

**IzoT™
NodeBuilder
User's Guide**

Develop hardware devices and software applications using Echelon's Series 6000, 5000, and 3100 chips and Smart Transceivers.

Echelon, LON, LonWorks, Neuron, 3120, 3150, Digital Home, i.LON, IzoT, FTXL, LonScanner, LonSupport, LNS, LonMaker, LONMARK, LonPoint, LonTalk, NodeBuilder, ShortStack, and the Echelon logo are trademarks of Echelon Corporation that may be registered in the United States and other countries.

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

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

Printed in the United States of America.
Copyright ©1997–2014 by Echelon Corporation.
Echelon Corporation
www.echelon.com

CONTENTS

Preface	ix
Purpose	x
Audience.....	x
Hardware Requirements.....	x
Content	x
Related Manuals.....	xi
For More Information and Technical Support.....	xii
1 Introduction	1
Introduction to the IzoT NodeBuilder Tool.....	2
New Features in the IzoT NodeBuilder Tool	2
LonTalk/IP Support	2
BACnet/IP Support.....	3
Series 6000 Chip Support.....	3
Transient Functions and Automatic Memory Maps	3
FT 6000 EVB Evaluation Board	4
Extended Address Table.....	4
Network Variables Up To 228 Bytes	4
Neuron C Version 2.3 Enhancements	5
What's Included with the IzoT FT 6000 EVK.....	6
IzoT NodeBuilder Development Tool.....	6
FT 6000 EVB Evaluation Boards	7
IzoT Commissioning Tool	8
IzoT Network Services Server	8
IzoT Router	8
IzoT Plug-in for Wireshark	8
Introduction to NodeBuilder Device Development and Network Integration ...	9
Channels	9
Routers.....	9
Applications.....	10
Program IDs	10
Network Variables.....	11
Configuration Properties	12
Functional Blocks	13
Functional Profiles.....	13
Hardware Templates.....	14
Neuron C.....	14
Device Templates	14
Device Interface Files.....	14
Resource Files	14
Targets	15
2 Installing the IzoT NodeBuilder Development Tool	17
Installing the IzoT FT 6000 EVK.....	18
Installing the IzoT NodeBuilder Software.....	18
3 IzoT NodeBuilder Quick-Start Exercise	25
IzoT NodeBuilder Quick-Start Exercise.....	26
Step 1: Creating an IzoT NodeBuilder Project.....	26
Step 2: Creating a NodeBuilder Device Template	29
Step 3: Defining the Device Interface and Creating its Neuron C	
Application Framework	33
Step 4: Developing the Device Application.....	38

FT 6000 Evaluation Boards	39
LTM-10A Platform and Gizmo 4 I/O Board	41
Step 5: Compiling, Building, and Downloading the Application	42
Step 6: Testing the Device Interface.....	46
Step 7: Debugging the Device Application	48
Step 8: Connecting and Testing the Device in a Network	54
Additional Device Development Steps.....	60
Creating an IzoT CT Stencil	60
Creating an IzoT Device Plug-in.....	64
Developing an HMI	65
Creating a Device Installation Application	65
Applying for LONMARK Certification	67
4 Creating and Opening IzoT NodeBuilder Projects	69
Introduction to the NodeBuilder Project Manager	70
Using the Project Pane	71
Creating a NodeBuilder Project.....	72
Creating a NodeBuilder Project from IzoT CT	73
Creating a NodeBuilder Project from the NodeBuilder Project Manager	73
Creating a NodeBuilder Project from the New Device Wizard.....	76
Opening a NodeBuilder Project.....	78
Opening a NodeBuilder Project from the IzoT Commissioning Tool	78
Opening a NodeBuilder Project from the NodeBuilder Project Manager	80
Copying NodeBuilder Projects.....	81
Using the IzoT Commissioning Tool to Backup and Restore a	
NodeBuilder Project	81
Manually Copying NodeBuilder Project Files.....	84
Copying NodeBuilder Device Templates.....	85
Copying User-Defined Resource Files	86
Viewing and Printing NodeBuilder XML Files	86
5 Creating and Using Device Templates	88
Introduction to Device Templates	89
Creating Device Templates	89
Starting the New Device Template Wizard	89
Specifying the Device Template Name.....	90
Specifying the Program ID	91
Specifying Target Platforms.....	96
Managing and Editing Device Templates.....	98
Managing Device Templates	98
Viewing and Editing Device Templates.....	99
Viewing Device Template Components	100
Managing Development and Release Targets	102
Setting Device Template Target Properties: Compiler	103
Setting Device Template Target Properties: Linker	106
Setting Device Template Target Properties: Exporter.....	107
Setting Device Template Target Properties: Configuration	109
Inserting a Library into a NodeBuilder Device Template	111
Using Hardware Templates	114
Creating Hardware Templates	115
Editing Hardware Templates.....	117
Setting Hardware Properties	117
Setting Memory Properties	120
5000 Series Chips.....	121
6000 Series Chips.....	122
3150 Neuron Core	122

3120 and 3170 Neuron Core	122
Setting the Hardware Template Description.....	122
6 Defining Device Interfaces and Creating their Neuron C Application Framework.....	124
Introduction to Device Interfaces.....	125
Starting the Code Wizard.....	125
Using the Resource Pane.....	126
Introduction to Resource File Sets.....	127
Introduction to Resources	128
Using the NodeBuilder Resource Editor	130
Using the Program Interface Pane	130
Defining the Device Interface.....	132
Adding Functional Blocks	135
Using Large Functional Block Arrays.....	138
Editing Mandatory Network Variables	138
Editing Mandatory Configuration Properties.....	145
Implementing Optional Network Variables	151
Implementing Optional Configuration Properties	153
Adding Implementation-specific Network Variables	155
Adding Implementation-specific Configuration Properties	158
Setting Initial Values for Network Variables and Configuration Properties.....	161
Setting Initial Values for Structured Data Types	162
Setting Initial Values for Enumerations.....	164
Setting Initial Values for Floating Point and s32 Data Types...	165
Using Changeable-Type Network Variables	166
Generating Code with the Code Wizard	167
Files Created by the Code Wizard	167
Using Code Wizard Templates.....	170
Version 3 Templates	170
Version 2 Templates	170
Version 1 Templates	171
Creating the Device Application	171
7 Developing Device Applications.....	173
Introduction to Neuron C	174
Unique Aspects of Neuron C	174
Neuron C Variables.....	176
Neuron C Variable Types	176
Neuron C Storage Classes.....	176
Variable Initialization.....	177
Neuron C Declarations	177
Introduction to Neuron C Code Editing.....	178
Modifying Neuron C Code Generated by the Code Wizard.....	179
Code Commands.....	179
Code Guidelines	180
Add I/O and Timer Declarations.....	180
Add when-tasks Responding to I/O and Timer Events	181
Add interrupt-tasks Responding to Interrupt Requests.....	181
Add Code to when(nv_update_occurs(<nv>)) when-task of Functional Blocks with Input NVs.....	181
Share Code with filexfer.nc when Handling Explicit Messages on a Device Implementing FTP	181
Ignore NCC#310 and NC#463 Compiler Warnings	181
Implementing Changeable-Type Network Variables	181

Neuron C Version 2 Features Not Supported by the Code Wizard	183
Message Tags	183
I/O Models	183
Network Variables	183
Configuration Properties	183
when() clauses	184
LONMARK Style	184
Director Functions	184
Interrupt Tasks	184
Using the NodeBuilder Editor	184
Using Syntax Highlighting	185
Searching Source Files	185
Searching a Single File for a String	185
Replacing Text	186
Searching Multiple Files for a String	186
Using Bookmarks	189
Setting Editor Options	189
8 Building and Downloading Device Applications	191
Introduction to Building and Downloading Applications	192
Building an Application Image	192
Excluding Targets from a Build	197
Cleaning Build Output Files	197
Viewing Build Status	198
Setting Build Options	200
Downloading an Application Image	201
Programming 5000 and 6000 Off-chip Memory	203
Programming 5000 and 6000 Series Chips In-Circuit	203
Programming 3150 Off-chip Memory	208
Programming 3150 On-chip Memory	209
Programming 3120 and 3170 On-chip Memory	210
Programming PL 3120 and PL 3170 Smart Transceiver Parameters	210
Upgrading Device Applications	211
Adding and Managing Target Devices	211
Adding a Target Device with the IzoT Commissioning Tool	211
Adding a Target Device with the NodeBuilder Project Manager	215
Managing Target Devices	217
Editing Target Device Settings	218
9 Testing a NodeBuilder Device Using the IzoT Commissioning Tool	221
Introduction to Testing NodeBuilder Devices	222
Monitoring and Controlling NodeBuilder Devices	222
Using the Data Point Shape	222
Using the LonMaker Browser	224
Connecting NodeBuilder Devices	227
10 Debugging a Neuron C Application	233
Introduction to Debugging	234
Starting the NodeBuilder Debugger	234
Using the Debugger Toolbar	236
Stopping an Application	237
Halting an Application	238
Running to the Cursor	238
Setting and Using Breakpoints	238
Stepping Through Applications	239

Debugging Interrupts for 5000 or 6000 Series chips	239
Using Statement Expansion.....	239
Using the Watch List Pane.....	239
Using the Call Stack Pane	243
Using the Debug Device Manager Pane.....	243
Peeking and Poking Memory	244
Executing Code in Development Targets Only	245
Using the Debug Error Log Tab	245
Setting Debugger Options.....	245
Appendix A Using the Command Line Project Make Facility	249
Using the NodeBuilder Command Line Project Make Facility.....	250
Appendix B Using Source Control With a NodeBuilder Project.....	253
Using Source Control with a NodeBuilder Project.....	254
Appendix C Glossary	257
Appendix D NodeBuilder Software License Agreement	271

Preface

The IzoT™ NodeBuilder® Development Tool is a complete hardware and software platform that is used to develop applications for Neuron® Chips and Echelon® Smart Transceivers. The IzoT NodeBuilder tool lets you create, debug, test, and maintain IzoT and LONWORKS® devices. It includes a suite of device development software that you can use to develop device applications, and hardware platforms that you can use to build and test prototype and production devices.

Purpose

This document describes how to use the IzoT NodeBuilder tool to develop IzoT and LONWORKS device applications and build and test prototype and production IzoT and LONWORKS devices.

Audience

This guide is intended for device and system designers with an understanding of control networks.

Hardware Requirements

Requirements for computers running the IzoT NodeBuilder tool are listed below:

- Microsoft® Windows. Windows 8 64-bit and 32-bit, or Windows 7 64-bit and 32-bit, or Windows XP 32-bit. Echelon recommends that you install the latest service pack available from Microsoft for your version of Windows.
- An Intel® Pentium® or compatible processor meeting the minimum Windows requirements for the selected version of Windows.
- 300 to 550 megabytes (MB) free hard-disk space, plus the minimum Windows requirements for the selected version of Windows.
 - The IzoT NodeBuilder tool requires 100 MB of free space.
 - The IzoT Commissioning Tool, which is required to install the IzoT NodeBuilder tool, requires 172 MB of free space.
 - Microsoft .NET Framework 3.5 SP1, which is required to run the IzoT NodeBuilder tool, requires 30 MB of free space. This is not included and must be downloaded separately.
- 512 MB RAM minimum.
- DVD-ROM drive.
- 1024x768 or higher-resolution display with at least 256 colors.
- Mouse or compatible pointing device.

Content

This guide includes the following content:

- *Introduction.* Lists the new features in the IzoT NodeBuilder tool, summarizes the components included with the IzoT NodeBuilder tool, and provides an overview of IzoT NodeBuilder device development and network integration.
- *Installing the IzoT NodeBuilder Development Tool.* Describes how to get started with your IzoT NodeBuilder tool, including how to install the IzoT NodeBuilder software and connect the IzoT FT 6000 EVK hardware.
- *IzoT NodeBuilder Quick-Start Exercise.* Demonstrates how to create an IzoT or LONWORKS device using the IzoT NodeBuilder tool.
- *Creating and Opening NodeBuilder Projects.* Describes how to create, open, and copy NodeBuilder projects, and explains how to copy NodeBuilder projects and NodeBuilder device templates to another computer.
- *Creating and Using Device Templates.* Describes how to use the New Device Template wizard in the NodeBuilder Project Manager to create, manage, and edit NodeBuilder device templates.

Explains how to manage development and release targets and insert libraries into a device template. Describes how to use the Hardware Template Editor to create and edit hardware templates.

- *Defining Device Interfaces and Creating their Neuron C Application Framework.* Describes how to use the NodeBuilder Code Wizard to define your device interface and generate Neuron C code that implements it. Explains how to start the NodeBuilder Code Wizard, how to add functional blocks, network variables, and configuration properties to your device template, and how to create the Neuron C framework for your device interface.
- *Developing Device Applications.* Provides an overview of the Neuron C Version 2.3 programming language. Describes how to edit the Neuron C source code generated by the NodeBuilder Code Wizard to implement your device functionality. Explains how to use the NodeBuilder Editor to edit, search, and bookmark Neuron C code.
- *Building and Downloading Device Applications.* Describes how to compile Neuron C source code, build an application image, and download the application image to a device. Explains how to add target devices to a NodeBuilder project and how to manage them.
- *Testing a NodeBuilder Device Using the IzoT Commissioning Tool.* Describes how to use the Data Point shape and IzoT Browser in the IzoT Commissioning Tool to monitor and control your device. It explains how to use the IzoT Commissioning Tool to connect your IzoT NodeBuilder device to other IzoT or LONWORKS devices in a network.
- *Debugging a Neuron C Application.* Describes how the use the NodeBuilder debugger to troubleshoot your Neuron C application.
- *Appendices.* Provides information for using the command line project make facility and managing an IzoTNodeBuilder project using a source control application. Also includes a glossary with definitions for many terms commonly used with an IzoT NodeBuilder device development.

Note: Screenshots in this document were taken during the development of the IzoT NodeBuilder tool; therefore, some images may vary slightly from the release version of the user interface.

Related Manuals

The documentation related to the IzoT NodeBuilder tool is provided as Adobe® PDF files and online help files. The PDF files are installed in the **Echelon NodeBuilder** program folder when you install the IzoT NodeBuilder tool. You can download the latest NodeBuilder documentation, including the latest version of this guide, from Echelon's Web site at www.echelon.com/docs.

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

FT 6000 EVB Examples Guide
(078-0505-01)

Describes how to run the Neuron C example applications included with the FT 6000 EVK on an FT 6000 EVB.

FT 6000 EVB Hardware Guide
(078-0504-01)

Describes how to connect the FT 6000 EVBs, and describes the Neuron core, I/O devices, service pin and reset buttons and LEDs, and jumper settings on the FT 6000 EVB hardware. One or two FT 6000 EVBs are included with the IzoT FT 6000 EVK.

Introduction to the LONWORKS Platform
(078-0183-01B)

Provides a high-level introduction to LONWORKS networks and the tools and components that are used for developing, installing, operating, and maintaining them.

OpenLNS Plug-in Programmer's Guide (078-0393-01A)

Describes how to write plug-ins using .NET programming languages such as C# and Visual Basic .NET

<i>IzoT Commissioning Tool User's Guide (078-0514-01)</i>	Describes how to use the IzoT Commissioning Tool to design, commission, modify, and maintain LONWORKS networks.
<i>LONMARK SNVT and SCPT Guide</i>	Documents the standard network variable types (SNVTs), standard configuration property types (SCPTs), and standard enumeration types that you can declare in your applications.
<i>IzoT Plug-in for WireShark Guide (078-9511-01A)</i>	Describes how to use the IzoT Plug-in for WireShark. This is the packet analyzer to monitor, analyze, and troubleshoot network protocol problems
<i>Neuron C Programmer's Guide (078-0002-02I)</i>	Describes how to write programs using the Neuron [®] C Version 2.3 language.
<i>Neuron C Reference Guide (078-0140-02G)</i>	Provides reference information for writing programs using the Neuron C language.
<i>Neuron Tools Error Guide (078-0402-01D)</i>	Provides reference information for Neuron tool errors.
<i>NodeBuilder Resource Editor User's Guide (078-0194-01C)</i>	Describes how to use the NodeBuilder Resource Editor to create and edit resource file sets and resources such as functional profile templates, network variable types, and configuration property types.

For More Information and Technical Support

The **NodeBuilder ReadMe** document provides descriptions of known problems, if any, and their workarounds. To view the **NodeBuilder ReadMe**, click **Start**, point to **Programs**, point to **NodeBuilder**, and then select **NodeBuilder ReadMe First**. You can also find additional information about the IzoT NodeBuilder tool at the NodeBuilder Web page at www.echelon.com/nodebuilder.

If you have technical questions that are not answered by this document, the NodeBuilder online help, or the NodeBuilder ReadMe file, you can contact technical support. You can get free e-mail support by sending your support questions to lonsupport@echelon.com. To receive priority technical support from Echelon, you can purchase support services from Echelon or an Echelon support partner. See www.echelon.com/support for more information on Echelon support and training services.

You can also enroll in training classes at Echelon or an Echelon training center to learn more about developing devices. You can find additional information about device development training at www.echelon.com/training.

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

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

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

Introduction

This chapter introduces the IzoT NodeBuilder Development Tool. It lists the new features in the IzoT NodeBuilder tool, summarizes the components included with the IzoTNodeBuilder tool, and provides an overview of NodeBuilder device development and network integration.

Introduction to the IzoT NodeBuilder Tool

The IzoT NodeBuilder Development Tool is a complete hardware and software platform for developing, debugging, testing, and maintaining IzoT and LONWORKS devices based on the Neuron 6000 Processor and FT 6000 Smart Transceiver and all previous-generation Series 5000 and Series 3100 chips. You can use the IzoT NodeBuilder tool to create many types of devices, including VAV controllers, thermostats, washing machines, card-access readers, refrigerators, lighting ballasts, blinds, and pumps. You can use these devices in a variety of systems including building and lighting controls, factory automation, energy management, and transportation.

You can use the IzoT NodeBuilder tool to do the following:

- View standard resource file definitions for standard network variable types (SNVTs), standard configuration property (SCPTs), and standard functional profile templates (SFPTs).
- Create your own resource files with user-defined network variable types (UNVTs), user-defined configuration property (UCPTs), and user-defined functional profile templates (UFPTs).
- Automatically generate Neuron C code that implements your device's interface and provides the framework for your device application.
- Edit your Neuron C code to implement your device's functionality.
- Compile, build, and download your application to a development platform or to your own devices.
- Test with prototype I/O hardware on either the FT 6000 EVB Evaluation Boards included with the FT 6000 EVK and available separately, or LTM-10A Platform and Gizmo 4 I/O Board included with the NodeBuilder FX/PL tool and available separately, or use your own custom device to build and test your own I/O hardware.
- Install your device into an IzoT or LONWORKS network and test how your device interoperates with other IzoT and LONWORKS devices.

New Features in the IzoT NodeBuilder Tool

The IzoT NodeBuilder tool includes support for Echelon's new Series 6000 chips (the term used to collectively refer to the FT 6050 and FT 6010 Smart Transceivers and the Neuron 6050 Processor), support for Echelon's new FT 6000 EVB, support for the previous generation Series 5000 and Series 3100 chips, and the following key features:

- Support for the IzoT LonTalk/IP and BACnet/IP protocols
- Support for transient functions and automatically tuned memory maps
- Extended address table support with support for up to 254 address table entries
- Support for network variables up to 228 bytes (exceeding the previous limit of 31 bytes)
- Neuron C Version 2.3 Enhancements

The following sections describe these new features.

LonTalk/IP Support

The Series 6000 chips and firmware add support for the LonTalk/IP protocol. The LonTalk/IP protocol is the control network protocol that implements IzoT Control Services. IzoT Control Services are based on the LONMARK Layer 7 protocol, the ISO/IEC 14908-1 Layers 3 to 6 protocols, with native IP addressing at Layer 3, and link-specific protocols for Layers 1 and 2. The link-specific protocols may implement compression of the Layer 3 packets depending on the requirements of the underlying links. There are two modes for LonTalk/IP called Compatibility Mode and Enhanced Mode. In Compatibility Mode LonTalk/IP devices can communicate with LonTalk devices directly without the need for a router as long as they are on the same LonTalk-compatible channel. LonTalk/IP

devices in Compatibility Mode can also communicate with LonTalk devices on different channels as long as there is a route created between the channels with one or more IzoT routers. In Enhanced Mode, LonTalk/IP devices cannot communicate with LonTalk devices, even with the use of IzoT routers.

You can configure the IzoT Router included with the FT 6000 EVK to operate in either Enhanced Mode or Compatibility Mode. If you are developing devices that will potentially be used with devices based on the Series 5000 or Series 3100 processors, select Compatibility Mode. If you are developing devices that will exclusively be used with devices based on Series 6000 processors or other IzoT-compatible devices such as devices based on the IzoT Device Stack EX, select Enhanced Mode.

BACnet/IP Support

The Series 6000 chips and firmware add support for the BACnet/IP protocol. The BACnet/IP protocol is the protocol that implements the BACnet services defined by ASHRAE and specified in ISO 16484-5, with native IP addressing at Layer 3, and link-specific protocols for Layers 1 and 2. The link-specific protocols may implement compression of the Layer 3 packets depending on the requirements of the underlying links.

Series 6000 Chip Support

The IzoT NodeBuilder software adds development support for the Series 6000 chips. Development for Series 5000 and Series 3100 chips is also supported. An *FT 6000 Evaluation Board* hardware template file (.NbHwt extension) is included, matching the FT 6000 EVB hardware.

See the *FT 6000 EVK Hardware Guide* for instructions to emulate the FT 6010 hardware with the FT 6050 Evaluation board. The *FT 6010 Evaluation Board* hardware template required for this emulation is included.

An *FT 6050* hardware template is included as an example of a typical hardware template for a generic FT 6050 device.

The Neuron C compiler and its companion tools automatically take advantage of the Series 6000 chips' features. For example, the Neuron C compiler automatically takes advantage of the extended address table where available, and the dynamic host configuration protocol (DHCP) is automatically enabled where supported.

Transient Functions and Automatic Memory Maps

The IzoT NodeBuilder software adds support for *transient* functions for applications targeting a Series 6000 chip. A transient function is loaded from serial flash memory on demand and automatically managed by the Neuron firmware, allowing the application's total code size to exceed the available physical memory. Functions declared in Neuron C 2.3 compile to *transient* functions by default, while *when-tasks* and *interrupt-tasks* always compile into resident code.

Individual functions can be made resident with the new Neuron C `__resident` keyword (note the leading *double* underscore characters):

```
unsigned __resident add(unsigned a, unsigned b) {  
    return a+b;  
}
```

See the *Neuron C Programmer's Guide* for more details about resident and transient functions.

To ease device development and ensure suitable space for transient functions, the Neuron C 2.3 tools automatically size the memory map to suit the requirements of the Neuron firmware, the Series 6000 chips and your application combined. For these chips, it is no longer necessary to estimate the amount of volatile memory ("RAM"), persistent data storage ("EEPROM") or space needed for constant data and code.

