 2.1 Application* for more information). You can use conditional compilation to isolate changes if you want to continue to use both compilers.
8. While Neuron C Version 2.1 is designed to be backwards compatible, it introduces a number of new reserved words not used by version 1 of the language. Be sure your source code does not use any of the new Neuron C reserved words as variable names. See the *Neuron C Reference Guide* for a complete listing of reserved words.

9. Remove the `#pragma set_std_prog_id` statement in the source code. This statement is not necessary for NodeBuilder 3.1 projects and will cause a compiler warning if it does not match the value in the NodeBuilder device template.
10. Remove the `#include <netdbg.h>` statement from the source code. This statement may collide with the debug kernel option settings made in the NodeBuilder *target settings*.
11. Make sure your code contains a `#pragma num_alias_table_entries <N>` statement, with `<N>` being an integer from 0 (zero) to 62. Neuron C Version 2.1 requires this pragma to be specified. If you do not wish to support any network variable aliases, corresponding with the Neuron C Version 1 default, set `<N>` to 0 (zero).
12. Build the development target. The development target requires approximately 2Kb of additional code space, since it links in the debug kernel.
13. You can simplify maintenance of your application by converting your functional block and configuration property declarations to Neuron C Version 2.1 syntax. This enables the NodeBuilder tool to automatically generate self-documentation strings required for your functional blocks and functional block members, and to correctly document your functional blocks in device interface files. See *Converting a Neuron C Version 1 Application to a Neuron C Version 2.1 Application* for information on updating your application to use Neuron C Version 2.1 features.
14. If you did not convert your application to Neuron C Version 2.1 as described in the previous step, the NodeBuilder 3.1 tool will not automatically generate your self-documentation strings, and it will not automatically include functional block information in your device interface files. This is the same behavior as the NodeBuilder 1.5 tool. If you previously manually updated the device interface files to include configuration file information, you will have to continue to do so. The NodeBuilder 3.1 tool simplifies this process by automatically appending a device interface appendix file (.xf2 extension) to your device interface file if you specify one. To specify a device interface appendix file, create a file in your output folder with the same base name as your device interface file (i.e., without the .xif extension), name it with a .xf2 extension, and then add it to your source files folder in the NodeBuilder Project Manager. Include the configuration template and value files in your device interface appendix file. The NodeBuilder 3.1 tool does not automatically update your device interface appendix file—it only appends it to your device interface file. You will have to manually update the device interface appendix file if you make any changes to your configuration properties implemented within configuration files. The device interface appendix file is not required if you convert the functional block and configuration property declarations in your application to Neuron C Version 2.1 syntax. See *Converting a Neuron C Version 1 Application to a Neuron C Version 2.1 Application* for more information.

For more information about converting a NodeBuilder 1.5 project to NodeBuilder 3.1, see *NodeBuilder Project Conversion Tips*.

---

# Converting a Neuron C Version 1 Application to a Neuron C Version 2.1 Application

The NodeBuilder 3.1 tool supports the Neuron C Version 2.1 language, whereas the NodeBuilder 1.5 and LonBuilder 3.01 development tools use the Neuron C Version 1 language. The Neuron C Version 2.1 language greatly simplifies development of applications that use functional blocks and configuration properties. If you have any applications that you originally developed using version 1 syntax, you should convert them to version 2.1 syntax if you plan to do make any changes for future maintenance or enhancements.

The NodeBuilder 3.1 tool continues to support Neuron C Version 1 (see *NodeBuilder Project Conversion Tips* for more information about using NodeBuilder 3.1 with legacy source code). However, the NodeBuilder 3.1 tool does not permit functional block and functional block member declarations to be made using a mixture of version 1 and version 2 syntax within the same application.

The following describes procedure for converting version 1 applications to version 2. This may be a complex process for a complex application, so be sure to thoroughly test the result of your porting when you are done. For simplicity, this section refers to the Neuron C Version 1 application as the "old application" and refers to the Neuron C Version 2.1 application as the "new application". It also assumes that the name of the new application's NodeBuilder 3.1 device template is "NewApp" for reference to filenames in use, but you can change this name to any name of your choice.

To convert a Neuron C application, follow these steps:

---

## Step 1: Build the old application

Eliminate all possible incompatibilities with the Neuron C Version 2.1 language by applying all considerations and changes described in *Converting a NodeBuilder 1.5 Project to a NodeBuilder 3.1 Project*. After completing these changes, build and load the old application with the NodeBuilder 3.1 tool, and verify that it still works as desired.

---

## Step 2: Create a new device template

Create a new NodeBuilder device template as described in *Using the New Device Template Wizard*. This device template will be used to create the new application. Make sure to use a program ID that is different from that used by the old application; ideally, you would increase the model number field using the Standard Program ID calculator when you create the device template. Alternatively, you can use a range of model numbers with program ID management (don't select a range that includes the program ID of the old application).

Specify a new name and a new base folder for this new application so that the one you've created and converted in step 1 won't be affected by this step. For the sake of this discussion, it is assumed the new NodeBuilder device template is named `NewApp`, but you can choose any name.



Adjust all other preferences in the device template and target settings dialogs, and set the hardware templates as desired.

---

### ***Step 3: Create Resource Files***

Use the NodeBuilder Resource Editor to define resource files for your user functional profiles, user network variable types, and user configuration property types. You can skip this step if you only used standard functional profiles and types. See *Editing Resource Files* for more information.

---

### ***Step 4: Create code using the code wizard***

Use the NodeBuilder Code Wizard to create source code that implements your device interface. Add the functional profiles, network variables, and configuration properties that comprise your device interface. See *Using the NodeBuilder Code Wizard* for more information. Set related attributes such as the external names for functional blocks or the configuration file access method, as required. Generate code and exit the Code Wizard.

By the end of this step, your device should compile correctly, but will not yet contain any of your data, private types and algorithms that were contained within the original application.

---

### ***Step 5: Move global declarations***

Move global variables, type definitions and enumerations, constants, preprocessor macros, I/O object declarations, global pragma directives and timer declarations from the old application into `NewApp.h`.

Do not copy network variable declarations, or type definitions for UNVTs, SNVTs, UCPTs, or SCPTs.

At the end of this step, make sure the new application builds correctly.

---

### ***Step 6: Move global utility functions and system event handlers***

Move global utility functions from the old application into `NewApp.nc` file. Move code that responds to system events (`when reset`, `when offline`, `when online`, and `when wink`) from the old application into `NewApp.nc`. The code wizard generated code framework already contains the bodies for these `when` tasks. Also move any handlers for user-defined events (e.g. `when (bMyFlag)`) in the same way.

---

### ***Step 7: Move functional block-specific state management***

Move any functional block-specific state management code from the old application into the new application. These are code sequences that deal

with managing the functional block and its state transitions (*enable*, *disable*, *override*, etc).

The code wizard automatically produces code to implement the basic operations for functional block state management. It also automatically generates a default implementation of a director function for each functional block and array of functional blocks, which can be found in the Neuron C file for each functional block or functional block array. Use the functional block management framework generated by code wizard and eliminate any code from the old application that serves the same purpose.

Move functional block-specific housekeeping and management code from the old application into the relevant sections of the new application's director functions.

---

## Step 8: Set resource scopes

The LonMaker tool automatically sets the resource scope for all configuration properties; for all standard functional profiles and network variables; and for all user functional profiles and network variables that are defined by scope 3 resource files. If you have any functional profiles or network variables that are defined in scope 4, 5, or 6 resource files, you must explicitly set their scope. The easiest way to do this is to implement a plug-in that automatically sets the scope for you and for the users of your device. See the *LNS Plug-in Programmer's Guide* for details. You can also override the scope settings using the LonMaker tool as described in Chapter 11, *Creating and Using Custom LonMaker Shapes and Stencils*, of the *LonMaker User's Guide*.

---

## Step 9: Test #1

Disable all code in the system events that is specific to your hardware, or that references functions or variables that you have not ported yet.

Build the new application, load the device and integrate the device into a LonMaker network. Make sure you can browse the device (it should have all functional blocks as desired including all network variables and configuration properties). Test the device and the individual functional blocks using the LonMaker device manager (the self-test, that is part of a comprehensive test, will be shown as `not supported`). Make sure that you can enable and disable the individual functional blocks, and that the device responds correctly to system events (`reset`, `online`, `offline`, `wink`).

After completion of this testing, re-enable the code you have disabled at the beginning of this step.

---

## Step 10: Move input network variable handler

Move code that processes the arrival of input network variable data from the old to the new application. You can find that code in qualified and unqualified `when (nv_update_occurs)` tasks in the old application.

The code generated by the code wizard in the new application contains an empty `<FbName>ProcessNv()` function (with `<FbName>` being the name of the functional block). Move your code into this function.

The code wizard always generates one `when (nv_update_occurs)` task per functional block or functional block array (if there are input network variables associated with the functional block). See step 11 of the *NodeBuilder Example Task 8: Real Time Keeper* example for details about splitting this single `when`-task into multiple event handlers. If you increase the total number of `when` tasks, you might reduce the device's overall responsiveness. This is because scheduler turnaround times grow with the number of `when` tasks to be processed.

---

## ***Step 11: Move declarations and handlers for timer and I/O-related events***

Move code that handles I/O events (e.g. `when (io_changes(...))`) from the old to the new application, and repeat the same for code that handles timer events, or user-defined but object-specific events. Place global `when` tasks and functions in `NewApp.nc`, and functional block-specific tasks in the functional block-specific Neuron C file.

For functional block-specific code, place the code just ahead of the director function close to the bottom of the file.

Move declarations of functional block-specific timers, I/O objects, variables, macros, type definitions and enumerations into the functional block-specific header file (.h extension), where applicable.

---

## ***Step 12: Move application messaging code***

Move code for sending and receiving of application messages and for responding to application message requests to the most suitable location into the new application, where applicable. Add global code to the `NewApp.nc` file, or include it from that file. Add functional block-code to the functional block-specific Neuron C file, or include it from that file.

If you generated code with the option to access configuration files via the file transfer protocol (FTP), see *Code Generated by the Code Wizard* for details about the FTP server generated by the code wizard and its use of application messaging-related constructs.

---

## ***Step 13: Test #2***

Build your new device, and test it using the LonMaker. Repeat the test outlined in step 9 above. Also use the debugger to verify the correct processing of events. In case your code contains an FTP server, verify both installation scenarios - the ad-hoc scenario where you upload the interface definition from the device, and the engineered system scenario where you specify the interface by importing the device interface file.

---

## NodeBuilder Project Conversion Tips

This section contains a list of tips for converting a NodeBuilder 1.5 project to a NodeBuilder 3.1 project. See *Converting a NodeBuilder Project* for step-by-step instructions.

- In the NodeBuilder 1.5 tool, the **Build** tab of the device window allowed you to turn on a number of different build reports, including the Output Link Summary and Assembly Listing; these options can now be accessed by right-clicking the Development or Release target folder and selecting Settings from the shortcut menu. This opens the device template target properties dialog, which contains four tabs: *Compiler*, *Linker*, *Exporter*, and *Configuration*. See the NodeBuilder help file for details.
- The NodeBuilder 3.1 tool creates version 4 device interface files. These files are more complete than the version 3 files generated by the NodeBuilder 1.5 and LonBuilder 3.01 tools. A version 4 device interface file contains configuration file declarations consisting of FILE and NVVAL records, which you had to manually construct with the NodeBuilder 1.5 and LonBuilder 3.01 tools.
- When the NodeBuilder 3.1 tool is installed on a computer that already has the NodeBuilder 1.5 tool installed, the `stdlibs.lst` file in the LONWORKS Images folder will be backed up to `stdlibs.1` before being overwritten by the `stdlibs.lst` file that the NodeBuilder 3.1 tool installation installs. This file contains a list of standard libraries.
- For devices that use `uninit` application variables at fixed memory locations, make sure to set the **Compatible** and **Order Preserving** linker options for backwards compatibility of the allocation algorithm. To view linker properties, right-click a target, select **Settings** from the shortcut menu, and click the **Linker** tab.
- For legacy devices that have been ported to NodeBuilder 3.1 but that continue to use the Neuron C Version 1 syntax for declaring functional blocks and configuration properties, make sure not to use the Neuron C Version 2.1 language features that trigger the compiler to generate functional block self-documentation information. If the compiler generates self-documentation information, any pre-existing self-documentation strings will be moved into the comment section (following the semicolon in a self-documentation string). You must either convert these projects to using the new, enhanced, Neuron C Version 2.1 syntax, or continue maintaining these devices only using the legacy Neuron C Version 1 syntax. The following Neuron C Version 2.1 constructs trigger the generation of self-documentation information: `cp` network variable modifier, `cp_family` keyword, `device_properties`, `fb_properties`, `nv_properties` constructs, and the use of the `fblock` keyword.
- Several standard network variable types (SNVTs) in the version 11 resource files shipped with the NodeBuilder 3.1 tool use the `s32_type` structure to contain signed, 32-bit integers. The `s32_type` structure has the following format:

```
typedef struct s32_type {
    int bytes[4];
} s32_type;
```

In previous versions of the resource files, a 4-byte array was used in place of the `s32_type` structure. In most cases, this change does not affect how the data is accessed. However, in the event that you want to access individual bytes of the signed 32-bit value, you must now reference the `bytes` array within the structure.

For example, the `filexfer.nc` example provided on the Echelon web site contains the following line of code (used to convert 16-bit signed types to 32-bit types):

```
memcpy(&xfer_offset, &nviFilePos.rw_ptr[2],
sizeof(xfer_offset))
```

`rw_ptr` is an `s32_type` value. The code shown above will not compile in the NodeBuilder 3.1 tool because `rw_ptr` is a structure, not an array. This code can be corrected as shown below:

```
memcpy(&xfer_offset, &nviFilePos.rw_ptr.bytes[2],
sizeof(xfer_offset))
```

The following SNVTs use the `s32_type` structure:

```
SNVT_file_status
SNVT_currency
SNVT_file_pos
SNVT_time_zone
SNVT_reg_val
SNVT_reg_val_ts
SNVT_elec_kwh_1
SNVT_alarm_2
```

- If you are not using an authentication key, change the authentication key from `FF:FF:FF:FF:FF:FF` (48 bits, all set to 1) to `00:00:00:00:00:00` (48 bits, all bits cleared to zero). The old null key will continue to work, but the new null key results in better performance when your device is installed.
- NodeBuilder 3.1 uses the `PSG.lib` library in place of the `SLTA.lib` library. If you want to use the older `SLTA.lib` library, remove `PSG.lib` from the **Libraries** folder under the **Device Template** in the Program Manager and add the `SLTA.lib` library.

---

## Running NodeBuilder 1.5 and NodeBuilder 3.1 Concurrently

When working with both old and new NodeBuilder projects, it may be desirable to have both the NodeBuilder 1.5 and NodeBuilder 3.1 tools installed on the same computer. In order to do this, you must install the two applications in this order:

1. Install the NodeBuilder 1.5 tool. See the NodeBuilder 1.5 documentation for more information.
2. Install NodeBuilder 1.5 service pack 6. This service pack is available from the product updates page at [www.echelon.com/toolbox](http://www.echelon.com/toolbox). You must apply this patch before installing the NodeBuilder 3.1 tool.
3. Install the NodeBuilder 3.1 tool as described in *Installing the NodeBuilder*

*Software.*

When you install the NodeBuilder 3.1 software, the NodeBuilder 1.5 `default.ver` file will be overwritten by the NodeBuilder 3.1 `default.ver` file. This changes the default version number for all 3150 system images from version 7 to version 12. If you attempt to load a device that has the version 7 system image in its hardware using the NodeBuilder 1.5 tool after the NodeBuilder 3.1 software has been installed, it will fail. You can prevent this by editing your existing NodeBuilder 1.5 device templates so that the system image version is 7 instead of the default system image.







# Appendix C

## The Command Line Project Make Utility

This appendix describes how to use the command line project make utility with the project make command.

---

## Using the Command Line Project Make Utility

You can invoke the NodeBuilder build tools from the Windows command line. You can use this feature to generate automated build scripts for your devices. To build a project from the command line, invoke the NodeBuilder Command Line Project Make Utility by opening a Windows command prompt and entering the following command:

```
pmk [-p<Project> <command line switches> -t<Target>
```

You must specify what kind of operation will take place: a build (see the `-b` command switch), a query (see the `-q` command switch), or a clean (see the `-x` command switch). All other command line switches are optional. The **pmk** command performs one build, query, or clean operation.

The following command line switches are available:

<code>--help &lt;cmd&gt;</code>	Displays usage help for the <code>&lt;cmd&gt;</code> command. Providing no command at all also displays the list of the available commands and a brief usage hint. Alternative syntax: <code>-? &lt;cmd&gt;</code>
<code>--file &lt;file&gt;</code>	Uses <code>&lt;file&gt;</code> as input to the project make. This file can contain command line switches to be used by the project make utility. You can set <b>Generate build script</b> in the <i>Build</i> tab of the NodeBuilder Project Properties dialog to have the NodeBuilder tool automatically generate a command file (.cmd extension) that will allow you to reproduce the current build from the command line. This command file will be placed in the device template target folder, and will have the name <code>&lt;device template name&gt;.cmd</code> . If multiple targets are built, a separate command file will be generated for each. Alternative syntax: <code>-@ &lt;file&gt;</code>
<code>--always</code>	Causes the NodeBuilder tool to perform an unconditional build. See <i>Building an Application Image</i> for more information. This causes a clean command to be executed before the build. Alternative syntax: <code>-a</code>
<code>--build &lt;nbd&gt;</code>	Indicates that a build operation will be made on the selected NodeBuilder device template (“ <code>.NbdT</code> ” extension) for the target specified by the <code>-t</code> command switch. The device template will be compiled, linked, and exported. You can only specify a single device template per make command. Alternative syntax: <code>-b &lt;nbd&gt;</code>
<code>--compile &lt;nbd&gt;</code>	Specifies a NodeBuilder device template file (“ <code>.NbdT</code> ” extension) to be compiled. You can only specify a single device template per make command. Alternative syntax: <code>-c &lt;nbd&gt;</code>

<code>--defloc [&lt;dir&gt;]</code>	<p>Specifies a directory to search for the default command file. The default command file for the project make facility is named <code>lonpmk32.def</code>. If this file does not exist in the specified directory, the command will fail silently. If no directory is specified, the current directory will be searched for <code>lonpmk32.def</code>.</p> <p>The default command file can contain any number of command switches for the <code>pmk</code> command. These commands will be executed in addition to any commands that are entered on the command line, or passed along using the <code>--file</code> command switch. For example, a default command file consisting of the following line would generate a log of the build script for every build in a <code>lonpmk.32.log</code> file:</p> <pre>--mkscript c:\temp\lonpmk32.log</pre>
<code>--mkscript &lt;file&gt;</code>	<p>Generates a file that contains all the command switches and arguments that are used in this invocation of the project make utility. You can use this file as a log of the build or to recreate the build on another computer.</p>
<code>--confirm</code>	<p>Reconfirms build status after build completion. Alternative syntax: <code>-n &lt;cmd&gt;</code></p>
<code>--nadep &lt;nadep&gt;</code>	<p>Specifies the location of the assembler dependency file. By default, this file is located in the <b>IM</b> subfolder of the target folder (i.e. <b>Development</b> or <b>Release</b>).</p>
<code>--ncdep &lt;ncdep&gt;</code>	<p>Specifies the location of the compiler dependency file. By default, this file is located in the <b>IM</b> subfolder of the target folder (i.e. <b>Development</b> or <b>Release</b>).</p>
<code>--nldep &lt;nldep&gt;</code>	<p>Specifies the location of the linker dependency file. By default, this file is located in the <b>IM</b> subfolder of the target folder (i.e. <b>Development</b> or <b>Release</b>).</p>
<code>--nodefaults</code>	<p>Disables processing of default command files (see the description of the <code>--defloc</code> command switch for more information).</p>
<code>--nxdep &lt;nxdep&gt;</code>	<p>Specifies the location of the exporter dependency file. By default, this file is located in the <b>IM</b> subfolder of the target folder (i.e. <b>Development</b> or <b>Release</b>).</p>
<code>--project &lt;proj file&gt;</code>	<p>Specifies the NodeBuilder project that contains the NodeBuilder device template to be built.</p>

	NodeBuilder project files have the <code>.NbPrj</code> extension. Alternative syntax: <code>-p &lt;proj file&gt;</code>
<code>--query &lt;nbd&gt;</code>	Indicates that a query operation will be performed on the specified NodeBuilder device template for the target specified by the <code>-t</code> command switch. This command indicates whether the target needs to be built. Alternative syntax: <code>-q &lt;nbd&gt;</code>
<code>--silent</code>	Prevents the copyright and version information from being displayed.
<code>--target &lt;Development/Release&gt;</code>	Specifies what target the build, clean, or query operation will be invoked on. This switch is required. Alternative syntax: <code>-t &lt;Development/Release&gt;</code>
<code>--verbose</code>	Causes the project make facility to be run in verbose mode. The default is non-verbose mode.
<code>--warning &lt;message&gt;</code>	Produces a warning with the message specified. This command is useful in build scripts.
<code>--clean &lt;nbd&gt;</code>	Indicates that a clean operation will be performed on the specified NodeBuilder device template for the target specified by the <code>-t</code> command switch. A clean operation removes all files and folders produced by a build. Alternative syntax: <code>-x &lt;nbd&gt;</code>

Here's an example for a minimal command line invocation of the project make facility:

```
PMK -pTest.nbprj -bMyDevice.nbd -tDevelopment
```

This command performs a conditional build on the development target that is contained within the `MyDevice` device template which is part of the `Test` project.

For more information about the use of the NodeBuilder and Neuron C command line tools, see the *Neuron C Programmer's Guide*, Appendix A, *Neuron C Tools Stand-alone Use*. Details provided there under *Common Standalone Tool Use* also apply to the NodeBuilder Project Make Facility.

# Appendix D

## Using the LonBuilder Emulator

This appendix describes how to use a LonBuilder Emulator as a development platform for the NodeBuilder 3.1 tool.