FT 6000 EVB Evaluation Board

The FT 6000 EVB is a complete 6000 Series IzoT and LONWORKS device that you can use to create IzoT and LONWORKS devices. The FT 6000 EVB includes a FT 6050 Smart Transceiver with an external 10 MHz crystal (you can adjust the system's internal clock speed from 5MHz to 80MHz), an FT-X3 communication transformer, 64KB external serial EEPROM and flash memory devices, and a 3.3V power source. The FT 6000 EVB features a compact design that includes the following I/O devices that you can use to develop prototype and production devices and test the FT 6000 EVB example applications:

- 4 x 20 character LCD
- 4-way joystick with center push button
- 2 push-button inputs
- 2 LED outputs
- Light-level sensor
- Temperature sensor

The FT 6000 EVB Evaluation Board also includes EIA-232/TIA-232 (formerly RS-232) and USB interfaces that you can use to connect the board to your development computer and perform application-level debugging.

Each FT 6000 EVB also features a flash in-circuit programmer header that supports the SPI interface for fast downloads when programming the external non-volatile memory of the FT 6050 Smart Transceiver on the board.

For more information on the FT 6000 EVB hardware, including detailed descriptions of its Neuron core, I/O devices, service pin and reset buttons and LEDs, and jumper settings, see the *FT 6000 EVB Hardware Guide*.

Extended Address Table

Series 6000 chips can support up to 254 address table records, subject to available system resources (for example, RAM and EEPROM) and application requirements. The Series 5000 and Series 3100 chips are limited to a maximum of 15 address table entries.

Network Variables Up To 228 Bytes

Series 6000 chip support network variables up to 228 bytes in size. The Series 5000 and Series 3100 chips are limited to a maximum network variable size of 31 bytes.

Creating new applications or adding larger than 31 bytes network variables to existing applications is completely transparent except when larger than 31 bytes network variables are added to existing applications which also implement changeable-type applications.

Applications implementing changeable-type network variables and larger than 31-bytes network variables must return the current size of changeable-type network variables from the application-specific implementation of the *get_nv_length_override()* callback. In previous releases, the *get_nv_length_override()* callback would return a constant 0xFF by default. Applications which support larger than 31 bytes network variables must default the callback result to the value obtained from the *get_declared_nv_length()* API:

```
unsigned _RESIDENT get_nv_length_override(unsigned nvIndex)
{
    #if defined(_SUPPORT_LARGE_NV)
        unsigned uResult = get_declared_nv_length(nvIndex);
    #else
        unsigned uResult = 0xFF;
    #endif
}
```

```

// TO DO: add code to return the current length of the network variable
// with index "nvIndex."
// Example code follows:
//
// switch (nvIndex) {
//   case nviChangeableNv::global_index:
//     if (nviChangeableNv::cpNvType.type_category != NVT_CAT_INITIAL
//         && nviChangeableNv::cpNvType.type_category != NVT_CAT_NUL) {
//       uResult = nviChangeableNv::cpNvType.type_length;
//     }
//     break;
// } // switch

return uResult;
}

```

See *Implementing Changeable-Type Network Variables* in chapter 4 of this guide and *Changeable-Type Network Variables* in the *Neuron C Programmer's Guide* for more detail.

Neuron C Version 2.3 Enhancements

The new features in the Neuron C Version 2.3 programming language include support for transient and resident functions, automatic memory maps, a new preprocessor, support for initialization of automatic variables implementing scalar types, and new or enhanced compiler directives and other language enhancements. These new features are detailed in the *Neuron C Programmer's Guide* and *Neuron C Reference Guide*.

Support for transient functions and automatic memory maps is detailed earlier in this section.

The new preprocessor, based on the open source MCPP implementation by Kioshi Matsui, provides previously unsupported directives such as **#if** and **#elif**.

Automatic variables implementing scalar types can now be initialized. For example:

```

unsigned long add(unsigned num, unsigned *values) {
    unsigned long result = 0;
    while (num--) {
        result += *values++;
    }
    return result;
}

```

Automatic variables which implement aggregate types such as structures, unions and arrays cannot be initialized in this manner. To initialize such a variable, declare the variable and provide its initial data in two distinct expressions.

Enhanced compiler directives include the **pragma addresses** and **pragma aliases** directives (also known as **pragma num_addr_table_entries** and **pragma num_alias_table_entries**, respectively). These directives are enhanced with a more user-friendly new name, and automatic allocation of the corresponding resource based on the compiler's inspection of your application. The directives can be used to override this allocation.

The **pragma num_domains** directive is also supported with a user-friendly alias, **pragma domains**.

New directives are supplied to control new features (**pragma dhcp**, **pragma enhanced_mode**, **pragma resident**) and to control placement of certain portions of your application within the available space (**pragma locate**).

In addition, the new **__type_index** and **__type_scope** built-in properties are supported for network variables, and yield a numeric constant for the type index and type scope number, respectively. These properties begin with a double underscore character to avoid conflicts with existing application code. For example, a **SNVT_switch** typed network variable reports 95 for the type index, and a scope of zero.

What's Included with the IzoT FT 6000 EVK

The FT 6000 EVK includes the following components:

- **IzoT NodeBuilder Development Tool.** The IzoT NodeBuilder Development Tool is an integrated development environment (IDE) for developing applications for Series 6000, 5000, and 3100 chips. It is available as a free download that requires a serial number to be installed. A serial number for the IzoT NodeBuilder software is included on the back of the IzoT Commissioning Tool EVK Edition DVD case that is included with the FT 6000 EVK.
- **Two FT 6000 EVB Evaluation Boards.** The FT 6000 EVBs are evaluation boards that you can use to run example applications and to prototype and debug your own applications. Each includes an FT 6050 Smart Transceiver and can be attached to an IzoT or LONWORKS free topology (FT) twisted pair channel.
- **IzoT Commissioning Tool EVK Edition DVD.** Contains the software tool for designing, installing, and maintaining IzoT and LONWORKS control networks. The EVK Edition includes the Microsoft Visio 2010 diagramming and vector graphics edition application.
- **IzoT Network Services Server CD.** Contains the network operating system that is the standard network management software platform for commercial and industrial IzoT and LONWORKS networks.
- **IzoT Router.** A ready-to-run application for connecting IzoT devices on an Ethernet channel with IzoT and LonWorks devices on an FT channel. The router automatically forwards packets between the Ethernet and FT channels.
- **IzoT Plug-in for Wireshark.** Wireshark is a free and open-source packet analyzer for IP networks. It is used for network troubleshooting and analysis. A plug-in for Wireshark is included with the IzoT NodeBuilder software that enabled Wireshark to decode LonTalk/IP packets.
- **Quick Start Guide.** This document describes how to install the software included with your FT 6000 EVK; connect the FT 6000 EVBs and your development computer to an FT-10 channel; and create a simple network using the example application pre loaded on the FT 6000 EVB.
- FT 6050 Smart Transceiver sample chips.
- Accessories including power supplies and cables.

The following sections describe each of the components.

IzoT NodeBuilder Development Tool

The IzoT NodeBuilder Development Tool is an integrated development environment (IDE) for developing and debugging applications for Series 6000, 5000, and 3100 chips. It is available as a free download that requires a serial number to be installed. A serial number for the IzoT NodeBuilder software is included on the back of the IzoT Commissioning Tool EVK Edition DVD case that is included with the FT 6000 EVK. It includes Neuron C example applications that you can run on your FT 6000 EVBs and use to further learn how to develop your own device applications.

The IzoT NodeBuilder software includes the following components:

- **NodeBuilder Resource Editor.** View standard types and functional profiles, and create user-defined types and profiles if the standard resource files do not include the resources you need.
- **NodeBuilder Code Wizard.** Use a drag-and-drop interface to create your device's interface and then automatically generate Neuron C source code that implements the device interface and creates the framework for your device application.

- *NodeBuilder Editor*. Edit the Neuron C source code generated by the Code Wizard to create your device's application, or create and edit your own Neuron C code.
- *NodeBuilder Debugger*. Debug your application with a source-level view of your application code as it executes.
- *NodeBuilder Project Manager*. Build and download your application image to your development platform or to your own device hardware.

The IzoT NodeBuilder software include three Neuron C example applications that you can run on your FT 6000 EVBs. You can use these examples to test the I/O devices on the FT 6000 EVB, and create simple IzoT and LONWORKS networks. You can view the Neuron C code used in the example applications, and then create a new device application by modifying the existing example applications or by developing the device application from scratch.

For more information on using the FT 6000 EVB example applications, see the *FT 6000 EVB Examples Guide*.

FT 6000 EVB Evaluation Boards

The IzoT FT 6000 EVK includes two FT 6000 EVBs. Each EVB is a complete Series 6000 IzoT and LONWORKS device that you can use to evaluate the IzoT and LONWORKS platforms and create IzoT and LONWORKS devices. The FT 6000 EVB includes an FT 6050 Smart Transceiver with an external 10 MHz crystal (you can adjust the system's internal clock speed from 5MHz to 80MHz), an FT-X3 communication transformer, 512KB external serial flash memory devices, and a 3.3V power source. The FT 6000 EVB features a compact design that includes the following I/O devices that you can use to develop prototype and production devices and test the FT 6000 EVB example applications:

- 4 x 20 character LCD
- 4-way joystick with center push button
- 2 push-button inputs
- 2 LED outputs
- Light-level sensor
- Temperature sensor



The FT 6000 EVB Evaluation Board also includes an EIA-232/TIA-232 (formerly RS-232) and USB interfaces that you can use to connect the board to your development computer and perform application-level debugging.

Each FT 6000 EVB also features a flash ICE header that supports the SPI and I²C interfaces for fast downloads when programming the external non-volatile memory of the FT 6000 Smart Transceiver on the board.

For more information on the FT 6000 EVB hardware, including detailed descriptions of its Neuron core, I/O devices, service pin and reset buttons and LEDs, and jumper settings, see the *FT 6000 EVB Hardware Guide*.

IzoT Commissioning Tool

The IzoT Commissioning Tool is a software tool that you can use to install, connect, configure, test, and update devices that you develop with the IzoT NodeBuilder software. It is a software package for designing, installing, and maintaining IzoT and LONWORKS control networks. Based on Echelon's IzoT Network Services Server, the IzoT Commissioning Tool combines a powerful network services platform with an easy-to-use Visio user interface. The IzoT Commissioning tool is compatible with a number of OpenLNS and LNS plug-ins, simplifying network installation and integration for devices with plug-in support.

The IzoT Commissioning Tool can be used to manage all phases of a network's life cycle, from the initial design and commissioning to the ongoing operation, because it provides the functionality of several network tools in one single solution:

- *Network Design Tool.* You can design a network onsite or offsite (either connected to the network over the Internet or not connected to it all), and then modify it anytime. The IzoT Commissioning Tool can also learn an existing network's design through a process called *network recovery*.
- *Network Installation Tool.* You can rapidly install a network designed offsite once it is brought onsite. The device definitions can be quickly and easily associated with their corresponding physical devices to reduce on-site commissioning time. The IzoT Browser provides complete access to all network variables and configuration properties.
- *Network Documentation Tool.* You can create an IzoT CT drawing during the network design and installation process. This drawing is an accurate, logical representation of the installed physical network. The drawing is therefore an essential component of as-built reports.
- *Network Operation Tool.* You can operate the network using the operator interface pages contained within the IzoT CT drawing.
- *Network Maintenance Tool.* You can easily add, test, remove, modify, or replace devices, routers, channels, subsystems, and connections to maintain the network.

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

IzoT Network Services Server

The IzoT Network Service Server is a network operating system that is the standard network management software platform for commercial and industrial IzoT and LONWORKS networks.

IzoT Router

The IzoT Router is a ready-to-run application for connecting IzoT devices on an Ethernet channel with IzoT and LonWorks devices on an FT channel. The router automatically forwards packets between the Ethernet and FT channels.

IzoT Plug-in for Wireshark

Wireshark is a free and open-source packet analyzer for IP networks. It is used for network troubleshooting and analysis. A plug-in for Wireshark is included with the IzoT NodeBuilder software that enabled Wireshark to decode LonTalk/IP packets. See the *IzoT Plug-in for Wireshark Guide* for details on how to download Wireshark, install the IzoT plug-in for Wireshark, and use Wireshark with LonTalk/IP networks.

Introduction to NodeBuilder Device Development and Network Integration

An IzoT or 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. You can use the IzoT NodeBuilder tool to develop, test, and debug Neuron C applications for Neuron hosted devices.

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 on the host processor. The IzoT NodeBuilder tool supports some of the common tasks occurring in the creation of host-based devices; however, an additional host-based device development tool is required.

Each device includes one or more processors that implement the LonTalk/IP or the ISO/IEC 14908-1 Control Network Protocol (CNP). Each device also includes a component called a *transceiver* to provide its interface to the communications channel.

A device publishes and consumes 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*, which may be referred to simply as 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. The CNP is an open protocol defined by the ISO/IEC 14908-1 standard (defined nationally in the United States, Europe, and China by the ANSI/EIA 709.1, EN 14908, and GB/Z 20177 standards, respectively). LonTalk/IP is based on the ISO/IEC 14908-1 standard, with extensions to use IP as the transport protocol.

Channels

A *channel* is the physical media between devices upon which the devices communicate. The LonTalk/IP and CNP protocols are media independent; therefore, numerous types of media can be used for channels: twisted pair, power line, fiber optics, IP, and radio frequency (RF) to name a few. Channels are categorized into *channel types*, and the channel types are characterized by the device transceiver. Common channel types include TP/FT-10 (ISO/IEC 14908-2 twisted pair free topology channel) which is also called FT, TP/XF-1250 (high-speed twisted pair channel), PL-20 (ISO/IEC 14908-3 power line channel), FO-20 (ANSI/CEA-709.4 fiber optics channel), and IP-852 (ISO/IEC 14908-4 IP-communication).

Different transceivers may be able to interoperate on the same channel; therefore, each transceiver type specifies the channel type or types that it supports. The choice of channel type affects transmission speed and distance as well as the network topology.

The IzoT NodeBuilder tool, IzoT Commissioning Tool, and Neuron Chips support all standard channel types, but not all Neuron Chips support all transceiver and channel types. Smart Transceivers combine the transceiver and Neuron core in the same chip, and therefore support the channel types supported by the integrated transceiver.

Routers

Multiple channels can be connected using *routers*. Routers are used to manage network message 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 at least two channels.

The IzoT Router can be configured to act as a DHCP relay for address allocation on the LonTalk/IP network. DHCP relay can be used to have a single server perform all of the allocation of addresses in a network, and then have the server's allocations forwarded onto another subnet.

To enable the DHCP relay functionality, go to the DHCP page on configuration Web page of the IzoT Router. Once there, select the **Relay** option and a box to enter the address of the DHCP server appears. Enter the IP address of your server and click the **Save Configuration** button. DHCP requests are now forwarded from the LonTalk/IP network to the DHCP server, and responses from the server will be forwarded to devices on the LonTalk/IP network.

The DHCP server must have a separate subnet set up to allocate addresses from the LonTalk/IP subnet. This is because the IzoT Router treats the IP and LonTalk/IP subnets as separate subnets, and uses the subnets to determine which subnet to send messages on. The subnet on the FT side needs to be different than the subnet on the Ethernet side. If both the LonTalk/IP and IP subnets share the same address range, the router cannot determine where to send the messages.

Applications

Every IzoT and 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 LonTalk/IP and LONWORKS channels from other devices, as well as information from the device hardware (for example, the temperature from a temperature sensing device). The outputs from a device can include information sent on LonTalk/IP and LONWORKS channels to other devices, as well as commands sent to the device hardware (for example, a fan, light, heater, or actuator). You can use the IzoT NodeBuilder tool to write a device's Neuron C application.

Program IDs

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

Field	Description
Format (F)	<p>A 1 hex-digit value defining the structure of the program ID. The upper bit of the format defines the program ID as a <i>standard program ID (SPID)</i> or a <i>text program ID</i>. The upper bit is set for standard program IDs, so formats 8–15 (0x8–0xF) are reserved for standard program IDs.</p> <ul style="list-style-type: none">• 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, cannot be used for new devices.

The IzoT NodeBuilder tool can be used to create applications with program ID format 8 or 9.

Manufacturer ID (M)	<p>A 5 hex-digit ID that is unique to each LONWORKS device manufacturer. The upper bit identifies the manufacturer ID as a <i>standard manufacturer ID</i> (upper bit clear) or a <i>temporary manufacturer ID</i> (upper bit set).</p>
----------------------------	---

Field	Description
	<ul style="list-style-type: none"> Standard manufacturer IDs are assigned to manufacturers when they join LONMARK International, and are also published by LONMARK International so that the device manufacturer of a LONMARK certified device is easily identified. Standard manufacturer IDs are never reused or reassigned. If your company is a LONMARK member, but you do not know your manufacturer ID, you can find your ID in the list of manufacturer IDs at www.lonmark.org/spid. The most current list at the time of release of the NodeBuilder tool is also included with the IzoT NodeBuilder software. Temporary manufacturer IDs are available at no charge to anyone on request by filling out a simple form at www.lonmark.org/mid.
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 International 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 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)	<p>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 your published model number.</p> <p>See the <i>LonMark Application Layer Interoperability Guidelines</i> for more information about program IDs.</p>

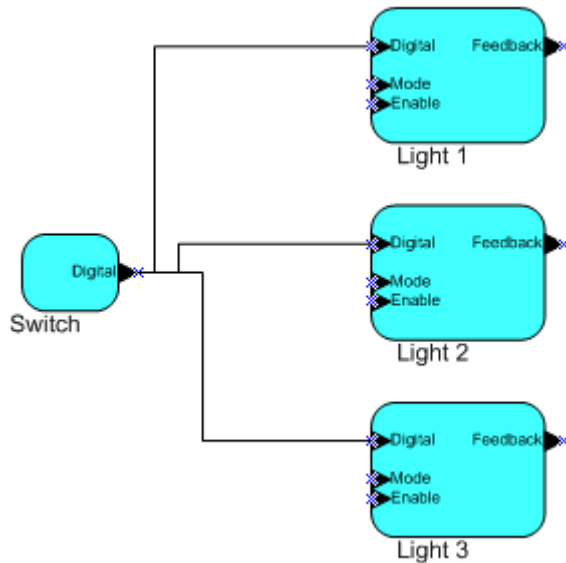
Network Variables

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 management tool such as the IzoT Commissioning Tool could be used to connect these two devices, allowing the switch to control the lighting device, as shown in the following figure:



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 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 assigns the new value to the output network variable.

Through a process called *binding* that takes place during network design and installation, the device is configured to know the logical address of the other device or group of devices in the network expecting that network variable's values. The device's embedded firmware assembles and sends the appropriate message(s) to these destinations. Similarly, when the device receives an updated value for an input network variable required by its application program, its firmware 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.

The NodeBuilder Code Wizard automatically generates the required network variable declarations for your device's interface in your device's Neuron C application. Typically, you don't need implement any code in the device application to handle the binding process, or the source or destination devices for network variable values. Neuron C provides an easy-to-use programming model familiar to any C language programmer that encapsulates the complexity of distributed applications.

Configuration Properties

IZoT AND LONWORKS applications may also contain *configuration properties*. Configuration properties allow the device's behavior to be customized using a network management tool such as the

IzoT Commissioning Tool or a customized plug-in created for the device (see the *OpenLNS Plug-in Programmer's Guide* for more information on creating OpenLNS device plug-ins).

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. Another example of a configuration property is a heartbeat that determines how often a device transmits network variable updates over the network.

Like network variables, configuration properties have types that determine the type and format of the data they contain.

The NodeBuilder Code Wizard automatically generates the required configuration property declarations for your device's interface and most of the required infrastructure code in your device's Neuron C application. The IzoT NodeBuilder tool supports configuration properties with an easy-to-use programming model in Neuron C.

Functional Blocks

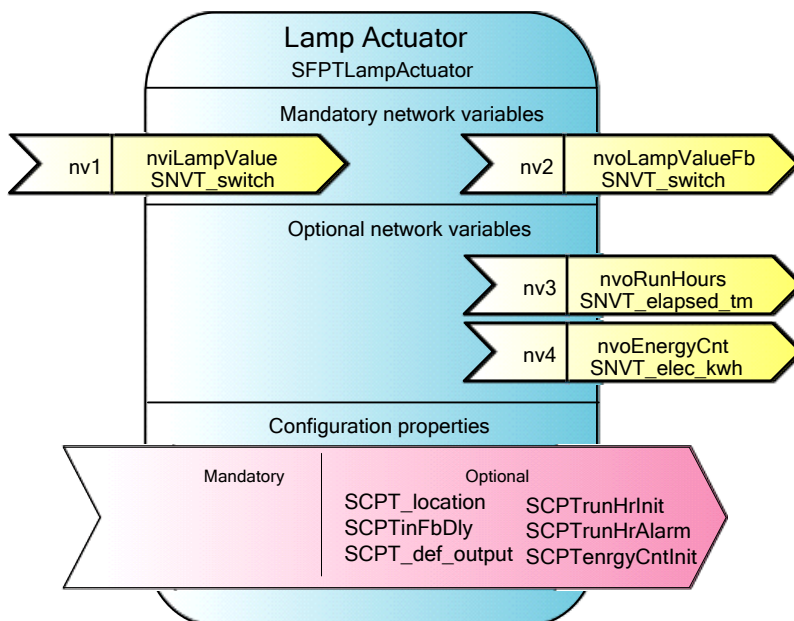
Applications in devices are divided into one or more *functional blocks*. A functional block is a collection of network variables and configuration properties, which are used together to perform one task. These network variables and configuration properties are called the *functional block members*. For example, a digital input device could have 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 device.

The NodeBuilder Code Wizard automatically generates the required functional block declarations for your device's interface in your device's Neuron C application.

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, the standard functional profile for a light actuator has mandatory **SNVT_switch** input and output network variables, optional **SNVT_elapsed_tm** and **SNVT_elec_kwh** output network variables, and a number of optional configuration properties. The following diagram illustrates the components of the standard light actuator 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.

Hardware Templates

A *hardware template* is a file with a **.NbHwt** extension that defines the hardware configuration for a device. It specifies hardware attributes that include the transceiver type, Neuron Chip or Smart Transceiver model, clock speed, system image, and memory configuration. Several hardware templates are included with the IzoT 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 IzoT 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 IzoT NodeBuilder tool to build the application for a device. It contains a list of the application Neuron C source files, device-related preferences, and the hardware template name. When the application is built, the IzoT NodeBuilder tool automatically produces an *IzoT device template* and passes it to the IzoT Commissioning Tool and other network tools. The IzoT device template defines the device interface, and it is used by the IzoT Commissioning Tool and other network tools to configure and bind the device.

Device Interface Files

A *device interface file* (also known as an *XIF file* or an *external interface file*) is a file that specifies the 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's application. IzoT tools such as the IzoT Commissioning Tool use device interface files to create an IzoT device template. This enables the network tool to be used to create network designs without being connected to the physical devices, and it speeds up some configuration steps when the network tool is connected to the physical device. 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 IzoT NodeBuilder tool when you build an application. The IzoT 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 IzoT tools such as the IzoT Commissioning 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 LONMARK International. The standard resource files define standard network variable types (SNVTs), standard configuration property types (SCPTs), and standard functional profiles. For example, **SCPT_location** 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 temperature as a floating-point number. The standard network variable and configuration property types are defined at types.lonmark.org.