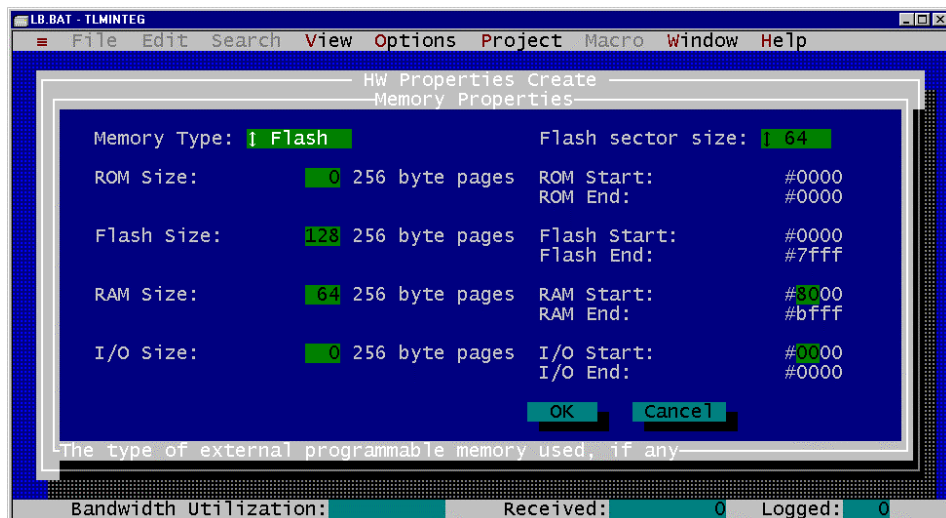
## Using the LonBuilder Emulator

You can use a LonBuilder Emulator as a NodeBuilder 3.1 development and testing platform. The NodeBuilder 3.1 debugger does not use the hardware debugging support built into the LonBuilder Emulator, but the firmware debug kernel used by the NodeBuilder 3.1 debugger will function correctly in the LonBuilder Emulator. To use the LonBuilder Emulator with the NodeBuilder 3.1 tool, follow these steps:

1. Ensure that the NodeBuilder 3.1 and LonBuilder 3.01 tools are installed on your computer.
2. The NodeBuilder 3.1 software installation installs NBE3150.ib, NBE3150.nx, NBE3150.nxb, and NBE3150.sym to the LONWORKS Images\Ver<X> folder (where <X> is the latest firmare version) to support using the LonBuilder Emulator as a NodeBuilder target platform. Copy these files to \lb\images\Ver<X> so the LonBuilder software can find them.
3. Open the LonBuilder software.
4. Open a new set of LonBuilder hardware properties as described in the *LonBuilder User's Guide*.
5. Set the following properties in the LonBuilder **HW Properties Modify** dialog:

<b>HW Property Name</b>	Set to a descriptive name, such as NodeBuilderEmu.
<b>Neuron Chip Firmware</b>	Set to <b>NBE3150</b> .
<b>Neuron Chip</b>	Set to <b>3150</b> .
<b>Firmware Version</b>	Set to <b>13</b> .
<b>Input Clock Rate</b>	Set to <b>10 MHz</b> .

6. Click **More**. The **Memory Properties** dialog opens.
7. Set the properties as shown in the following figure:



The memory map should match the eventual target platform (so the

LonBuilder Emulator will emulate it appropriately).

8. Re-install the emulator using the LonBuilder tool as described in the *LonBuilder User's Guide*.
9. Create a new NodeBuilder hardware template as described in *Using Hardware Templates* in the *Using Device Templates* chapter. Set **Platform** in the *Hardware* tab to **LonBuilder Emulator 3150**.
10. Create a new Develop Target Device Shape and associate it with the NodeBuilder project.
11. Set the development target hardware template to the template that you created in step 9.

Build and load the project. You will now be able to test your application using the LonBuilder Emulator.





# Appendix E

## Using Source Control

This appendix describes how to manage a NodeBuilder project using a source control application.

---

## Using Source Control

When developing a large NodeBuilder project, you can put the project under source control to allow multiple developers to work concurrently on different parts of the project and maintain configuration control of their changes. This appendix lists all the files associated with a NodeBuilder project that should be kept under source control.

The following abbreviations for file locations are used throughout the table:

<code>&lt;LonWorks&gt;</code>	Stands for the LONWORKS folder. This is <code>C:\LonWorks</code> by default.
<code>&lt;NbDtFolder&gt;</code>	Stands for the folder that contains the NodeBuilder device template file. NodeBuilder device template files use the “.NbDt” file extension. By default <code>&lt;NbDtFolder&gt;</code> is a subfolder of the NodeBuilder project folder.
<code>&lt;mnfr&gt;</code>	Stands for your manufacturer name. For example: ACME Corporation.
<code>&lt;lang&gt;</code>	Stands for any valid device resource file language identifier such as ENU, GER, FRA, and so on.
<code>&lt;project&gt;</code>	The name of the NodeBuilder project.

Check the following files into a source code control system to allow several developers to work on the same code base and to enable a LONWORKS device file set to be completely recreated from source:

**Project Files** These files have the .NbPrj and NbOpt extensions.

Check in the `<project>.NbPrj` file. It is the NodeBuilder project file. It holds pointers to all the NodeBuilder device templates and any user-defined hardware templates required for a build.

You do not need to check in the `<project>.NbOpt` file. It is a NodeBuilder options file. It holds information about which devices have been inserted into the project, breakpoint lists for the debugger and other user settings. The options in this file are a matter of personal preference, and do not effect device file set.

Although NodeBuilder project folders and all their subfolders can be moved and re-opened from the new location via the NodeBuilder Open Project dialog, moving a project folder can cause compilation errors due to absolute file references in use, or due to resource files being moved. Try to use relative references rather than absolute file name paths whenever possible.

Avoid using the **Include Search Path** option (see the NodeBuilder *Project options*) in order to improve project-to-project compatibility.

The default location: for the project files is  
C:\lm\source\

### **NodeBuilder Device Template Files**

These files hold most of the data required to build a device file set and NodeBuilder device template. They have a “.NbDt” extension.

The device template folder and all its contents can be moved and re-inserted into an existing project. Moving a device template folder can cause compilation errors due to absolute file references in use, or due to resource files being moved.

The default location for the NodeBuilder device template files is <NbDtFolder>.

### **Neuron C Source Files**

These files have .nc, .c, and .h extensions. The main .nc file name is specified by right-clicking the device template and selecting **Set Source File** from the shortcut menu. You must check in this file and any files brought in with include directives.

Standard headers are stored in the <LonWorks>\NeuronC\Include directory. These files should never be edited because future updates to the NodeBuilder tool will overwrite modified files and your changes would be lost. Check these in to ensure that you can go back to the version used to create your device, but be cautious when restoring them so that you do not overwrite newer versions.

You can determine the set of dependent files from the Project pane by performing a successful unconditional build operation and inspecting the files listed under the **Dependencies** folder.

### **Miscellaneous files**

These files may have any extension. These files include user-defined libraries, build script files, and other user-defined files. Check these in if they are required to build or document your application image.

### **Hardware Template files**

NodeBuilder hardware templates describe the hardware that will be used to host the application. This data includes Neuron Chip model, clock rate, and memory map. They have a .NbHwt extension.

Standard hardware templates are stored in <LonWorks>\NodeBuilder\Templates\Hardware\Standard. These files should never be

edited because future updates to the NodeBuilder tool will overwrite modified files and your changes would be lost. Check these in to ensure that you can go back to the version used to create your device, but be cautious when restoring them so that you do not overwrite newer versions.

You can place user hardware templates in any folder. A cross-project collection of user hardware templates may be found in the User hardware templates folder, which by default is in  
C:\LM\Source\Templates\Hardware\User.

## Resource Files

Resource files are produced by the *Resource Editor*. The set of files is called a resource file set. The resource file set holds definitions of functional profiles, network variable types, and configuration property types. These files have .typ, .fmt, .fpt, and .<lang> extensions.

You can move resource files by removing the reference to the previous resource folder from the resource file catalog using the NodeBuilder Resource Editor, moving the resource folder and all its content to a new location, and then adding the new resource folder to the resource catalog using the resource editor. You must also add all required resource folders to the resource catalog when moving or restoring a NodeBuilder project to a new computer.

To register a resource file from a build script, change the current directory to the <LonWorks>\Types folder and enter the following command:

```
mkcat -a<ResourceFolderPath>
```

Do not check-in the resource catalog itself (LDRF.cat by default), since it might contain references to resource files that are unique to each computer.

# Appendix F

## NodeBuilder Software License Agreement

This appendix contains a copy of the NodeBuilder Software License Agreement that you must accept to install or use the NodeBuilder software:

---

## NOTICE

This is a legal agreement between you and Echelon Corporation (“Echelon”). YOU MUST READ AND AGREE TO THE TERMS OF THIS SOFTWARE LICENSE AGREEMENT BEFORE ANY LICENSED SOFTWARE CAN BE DOWNLOADED OR INSTALLED OR USED. BY CLICKING ON THE “I AGREE” OR “ACCEPT” BUTTON OF THIS SOFTWARE LICENSE AGREEMENT, OR DOWNLOADING LICENSED SOFTWARE, OR INSTALLING LICENSED SOFTWARE, OR USING LICENSED SOFTWARE, YOU ARE AGREEING TO BE BOUND BY THE TERMS AND CONDITIONS OF THIS SOFTWARE LICENSE AGREEMENT. IF YOU DO NOT AGREE WITH THE TERMS AND CONDITIONS OF THIS SOFTWARE LICENSE AGREEMENT, THEN YOU SHOULD EXIT THIS PAGE AND DO NOT DOWNLOAD OR INSTALL OR USE ANY LICENSED SOFTWARE. BY DOING SO YOU FOREGO ANY IMPLIED OR STATED RIGHTS TO DOWNLOAD OR INSTALL OR USE LICENSED SOFTWARE.

---

## SOFTWARE LICENSE AGREEMENT

In consideration of Your agreement to the terms of this Agreement, Echelon grants You a limited, non-exclusive, non-transferable license to use up to two (2) copies of the Licensed Software and any updates or upgrades thereto provided by Echelon according to the terms set forth below. If the Licensed Software is being provided to You as an update or upgrade to software which You have previously licensed, then You agree the Licensed Software may be used and transferred only as part of a single product package and may not be separated for use on more than two (2) computers as expressly provided below.

---

## DEFINITIONS

For the purpose of this Agreement, the following terms shall have the following meanings:

- “Documentation” means the documentation included with the Licensed Software.
- “Licensed Software” means all computer software programs and associated media, printed materials, and online or electronic documentation that accompany the NodeBuilder Development Tool product; including, without limitation, the NodeBuilder Example Applications. The Licensed Software also includes any software updates, add-on components, stencils, templates, shapes, SmartShapes symbols, web services and/or supplements that Echelon may provide to You or make available to You, or that You obtain from the use of features or functionality of the Licensed Software, after the date you obtain your initial copy of the Licensed Software (whether by delivery of a CD, permitting downloading from the Internet or a dedicated web site, or otherwise) to the extent that such items are not accompanied by a separate license agreement or terms of use. Licensed Software does not include the LonMaker Integration tool, LNS™ DDE Server, Microsoft Visio, or any other software product shipped with the NodeBuilder Development Tool product

and not contained in the NodeBuilder directories as identified in the Documentation.