As new SNVTs and SCPTs are defined, updated resource files and documentation are posted to the LONMARK Web site. Standard functional profiles are included with the IzoT NodeBuilder tool, and their documentation is also available on the LONMARK Web site. To view and download the latest resource files and documentation, go to the LONMARK Web site at www.lonmark.org.

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 (UFPTs).

You can create applications that only use the standard types and profiles. In this case, you do not need to create user-defined resource files. If you need to define any new user types or profiles, you will use the *NodeBuilder Resource Editor* to create them.

Targets

A *target* is a LONWORKS device whose application is built by the IzoT 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.

Installing the IzoT NodeBuilder Development Tool

This chapter describes how to get started with your IzoT NodeBuilder tool, including how to install the IzoT NodeBuilder software and connect the FT 6000 EVK hardware.

Installing the IzoT FT 6000 EVK

To install your IzoT FT 6000 EVK, follow these steps:

1. Verify that you have a manufacturer ID. A manufacturer ID is required for many IzoT NodeBuilder tool functions.

Standard manufacturer IDs are assigned to manufacturers when they join LONMARK International, and are also published by LONMARK International so that the device manufacturer of a LONMARK certified device is easily identified. If your company is a LONMARK member, but you do not know your manufacturer ID, you can go to www.lonmark.org/spid and find your ID in the list of manufacturer IDs. The most current list at the time of release of the IzoT NodeBuilder tool is also included with the IzoT NodeBuilder software.

If you do not have a manufacturer ID, you can instantly get a temporary manufacturer ID by filling out a simple form at www.lonmark.org/mid.

2. Register your IzoT FT 6000 EVK. This entitles you to a free replacement software download or serial number if you lose either one in the future. To register your IzoT FT 6000 EVK, go to www.echelon.com/register, select the **FT 6000 EVK** product, enter the serial number from the back of your OpenLNS Commissioning Tool DVD case, enter the other information requested by the form, and then click **Register Now**.
3. Insert the **IzoT Commissioning Tool EVK Edition DVD** into your computer, install the IzoT Commissioning Tool software, and then activate the IzoT Commissioning Tool as described in Chapter 2 of the *IzoT Commissioning Tool User's Guide*. The IzoT Commissioning Tool must be installed on your computer in order to install the IzoT NodeBuilder software.
4. Install the IzoT NodeBuilder software as described in the next section.
5. Connect the FT 6000 EVK hardware as described in *Connecting the FT 6000 EVB Hardware* chapter of the *FT 6000 EVB Hardware Guide*.
6. Complete the quick-start exercise in Chapter 3, *IzoT NodeBuilder Quick-Start Exercise*. In the quick-start exercise, you will develop a device with one sensor and one actuator. The sensor is a simple sensor that monitors a push button on the FT 6000 EVB and toggles a network variable output each time the button is pressed. The actuator drives the state of an LED on the FT 6000 EVB based on the state of a network variable input.

This quick-start guides you through all the steps of creating a device with the IzoT NodeBuilder tool, including creating the NodeBuilder project, the device template, the device interface, and the Neuron C code that implements your device interface; implementing device functionality in the Neuron C code; building and downloading the device application; testing the device in an IzoT or LONWORKS network; and debugging the device application.

7. Run the Neuron C example applications included with the IzoT NodeBuilder tool on your FT 6000 EVBs. The IzoT NodeBuilder tool includes three Neuron C example applications (*NcSimpleExample*, *NcSimpleIsiExample*, and *NcMultiSensorExample*) that you can use to test the I/O devices on the FT 6000 EVBs, and create simple managed and self-installed IzoT or LONWORKS networks.

The *NcMultiSensorExample* application is pre-loaded on the FT 6000 EVBs and runs in Interoperable Self-installation (ISI) mode by default. You install and connect this example application and the other examples using the IzoT Commissioning Tool, or using the ISI protocol. See the *FT 6000 EVB Examples Guide* for more information on using these example applications.

Installing the IzoT NodeBuilder Software

To install the IzoT NodeBuilder software, follow these steps:

1. Download the IzoT NodeBuilder software from www.echelon.com/downloads.

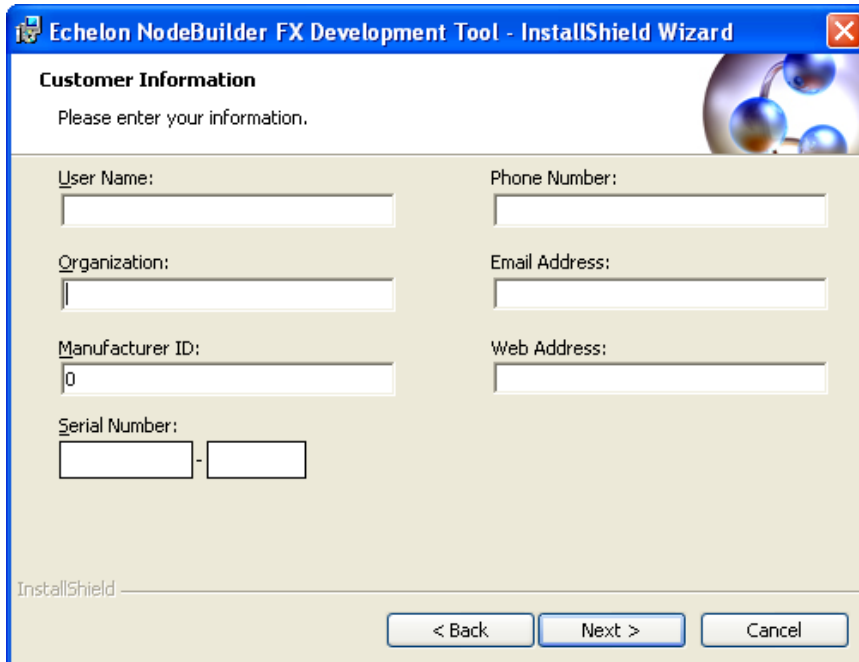
2. Run the IzoT NodeBuilder installer.
3. Run the NodeBuilder430.exe self-extracting installation program. The Welcome window of the NodeBuilder software installer opens.



4. Click **Next**. The NodeBuilder Development Tool License Agreement window opens.



5. Read the license agreement (see Appendix D, *NodeBuilder Software License Agreement*, for a printed version of this license agreement). If you agree with the terms, click **Accept the Terms** and then click **Next**. The Customer Information window appears.

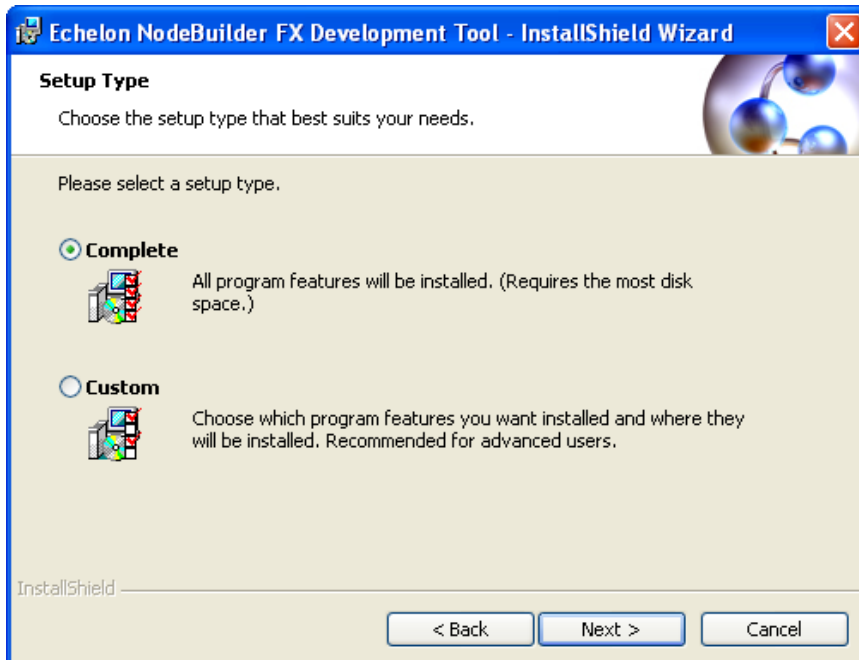


- Enter the NodeBuilder serial number on the back of IzoT Commissioning Tool DVD case in the **Serial Number** box. Optionally, you can enter the following registration information. The IzoT NodeBuilder tool automatically enters this information into your resource files. Your phone number, e-mail address, and Web address will be included with any resource file that you create and distribute.

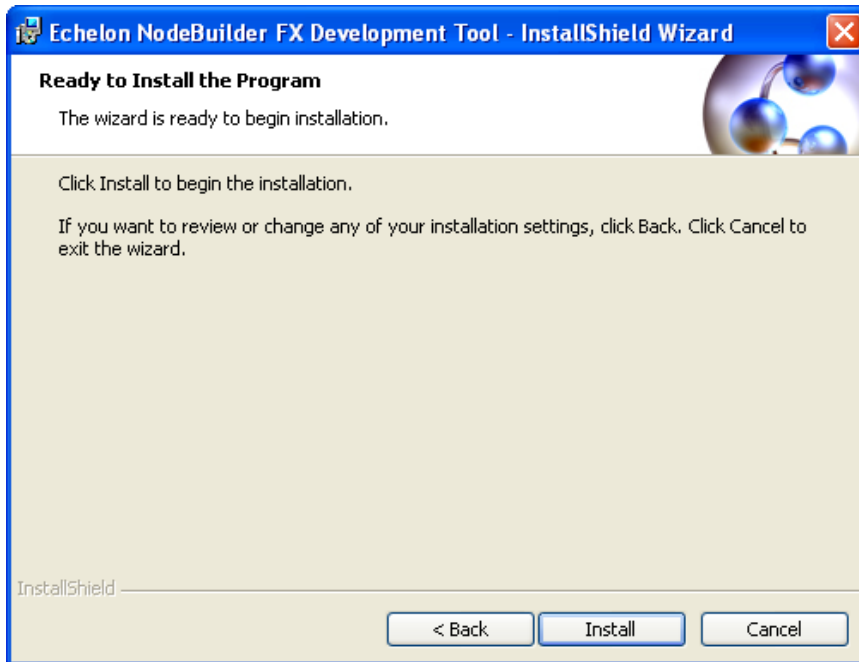
<i>User Name</i>	Your name. The name may be entered automatically based on the user currently logged on and whether other Echelon products are installed on your computer.
<i>Organization</i>	The name of your company. The name may be entered automatically based on the user currently logged on and whether other Echelon products are installed on your computer.
<i>Manufacturer ID</i>	If you have a standard manufacturer ID, enter it decimal format. If your company is a LONMARK member, but you do not know your manufacturer ID, you can find your ID in the list of manufacturer IDs at www.lonmark.org/spid . The most current list at the time of release of the IzoT NodeBuilder tool is also included with the IzoT NodeBuilder software. If you do not have a standard manufacturer ID, you can request a temporary manufacturer ID by filling out a simple form at www.lonmark.org/mid .
<i>Phone Number</i>	The phone number where you can be contacted.
<i>Email Address</i>	The e-mail address where you can be contacted.
<i>Web Address</i>	Your company's Web site.

Note: You can enter or modify this information after installing the IzoT NodeBuilder software in the NodeBuilder Project Manager. To do this, create or open a NodeBuilder project, click **Project**, click **Settings** (or right-click the Project folder in the Project pane and click **Settings** on the shortcut menu), and then click the **Registration** tab in the **NodeBuilder Project Properties** dialog.

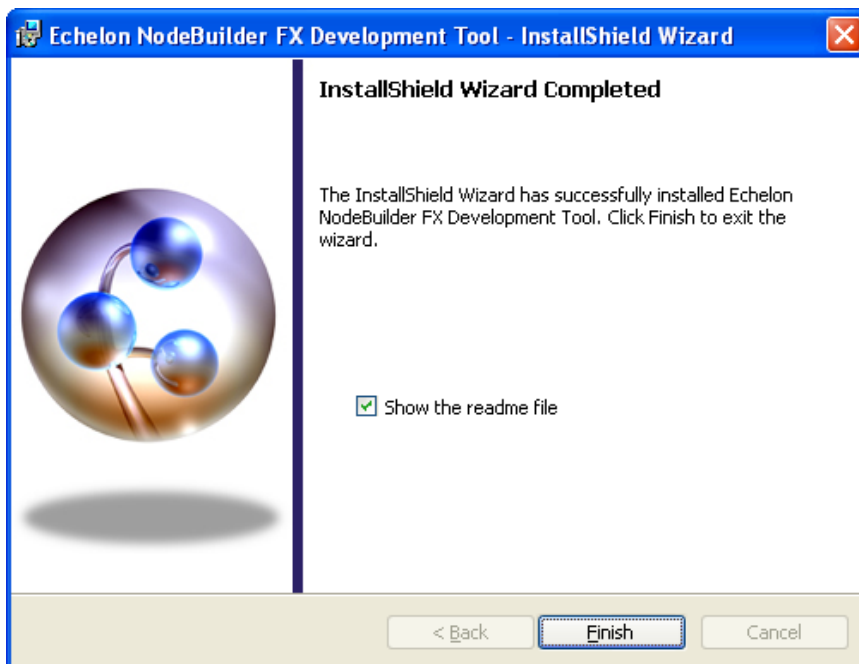
7. Click **Next**. If your computer does not have a LONWORKS directory, the Destination Location window appears. Choose a LONWORKS folder in which you want the IzoT NodeBuilder software installed. By default, the IzoT NodeBuilder software is installed in the **C:\Program Files (x86)\LonWorks** folder if you have not previously installed any Echelon or LONMARK products (this will not likely be the case because you should have already installed the IzoT Commissioning Tool, which is installed in the **C:\Program Files (x86)\LonWorks** folder by default on 64-bit versions of Windows). Click **Next**.
8. The Setup Type window opens.



9. Select the type of installation to be performed. Select **Complete** to install NodeBuilder features or select **Custom** to choose whether to install the FT 6000 EVB examples, NodeBuilder LTM-10A examples, both sets of examples, or neither on your computer. Click **Next**. The Ready to Install window appears.



10. Click **Install** to begin the NodeBuilder software installation. Before installing the IzoT NodeBuilder software, the following programs are automatically installed or upgraded on your computer (if they are not already installed on your computer, or if they are installed, but have a lower version): NodeBuilder Resource Editor 4.0, LONMARK Resource Files 14.00, LNS Plug-in Framework 1.10, and ISI Developer's Kit 3.02.
11. After the IzoT NodeBuilder software has been installed, a window appears stating that the installation has been completed successfully.



12. Click **Finish**. If a window appears prompting you to reboot your computer now or later, click **Yes** to reboot your computer now.

13. 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.
14. If you do not have a PDF document viewer, install Adobe Reader from get.adobe.com/reader/.

IzoT NodeBuilder Quick-Start Exercise

This chapter demonstrates how to create an IzoT or LONWORKS device using the IzoT NodeBuilder Development tool.

IzoT NodeBuilder Quick-Start Exercise

The following quick-start exercise demonstrates how to create an IzoT or LONWORKS device with the IzoT NodeBuilder tool. It introduces NodeBuilder features and provides some familiarity with the NodeBuilder interface.

The first step required to develop a device is to define the requirements for the device. For this quick-start exercise, you will develop a device with one sensor and one actuator. The sensor is a simple sensor that monitors a push button and toggles a network variable output each time the button is pressed. The actuator drives the state of an LED based on the state of a network variable input.

To develop an IzoT or LONWORKS device with the IzoT NodeBuilder tool, perform the following steps:

1. Create a NodeBuilder project.
2. Create a NodeBuilder device template.
3. Define the device interface and generate Neuron C source code that implements it.
4. Develop the device application by editing your Neuron C source code.
5. Compile, build, and download your application.
6. Test your device interface.
7. Debug your device's application.
8. Connect and test your device in a network.

Additional steps in the device development process include creating an IzoT CT stencil, an IzoT device plug-in, a human-machine interface (HMI), and an installation application for your device; and applying for LONMARK certification for your device. These steps are summarized in the *Additional Device Development Steps* section that follows this quick-start exercise.

After you complete this exercise, you can load and run the Neuron C example applications that are included with the IzoT NodeBuilder tool. The IzoT NodeBuilder software includes three Neuron C example applications that you can load into your FT 6000 EVBs, and one Neuron C example application that you can load into an LTM-10A platform with Gizmo 4 I/O Board (included with the NodeBuilder FX/PL Tool, and available separately). You can use these examples to test the I/O devices on the FT 6000 EVB or Gizmo 4 I/O board, and create simple LONWORKS networks. You can browse the Neuron C code used by these examples to further learn how to develop your own device applications.

For more information on using the FT example applications, see the *FT 6000 EVB Examples Guide*. For more information on using the PL example application, see the *NodeBuilder FX/PL Examples Guide*.

Step 1: Creating an IzoT NodeBuilder Project

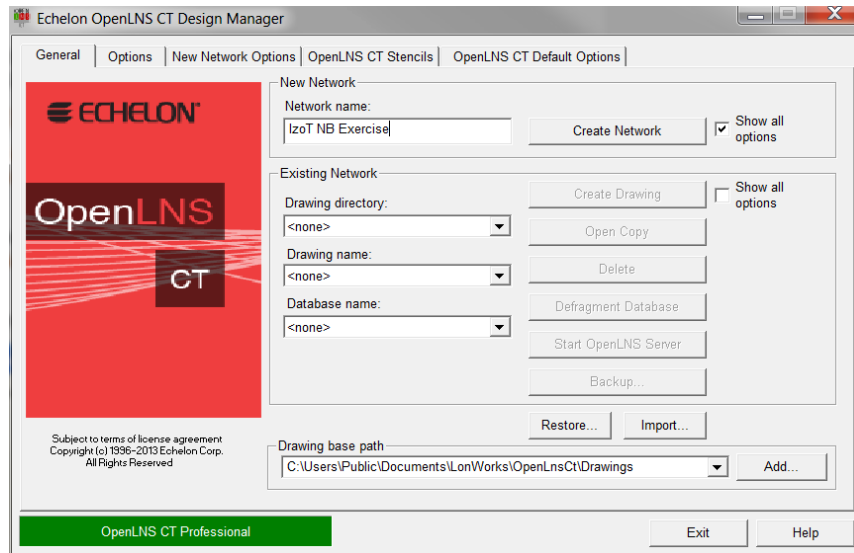
A *NodeBuilder project* collects all the information about a set of devices that you are developing. You will create, manage, and use NodeBuilder projects from the *NodeBuilder Project Manager*. The project manager provides an integrated view of your entire project and provides the tools you will use to define and build your project.

To create an IzoT NodeBuilder project, start the NodeBuilder Project Manager from the IzoT Commissioning Tool (CT) or directly from the NodeBuilder program folder. You will typically start the project manager from the IzoT CT because it simplifies the association of an IzoTNodeBuilder project with an IzoT CT network.

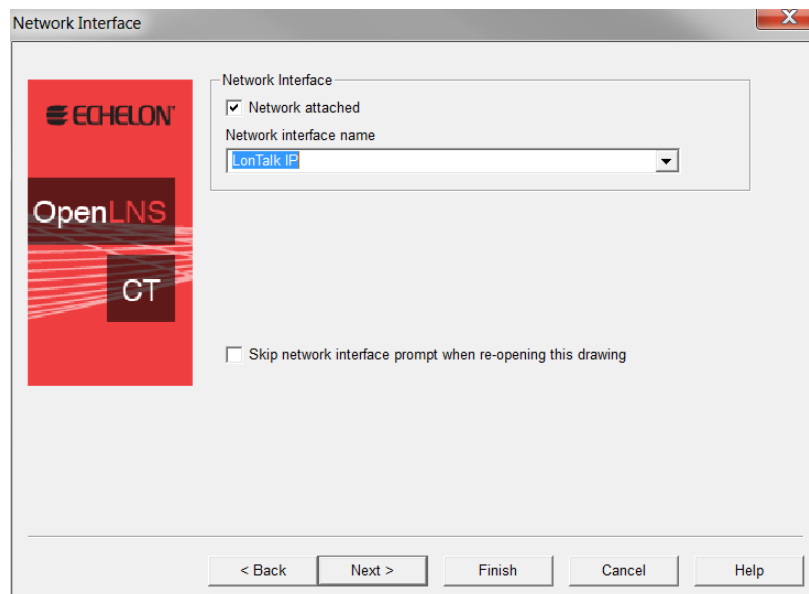
You can use the same NodeBuilder project with multiple IzoT CT networks, and you can use a IzoT CT network with multiple NodeBuilder projects. However, an IzoT CT network can only be used with **one** NodeBuilder project at a time.

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

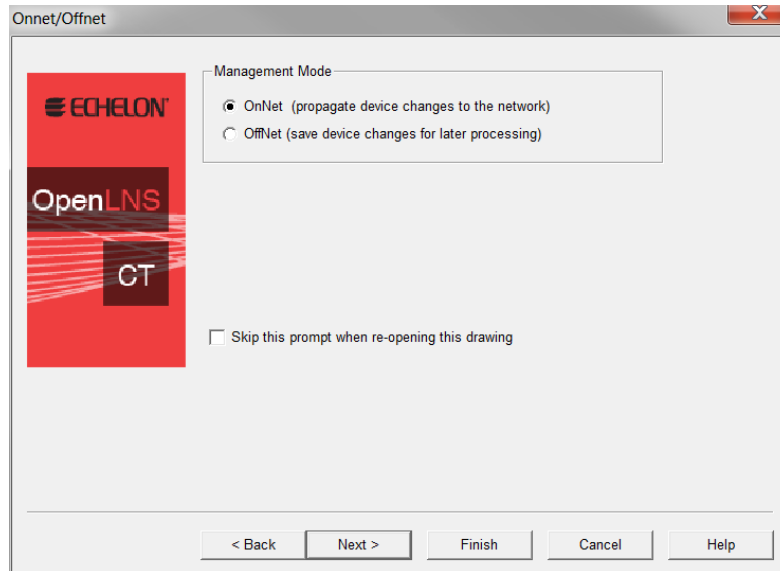
1. Create a new IzoT CT network. To do this, follow these steps:
 - a. Click **Start** on the taskbar, point to **Programs**, point to **Echelon OpenLNS CT**, and then select **OpenLNS Commissioning Tool**. The **LonMaker Design Manager** opens.
 - b. In the **Network Name** property under **New Network**, enter **IzoT NB Exercise**.



- c. Clear the **Show All Options** check box under **New Network** if it is selected.
- d. Click **Create Network** to create the new network.
 - A message may appear informing you that Visio must be launched and initialized so that it can work with IzoT CT. Click **OK**.
 - A warning may appear asking you if you want to enable macros. You must enable macros for the IzoT CT to function.
- e. Visio 2010 starts and the Naming page in the Network Wizard appears. Click **Next**. The Network Interface page appears.



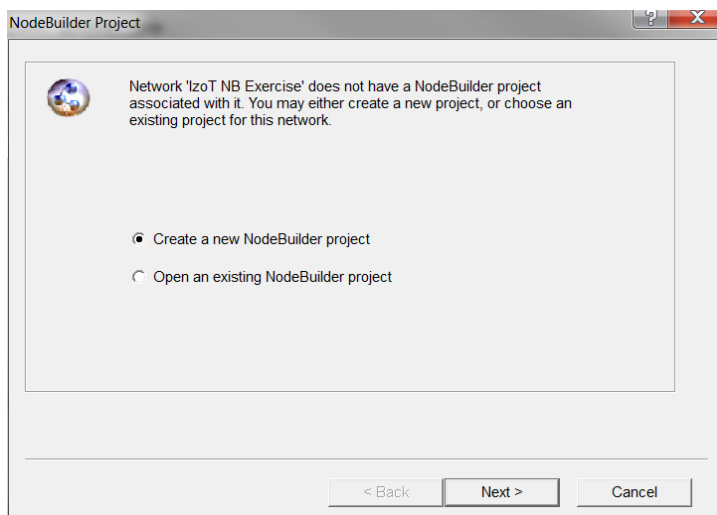
- f. Select the **Network Attached** check box and then select the LonTalk/IP network interface you created when you installed your FT 6000 EVK.
- g. Click **Next**. The Management Mode page appears.



- h. Select **OnNet**. This means that changes to the IzoT CT drawing are sent immediately to your NodeBuilder devices on the network. Click **Finish**.
- i. IzoT CT creates and opens a new network drawing.

For more information on creating and opening IzoT CT networks, see Chapter 3 of the *IzoT Commissioning Tool User's Guide*.

2. Click **Add-Ins**, click **OpenLNS CT**, and then click **NodeBuilder**.
3. The New Project wizard opens.



4. Accept the default **Create a New NodeBuilder Project** option, and then click **Next**.

5. Accept the default NodeBuilder **Project Name**, which is the same name as the IzoT CT network, and then click **Next**.
6. Accept the defaults in the **Specify Default Project Settings** dialog, and then click **Finish**.
7. The NodeBuilder New Device Template wizard starts. Proceed to the next section to create a NodeBuilder device template.

For more information on creating NodeBuilder projects, see Chapter 4, *Creating and Opening NodeBuilder Projects*.

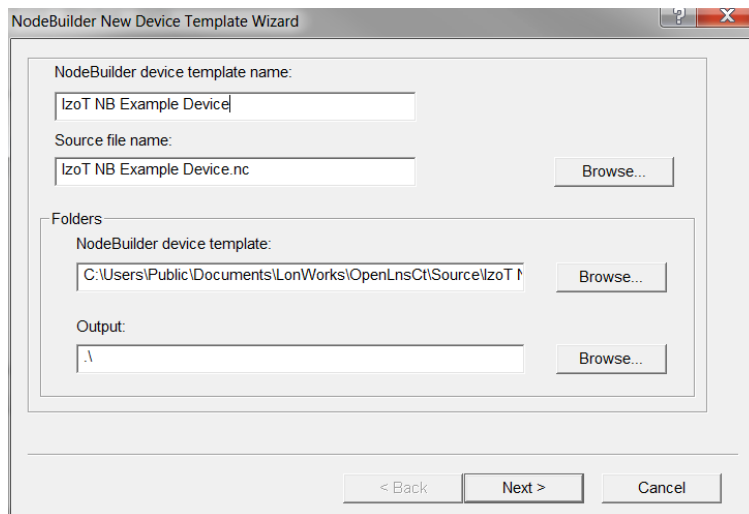
Step 2: Creating a NodeBuilder Device Template

Each type of device that you develop with the IzoT NodeBuilder tool is defined by a pair of device templates: a *NodeBuilder device template* and an *IzoT 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 IzoT device template defines the network interface to the device, and is used by IzoT tools such as the IzoT Commissioning Tool to configure and bind the device.

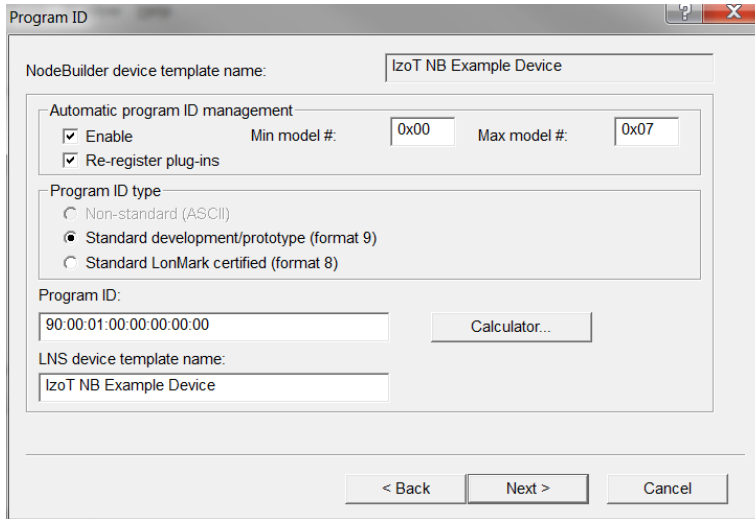
Each pair of device templates is identified by a unique *program ID*. Every device on a network with the same program ID must have the same device interface.

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

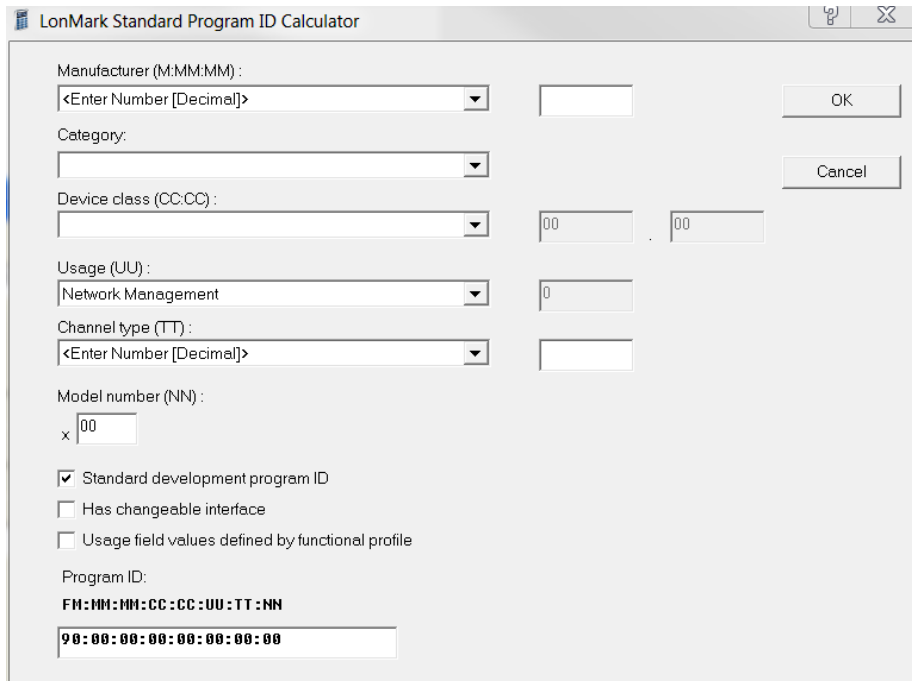
1. In the **NodeBuilder Device Template Name** property in the New Device Template wizard, enter **IzoT NB Example Device**.



2. Click **Next**. The Program ID window appears.



3. Click **Calculator**. The **Standard Program ID Calculator** dialog opens.



4. Enter the following values for the program ID fields:
 - In the **Manufacturer ID (M:MM:MM)** property, enter your standard manufacturer ID or temporary manufacturer ID in decimal format, or select the **Examples** manufacturer ID. By default, the manufacturer ID that you entered during of the IzoT NodeBuilder tool installation is shown by default.

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.

If you do not have a standard manufacturer ID, you can request a temporary manufacturer ID by filling out a simple form at www.lonmark.org/mid.
 - In the **Category** property, select the **I/O** option.

- In the **Device Class (CC:CC)** property, select the **Multi-I/O module (5.01)** option.
- In the **Usage (UU)** property, select the **General** option.
- In the **Channel Type (TT)** property, select the **TP/FT-10** option.
- In the **Model Number (NN)** property, enter **01**.

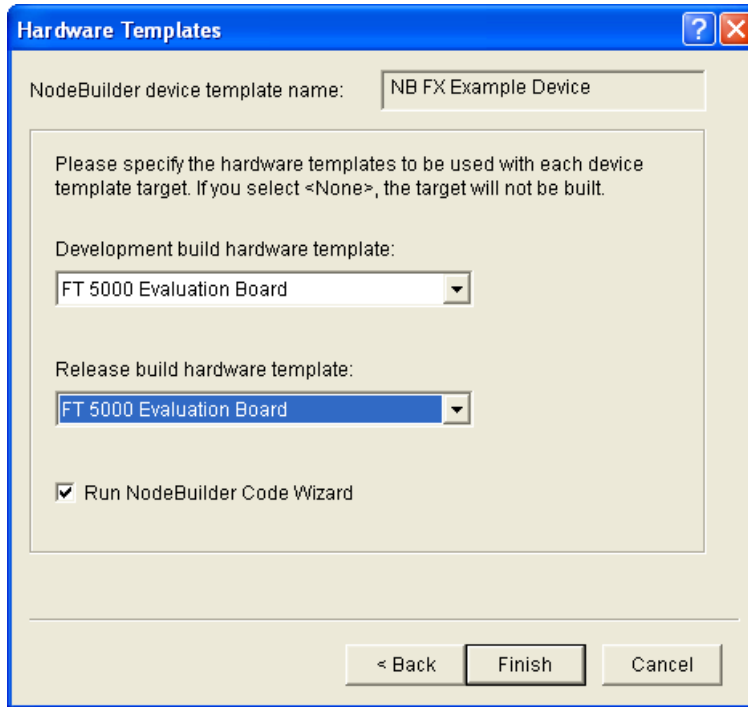
The screenshot shows the 'LonMark Standard Program ID Calculator' dialog box. It contains the following fields and values:

- Manufacturer (M:MM:MM): Examples (dropdown), 1048575 (text box)
- Category: I/O (dropdown)
- Device class (CC:CC): Multi-I/O module (5.01) (dropdown), 5 (text box), 01 (text box)
- Usage (UU): General (dropdown), 10 (text box)
- Channel type (TT): TP/FT-10 (dropdown), 4 (text box)
- Model number (NN): 01 (text box)
- Program ID: 9F:FF:FF:05:01:0A:04:01

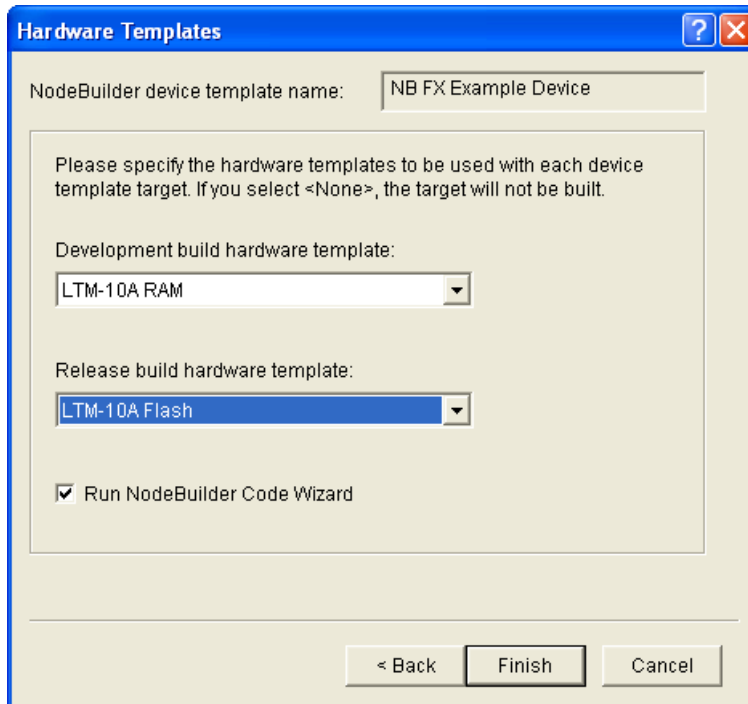
There are also checkboxes for 'Standard development program ID' (checked), 'Has changeable interface', and 'Usage field values defined by functional profile'. 'OK' and 'Cancel' buttons are present.

Note: The current list of manufacturer IDs, device classes, usage values, and channel types are defined in an XML file (**spidData.xml**) that is available at www.lonmark.org/spid. This file is updated as LONMARK International adds new manufacturer IDs, device classes, usage values, and channel types.

5. Click **OK** to return to the New Device Template wizard. The **Program ID** property contains the program ID you specified in the **Standard Program ID Calculator** dialog.
- Tip:** Do not clear the **Enable** check box under **Automatic Program ID Management**. This enables the Model Number (NN) field in the program ID to be incremented automatically when the external interface of the device is changed. This allows for the easy development of a device with a changing network 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 external interfaces.
6. Click **Next**. The Hardware Template window opens.
 7. Specify the development build and release build hardware template.
 - If you are using the FT 6000 EVK hardware (FT 6000 EVBs), select **FT 6000 EVB** in both the **Development Build Hardware Template** and **Release Build Hardware Template** properties.



- If you are using the NodeBuilder FX/PL hardware (LTM-10A Platform with Gizmo 4 I/O Board), select **LTM-10A RAM** in the **Development Build Hardware Template** property, and then select **LTM-10A Flash** in the **Release Build Hardware Template** property.

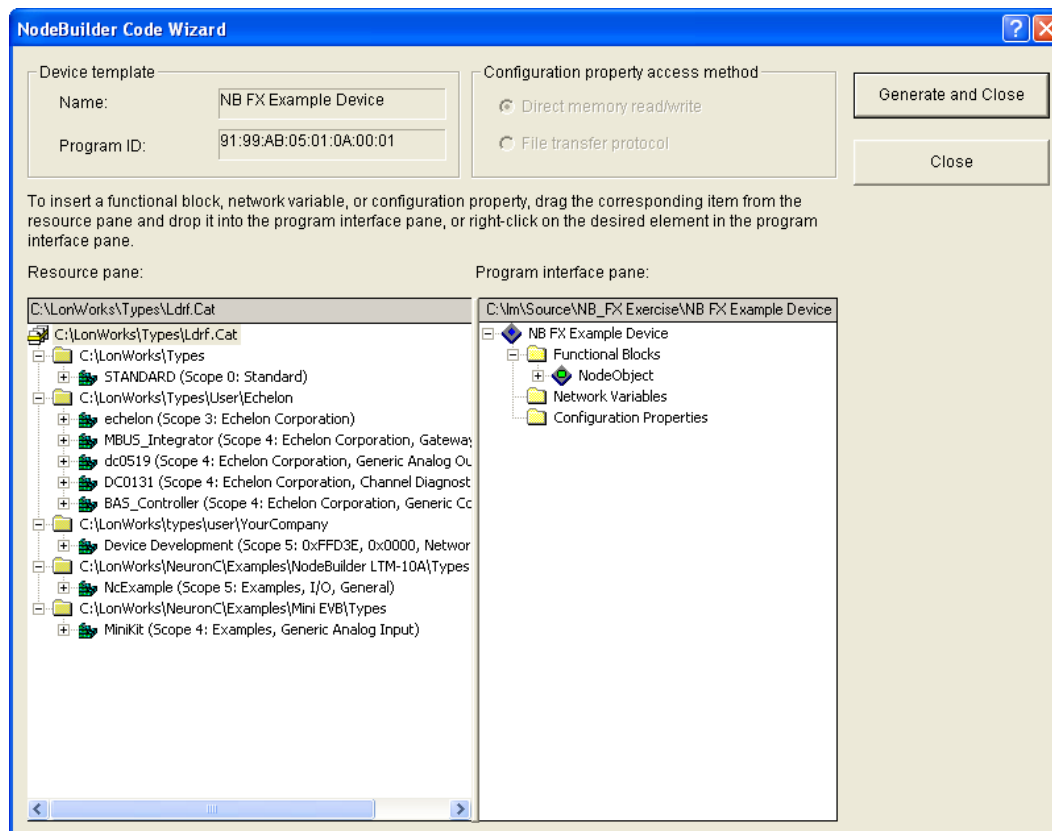


8. Click **Finish**. The NodeBuilder Code Wizard starts. There will be an initial pause as it reads the available resource files. Proceed to the next section to generate Neuron C code that defines your device's interface.

Step 3: Defining the Device Interface and Creating its Neuron C Application Framework

You can develop device applications with the IzoT NodeBuilder tool using the *Neuron C* programming language. Neuron C is based on ANSI C, with extensions for network communication, hardware I/O, timing, and event handling.

The IzoT NodeBuilder tool includes a NodeBuilder Code Wizard, which automatically generates Neuron C source code that defines the *device interface (XIF)*. The device interface includes all the functional blocks, network variables, and configuration properties implemented by your device. The NodeBuilder Code Wizard also generates much of the code for the Node Object functional block, which is a standard functional block that is used for maintaining and managing the device and its functional blocks.



The left pane of the NodeBuilder Code Wizard is the Resource pane, which is used to display the resources that are available for your application. The right pane is the Program Interface pane, which is used to display and modify your device's interface. You will define your device's interface by dragging functional profile templates and network variable and configuration property types from the Resource pane to the Program Interface pane.

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

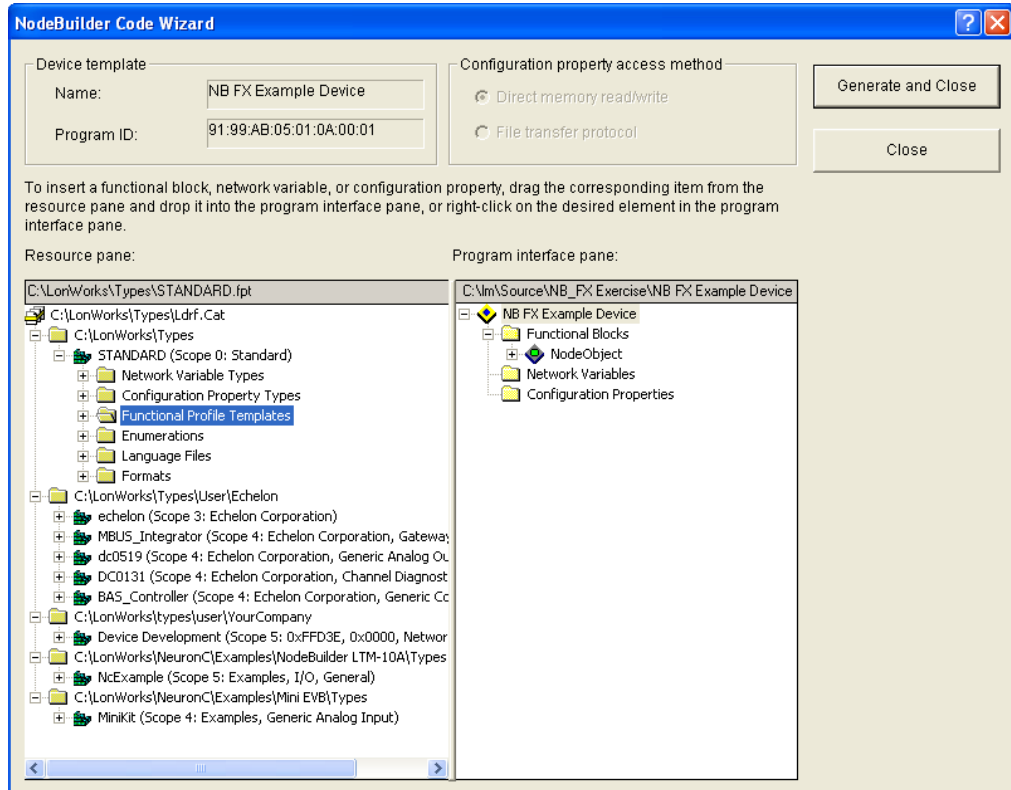
In this step, you will automatically create Neuron C source code for a device with the following functional blocks:

- An open-loop sensor functional block with a **SNVT_switch** output network variable.
- An open-loop actuator with a **SNVT_switch** input network variable.

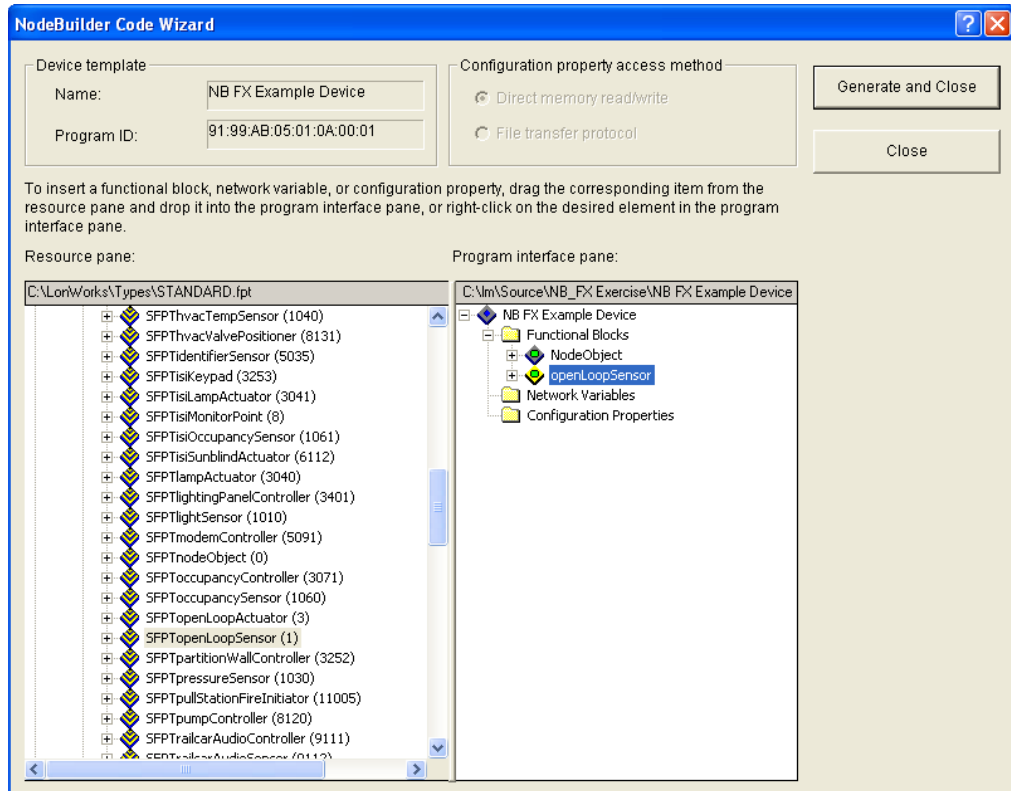
- A simple Node Object with no configuration properties (the NodeBuilder Code Wizard automatically creates this functional block).