- “NodeBuilder Example Applications” means the Neuron C and Visual Basic source code example applications included as part of the Licensed Software which demonstrate the use of the Licensed Software, (i) as provided in the “Examples” and “Gizmo4” directories and their subdirectories, (ii) as generated by the NodeBuilder Code Wizard, (iii) as generated by the LNS Plug-in Wizard, or (iv) otherwise containing wording in the source code clearly identifying such source code as an “Example Application”.
- “LONWORKS® Device” means a product designed for use in a network based upon Echelon’s LONWORKS platform, including without limitation LONWORKS Application(s) as set forth in the LONWORKS OEM License Agreement between You and Echelon.
- “Your Device” means a LONWORKS Application that you develop as set forth in the LONWORKS OEM License Agreement between You and Echelon.
- “Your Device Plug-in” means Your software product that makes calls to the LNS server or LNS remote client (as both terms are described in the Documentation) and which (i) operates only with Your Device, (ii) allows the user to set or retrieve application configuration properties, to read or write application data, or to perform diagnostics on only a single device at a time, (iii) provides a user interface that is customized for Your Device, (iv) does not recover, commission, or install any LONWORKS Device, including the LONWORKS Device being operated on, (v) conforms to the device plug-in specifications described in the Documentation, and (vi) does not include any of the following: (a) code that increases or decreases the number of available device credits or LonMaker credits, or (b) the Licensed Software with the exception of derivative works of the NodeBuilder Example Applications.
- “You(r)” means Licensee, i.e. the company, entity or individual who has rightfully acquired the NodeBuilder Development Tool.

---

## LICENSE

### You may:

- a) use the Licensed Software solely to develop Your Devices and Your Device Plug-ins and prepare your derivative works of the NodeBuilder Example Applications to develop Your Devices and Your Device Plug-ins;
- b) install and use the Licensed Software for such purposes on one (1) primary computer (the “Primary Computer”);
- c) install and use a second copy of the Licensed Software for such purposes on one (1) additional computer (the “Additional Computer”) for the exclusive use of the individual who is the primary user of the copy of the Licensed Software installed on the Primary Computer, provided that the Licensed Software may only be used on one computer at a time, and provided that such installation and use otherwise comply with all the terms and conditions of this Agreement;

- d) keep the original media on which the Licensed Software was provided by Echelon solely for backup or archival purposes;
- e) make, use, and sell Your Devices that You developed pursuant to the terms of the LONWORKS OEM License Agreement between You and Echelon;
- f) distribute Your Device Plug-ins; and
- g) physically transfer any authorized copy of the Licensed Software from one (1) computer to another, provided that such copy is removed from the computer on which it was previously installed and the Licensed Software is used on only one (1) computer at a time.

**You may not, and shall not permit others to:**

- a) install the Licensed Software for development on more than one (1) Primary Computer and one (1) Additional Computer, use the Licensed Software on more than one (1) computer at a time, or allow any individual other than the primary user to use the Licensed Software on the Additional Computer;
- b) copy the Licensed Software except as permitted above;
- c) except for the limited rights granted above, modify, translate, reverse engineer, decompile, disassemble or otherwise attempt (i) to defeat, avoid, bypass, remove, deactivate or otherwise circumvent any software protection mechanisms in the Licensed Software, including without limitation any such mechanism used to restrict or control the functionality of the Licensed Software, or (ii) to derive the source code or the underlying ideas, algorithms, structure or organization from any of the Licensed Software that has not been provided in source code form (except to the extent that such activities may not be prohibited under applicable law);
- d) alter, adapt, prepare derivative works of, modify or translate the Licensed Software in any way for any purpose, including without limitation error correction, except for the limited rights expressly granted above with respect to NodeBuilder Example Applications; or
- e) except for the limited rights granted above, distribute, rent, loan, lease, transfer or grant any rights in the Licensed Software or modifications thereof in any form to any person without the prior written consent of Echelon.

You hereby acknowledge and agree that Your Device is a LONWORKS Application as such term is defined in the LONWORKS OEM License Agreement between Echelon and Licensee and therefore, Your Device is subject to the terms thereof and you shall have no rights to distribute Your Devices or Your Plug-ins as set forth above unless You and Echelon shall have entered into a LONWORKS OEM License Agreement prior any such distribution.

This license is not a sale. Title, copyrights and all other rights to the Licensed Software and any copy made by You remain with Echelon and its suppliers. Unauthorized copying of the Licensed Software or the Documentation, or failure to comply with the above restrictions, will result in



automatic termination of this license and will make available to Echelon other legal remedies.

---

## **TERMINATION**

This license will continue until terminated. Unauthorized copying of the Licensed Software or failure to comply with the above restrictions will result in automatic termination of this Agreement and will make available to Echelon other legal remedies. This license will also automatically terminate if you go into liquidation, suffer or make any winding up petition, make an arrangement with Your creditors, or suffer or file any similar action in any jurisdiction in consequence of debt. Upon termination of this license for any reason you will destroy all copies of the Licensed Software. Any use of the Licensed Software after termination is unlawful.

---

## **TRADEMARKS**

You may make appropriate and truthful reference to Echelon and Echelon products and technology in Your company and product literature; provided that You properly attribute Echelon's trademarks and do not use the name of Echelon or any Echelon trademark in Your name or product name. No license is granted, express or implied, under any Echelon trademarks, trade names, trade dress, or service marks.

---

## **LIMITED WARRANTY AND DISCLAIMER**

Echelon warrants to you that, for a period of ninety (90) days from the date of delivery or transmission to You, the Licensed Software programs under normal use will perform substantially in accordance with the Licensed Software specifications contained in the Documentation. Echelon's entire liability and Your exclusive remedy under this warranty will be, at Echelon's option and expense, to use reasonable commercial efforts to attempt to correct or work around errors, to replace the Licensed Software with functionally equivalent Licensed Software, or to terminate this Agreement and accept return of the NodeBuilder Development Tool and refund Your purchase price less a reasonable amount for use. **NOTWITHSTANDING THE FOREGOING, ECHELON MAKES NO WARRANTIES WHATSOEVER WITH RESPECT TO THE NODEBUILDER EXAMPLE APPLICATIONS.**

**EXCEPT FOR THE EXPRESS LIMITED WARRANTIES AND CONDITIONS GIVEN BY ECHELON ABOVE, ECHELON AND ITS SUPPLIERS MAKE AND YOU RECEIVE NO OTHER WARRANTIES OR CONDITIONS, EXPRESS, IMPLIED, STATUTORY OR OTHERWISE OR IN ANY COMMUNICATION WITH YOU, AND ECHELON AND ITS SUPPLIERS SPECIFICALLY DISCLAIM ANY IMPLIED WARRANTY OF MERCHANTABILITY, SATISFACTORY QUALITY, FITNESS FOR A PARTICULAR PURPOSE OR NONINFRINGEMENT AND THEIR EQUIVALENTS.** Echelon does not warrant that the operation of the Licensed Software will be uninterrupted or error free or that the Licensed Software will meet Your specific requirements.

**SOME STATES OR OTHER JURISDICTIONS DO NOT ALLOW THE EXCLUSION OF IMPLIED WARRANTIES, SO THE ABOVE EXCLUSIONS**

MAY NOT APPLY TO YOU. YOU MAY ALSO HAVE OTHER RIGHTS THAT VARY FROM STATE TO STATE AND JURISDICTION TO JURISDICTION.

---

## **LIMITATION OF LIABILITY**

IN NO EVENT WILL ECHELON OR ITS SUPPLIERS BE LIABLE FOR LOSS OF OR CORRUPTION TO DATA, LOST PROFITS OR LOSS OF CONTRACTS, COST OF PROCUREMENT OF SUBSTITUTE PRODUCTS OR OTHER SPECIAL, INCIDENTAL, PUNITIVE, CONSEQUENTIAL OR INDIRECT DAMAGES, LOSSES, COSTS OR EXPENSES OF ANY KIND ARISING FROM THE SUPPLY OR USE OF THE LICENSED SOFTWARE, HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY (INCLUDING WITHOUT LIMITATION NEGLIGENCE). THIS LIMITATION WILL APPLY EVEN IF ECHELON OR AN AUTHORIZED DISTRIBUTOR HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES AND NOTWITHSTANDING THE FAILURE OF ESSENTIAL PURPOSE OF ANY LIMITED REMEDY. EXCEPT TO THE EXTENT THAT LIABILITY MAY NOT BY LAW BE LIMITED OR EXCLUDED, IN NO EVENT SHALL ECHELON'S OR ITS SUPPLIERS' LIABILITY EXCEED TEN THOUSAND DOLLARS (\$10,000). YOU ACKNOWLEDGE THAT THE AMOUNTS PAID BY YOU FOR THE LICENSED SOFTWARE REFLECT THIS ALLOCATION OF RISK.

SOME STATES OR OTHER JURISDICTIONS DO NOT ALLOW THE EXCLUSION OR LIMITATION OF LIABILITY FOR INCIDENTAL OR CONSEQUENTIAL DAMAGES, SO THE ABOVE LIMITATIONS AND EXCLUSIONS MAY NOT APPLY TO YOU.

---

## **SAFE OPERATION**

YOU ASSUME RESPONSIBILITY FOR, AND HEREBY AGREE TO USE YOUR BEST EFFORTS IN, DESIGNING AND MANUFACTURING PRODUCTS USING THE LICENSED SOFTWARE TO PROVIDE FOR SAFE OPERATION THEREOF, INCLUDING, BUT NOT LIMITED TO, COMPLIANCE OR QUALIFICATION WITH RESPECT TO ALL SAFETY LAWS, REGULATIONS AND AGENCY APPROVALS, AS APPLICABLE. THE LICENSED SOFTWARE, SMART TRANSCEIVERS, NEURON CHIPS, YOUR DEVICE, YOUR DEVICE PLUG-IN AND OTHER ECHELON PRODUCTS AND TECHNOLOGY ARE NOT DESIGNED OR INTENDED FOR USE AS COMPONENTS IN EQUIPMENT INTENDED FOR SURGICAL IMPLANT INTO THE BODY, OR OTHER APPLICATIONS INTENDED TO SUPPORT OR SUSTAIN LIFE, FOR USE IN FLIGHT CONTROL OR ENGINE CONTROL EQUIPMENT WITHIN AN AIRCRAFT, OR FOR ANY OTHER APPLICATION IN WHICH THE FAILURE THEREOF COULD CREATE A SITUATION IN WHICH PERSONAL INJURY OR DEATH MAY OCCUR, AND YOU SHALL HAVE NO RIGHTS HEREUNDER FOR ANY SUCH APPLICATIONS.

---

## LANGUAGE

The parties hereto confirm that it is their wish that this Agreement, as well as other documents relating hereto, have been and shall be written in the English language only.

Les parties aux présentes confirment leur volonté que cette convention de même que tous les documents y compris tout avis qui s'y rattache, soient rédigés en langue anglaise.

---

## SUPPORT

You acknowledge that You shall either (i) inform the end-user that You are the primary support contact for Your Devices and Your Device Plug-ins, and that Echelon Corporation will not support Your Devices and Your Device Plug-ins, or (ii) inform the end-user that there will be no support for Your Devices and Your Device Plug-ins.

---

## GENERAL

This Agreement shall not be governed by the 1980 U.N. Convention on Contracts for the International Sale of Goods; rather, this Agreement shall be governed by the laws of the State of California, including its Uniform Commercial Code, without reference to conflicts of laws principles. This Agreement is the entire agreement between You and Echelon and supersedes any other communications, representations or advertising with respect to the Licensed Software. If any provision of this Agreement is held invalid or unenforceable, such provision shall be revised to the extent necessary to cure the invalidity or unenforceability, and the remainder of the Agreement shall continue in full force and effect. If You are acquiring the Licensed Software on behalf of any part of the U.S. Government, the following provisions apply. The Licensed Software programs and Documentation are deemed to be “commercial computer software” and “commercial computer software documentation”, respectively, pursuant to DFAR Section 227.7202 and FAR 12.212(b), as applicable. Any use, modification, reproduction, release, performance, display or disclosure of the Licensed Software programs and/or Documentation by the U.S. Government or any of its agencies shall be governed solely by the terms of this Agreement and shall be prohibited except to the extent expressly permitted by the terms of this Agreement. Any technical data provided that is not covered by the above provisions is deemed to be “technical data-commercial items” pursuant to DFAR Section 227.7015(a). Any use, modification, reproduction, release, performance, display or disclosure of such technical data shall be governed by the terms of DFAR Section 227.7015(b).



# Index

- #LO modifiers
  - localized list separators, 7-49
  - localized time and date formats, 7-49
- \_DEBUG directives, 10-12
- 3120 chips, 9-14
- 3150 chips
  - programming 3150 on-chip memories, 9-11
  - programming off-chip memories, 9-10
- 32-bit values
  - initializers, 6-27
- access methods. *See* configuration properties
- acknowledged service types, 6-17
- active language, 7-6
  - setting, 7-6
- active sets
  - defined, 7-52
- ad-hoc systems, 2-6
- agreements
  - software license, F-2
- alarms
  - tutorial, 2-26
- ANSI C, 1-8, 3-7
- ANSI/EIA 709-1 protocol
  - defined, 1-2
- application errors
  - fatal, 9-13
- application image files, 3-7
  - installing, 15-2
  - tutorial, 2-32
- application images
  - build files, 9-3
  - building, 3-7, 9-2, 9-21
  - building tutorial, 2-32
  - device interface files (.xif, .xfb, and .xfo), 9-4
  - downloadable application image files (.nxe and .apb), 9-3
  - downloading tutorial, 2-32
  - EEPROM application image files (.nei), 9-4
  - flash application image files (.nfi), 9-4
  - loading, 9-9, 9-19
  - programmable application image files (.nri, .nei, and .nfi), 9-3
  - programming, 9-11
  - ROM application image files (.nri), 9-4
  - viewing status, 9-6
- application keys, 2-13
- applicationless state
  - forcing, 9-21
  - reboot, 9-13
- applications
  - browsing, 11-3
  - building, 3-7, 5-4
  - compiling, 3-7
  - configuration properties, 1-6
  - creating, 3-7
  - debugging, 3-8. *See* debugger
  - defined, 1-2, 1-3
  - designing, 3-2
  - developing, 3-2
  - device file sets, 3-7
  - downloading, 3-7
  - functional blocks, 1-7
  - host-based, 1-2

- network variables, 1-5
- testing, 11-2
- arrays
  - code wizard, 6-35
- as-built reports, 2-6
- assembler dependency files, C-3
- attached, 3-5
- auto indent, 8-8
- auto-detect
  - channels, 9-17
- automatic program ID management, 5-8, 5-11
  - tutorial, 2-24
- backup files, 4-12
- binary device interface files, 9-5
- binding, 1-6
- bitfield types, 7-18, 7-24
- bookmarks, 8-7
- boot ID, 9-11
- boot image, 9-11
- Breakpoint pane
  - defined, 10-3
- breakpoints
  - defined, 10-5
  - reset, 10-6
  - resynchronization, 10-6, 10-13
  - setting, 10-4, 10-6
  - synchronization, 10-6, 10-13
  - toggling, 10-4
  - toolbar, 10-3
  - tutorial, 2-38
- BridgeView, 14-4
- browser
  - introduction, 3-8
- build options, 9-7
- build scripts
  - examples, C-4
  - generating, C-2
  - invoking, C-2
  - options, 9-9
- build status
  - viewing, 9-6
- build tools
  - examples, C-4
  - invoking, C-2
- building, 3-7. *See* application images, building
- call stack, 10-9
- Call Stack pane, 10-5, 10-9
  - defined, 10-3
- catalog utility, E-4
- certification, 3-10
- changeable interfaces
  - program IDs, 5-16
- changeable type network variables, 6-15
- changeable types
  - program IDs, 5-16
  - setting, 6-28
- channel types
  - defined, 1-2
  - program IDs, 1-4, 5-15
- channels
  - auto-detect, 9-17
  - defined, 1-2
  - specifying, 9-17
- checksum errors, 9-13
- clock speeds, 5-24
- code
  - generating, 6-27
- code wizard, 2-7
- code wizard
  - adding implementation-specific configuration properties, 6-19

- adding implementation-specific network variables, 6-13
- arrays, 6-35
- code sections, 6-31
- commands, 6-32
- comments, 6-32
- common.h file, 6-29, 6-31
- common.nc file, 6-29, 6-31
- configuration properties, 6-35
- configuration property access method, 6-3
- cp\_family reuse, 6-35
- declaration order, 6-36
- defined, 6-2
- defining the device interface, 6-4
- device template properties, 6-5
- device templates, 6-3
- director functions, 6-31, 6-37
- editing properties, 6-27
- event handling, 6-31
- file directory structure, 6-30
- files created, 6-29
- fileSYS.h file, 6-30
- fileSYS.nc file, 6-29
- fileXfer.h file, 6-30
- fileXfer.nc, 6-37
- fileXfer.nc file, 6-30
- FTP server, 6-37
- generating code, 6-3, 6-5, 6-27
- global configuration properties, 6-35
- implementing network variables, 6-7
- Interface pane, 6-3
- modifying code, 6-27, 6-31
- network variable arrays, 6-35
- Neuron C Version 2 features not supported, 6-34
- Node Objects, 6-31, 6-37
- NodeObject.h file, 6-30
- NodeObject.nc file, 6-30
- overriding generated code, 6-32
- polled network variables, 6-35
- refreshing, 6-5
- Resource pane, 6-3
- shared configuration properties, 6-35, 6-36
- source code features, 6-30
- source files, 6-29
- starting, 5-3, 5-12, 6-2
- suggested changes to generated code, 6-33
- Translator.h file, 6-32
- tutorial, 2-25, 2-26, 2-30
- validations, 6-28
- when tasks, 6-31, 6-36
- Code Wizard, 6-29, 6-32
- colors
  - debugger, 10-14
  - editor, 8-2, 8-9
- command lines
  - building projects, C-2
- common.h file, 6-29, 6-31
- common.nc file, 6-29, 6-31
- communication parameters
  - reboot, 9-14
- company information
  - resource file sets, 7-15
- compiler dependency files, C-3
- compilator, 3-7
  - tutorial, 2-32
- complete mode, 13-5
- computer requirements, 2-12
- conditional formats, 7-47
- configuration network variables, 6-12, 6-23
- configuration properties
  - optional, 6-9

- configuration properties
  - access methods, 6-5
  - adding device configuration properties, 6-24
  - adding implementation-specific, 6-19
  - adding to functional profiles, 7-31
  - applies to, 7-34
  - applying, 6-23
  - array types, 7-18
  - browsing, 11-4
  - code wizard, 6-35
  - configuration network variables, 6-23
  - constant option, 7-33
  - creating bitfield types, 7-24
  - creating enumerated types, 7-23
  - creating reference types, 7-24
  - creating structure or union types, 7-22
  - creating types, 7-16
  - defaults, 7-34
  - defined, 1-6, 3-4
  - device interfaces, 1-8
  - editing types, 7-16
  - formats. *See* formats
  - global, 6-35
  - implementation options, 6-12
  - implementing as configuration network variables, 6-12
  - implementing in code wizard, 6-9
  - implementing static configuration properties, 6-12
  - inheriting types, 7-18
  - initial values, 6-24, 6-27
  - initializers, 6-13, 6-23
  - mandatory, 6-9
  - manufacturer option, 7-33
  - member names, 6-11
  - member numbers, 7-33
  - modifying bitfield types, 7-24
  - modifying enumerated types, 7-23
  - modifying reference types, 7-24
  - modifying structure or union types, 7-22
  - naming, 6-10, 6-20, 7-32
  - naming types, 7-17
  - object disabled option, 7-33
  - offline option, 7-34
  - overrides, 7-34
  - reset option, 7-33
  - resource file paths, 6-21
  - restriction flags, 6-11, 6-22
  - scopes, 6-21
  - settings, 7-33
  - shared, 6-35, 6-36
  - standard types
    - defined, 1-9
    - static, 6-23
    - string information, 7-34
    - types, 1-6, 6-11, 6-20, 7-2, 7-3, 7-9
    - viewing properties, 11-4
- Configuration Properties, 6-11, 6-24
- configuration property types
  - defined, 3-4
- Configuration Property Types, 7-24
- configured state
  - boot image, 9-11
- connections
  - defined, 1-6
- const\_flg flags, 6-12, 6-22
- constant options
  - configuration properties, 7-33
- conversion specifications, 7-47
- convert values, 6-26
- cp\_family
  - reuse, 6-35



- custom devices. *See* devices
- date
  - localized formats, 7-49
- DDE server, 14-3
  - application keys, 2-13
  - defined, 2-5
  - human-machine interfaces (HMIs), 3-10
  - installing, 2-13
  - licensing, 2-13
- DDE Server
  - CD, 2-5
- debug directives, 10-12
- Debug Error Log Tab, 10-12
- Debug Log tab, 10-3
- Debug Status pane, 10-10
  - defined, 10-3
- debugger, 3-8
  - breakpoints. *See* breakpoints
  - call stack, 10-9
  - color options, 10-14
  - current device, 10-10
  - Debug Error Log tab, 10-12
  - defined, 10-2
  - editing source code, 10-13
  - executing in development targets only, 10-12
  - font options, 10-14
  - halting, 10-4, 10-5, 10-10
  - introduction, 3-8
  - jumping to current instruction, 10-4
  - message options, 10-13
  - peeking and poking memory, 10-11
  - radix option, 10-13
  - reseting, 10-4
  - resuming, 10-3, 10-5, 10-10
  - running to cursor, 10-4, 10-5
  - setting options, 10-13
  - starting, 10-2, 10-5
  - stepping. *See* stepping
  - stopping, 10-4, 10-5, 10-10
  - tick interval option, 10-13
  - toolbar, 10-3
  - tutorial, 2-37
  - viewing status, 10-10
  - watch list. *See* watches
  - watching. *See* watches
- decimal
  - watches, 10-8
- defaults
  - configuration properties, 7-34
- dependencies, 5-17
- dependency files
  - compiler, C-3
  - exporter, C-4
  - linker, C-3
- descriptions
  - device, 9-18
- development platforms
  - LonBuilder Emulators, 9-9
  - LTM-10A, 9-9
- development process, 3-2
- Development Target Device shapes, 4-9
- development target folders, 5-17
- development targets, 5-5, 5-12, 5-17 *See* targets
  - defined, 1-9
- Development Targets, 10-12
- device classes
  - program IDs, 1-4, 5-14
- device configuration properties
  - adding, 6-24
- device file sets
  - defined, 3-7
  - tutorial, 2-32