To define your device interface and automatically create Neuron C source code for it using the Code Wizard, follow these steps:

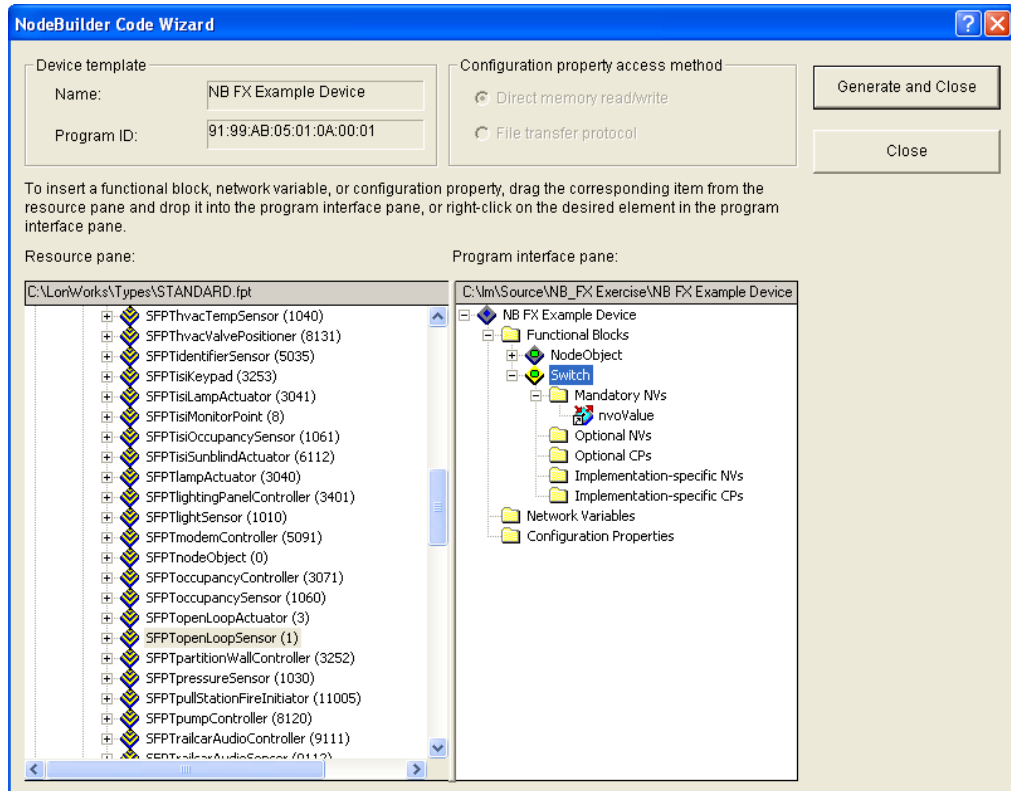
1. Create an open-loop sensor functional block with a **SNVT_switch** network variable. To do this, follow these steps:
 - a. Expand the **STANDARD (Scope 0: Standard)** resource file under the **LonWorks/Types** folder, and then expand the **Functional Profile Templates** folder to display the standard functional profile templates (SFPTs).



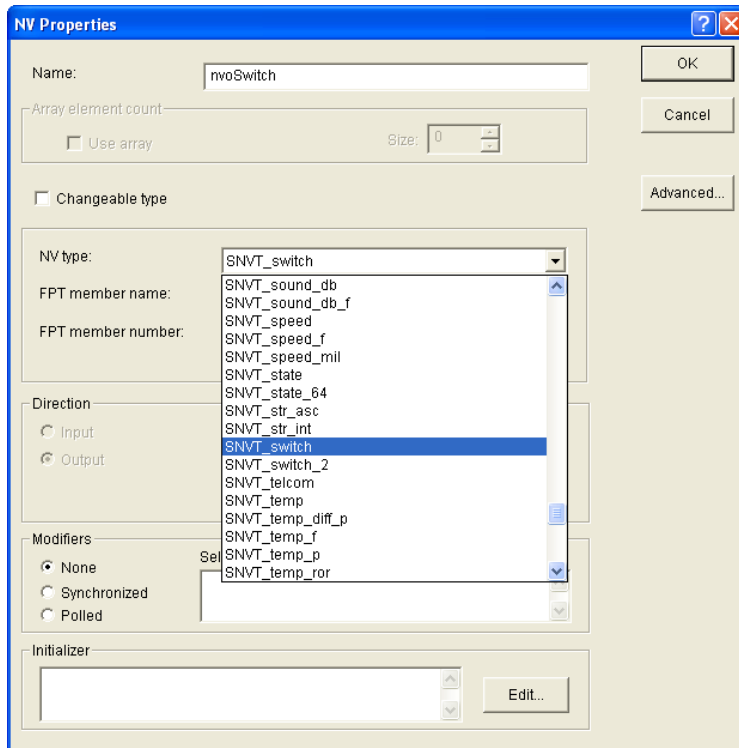
- b. Drag the **SFPTOpenLoopSensor (1)** functional profile template from the Resource Pane on the left side to the **Functional Blocks** folder in the Program Interface pane on the right side. An **openLoopSensor** functional block appears under the **Functional Blocks** folder.



- c. Rename the **openLoopSensor** functional block to “Switch”. To do this, right-click the **openLoopSensor** functional block in the Program Interface pane, click **Rename** on the shortcut menu, enter **Switch**, and then press ENTER or TAB. A warning message appears warning that new source files will be generated as a result of the name change. Click **OK**.
- d. Expand the **Switch** functional block, and then expand the **Mandatory NVs** folder to display the **nvoSwitch** network variable.



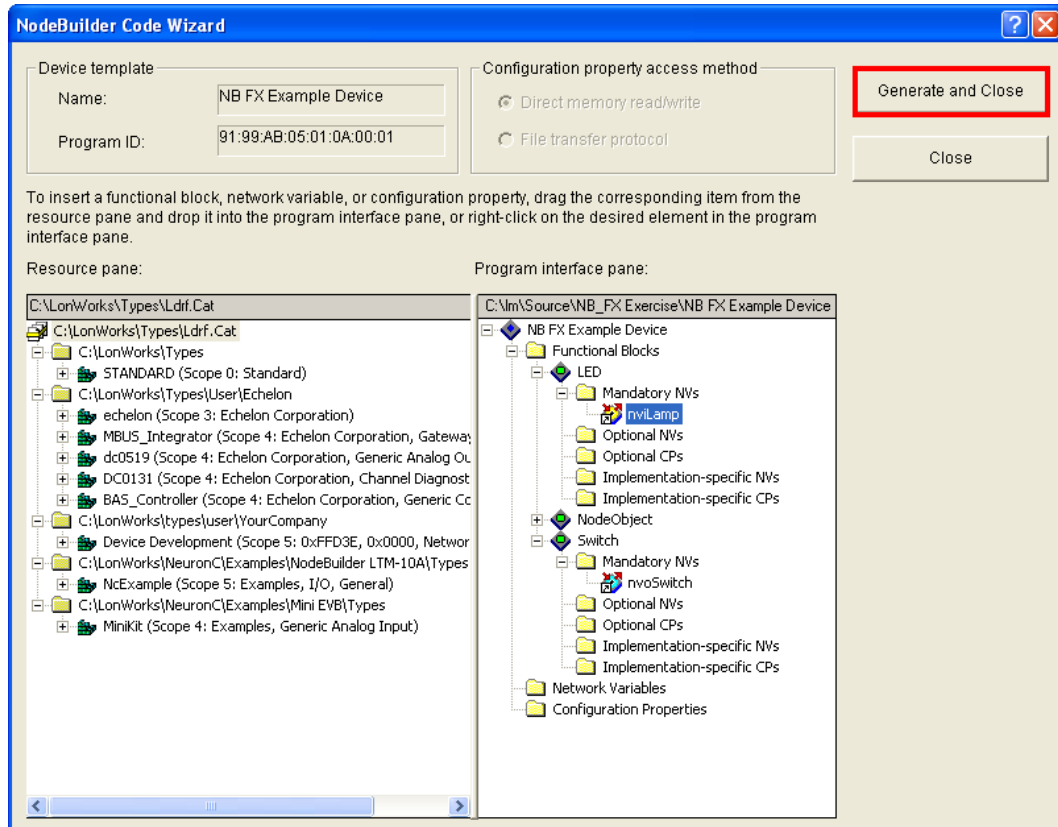
- e. Double-click the **nvoValue** network variable, or right-click it and then select **Properties** on the shortcut menu. The **NV Properties** dialog opens.
- f. In the **Name** property, change the network variable name to **nvoSwitch**.
- g. In the **NV Type** property, select **SNVT_switch**, and then click **OK**.



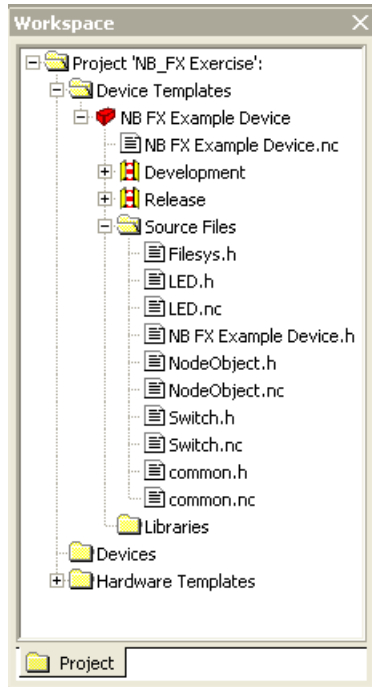
2. Create an open-loop actuator with a **SNVT_switch** network variable.
 - a. Drag a **SFPTopenLoopActuator** functional profile template from the Resource Pane on the left side to the **Functional Blocks** folder in the Program Interface pane on the right side.
 - b. Rename the **openLoopActuator** functional block to “LED”. A warning message appears warning that new source files will be generated as a result of the name change. Click **OK**.
 - c. Expand the **LED** functional block, and then expand the **Mandatory NVs** folder to display the **nviValue** network variable.
 - d. Double-click the **nviValue** network variable, or right-click it and then select **Properties** on the shortcut menu. The **NV Properties** dialog opens.
 - e. In the **Name** property, change the network variable name to **nviLamp**.
 - f. In the **NV type** property, select **SNVT_switch**, and then click **OK**.

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

3. Click the **Generate and Close** button in the top-right corner of the NodeBuilder Code Wizard to generate the Neuron C source files that implement your specified external interface.



4. The NodeBuilder Code Wizard closes and you are returned to the Project Manager window. The Project pane within the project manager displays the files and templates defined for your project.



5. Double-click the **IzoT NB Example Device.nc** file in the Project pane to open the main Neuron C file for this new device template.
6. Open the **Switch.h** and **LED.h** header files and view the functional block and configuration property declarations.
7. Open the **Switch.nc** and **LED.nc** Neuron C files and view the default implementation of the director function (named **SwitchDirector** or equivalent).

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” components in the device. This is done using the following line of code:

```
executeOnEachFblock (FBC_WHEN_RESET);
```

8. Proceed to the next section to implement your device’s functionality by editing your Neuron C code.

For more information on defining device interfaces and generating Neuron C code for them, see Chapter 6, *Defining Device Interfaces and Creating their Neuron C Application Framework*.

Step 4: Developing the Device Application

The Neuron C source code generated by the NodeBuilder Code Wizard implements your device’s interface. The Code Wizard also generates a skeleton application framework, including the most common tasks performed by the Node Object. When developing the device application, you will typically concentrate on writing the algorithms that implement your device’s functionality. To do this, you will edit the code generated by the Code Wizard and program any required interaction between the device application and the I/O devices on your device hardware.

In this step, you will add Neuron C I/O declarations to the **Switch.h**, and **LED.h** header files, and then implement your desired I/O functionality in the **Switch.nc** and **LED.nc** Neuron C files.

Note: The I/O object declarations used for the FT 6000 EVK hardware (FT 6000 EVBs) and the NodeBuilder FX/PL hardware (LTM-10A Platform with Gizmo 4 I/O Board) are different. Therefore,

follow the section corresponding with the development platform or platforms you are using for the appropriate code to use.

FT 6000 Evaluation Boards

1. Declare the I/O hardware for the Switch following these steps:
 - a. Double-click the **Switch.h** file in the Project pane to edit the source file.
 - b. Find the following line of code near the end of the Editor window:

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

- c. Add the following line of code after the line referenced in step b.

```
IO_9 input bit ioSwitch1;
```

2. Add functionality to the Switch I/O following these steps:
 - a. Double-click the **Switch.nc** file in the Project pane.

- b. Find the following line of code at the end of the Editor window:

```
#endif // _Switch_NC_
```

- c. Add the following when-clause before the line referenced in step b:

```
when (io_changes (ioSwitch1))
{
    nvoSwitch.state = !input_value;
    nvoSwitch.value = input_value ? 200u : 0;
}
```

3. Declare the I/O hardware for the LED. To do this follow these steps:
 - a. Double-click the **LED.h** file in the Project pane.

- b. Find the following line of code near the end of the Editor window:

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

- c. Add the following line of code after the line referenced in step b.

```
IO_2 output bit ioLamp = 1;
```

4. Add functionality to the LED I/O following these steps:

- a. Double-click the **LED.nc** file in the Project pane.
 - b. Find the following lines of code near the beginning of the Editor window:

```
when(nv_update_occurs(nviLamp))
//
//}}NodeBuilder Code Wizard End
{
```

- c. Add the following line of code after the lines referenced in step b:

```
io_out(ioLamp, !(nviLamp.value && nviLamp.state));
```

5. Click **File** and then click **Save All** to save all your changes to the source files.
6. Proceed to the next section to compile your Neuron C application, and then build an application image and download it to your device.

For more information on editing Neuron C code to implement your device's functionality, see Chapter 7, *Developing Device Applications*.

LTM-10A Platform and Gizmo 4 I/O Board

1. Declare the I/O hardware for the Switch following these steps:
 - a. Double-click the **Switch.h** file in the Project pane to edit the source file.
 - b. Find the following line of code near the end of the Editor window:

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

- c. Add the following line of code after the line referenced in step b.
2. Add functionality to the Switch I/O following these steps:

```
IO_6 input bit ioSwitch1;
```

- a. Double-click the **Switch.nc** file in the Project pane.
- b. Find the following line of code at the end of the Editor window:

```
#endif // _Switch_NC_
```

- c. Add the following when clause before the line referenced in step b:

```
when(io_changes(ioSwitch1))  
{  
    nvoSwitch.state = !input_value;  
    nvoSwitch.value = input_value ? 200u : 0;  
}
```

3. Declare the I/O hardware for the LED. To do this follow these steps:

- a. Double-click the **LED.h** file in the Project pane.
- b. Find the following line of code near the end of the Editor window:

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

- c. Add the following line of code after the line referenced in step b.

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

4. Add functionality to the LED I/O following these steps:

- a. Double-click the **LED.nc** file in the Project pane.
- b. Find the following lines of code near the beginning of the Editor window:

```
when(nv_update_occurs(nviValue))
```

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

- c. Add the following line of code after the lines referenced in step b:

```
io_out(ioLamp, !(nviLamp.value && nviLamp.state));
```

5. Click **File** and then click **Save All** to save all your changes to the source files.
6. Proceed to the next section to compile your Neuron C application, and then build an application image and download it to your device.

For more information on editing Neuron C code to implement your device's functionality, see Chapter 7, *Developing Device Applications*.

Step 5: Compiling, Building, and Downloading the Application

The IzoT 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.

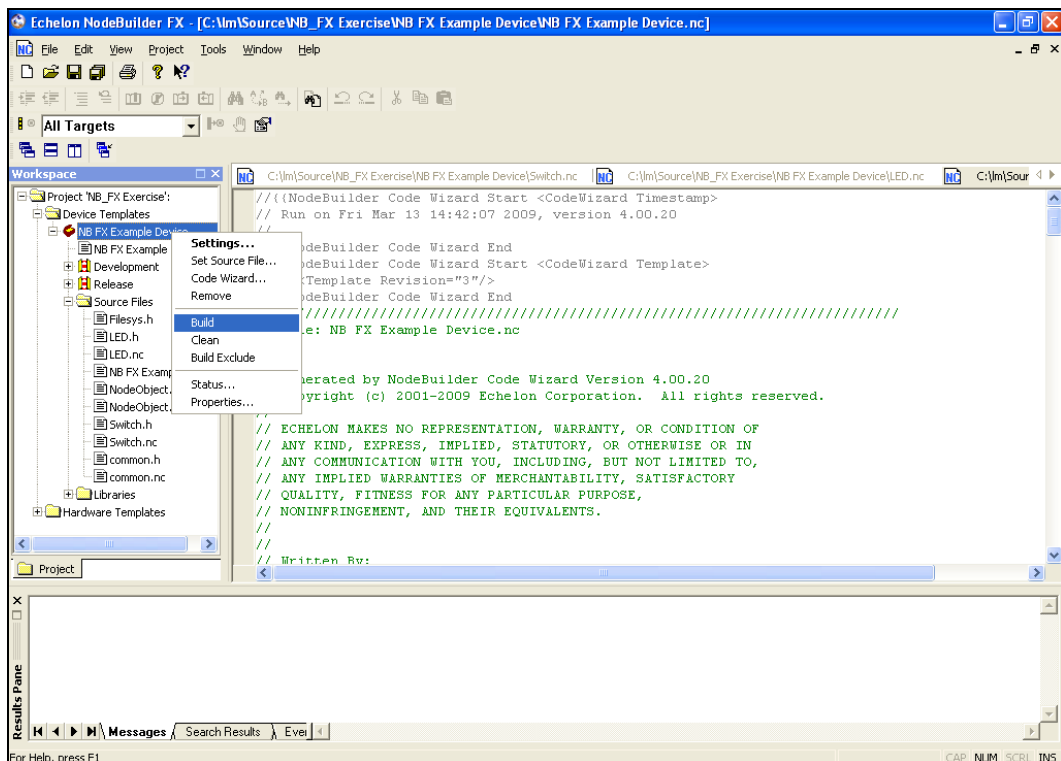
When you build your application, the IzoT NodeBuilder tool will create *downloadable application image files* and *device interface files*. The downloadable application image file is used by the IzoT Commissioning Tool and other network tools to download the compiled application image to a device. The device interface file describes the external interface for your device. It is used by network tools such as the IzoT Commissioning Tool to determine how to bind and configure your device. The device interface file is also used by the IzoT NodeBuilder tool to automatically create the LNS device template.

The IzoT 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 IzoT NB Exercise project is

C:\Users\Public\Documents\LonWorks\OpenLnsCt\Source\IzoT NB Exercise. The two device file sets are written to different directories—the **IzoT NB Example Device\Development** directory and the **IzoT NB Example Device\Release** directory. The development and release file set are both stored within your project directory.

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

1. Right-click the **IzoT NB Example Device** device template icon in the Project pane, then click **Build** on the shortcut menu.



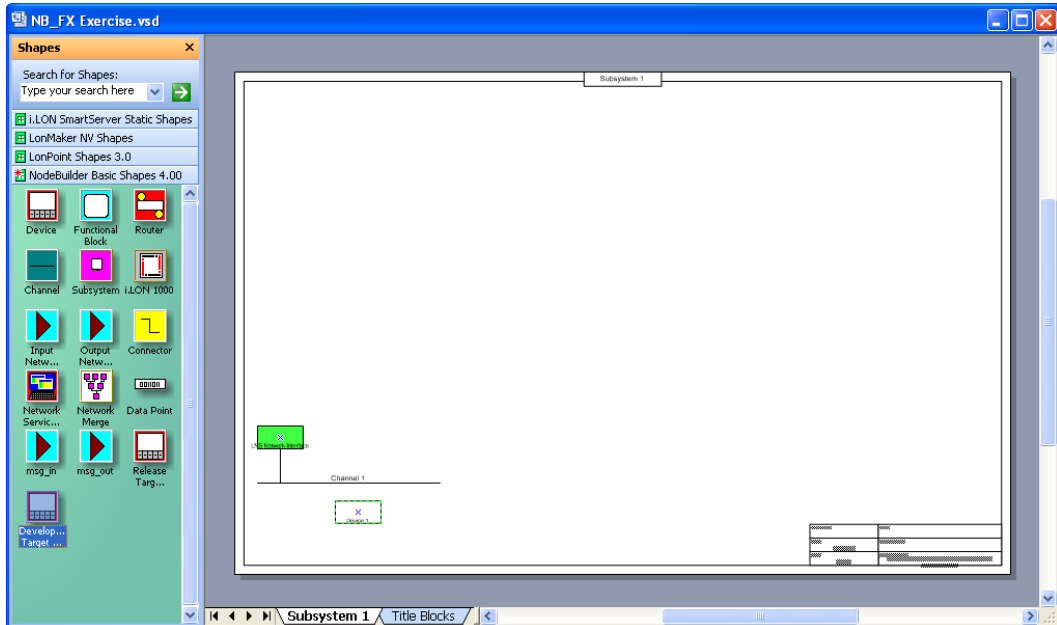
2. If you receive any build errors, double-check that the code you entered matches that listed in *Step 4: Developing Device Applications* (you may receive some warnings, which can be ignored in the context of this quick-start exercise).
3. Click the Echelon IzoT CT/Visio button in the Taskbar to switch to the IzoT Commissioning Tool (CT). You will use IzoT CT to install, bind, configure, and test the devices in your project. IzoT

CT displays a network drawing that shows the devices, functional blocks, and connections in your network.

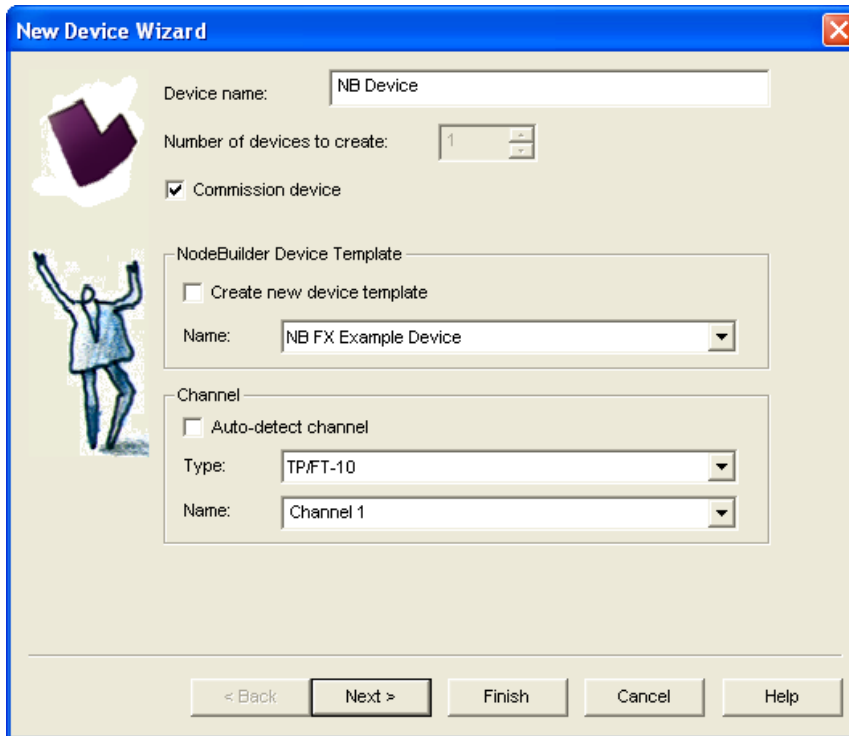
IzoT CTI also displays *stencils* that contain shapes that you can drag to your IzoT CT drawing. IzoT CT includes a **NodeBuilder Basic Shapes 4.00** stencil with shapes that you will use to add new devices, functional blocks, and connections to your network drawing. The **NodeBuilder Basic Shapes 4.00** 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 4.00** 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 IzoT NodeBuilder tool. The IzoT NodeBuilder tool allows you to create a mixed network of development hardware (FT 6000 EVB or LTM-10A Platforms), release hardware (your own hardware), and other devices.

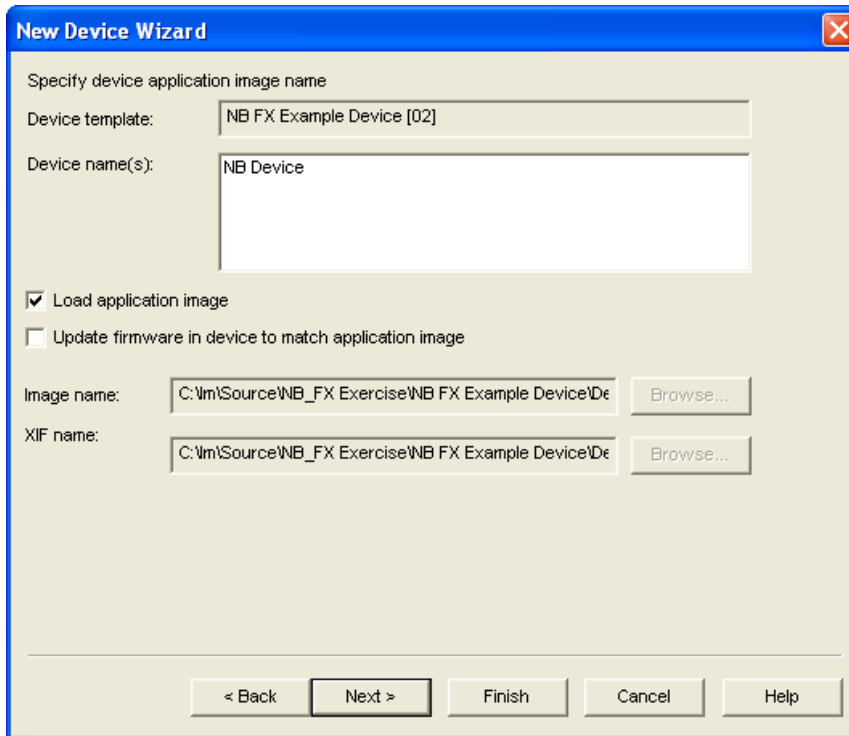
4. Drag a **Development Target Device** shape from the **NodeBuilder Basic Shapes 4.00** stencil to your network drawing. You can drop the shape anywhere, but a good location is just below the Channel 1 shape on your drawing.



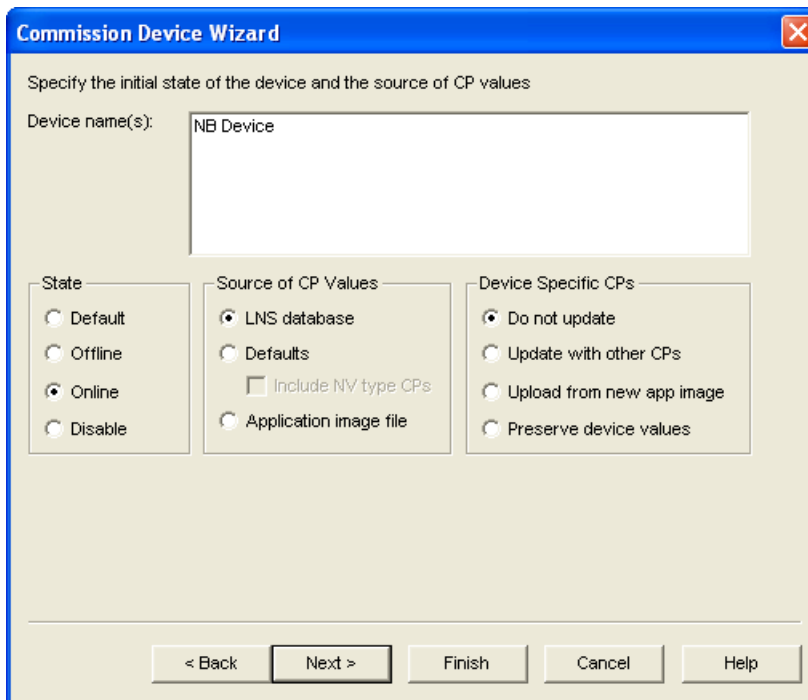
5. The New Device Wizard opens. In the **Device Name** property, enter **NB Device**, and then select the **Commission Device** check box. Verify that **IzoT NB Example Device** is selected in the **NodeBuilder Device Template** box.



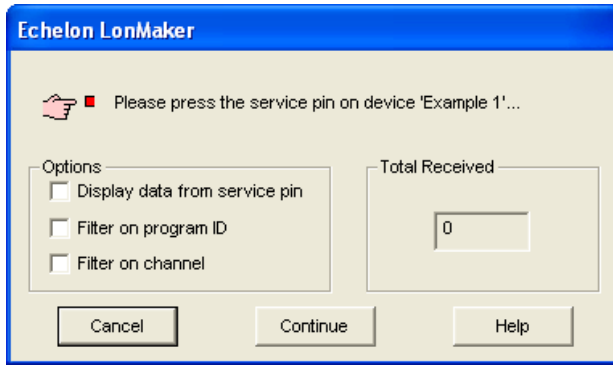
6. Click **Next** three times. The window in the New Device Wizard lets you select the application image to be downloaded to your device.
7. Select the **Load Application Image** check box and then click **Next**. This specifies that you will download to the device the binary application image file (.APB extension) that was automatically created when you built the device with the IzoT NodeBuilder tool. The application image files for your NodeBuilder development devices are stored in the **C:\Users\Public\Documents\LonWorks\OpenLnsCt\Source\<NodeBuilder Project>\<NodeBuilder Device Template>\Development** folder.



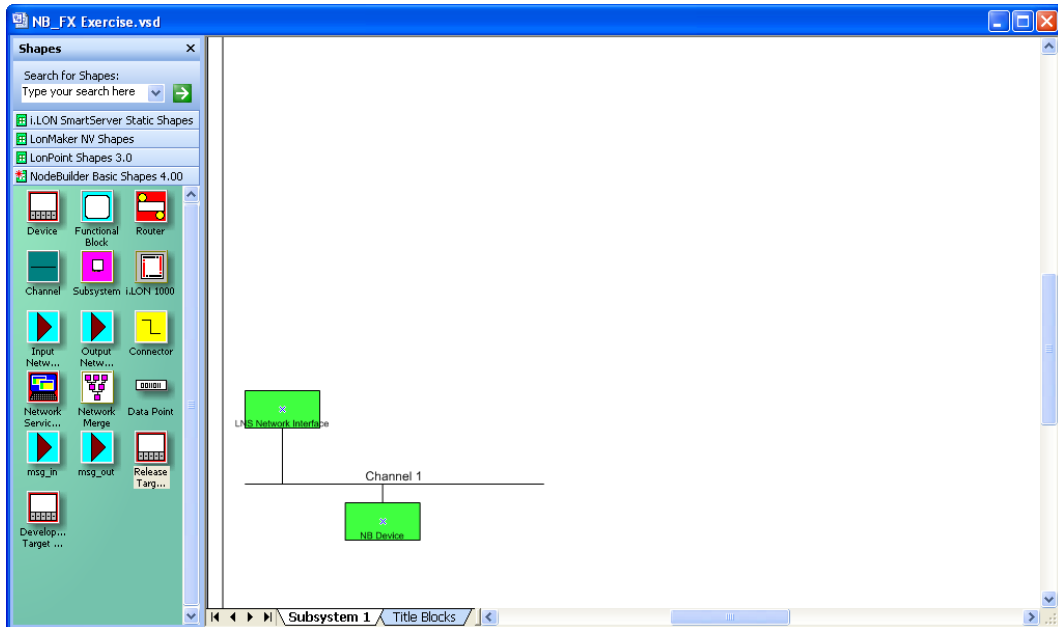
8. The next window lets you set the initial device state and the source of configuration property values when your device is commissioned.
9. Select the **Online** option under **State**. This means that your device will run its application after it has been commissioned.



10. Click **Finish**. The Press Service Pin window appears.



11. Press the service pin on the development platform to be loaded and commissioned. IzoT CT loads the application image for your IzoT NB Example Device application to the development platform and makes it operational. When IzoT CT is done commissioning, it will return to the IzoT CT drawing. The device shape will be solid green indicating that the device has been commissioned and is online. The device application will not do anything until you test the device or connect it to other devices.



12. Proceed to the next section to test your device's interface using the IzoT Browser.

For more information on building and downloading device applications, see Chapter 8, *Building and Downloading Device Applications*.



Step 6: Testing the Device Interface

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

The first tool that you will typically use for testing is the IzoT Browser. The browser displays all the input and output network variables and configuration properties for your device. You will typically exercise the hardware or network variable inputs to your device and observe the hardware and network outputs from your device.

4. Press and hold the left button at the bottom of your development board (SW1 on the FT 6000 EVB; IO_6 on the Gizmo 4 I/O Board). The value of the **nvoSwitch** network variable in the **Switch** functional block changes to 100.0 1, which means that the switch is at its maximum level (100%) and on.
5. Release the left button at the bottom of your development board. The value of the **nvoSwitch** network variable in the **Switch** functional block changes back to 0.0 0, which means that the switch is at its lowest level (0%) and off.

Note: The **nvoSwitch** network variable does not toggle each time you press the button. Instead, it depicts the current state of the button. You will modify the behavior of the Switch functional block in *Step 7: Debugging Your Device's Application* so that it acts as a toggle-switch.

6. Click anywhere in the row for the **nviLamp** network variable in the **LED** functional block. In the **Value** box in the browser toolbar, enter **100.0 1** and then press ENTER or click the **Set Value** button () in the browser toolbar. This sets the LED on the left side of your development board (LED1 on the FT 5000 EVB; IO_0 on the Gizmo 4 I/O Board) to its maximum level (100%) and turns it on.
7. In the **Value** box in the browser toolbar, enter **0.0 0**, and then press ENTER or click the **Set Value** button () in the browser toolbar. This returns the LED to its lowest level (0%) and turns it off. The LED functional block appears to be functioning correctly.
8. Proceed to the next section to debug your device's application. You will modify your device application so that the value of the **nvoSwitch** network variable in the **Switch** functional block toggles each time the button is pressed instead of when the button is pressed and released.

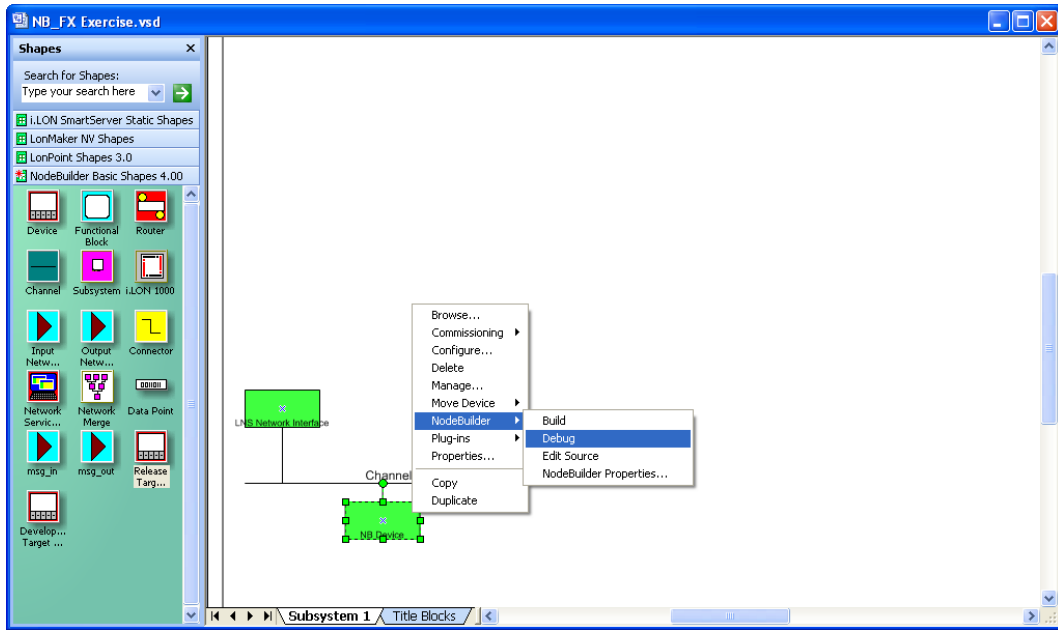
For more information on testing your device, see Chapter 9, *Testing a NodeBuilder Device Using the IzoT Commissioning Tool*.

Step 7: Debugging the Device Application

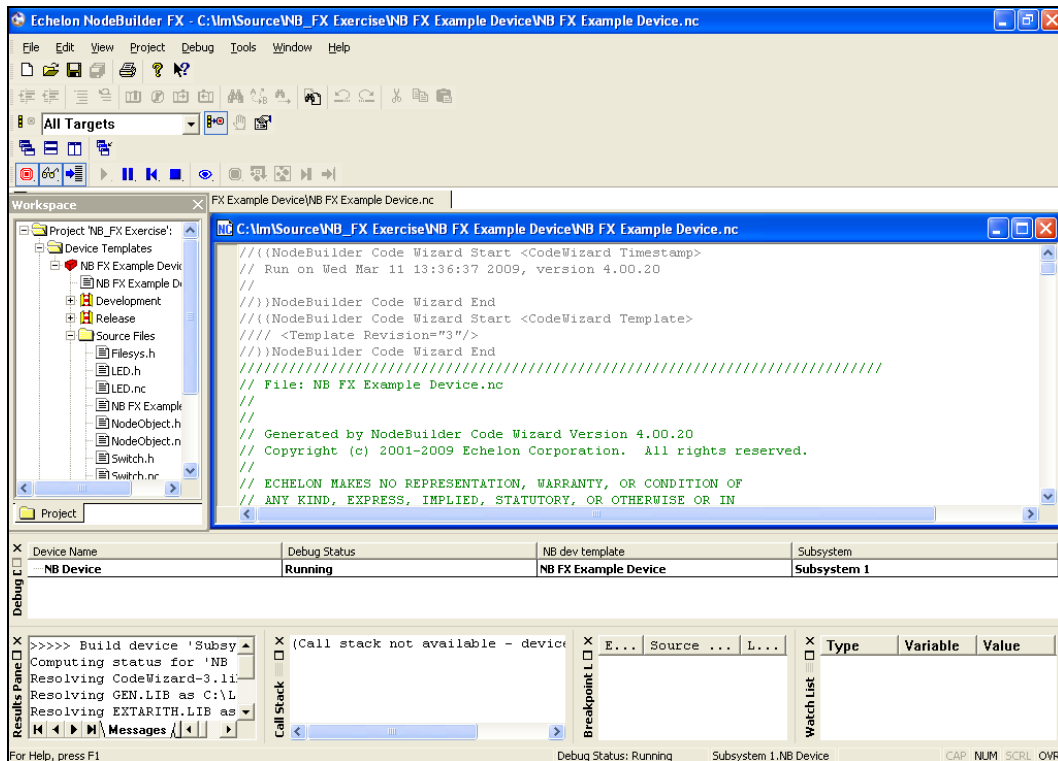
If your device does not function as expected, you can use the NodeBuilder Debugger to control and observe the behavior of the device application. The debugger allows you to set breakpoints, monitor variables, halt the application, step through the application, view the call stack, and peek and poke memory. You can make changes to the code as you debug your device.

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

1. Click the Echelon IzoT/Visio button in the Taskbar to switch to the IzoT Commissioning Tool.
2. Right-click the **NB Device** device shape in your IzoT CT drawing, point to **NodeBuilder**, and then click **Debug** on the shortcut menu.

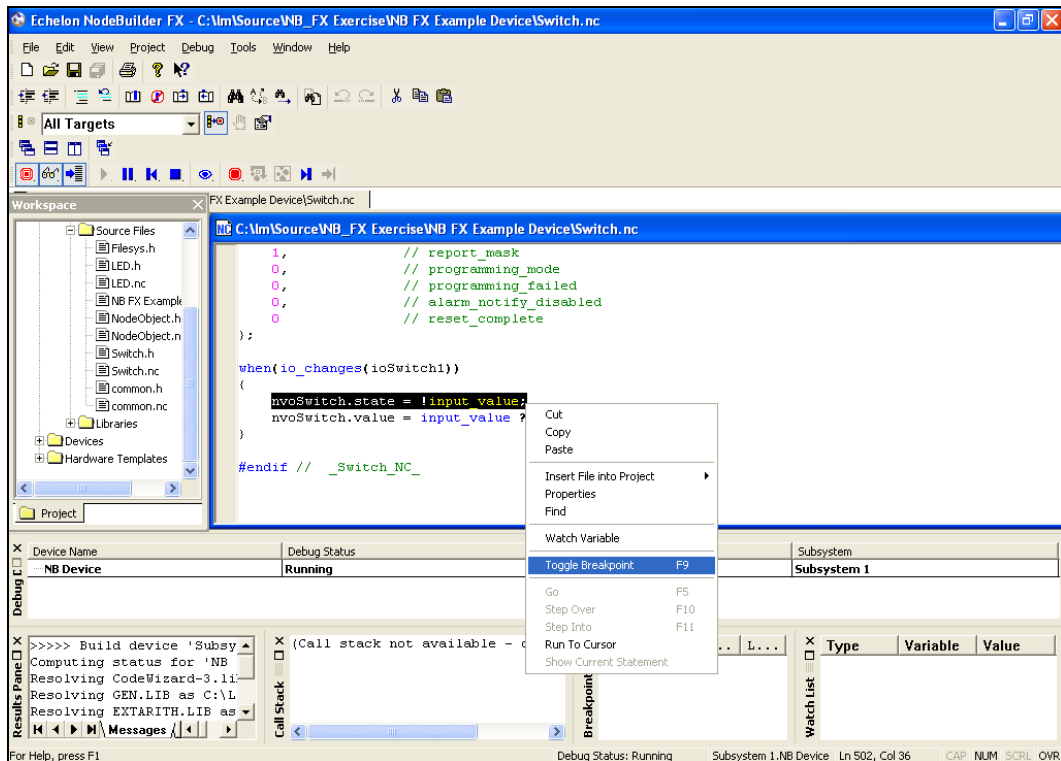


3. The NodeBuilder Project Manager appears, and a debug session for the device starts. There is a short pause as the debug session is started while the IzoT NodeBuilder tool establishes communication with the device's debug kernel.

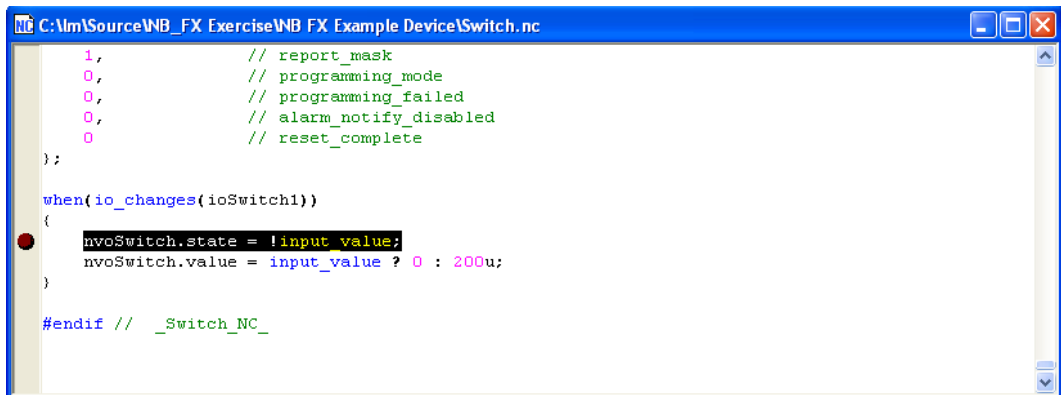


4. Double-click the **Switch.nc** file in the Project pane. A Debug window appears for the **Switch.nc** file.
5. Find the `when(io_changes(ioSwitch))` clause near the end of the file. This is the code you added in *Step 4: Developing the Device Application*.

- Right-click the `nvoSwitch.state = !input_value` line, and then click **Toggle Breakpoint** on the shortcut menu, or click anywhere in the line and press F9.



- A breakpoint marker (●) appears next to the line, and the line is added to the Breakpoint List pane at the bottom of the NodeBuilder Project Manager.