- device files
  - importing, 5-2
- device interface files, 3-7, 9-4
  - defined, 1-8
  - installing, 15-2
  - tutorial, 2-32
- device interfaces
  - adding device configuration properties, 6-24
  - adding device network variables, 6-24
  - adding functional blocks, 6-5
  - defined, 1-8
  - defining, 3-3, 3-4
  - defining with code wizard, 6-4
  - implementing, 6-2
  - testing, 3-8
  - testing tutorial, 2-36
- device network variables
  - adding, 6-24
- device settings
  - editing, 9-22
- device template files, 4-14
  - defined, 4-14
- device template folders, 5-7
- Device Template Wizard, 4-8, 5-6, 5-7
  - New Device Template, 5-6
  - target platforms, 5-11
- Device Template Wizard
  - Program ID, 5-7, 5-8, 5-10
- device templates, 3-6, 5-2
  - building, 5-3, 9-2
  - cleaning, 5-3, 5-4
  - copying, 4-12, 4-13, 5-2
  - creating, 3-7, 3-8, 5-2, 5-6
  - creating tutorial, 2-21
  - defined, 1-8, 3-6
  - editing, 5-2
  - excluding, 5-4
  - importing from NodeBuilder 1.5, 5-2
  - inserting, 5-2
  - LNS device templates
    - defined, 1-8
  - moving, 4-12, 4-13
  - multiple, 2-21
  - naming, 5-6
  - output files, 5-7
  - program IDs, 2-22
  - removing, 5-4
  - selecting, 4-10
  - settings, 5-3
  - source files, 5-3, 5-4, 5-7
  - specifying targets, 9-15
  - tutorial, 2-21, 2-32
  - viewing, 5-2
  - viewing properties, 5-4
  - viewing status, 5-3, 5-4
- Device Templates folders
  - defined, 4-4
- device\_specific\_flg flags, 6-11, 6-22
- devices, 1-2. *See* targets
  - browsing, 11-3
  - commissioning, 4-10
  - creating shapes. *See* shapes
  - debugging, 3-8, 9-21
  - developing, 3-2
  - editing target device settings, 9-22
  - forcing applicationless, 9-21
  - installing, 3-9
  - loading, 9-9
  - naming, 4-9
  - testing, 3-8, 11-2
  - viewing status, 9-22
- Devices folders

- defined, 4-4, 9-21
- direct memory read/write, 6-3
- direction, 6-16
- director functions
  - code wizard, 6-37
  - default, 6-31
  - tutorial, 2-31
- directories
  - LonWorks directories, 2-16
- discrete formats, 7-44
- documentation
  - included with the NodeBuilder tool, 2-3
- downloadable application image files, 9-3
- drivers, 2-13
- dynamic network variables
  - program IDs, 5-16
- Edit pane
  - defined, 4-3
- Editing pane
  - defined, 4-2
- editor
  - auto indent, 8-8
  - bookmarks, 8-7
  - colors, 8-2, 8-9
  - cut, copy, and paste, 8-2
  - find in files, 8-4
  - font, 8-9
  - introduction, 8-2
  - pattern matching, 8-5
  - regular expression syntax, 8-5
  - reload option, 8-9
  - replacing, 8-3
  - reset option, 8-9
  - searching, 8-3
  - searching multiple files, 8-4
  - setting options, 8-8
  - syntax highlighting, 8-2
  - tab width, 8-8
  - tutorial, 2-31
  - using your own, 3-7
- EEBLANK.NRI, 9-12
- EEPROM. *See* memories. *See* memories
- EEPROM application image files, 9-4, 9-11
- EEPROM variables
  - reboot, 9-14
- engineered systems, 2-6
- enumerated types, 7-18, 7-23
- enumeration types
  - creating and modifying, 7-34
- enumerations
  - defined, 7-4
  - types, 7-9
- Enumerations, 7-35
- errors
  - fatal application errors, 9-13
- event log
  - defined, 4-3
- Example, A-2, A-4, A-15, A-16
- examples
  - adding analog I/O, A-8
  - adding digital I/O, A-5
  - adding Node Objects, A-4
  - adding quadrature input, A-23
  - adding real-time keepers, A-19
  - adding shaft-encoder input, A-23
  - adding temperature sensors, A-16
  - adding translators, A-12
  - adding wheel input, A-23
  - contents, A-2
  - creating, A-2
  - enhancing, A-27
  - installing, 2-16
  - introduction, A-2

- locations, A-2
- manufacturer ID, 2-22
- setting up, A-3
- exhaustive mode, 13-5
- exporter dependency files, C-4
- extended arithmetic function library, 5-19
- external interface files. *See* device interface files
- external names. *See* functional blocks
- fatal application errors, 9-13
- FbModes user cells, 7-3
- file directory structure, 6-30
- file extensions
  - language files, 7-4
- file transfer protocol, 6-3
- files
  - application image build files, 9-3
  - application image files, 3-7
  - cleaning, 9-5
  - defined, 4-14
  - device interface files, 3-7
  - device interface files (.xif, .xfb, and .xfo), 9-4
  - downloadable application image files (.nxe and .apb), 9-3
  - EEBLANK.NRI, 9-12
  - EEPROM application image files (.nei), 9-4
  - flash application image files (.nfi), 9-4
  - inserting, 5-5
  - language file extensions, 7-4
  - printing, 4-14
  - programmable application image files (.nri, .nei, and .nfi), 9-3
  - project files, 4-2
  - removing, 5-5
  - ROM application image files (.nri), 9-4
  - spidData.xml, 2-23
  - using source control, E-2
  - viewing, 4-14
- filesys.h file, 6-30
- filesys.nc file, 6-29
- filexfer.h file, 6-30
- filexfer.nc, 6-37
- filexfer.nc file, 6-30
- FIX, 14-4
- flags. *See* restriction flags
- flash application image files, 9-4
- flash memories, 9-4. *See* memories. *See* memories. *See* memories
- float types, 7-18
- floating point, 5-19
  - initializers, 6-27
- font options
  - debugger, 10-14
- fonts
  - editor, 8-9
- format specifiers. *See* formats
- format strings, 7-45
- formats
  - conditional formats, 7-47
  - conversion specifications, 7-47
  - creating and modifying, 7-41, 7-42
  - defined, 7-5, 7-41
  - examples, 7-45
  - fields, 7-44
  - format specifiers, 7-44
  - format strings, 7-45
  - locale-specific, 7-42
  - localized date formats, 7-49
  - localized list separators, 7-49
  - localized time formats, 7-49
  - modifier
    - defined, 7-42
  - modifiers, 7-44

- naming, 7-42
- program IDs, 1-3, 5-16
- resource files, 7-9
- scaling factors, 7-48
- text formats, 7-45

Formats, 7-44

FPT keys. *See* functional profile numbers

Functional Block shapes

- tutorial, 2-39

functional blocks

- adding, 6-5, 6-6
- adding implementation-specific configuration properties, 6-19
- adding implementation-specific network variables, 6-13
- browsing, 11-3
- configuring, 12-4
- creating, 11-2
- creating arrays, 6-6
- creating shapes. *See* shapes
- defined, 1-7, 3-4, 6-5
- device interfaces, 1-8
- disabling, 11-3
- enabling, 11-3
- external names, 6-5
- implementation-specific members, 1-7
- implementing optional configuration properties, 6-9
- members, 1-7
- naming, 6-6

functional profile keys. *See* functional profile numbers

functional profile numbers, 3-4

- inherited functional profiles, 7-26

functional profiles

- adding, 6-5
- adding configuration property members, 7-31
- adding network variable members, 7-28
- creating, 7-25
- defined, 1-7, 3-4, 7-3
- documentation, 3-4
- functional profile index, 7-28
- functional profile keys, 7-27
- functional profile numbers, 7-27
- inherited profiles, 7-31
- inheriting, 7-25, 7-26
- introduction, 7-2
- manufacturer-defined, 1-9, 3-4
- member numbers, 7-30
- modifying, 7-25
- naming, 7-25, 7-27
- overriding members, 7-27
- principal network variables, 7-28, 7-30
- resource files, 1-8, 7-9
- selecting, 3-4
- standard
  - defined, 1-9
  - documentation, 1-9
- generating code, 6-27
- generating resource files, 7-52

Gizmo 4, 2-10

- defined, 2-8
- documentation, 2-3, 2-10
- I/O devices, 2-10
- I/O library, 2-10
- installing, 2-17
- LTM-10A Flash Control Modules, 2-9
- testing, 3-3

Gizmo 4 User's Guide, 2-2

hardware

- designing, 3-2
- developing, 3-3
- device requirements, 3-3
- I/O, 3-3

- I/O connectors, 3-3
- prototyping, 3-3
- testing, 3-3
- hardware installation, 2-16
- hardware template files
  - defined, 4-14
- Hardware Template Properties, 5-22, 5-24, 5-25, 5-27
- hardware templates
  - clock speed, 5-24
  - copying, 4-13, 5-21
  - creating, 5-21, 5-22
  - defined, 1-7, 5-12, 5-17, 5-20
  - description, 5-26
  - editing, 5-22
  - importing, 5-21
  - inserting, 5-21
  - model, 5-24
  - naming, 5-23
  - platform, 5-23
  - removing, 5-22
  - setting hardware properties, 5-22
  - setting properties, 5-23
  - system image name, 5-24
  - system image version, 5-24
  - target definitions, 1-9
  - transceiver type, 5-23
  - viewing, 5-20
- Hardware Templates folders
  - defined, 4-4
- headers
  - resource file sets, 7-15
- hex values
  - calculating, 7-21
- hexadecimal
  - watches, 10-8
- high-performance network interfaces, 2-11
- HMI. *See* human-machine interfaces (HMIs)
- host-based devices
  - defined, 1-2
- human-machine interfaces (HMIs), 2-5, 3-9
  - DDE server, 14-3
  - defined, 14-2
  - installing, 15-4
  - LNS DDE Server, 14-3
  - third-party, 14-3
  - using the LonMaker tool as an HMI, 14-2
- i.LON 1000 Internet Servers, 2-11
  - installing, 2-13
- I/O connectors, 3-3
- I/O hardware
  - prototyping, 3-3
- implementation-specific configuration properties, 6-19
- implementation-specific members, 1-7
  - adding, 6-6
- implementation-specific network variables, 6-13
- include files
  - search path, 4-8
- inheritance, 3-5
- inherited functional profiles, 7-25, 7-26, 7-31
  - functional profile key, 7-26
  - overriding members, 7-27
- initial values, 6-24
- initializers, 6-13
  - configuration properties, 6-23
  - network variables, 6-19
- installation, 2-12
- Installation, 15-4
  - Creating*, 15-4
- installations. *See* software installations

- installing
  - tutorial, 2-39
- InstallShield, 15-2
- int formats, 7-44
- Intellution FIX, 14-4
- Interface pane
  - tutorial, 2-26
- InTouch, 2-5, 3-10, 14-4
- Introduction to LONWORKS document, 1-1
- LabView, 14-4
- language files, 7-9
  - active language, 7-6
  - adding language strings, 7-38
  - creating and editing language strings, 7-37
  - defined, 7-4
  - exporting, 7-41
  - file extensions, 7-4
  - translating, 7-6, 7-40
  - viewing two side-by-side, 7-40
- language strings
  - adding, 7-38
  - adding while defining a resource, 7-39
  - copying, 7-38
  - creating and modifying, 7-37
- ldrf.cat files, 7-7
- libraries
  - adding, 5-5, 5-18, 5-19
  - contents, 5-19
  - custom, 5-18
  - extended arithmetic, 5-19
  - floating point, 5-19
  - programmable serial gateway, 5-19
  - standard, 5-18, 5-19
- license agreements, F-2
- licenses
  - OEM, 3-2
- limits, 7-19
  - setting, 7-20
- linker dependency files, C-3
- list separators
  - localized, 7-49
- LNS
  - plug-ins tutorial, 2-42
- LNS DDE Server. *See* DDE server. *See* DDE Server
  - documentation, 2-3
- LNS DDE Server User's Guide, 2-13
- LNS Device Plug-in Developer's Guide, 2-2
- LNS Device Plug-in Wizard, 2-42. *See* plug-in wizard
- LNS device templates, 1-8. *See* device templates
  - defined, 3-6
- Load After Build, 9-3
- localization. *See* translation
- localized formats. *See* formats
- localized list separators, 7-49
- localized time and date formats, 7-49
- locations
  - device, 9-18
- locked scope and program ID templates, 7-13
- logical addresses, 1-6
- LonBuilder Development Tool, 2-11
  - using for development, D-2
- LonBuilder Emulator
  - using as a development platform, D-2
- LonBuilder Emulators, 3-3
- LonMaker, 12-3
  - New Device Wizard, 4-9, 4-10
- LonMaker browser. *See* browser
  - calculating raw values, 7-21
- LonMaker Browser, 11-3