- Press and then release the left button at the bottom of your development board (SW1 on the FT 6000 EVB; IO_6 on the Gizmo 4 I/O Board). Observe that program execution stops at your breakpoint as denoted by the arrow symbol on top the breakpoint symbol (👉●).


```

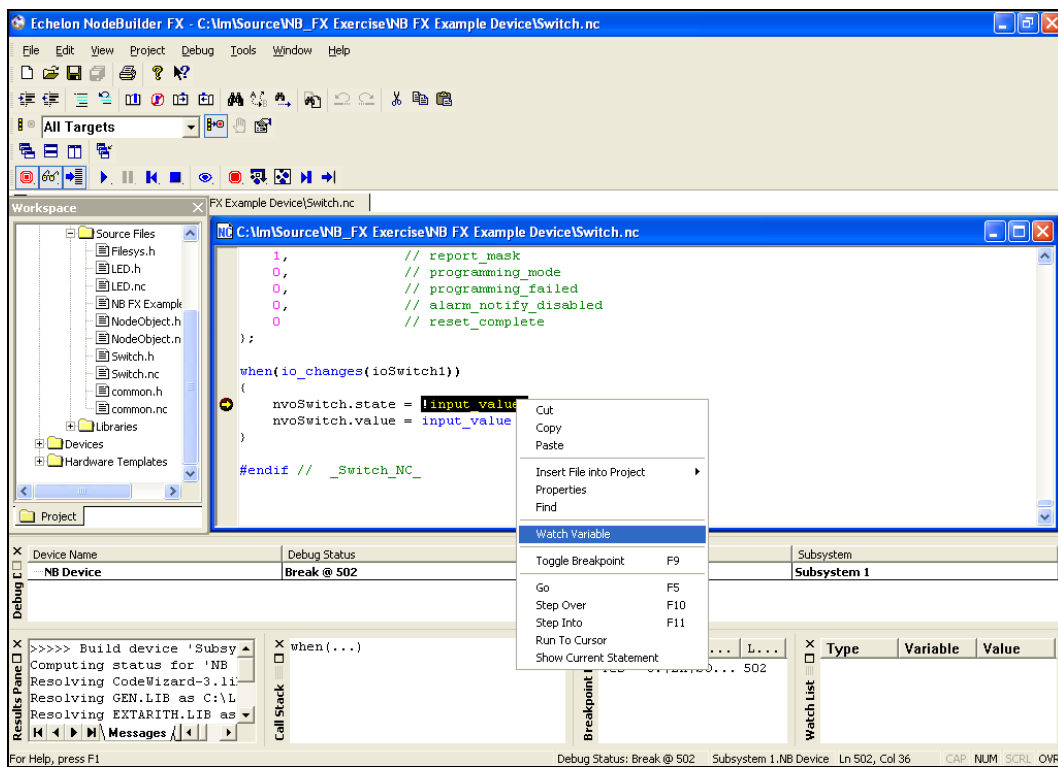
C:\m\Source\NB_FX_Exercise\NB_FX_Example_Device\Switch.nc
1,          // report_mask
0,          // programming_mode
0,          // programming_failed
0,          // alarm_notify_disabled
0,          // reset_complete
};

when(io_changes(ioSwitch1))
{
  nvoSwitch.state = !input_value;
  nvoSwitch.value = input_value ? 0 : 200u;
}

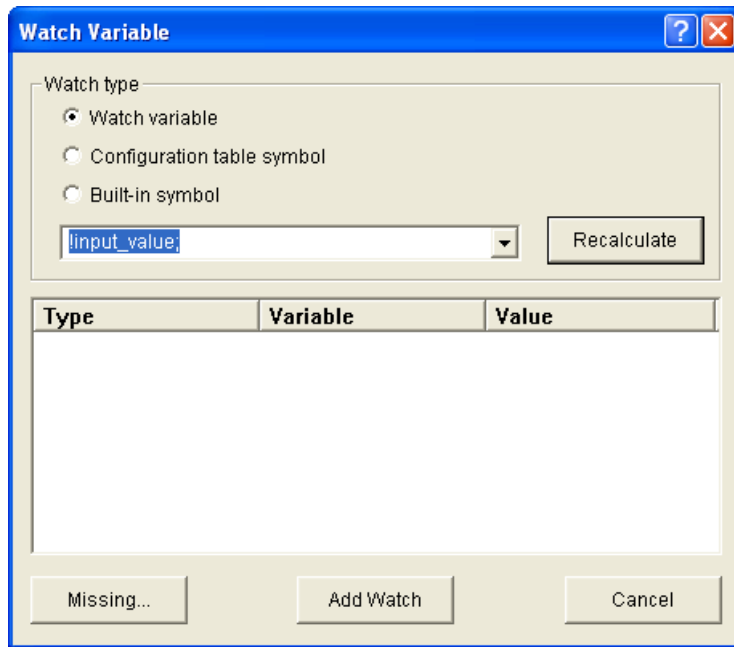
#endif // _Switch_NC_



```

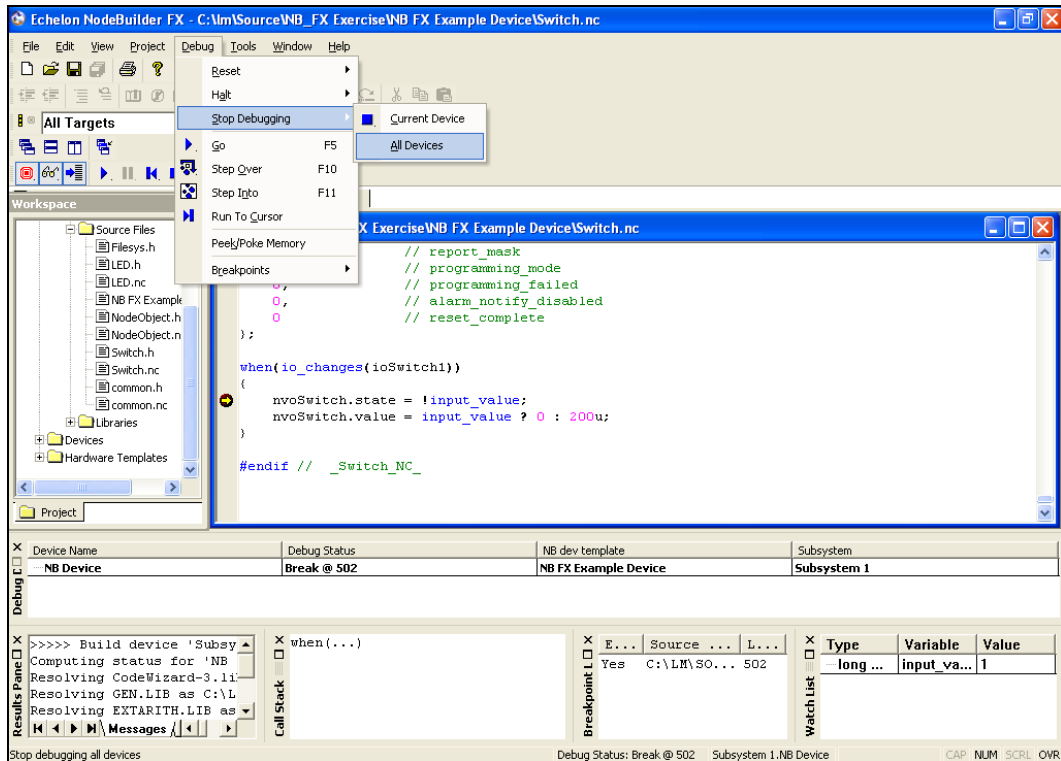
- Right-click the `input_value` variable in the line of code in which you set the breakpoint, and then click **Watch Variable** on the shortcut menu.



- The **Watch Variable** dialog opens.



11. Click **Add Watch**. The variable is added to the Watch List pane at the bottom of the NodeBuilder Project Manager. This pane displays each of the variables added to the watch list and their current values.
12. Click the **Step Into** button () in the debug toolbar to step through the code in the function until you reach the end of the when clause. The `input_value` variable is **0**.
13. Click the **Step Into** button to observe that the function executes a second time. The `input_value` variable is now **1**.
14. Click the **Resume** button () in the debug toolbar. Your device application resumes normal execution.
15. Click **Debug**, point to **Stop Debugging**, and then select **All Devices**.





16. The NodeBuilder debugger has demonstrated that events occur when the button is both pressed and released. To implement the desired behavior in which an event occurs only when the button is pressed, change the following lines of code in the **Switch.nc** file:

```
nvoSwitch.state = !input_value;
nvoSwitch.value = input_value ? 200u : 0;
```

to the following:

```
if (!input_value) {
    nvoSwitch.state ^= 1;
    nvoSwitch.value = nvoSwitch.state ? 200u : 0;
}
```

17. Verify that the **Load after Build** option () is set.
18. Right-click the **IzoT NB Example Device** device template in the Project pane, then click **Build** on the shortcut menu. The IzoT NodeBuilder tool rebuilds the IzoT NB Example Device application and downloads it to all devices using the IzoT NB Example Device device template.
19. Right-click the **IzoT NB Example Device** device in your IzoT CT drawing, then click **Browse** on the shortcut menu to open the IzoT Browser. Verify that the **Monitor All** button () on the toolbar is enabled.
20. Press the left button at the bottom of your development board (**SW1** on the FT 6000 EVB; **IO_6** on the Gizmo 4 I/O Board) repeatedly. Observe that the button now acts as a toggle-switch—the value of the **nvoSwitch** network variable in the **Switch** functional block changes when you press the button, but it no longer changes when you release the button.
21. Proceed to the next section to install and test your device in an IzoT or LONWORKS network.

For more information on debugging Neuron C applications, see Chapter 10, *Debugging a Neuron C Application*.

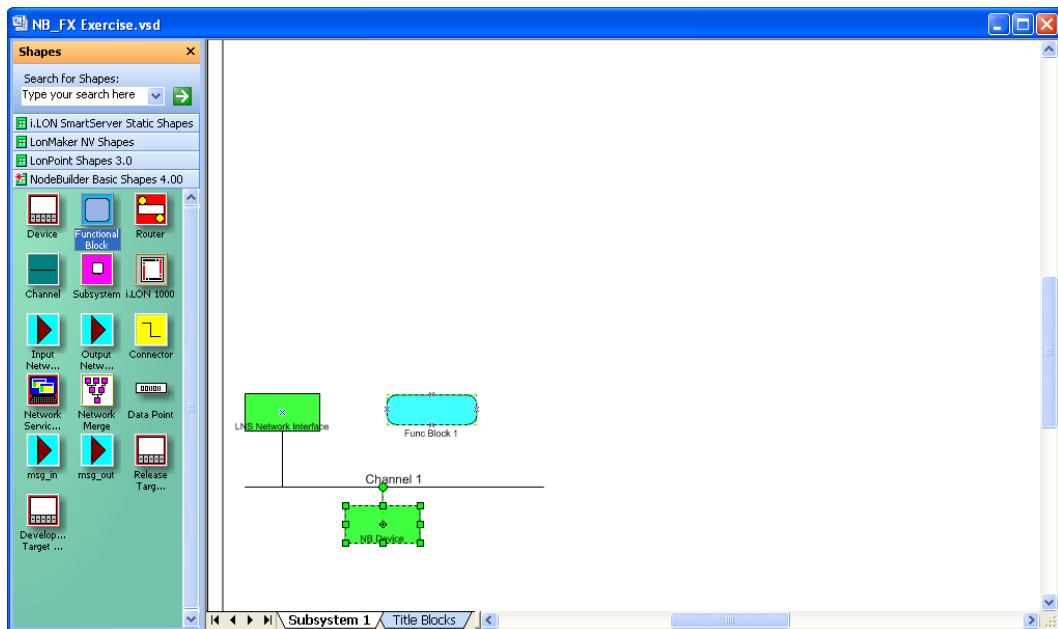
Step 8: Connecting and Testing the Device in a Network

Once you determine that your device is functioning as desired, you can test it as part of a network. You can use the IzoT Commissioning Tool to connect your development devices to other devices and verify their operation within a network. This entails creating functional blocks, connecting the network variables within the functional blocks, and verifying that the network variable values are updated appropriately when you use the I/O devices on your device.

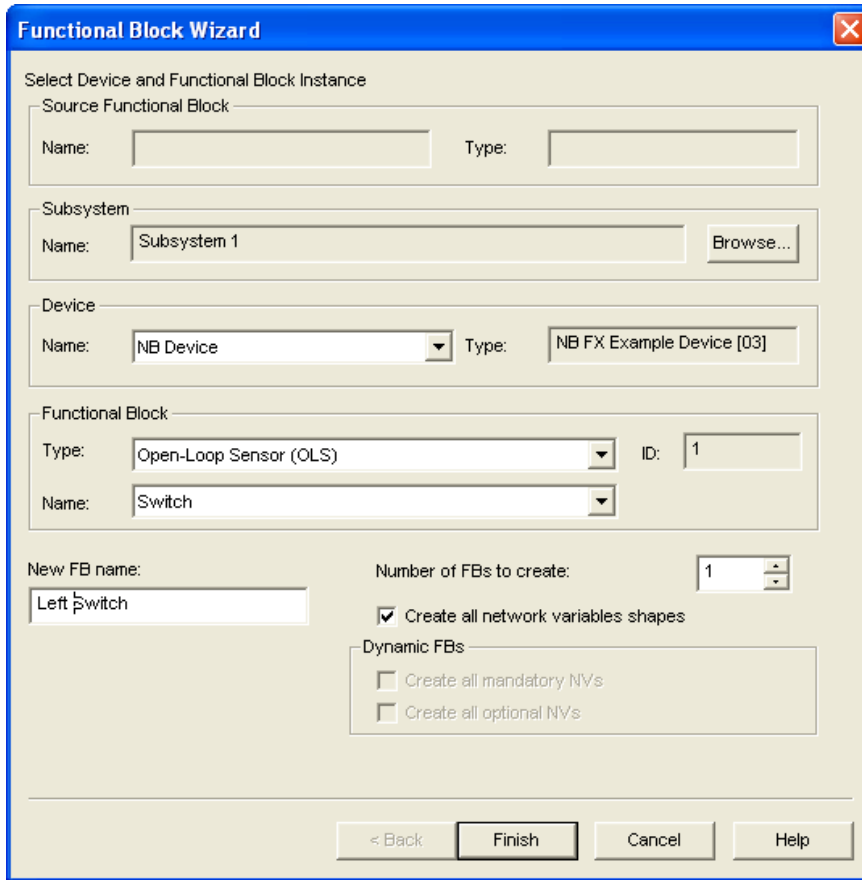
An output network variable of a device may be connected to compatible input network variables of the same device. These are called *turnaround connections*. For this exercise, you will create a turnaround connection so that a switch on your development board controls an LED. The procedure is the same 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:

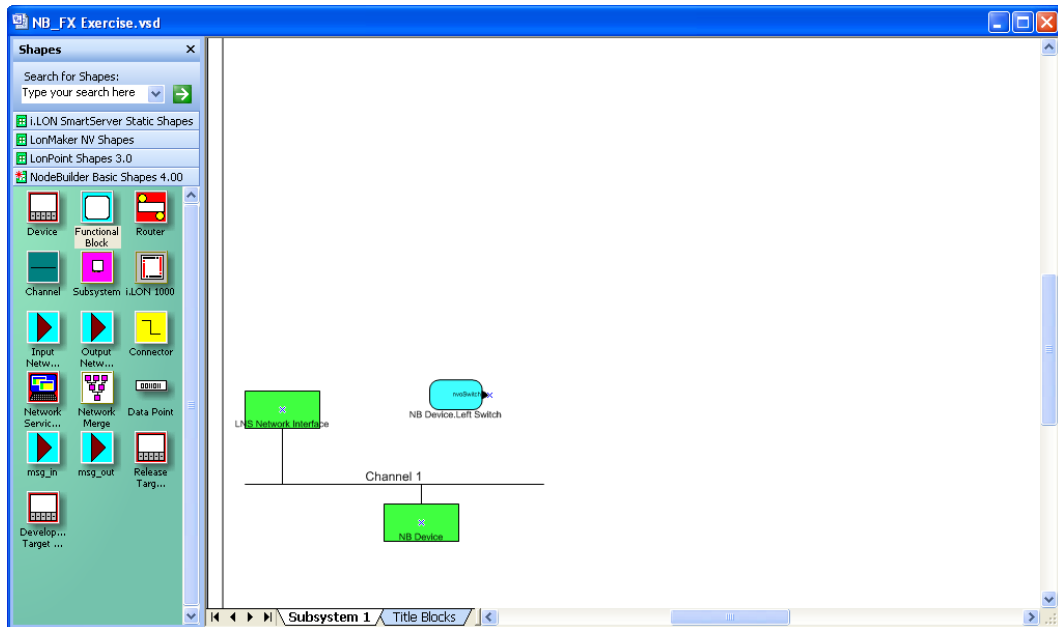
1. Click the Echelon IzoT CT/Visio button in the Taskbar to switch to the IzoT Commissioning Tool.
2. Drag a **Functional Block** shape from the **NodeBuilder Basic Shapes 4.00** stencil on the left of the IzoT CT window to the drawing.



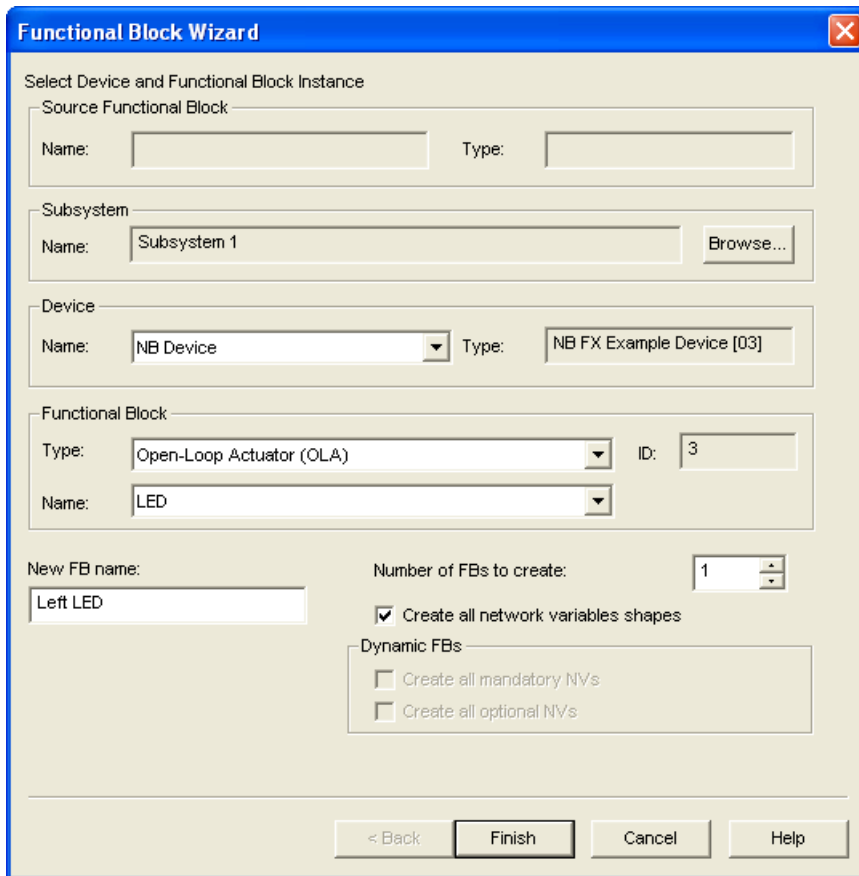
3. The Functional Block wizard opens. You will use this wizard to associate the new functional block shape with the **NB Device** device and the **Switch** functional block.
4. In the Functional Block wizard, do the following:
 - a. In the **Name** property under **Device**, select **NB Device** if it is not already selected.
 - b. In the **Name** property under **Functional Block**, select **Switch**.
 - c. In the **New FB Name:** property, enter **Left Switch**.
 - d. Select the **Create All Network Variable Shapes** check box.



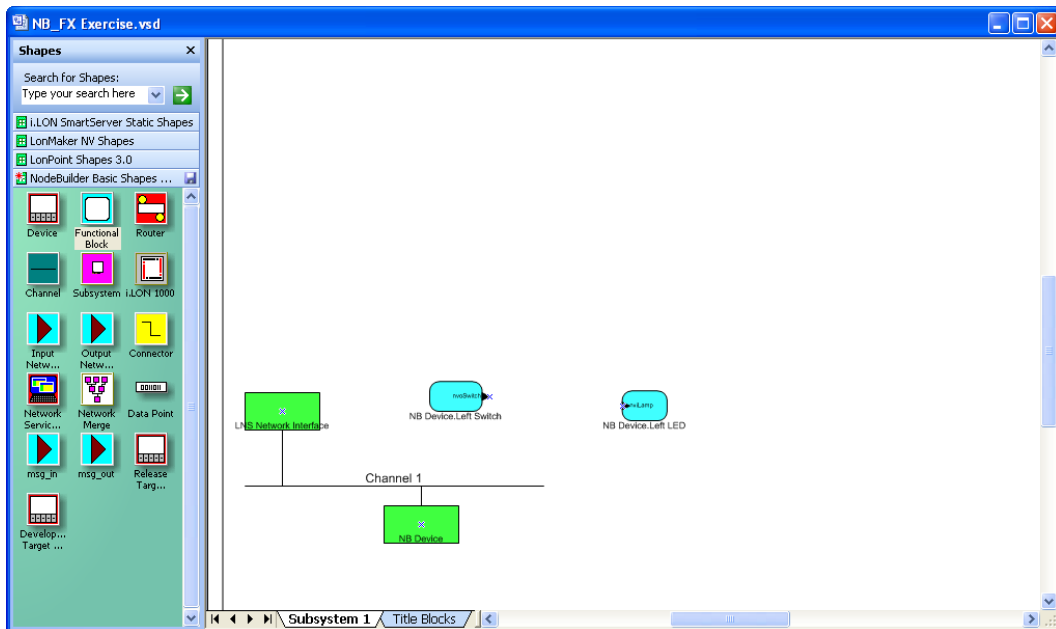
5. Click **Finish**. The New Functional Block wizard closes and the IzoT CT drawing appears. A new **Left Switch** functional block shape appears on the drawing.



6. Repeat steps 2–4 to create a new functional block shape named “Left LED”. In the **Name** property under **Functional Block** in the Functional Block Wizard, select **LED**. In the **New FB Name:** property, enter **Left LED**.

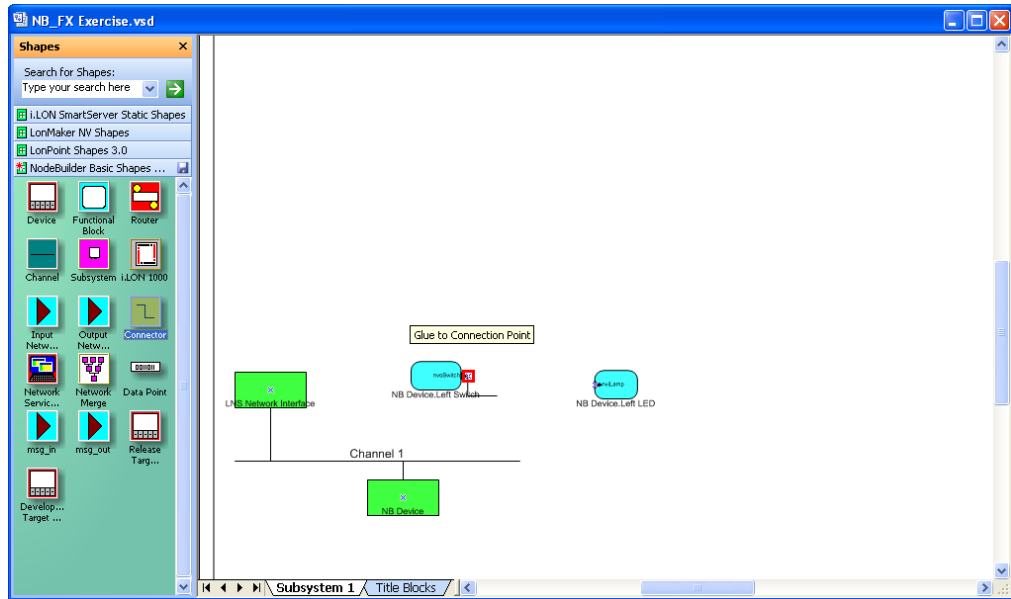


7. Click **Finish**. The New Functional Block wizard closes and the IzoT CT drawing appears. A new **Left LED** functional block shape appears on the drawing.