- tutorial, 2-36
- LonMaker credits, 2-13
- LonMaker Integration Tool
  - backing up, 4-12
  - CD, 2-5
  - configuring devices, 1-6
  - creating a network, 3-5
  - creating a network tutorial, 2-19
  - defined, 2-5, 3-5
  - installing, 2-13
  - installing a device tutorial, 2-39
  - making connections, 1-5
  - New Device Wizard, 4-9
  - production devices, 3-5
  - starting, 3-5
  - starting the NodeBuilder tool, 4-9
  - tutorial, 2-33
- LonMaker networks
  - multiple, 2-20, 3-6, 4-2
- LonMaker tool
  - creating shapes, 12-2. *See shapes*
  - creating stencils, 12-2. *See shapes*
  - shapes, 3-9
  - stencils, 3-9
- LonMaker User's Guide, 2-7, 2-13
- LONMARK Application Layer
  - Interoperability Guidelines, 1-5, 1-8
- LONMARK Interoperability Association
  - certifying devices, 3-10
  - program IDs, 5-13
- LONMARK Interoperability Association, 1-4
  - defined, 3-10
  - resource files, 1-9
- LONMARK SNVT and SCPT Guide, 1-9
- lonpmk32.def file, C-3
- LonTalk protocol
  - defined, 1-2
- LONWORKS directory, 2-16
- LonWorks networks
  - introduction, 1-2
- LonWorks platform
  - introduction, 1-2
- LTM-10A Flash Control Modules, 2-11, 2-12
  - defined, 2-4
  - documentation, 2-3
- LTM-10A Platforms
  - defined, 2-3
  - development platforms, 3-3
  - documentation, 2-3
  - Gizmo 4, 2-8
- major versions, 7-53
- mandatory configuration properties, 6-6
- mandatory network variables, 6-6
- manufacture options
  - configuration properties, 7-33
- manufacturer IDs
  - entering, 2-15
  - getting, 2-12
  - program IDs, 1-3, 5-13
  - tutorial, 2-22
- maximum model numbers, 5-9
- maximum values, 7-31
- maximums
  - setting, 7-20
- member numbers, 7-30
  - defining, 7-33
- members
  - functional blocks, 1-7
  - implementation-specific, 1-7
- memories
  - EEPROM, 5-26
  - flash, 5-26
  - initialization, 9-12



- NVRAM, 5-26
- off-chip, 5-24
- on-chip, 5-24
- peeking and poking, 10-11
- programming 3120 on-chip memory, 9-14
- programming flash memories, 9-11
- programming off-chip memory, 9-10
- programming on-chip memory, 9-11
- sector size, 5-26
- specifying, 5-24
- write time, 5-26
- memory signature, 9-12
- message options
  - debugger, 10-13
- Messages tab, 4-3, 9-2
- mfg\_flg flags, 6-11, 6-22
- Microprocessor Interface Program (MIP)
  - LTM-10A modules, 2-4
- Microsoft Visual Basic 6, 2-12
- minimal mode, 13-5
- minimum model numbers, 5-9
- minimum values, 7-31
- minimums
  - setting, 7-20
- minor versions, 7-53
- mkcat utility, E-4
- model numbers, 5-8
  - program IDs, 1-5, 5-9, 5-15
- models, 5-24
- modifier. *See* formats
- modifiers, 6-17
- National Instruments BridgeView, 14-4
- National Instruments LabView, 14-4
- NbDt, 4-14. *See* device template files. *See* device template files
- nbHwt. *See* hardware templates
- NbHwt. *See* hardware template files
- NbOpt, 4-2, 4-7. *See* options files
- NbPrj, 4-2, 4-7, 4-11, 4-12. *See* project files
- NbProj. *See* project files
- network interfaces, 2-11, 2-13
  - program IDs, 1-3
- Network Variable shapes
  - tutorial, 2-39
- network variable types
  - defined, 3-4
- Network Variable Types, 7-23
  - Creating*, 7-20
  - Editing*, 7-20
- network variables
  - adding device network variables, 6-24
  - adding implementation-specific, 6-13
  - adding to functional profiles, 7-28
  - array types, 7-18
  - arrays, 6-14, 6-35
  - binding, 1-6
  - browsing, 11-4
  - changeable types, 6-15
  - configuration network variables, 6-12
  - connecting, 1-5
  - creating arrays, 6-8
  - creating bitfield types, 7-24
  - creating enumerated types, 7-23
  - creating reference types, 7-24
  - creating structure or union types, 7-22
  - creating types, 7-16
  - defined, 1-5, 3-3
  - device interfaces, 1-8
  - direction, 6-8, 6-16
  - editing types, 7-16

- formats. *See* formats
- implementation-specific, 6-19
- implementing in code wizard, 6-7
- initial values, 6-24
- initializers, 6-9, 6-19
- member names, 6-8
- member numbers, 6-8
- modifiers, 6-17
- modifying bitfield types, 7-24
- modifying enumerated types, 7-23
- modifying reference types, 7-24
- modifying struture or union types, 7-22
- naming, 6-8, 6-14
- naming members, 7-29
- naming types, 7-17
- polled, 6-9, 6-18, 6-35, 7-31
- principal, 7-28
- renaming, 6-8
- scopes, 6-15
- self-documentation, 6-9, 6-18
- service types, 6-8, 6-16
- setting types, 6-6, 6-8
- standard types
  - defined, 1-9
- synchronous, 6-9, 6-18
- types, 6-15, 7-2, 7-3, 7-9
- viewing properties, 11-4
- Network Variables, 6-9, 6-14
  - Implementation-specific, 6-14
- networks
  - creating, 3-5
- Neuron 3120 Chip programmer, 9-14
- Neuron C, 6-36
  - converting version 1 to version 2, B-4
  - defined, 1-8, 3-7
  - device templates, 1-8
  - documentation, 3-7
  - editing tutorial, 2-31
  - tutorial, 2-26, 2-30
  - Version 2 backward compatibility, B-2
  - writing, 3-7
- Neuron C Errors Guide, 2-2
- Neuron C Programmer's Guide, 2-3
- Neuron C Reference Guide, 2-2, 2-3
- Neuron Chip models, 5-24
- Neuron Chip-hosted devices
  - defined, 1-2
- New Device Template Wizard, 5-2
- nlib library utility, 5-19
- Node Object
  - defined, 6-4
  - examples, A-4
  - tutorial, 2-26
- Node Objects
  - code wizard, 6-31, 6-37
- NodeBuilder 1.5, 5-2, 5-22
  - converting projects, B-2, B-7
  - running, B-2, B-9
- NodeBuilder 1.5 Development Tool, 2-10
- NodeBuilder Code Wizard. *See* code wizard
- NodeBuilder debugger. *See* debugger. *See* debugger
- NodeBuilder Development Tool. *See* NodeBuilder tool
- NodeBuilder device templates. *See* device templates
- NodeBuilder Errors Guide, 2-3
- NodeBuilder example. *See* examples
- NodeBuilder Gizmo 4 I/O Board. *See* Gizmo 4
- NodeBuilder Project, 4-11
- NodeBuilder Project Manager. *See* project manager

- NodeBuilder projects. *See* projects. *See* projects
- NodeBuilder Quick-Start Tutorial, 2-7
- NodeBuilder tool
  - CD, 2-7
  - components, 2-2, 2-7
  - contents, 2-2
  - defined, 2-2
  - documentation, 2-2
  - hardware requirements, 2-11
  - introduction, 2-2
  - requirements, 2-11
  - software requirements, 2-11
  - starting, 4-9
  - system requirements, 2-11
  - upgrades, 2-10, 2-12
- NodeObject.h file, 6-30
- NodeObject.nc file, 6-30
- nodes. *See* devices
- NVRAM, 9-4. *See* memories
- obj\_disabl\_flg flags, 6-12, 6-22
- object disabled options
  - configuration properties, 7-33
- obsolete items, 7-6, 7-50
- obsolete types, 7-19
- off-chip memory. *See* memories. *See* memories
- offline options
  - configuration properties, 7-34
- offline\_flg flags, 6-12, 6-22
- on-chip memory. *See* memories. *See* memories
- OnNet, 3-6
- operator interfaces, 3-9. *See* human-machine interfaces (HMIs)
- optimized device interface files, 9-5
- optional configuration properties, 6-9
- optional network variables, 6-6
- optional networkVariables, 6-7
- options files
  - defined, 4-14
- output files
  - cleaning, 9-5
- overrides, 7-19, 7-31
  - configuration properties, 7-34
- PCC-10 Network Interfaces, 2-11
- PCLTA-10 Network Interfaces, 2-11
- PCLTA-20 Network Interfaces, 2-11
- peeking memory, 10-11
- ping interval, 9-18
- platforms, 5-23
- plug-in wizard, 2-8
- plug-in wizard
  - introduction, 3-9
  - starting, 5-3, 13-2
  - starting tutorial, 2-42
  - testing a plug-in, 2-45
- plug-ins
  - configuring devices, 1-6
  - creating, 3-9, 13-2
  - defined, 13-2
  - deregistering, 13-4
  - documentation, 2-3
  - installing, 15-4
  - installing the plug-in wizard, 2-12
  - project manager, 2-6
  - registering, 13-3
  - registration, 2-45, 5-9
  - testing tutorial, 2-45
  - tutorial, 2-42
- pmk command
  - examples, C-4
  - invoking, C-2
- poking memory, 10-11
- polled, 6-18

- polled network variables, 7-31
- power line couplers, 2-4
- prefixes
  - maintaining backward compatibility, B-2
- principal network variables, 7-28, 7-30
- program ID calculator, 5-13
  - starting, 5-13
- program ID templates, 7-2
  - locked, 7-13
  - setting, 7-13
- program IDs
  - automatic program ID management, 2-24
  - calculating, 5-12
  - changeable interfaces, 5-16
  - changing, 9-3
  - channel types, 1-4, 5-15
  - defined, 1-3, 3-6, 5-10
  - defining, 5-7
  - device classes, 1-4, 5-14
  - dynamic network variables, 5-16
  - format, 5-10
  - formats, 1-3, 5-16
  - guidelines, 1-5
  - managing, 5-8
  - manufacturer IDs, 1-3, 5-13
  - model numbers, 1-5, 5-8, 5-9, 5-15
  - setting, 5-10
  - target definitions, 1-9
  - tutorial, 2-21, 2-22
  - type, 5-10
  - updating, 5-15
  - usage, 1-4, 5-15
- programmable application image files, 9-3
- project files, 4-2, 4-12
  - default, 4-12
  - defined, 4-14
- project folders, 4-7
  - copying, 4-13
  - defined, 4-4
- project manager
  - opening projects, 4-10
- project manager, 2-7
  - creating projects, 4-5
  - defined, 3-6, 4-2
  - editing. *See* editor
  - introduction, 4-2
  - panes, 4-3
  - Project pane, 4-4
  - starting, 3-6, 4-5, 4-10
  - tutorial, 2-20
- Project pane
  - defined, 4-3, 4-4
  - Device Templates folder, 5-2
  - targets, 5-17
  - using, 4-4
- project settings, 4-4
- projects
  - creating, 4-5
  - opening, 4-10
  - selecting, 4-11
- projects
  - adding targets, 9-20
  - build script options, 9-9
  - build type options, 9-8
  - building. *See* application images, building
  - cleaning, 9-5
  - cleaning from the command line, C-4
  - converting NodeBuilder 1.5 projects, B-2, B-7
  - copying, 4-12
  - creating, 2-20, 3-6, 4-5
  - debug options, 9-9

- default, 4-7
- defined, 3-6, 4-2
- directory, 4-7
- excluding targets, 9-5
- folder, 4-7
- Load After Build, 9-3
- load after build options, 9-8
- load options, 9-8
- making from the command line, C-2
- moving, 4-12
- naming, 4-6
- setting build options, 9-7
- settings, 4-4, 4-7
- stop build options, 9-8
- testing, 11-2
- using source control, E-2
- using targets, 9-21
- verbose options, 9-9
- viewing status, 9-6
- protocols
  - defined, 1-2
- PSG/3 programmable serial gateway, 5-19
- PSG-20 programmable serial gateway, 5-19
- Quick-Start tutorial. See tutorial
- radix options
  - debugger, 10-13
- range overrides, 7-19
- range\_mode\_string, 6-36
- raw values
  - calculating, 7-21
- real formats, 7-44
- reboot options, 9-11
  - setting, 9-12
- recovery, 2-6
- reference information, 7-24
- reference types, 7-18, 7-19, 7-24, 7-30
- refresh, 7-10
- regional settings, 7-49
- registration, 2-15
- regular expressions
  - syntax, 8-5
- Release Target Device shapes, 4-9
- release target folders, 5-17
- release targets, 5-5, 5-12, 5-17. See targets
  - defined, 1-9
- removed items, 7-6
- removing resources, 7-50
- repeated service types, 6-17
- replacing, 8-3
- requirements, 2-12
- reset options
  - configuration properties, 7-33
- reset\_flg flags, 6-11, 6-22
- resource catalog
  - resource catalog file, 7-5
- resource catalogs
  - browsing, 7-8
  - defined, 7-7
  - refreshing, 7-10
  - viewing properties, 7-51
- resource editor, 2-7
- resource editor
  - defined, 7-5
  - options, 7-6
  - starting, 7-5
- resource file sets
  - active sets, 7-52
  - company information, 7-15
  - creating, 7-12
  - editing, 7-12
  - headers, 7-15
  - locations, 7-14

- naming, 7-14
- program ID template, 7-13
- scopes, 7-13
- selecting a scope, 7-12
- versions, 7-15, 7-52
- viewing properties, 7-52
- resource files
  - default scopes, 7-3
  - defined, 1-8, 3-4
  - distribution, 1-9
  - generating, 7-52
  - installing, 15-3
  - introduction, 7-2
  - manufacturer-defined, 1-9
  - paths, 6-16
  - program ID templates, 7-2
  - resource file sets, 7-2
  - scopes, 7-2
  - standard
    - defined, 1-9
  - target definitions, 1-9
  - updates, 1-9
  - using source control, E-4
  - viewing file properties, 7-51
- Resource Files, 7-14, 7-15
- resource folders
  - adding, 7-9
  - defined, 7-7
  - moving, 7-10
  - removing, 7-9
- Resource pane
  - tutorial, 2-26
- resource strings. *See* language strings
- resources
  - backward compatibility, 7-16
  - copying, 7-50
  - creating, 7-16
  - editing, 7-16
  - obsoleting, 7-51
  - removing, 7-50
  - searching, 7-10
- restriction flags, 6-11, 6-22
  - const\_flg flags, 6-12
  - device\_specific\_flg flags, 6-11
  - mfg\_flg flags, 6-11
  - obj\_disabl\_flg flags, 6-12
  - offline\_flg flags, 6-12
  - reset\_flg flags, 6-11
- Results pane
  - Debug Error Log tab, 10-12
  - Debug Log tab, 10-3
  - defined, 4-2, 4-3
  - Messages tab, 9-2
- resume
  - tutorial, 2-38
- ROM. *See* memories
- ROM application image files, 9-4, 9-11
- routers
  - defined, 1-3
- RQ\_DISABLED, 6-31
- RQ\_ENABLE, 6-31
- RQ\_REPORT\_MASK, 6-31
- RQ\_UPDATE\_STATUS, 6-31
- SCADA. *See* human-machine interfaces (HMIs)
- scalar details, 6-26
- scale factors
  - setting, 7-20
- scaling factors
  - examples, 7-48
  - formats, 7-48
- scheduler\_reset directive, 6-37
- scope 6, 5-9
- scopes

- defaults, 7-3
- defined, 7-2
- locked, 7-13
- network variables, 6-15
- selecting, 7-12
- setting, 7-13
- SCPTs, 3-4
  - defined, 1-9
- SCPTsndDelta
  - tutorial, 2-29
- Script, C-3
- seaching, 8-3
- Search Results tab
  - defined, 4-3
- sector size. *See* memories
- self-documentation
  - network variables, 6-18
- serial gateway library, 5-19
- serial numbers, 2-15
- service types, 6-16, 7-30
- SFPTopenLoopActuator
  - tutorial, 2-30
- SFPTopenLoopSensor
  - tutorial, 2-27
- shapes
  - creating, 12-2
  - creating complex shapes, 12-4
  - creating device shapes, 12-2
  - creating functional block shapes, 12-3
  - installing, 15-4
  - tutorial, 2-33
- ShortStack Developer's Kit, 2-4
- signature, 9-12
- signed char types, 7-19
- signed long types, 7-19
- signed quad types, 7-19
- signed short types, 7-19
- single stepping. *See* stepping
- Smart Transceiver models, 5-24
- SNVTs, 3-4
  - defined, 1-9
- software installation, 2-12
- software installations
  - creating, 15-2
  - files, 15-2
- software license agreements, F-2
- sorting, 7-6
- source code
  - editing while debugging, 10-13
  - generating, 6-27
  - generating tutorial, 2-25
- source control, E-2
- source files, 5-5
  - device templates, 1-8
  - inserting, 5-5
  - removing, 5-5
  - searching, 8-3
  - setting, 5-3
  - using source control, E-2
- SPID. *See* program IDs
- SPID Calculator, 5-12
- spidData.xml, 2-23
- Stanard Program ID Calcluator. *See* program IDs
- standard program ID calculator, 5-12
- Standard Program ID Calculator, 5-7
- standard resource file set
  - defined, 3-4
- static configuration properties, 6-12, 6-23
- status
  - viewing, 9-6
- stencils
  - creating, 3-9, 12-2

- installing, 15-4
- tutorial, 2-33
- using targets, 9-15
- stepping
  - tutorial, 2-38
- stepping
  - defined, 10-6
  - starting, 10-6
  - step into, 10-4, 10-6
  - step over, 10-4, 10-6
- string information, 7-31
- strings. *See* language strings
  - setting, 7-19
- Strings, 7-6
- structure fields, 6-25
- structure types, 7-19
- structures, 7-22
- SuiteLink protocol, 2-5, 14-3
- synchronous, 6-18
- syntax coloring, 8-9
- syntax highlighting, 8-2
- system images
  - name, 5-24
  - version, 5-24
- system requirements, 2-11
- Système Internationale units. *See* formats
- tab width, 8-8
- Target Device shapes, 4-9
- target folders, 5-17
- target platforms, 5-11
  - tutorial, 2-24
- targets, 5-5, 5-17
  - adding, 9-15, 9-20
  - browsing, 11-3
  - building, 5-17, 9-2, 9-5, 9-21
  - cleaning, 5-18
  - compiling, 5-18
  - debugging, 9-21. *See* debugger
  - debugging only development targets, 10-12
  - defined, 1-9, 5-12, 9-15
  - device template, 9-22
  - displaying status, 5-18
  - editing device settings, 9-22
  - excluding, 5-18, 9-5
  - forcing applicationless, 9-21
  - hardware templates, 5-22
  - inserting, 9-21
  - naming, 9-16
  - removing, 9-21
  - selecting target type, 9-20
  - setting options, 9-21
  - setting the hardware template, 5-17
  - settings, 5-17
  - specifying hardware templates, 5-12
  - target types, 9-23
  - testing, 11-2
  - viewing status, 9-22
- template files
  - using source control, E-3
- templates. *See also* hardware templates. *See* device templates
  - device templates
    - defined, 1-8
  - hardware templates
    - defined, 1-7
- temporary manufacturer IDs, 2-12
- temporary manufacturer IDs, 1-4, 5-14
- testing
  - tutorial, 2-39
- text device interface files, 9-5
- text formats, 7-44, 7-45
- text program IDs. *See* program IDs
- tick interval options



- debugger, 10-13
- time
  - localized formats, 7-49
- Toolbar
  - Debugger, 10-4
- toolbars
  - debugger, 10-3
- TP/FT-10F Control Modules, 2-9
  - Gizmo 4, 2-9
- transceiver type, 5-23
- transceiver types
  - default, 4-8
- transceivers
  - defined, 1-2
  - program IDs, 1-4
- translation
  - exporting, 7-41
  - language files, 7-40
  - side-by-side, 7-40
- Translator.h file, 6-32
- turnaround connections
  - defined, 2-39
- tutorials
  - creating a LonMaker network, 2-19
  - creating a NodeBuilder device template, 2-21
  - creating a Nodebuilder project, 2-20
  - goals, 2-17
  - introduction, 2-17
  - sections, 2-18
  - testing a NodeBuilder device, 2-39
- unacknowledged service types, 6-17
- union types, 7-19
- unions, 7-22
- United States units. *See* formats
- unsigned char types, 7-19
- unsigned long types, 7-19
- unsigned short types, 7-19
- unspecified service types, 6-17
- usage
  - program IDs, 1-4, 5-15
- versions
  - resource file sets, 7-52
- Visio drawings, 2-6
- Visual Basic
  - plug-ins, 2-42, 3-9
- Visual C++
  - plug-ins, 2-42, 3-9
- Watch List, 10-8
- Watch List pane. *See* watches
  - defined, 10-3
  - toolbar, 10-3
- watches
  - adding, 10-7
  - decimal, 10-8
  - defined, 10-7
  - deleting, 10-8
  - editing values, 10-8, 10-9
  - formatting, 10-8
  - hexadecimal, 10-8
  - radix, 10-8
  - removing, 10-8
  - selecting, 10-7
  - setting, 10-4, 10-5
  - tutorial, 2-38
  - Watch List pane, 10-3
- Wonderware, 2-5
- Wonderware InTouch, 3-10, 14-4
- write times. *See* memories
- XIF. *See* device interface files
- XML, 4-2
- XML, 4-14
- XML files
  - printing, 4-14

viewing, 4-14