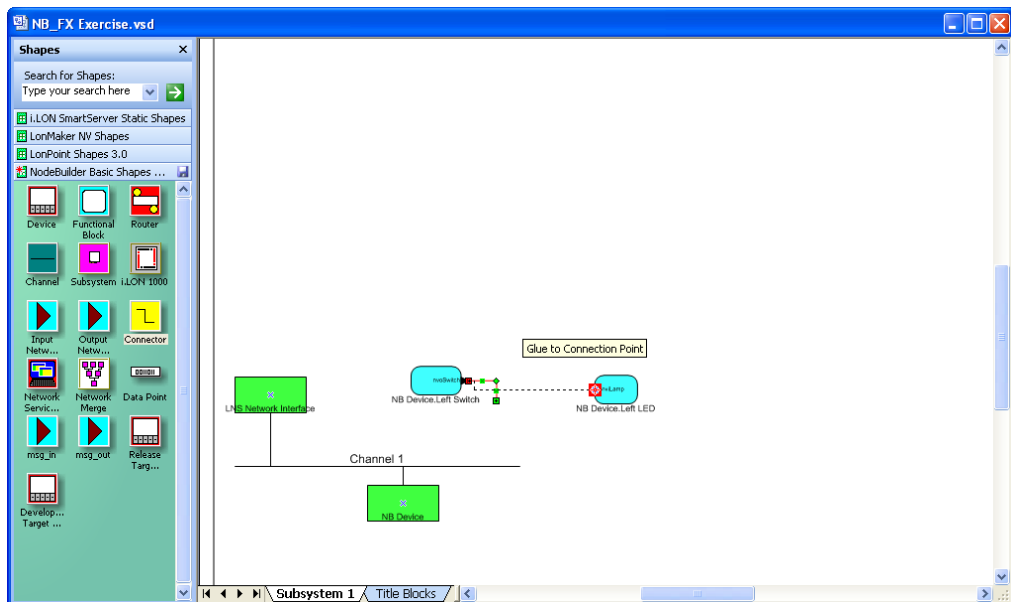


8. Connect the **nvoSwitch** output network variable of the **Left Switch** functional block to the **nviLamp** input network variable of the **Left LED** functional block. To do this follow these steps:

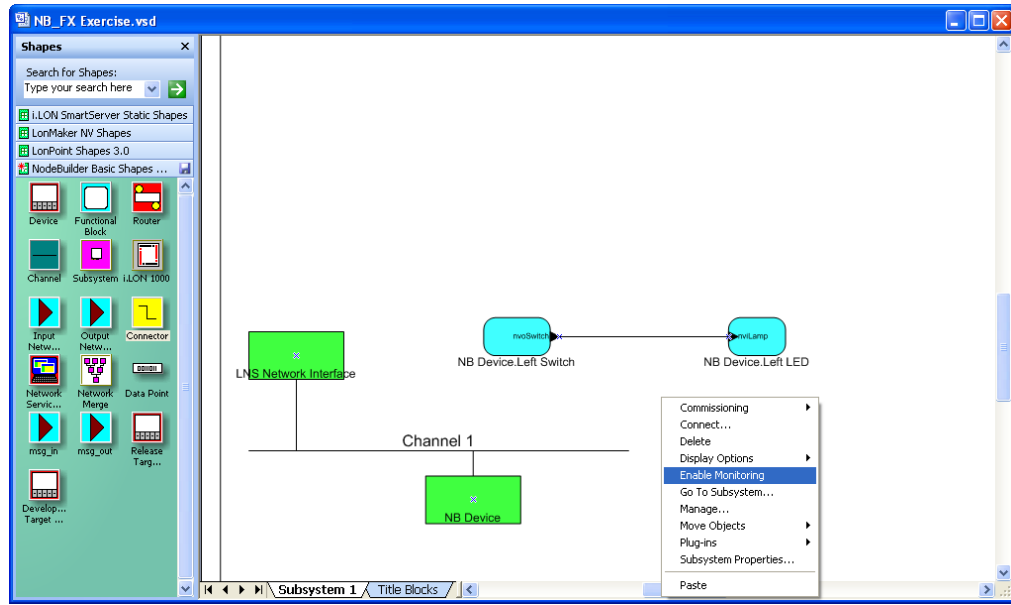
- a. Drag the **Connector** shape from the **NodeBuilder Basic Shapes 4.00** stencil to the drawing. Position the left end of the shape over the tip of the **nvoSwitch** output network variable on the **Left Switch** functional block before releasing the mouse button. A red box appears around the end of the **Connector** shape when you have positioned it correctly over the **Network Variable** shape.



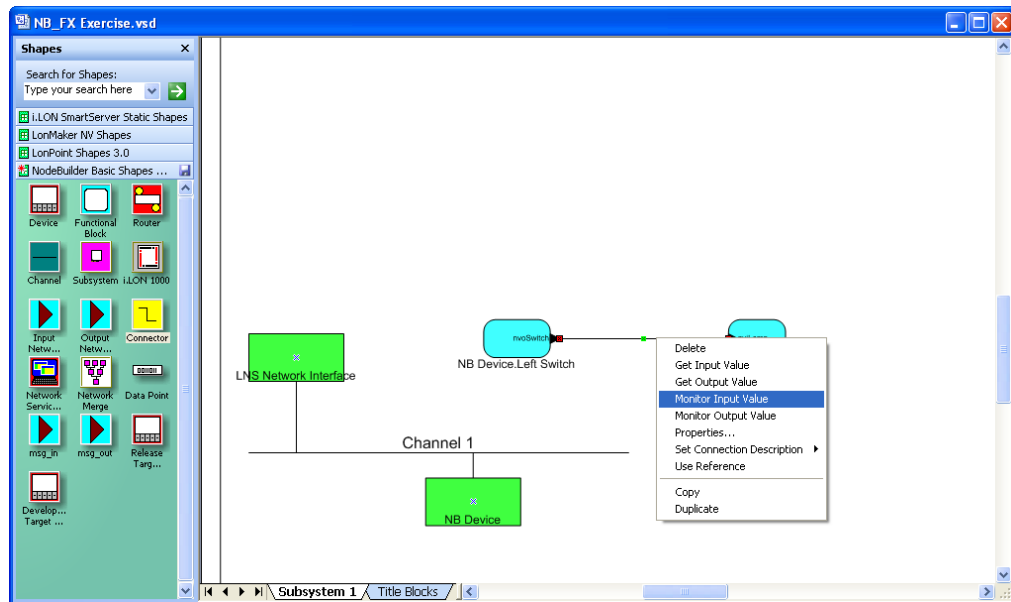
- b. Drag the other end of the **Connector** shape to the **nviLamp** input network variable of the **Left LED** functional block until it snaps into place and a square box appears around the end of the **Connector** shape. There is a brief pause as the IzoT Commissioning Tool updates the **NB Device** device over the network.



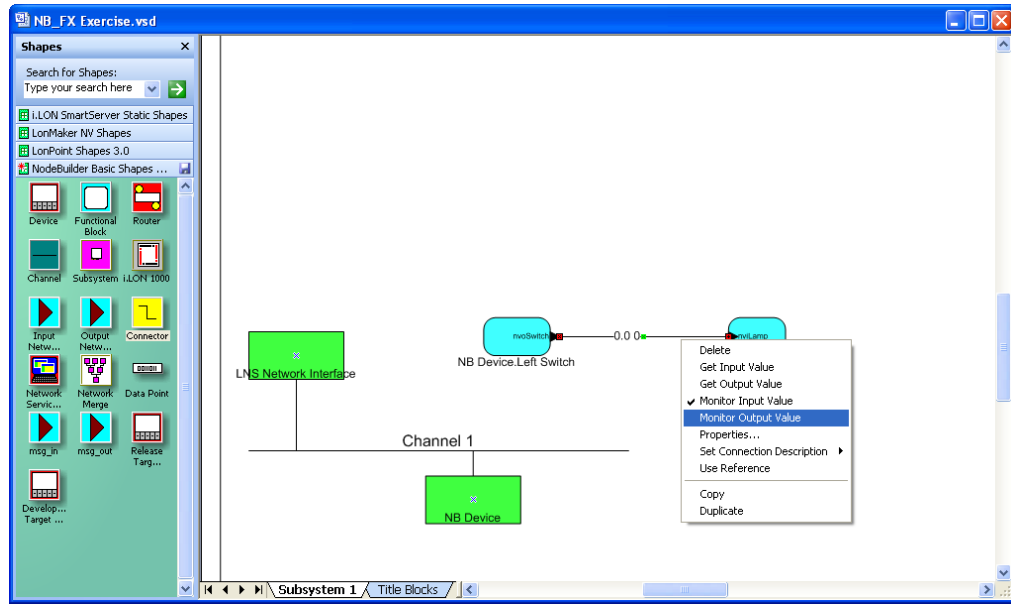
9. Monitor the values of the **nvoSwitch** output network variable of the **Left Switch** functional block and the **nviLamp** input network variable of the **Left LED** functional block. To do this, follow these steps:
 - a. Right-click an empty space in the IzoT CT drawing and then select **Enable Monitoring** on the shortcut menu.



- b. Right-click the new **Connector** shape and select **Monitor Input Value** to display the current value of the **nvoSwitch** network variable on the **Left Switch** functional block.

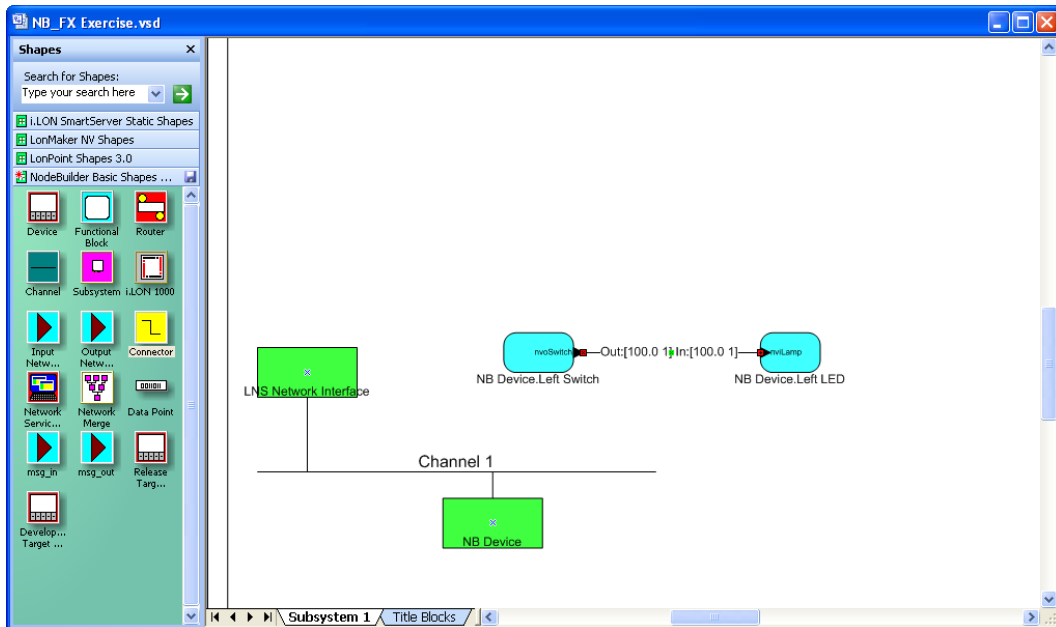


- c. Right-click the new **Connector** shape and select **Monitor Output Value** to display the current value of the **nviLamp** network variable on the **Left LED** functional block.



10. Press the left button at the bottom of your development board (SW1 on the FT 6000 EVB; IO_6 on the Gizmo 4 I/O Board) repeatedly to test the connection between the **nvoSwitch** output network variable of the **Left Switch** functional block and the **nviLamp** input network variable of the **Left LED** functional block.

Observe that the left LED at the bottom of your development board (LED1 on the FT 6000 EVB; IO_0 on the Gizmo 4 I/O Board) turns on and off each time you press the left button on your development board. In addition, the current values of the output and input network variable on the **Connector** shape toggle between 100.0 1 and 0.0 0 each time you press the button.



For more information on testing NodeBuilder devices in a LONWORKS network, see Chapter 9 *Testing a NodeBuilder Device Using the IzoT Commissioning Tool*.

Additional Device Development Steps

After you create your device application and successfully test your device in a network, you can perform the following additional steps in the device development process, which are summarized in the following sections:

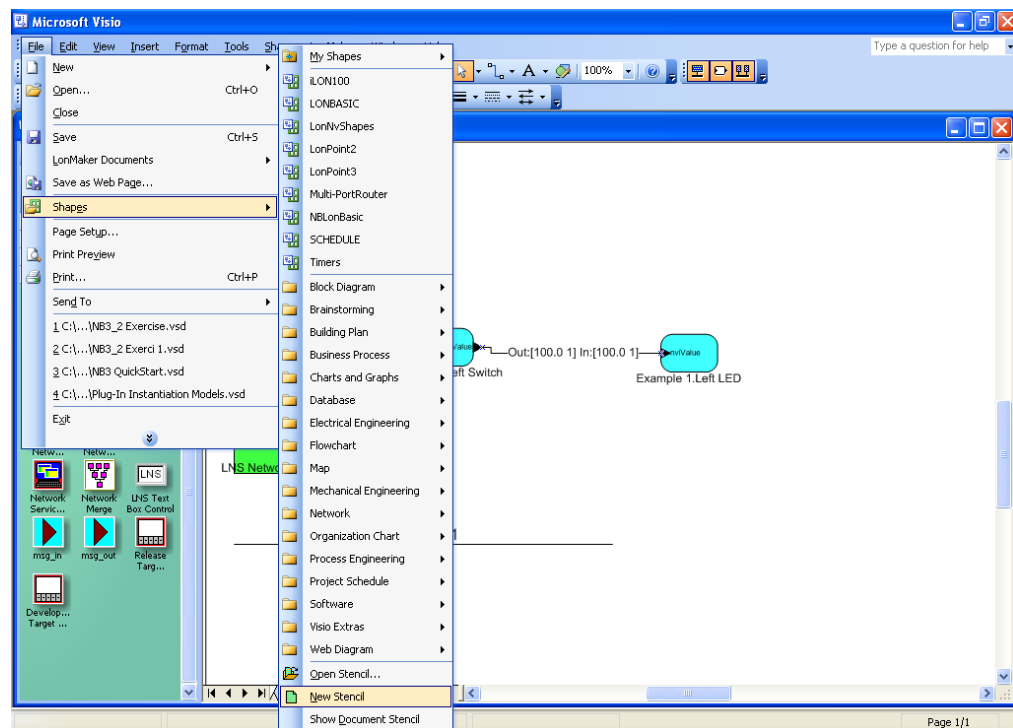
- Create an IzoT CT stencil.
- Create an IzoT device plug-in.
- Create an HMI.
- Create a device installation application
- Apply for LONMARK certification for your device.

Creating an IzoT CT Stencil

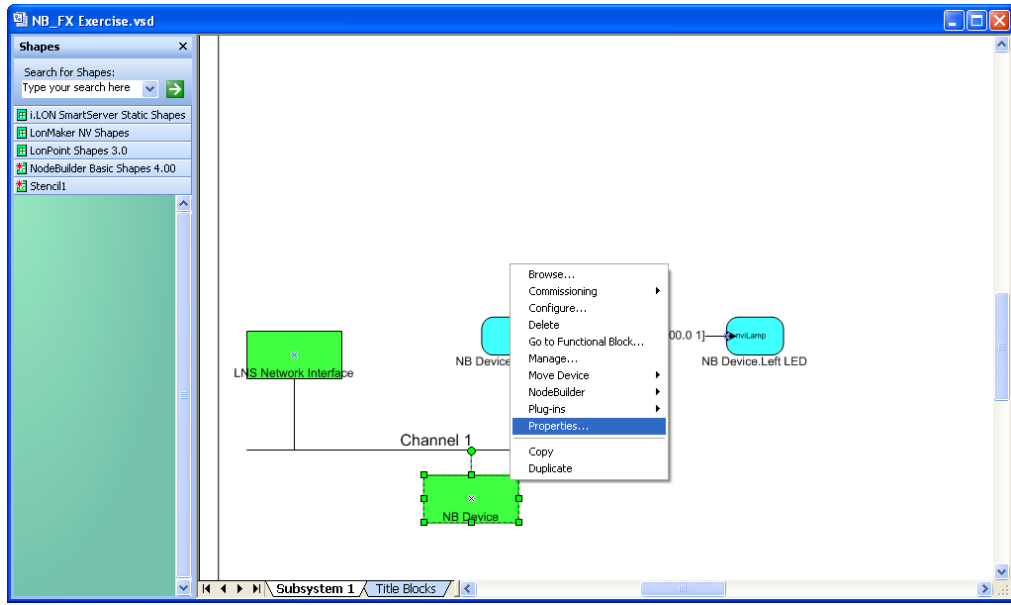
You can create a IzoT CT stencil for your device to make it easier for network integrators to install. A IzoT CT stencil should contain a custom IzoT CT shape for your device and for each functional block in the device interface. These custom shapes can then be provided to network integrators so that they can quickly integrate your device into their LONWORKS networks using the IzoT Commissioning Tool.

To create a IzoT CT stencil for your device, you do the following:

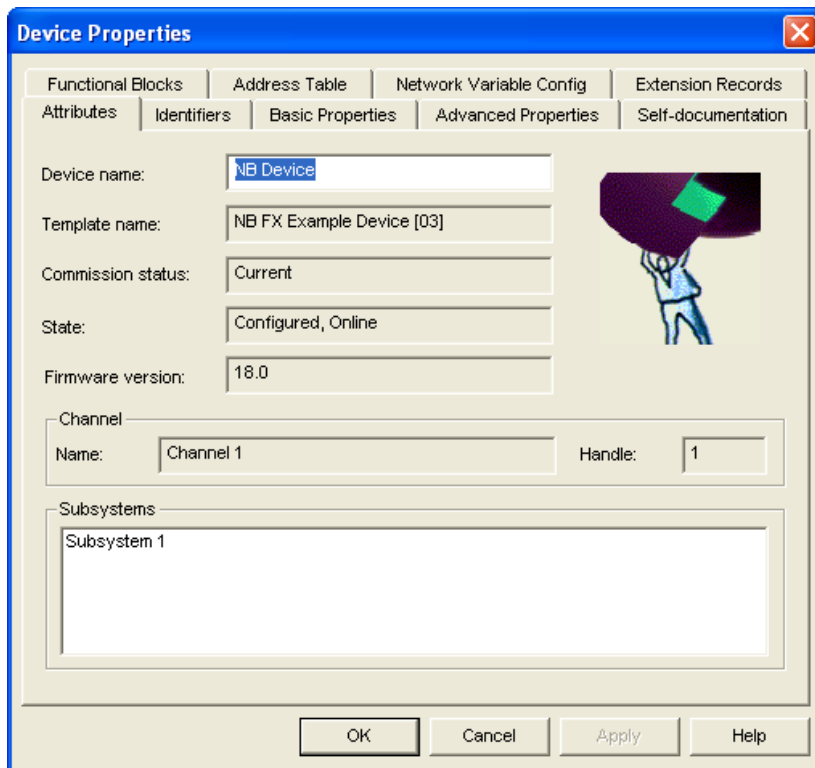
1. Create a new IzoT CT stencil. To do this follow these steps:
 - a. Open the IzoT CT drawing containing the NodeBuilder device for which you want to make custom shapes.
 - b. Click **File**, point to **Stencils**, and then click **New Stencil**.



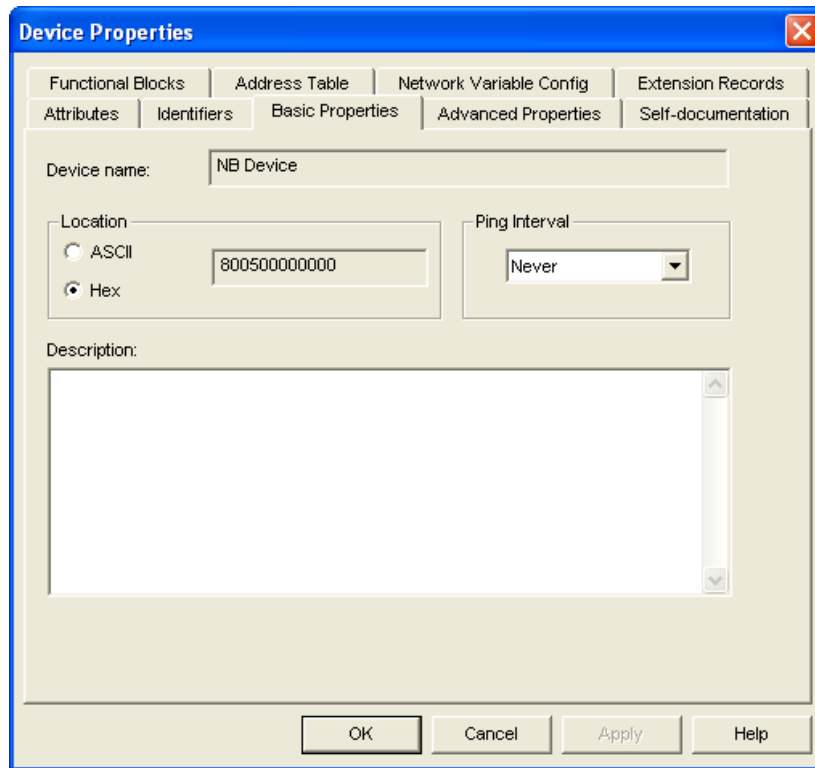
- c. A blank IzoT CT stencil named **Stencil** is added to the **Shapes** window.
2. Create a custom device shape. To do this follow these steps:
 - a. Right-click the NodeBuilder device in the IzoT CT drawing page and then select **Properties** on the shortcut menu.



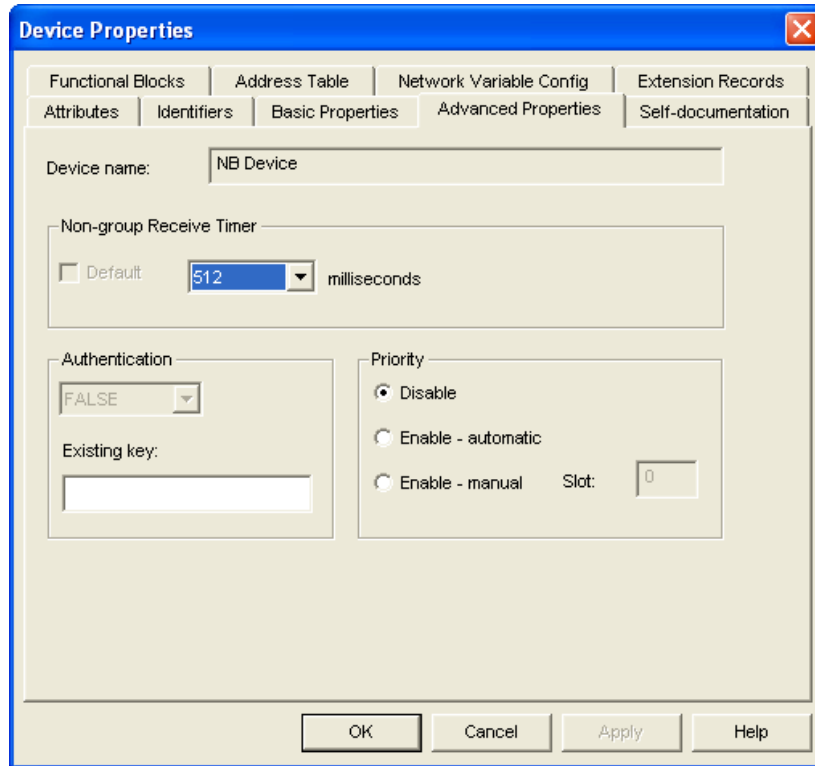
- b. The **Device Properties** dialog opens with the **Attributes** tab selected. This dialog allows you to read and write to the properties of the IzoT CT device.



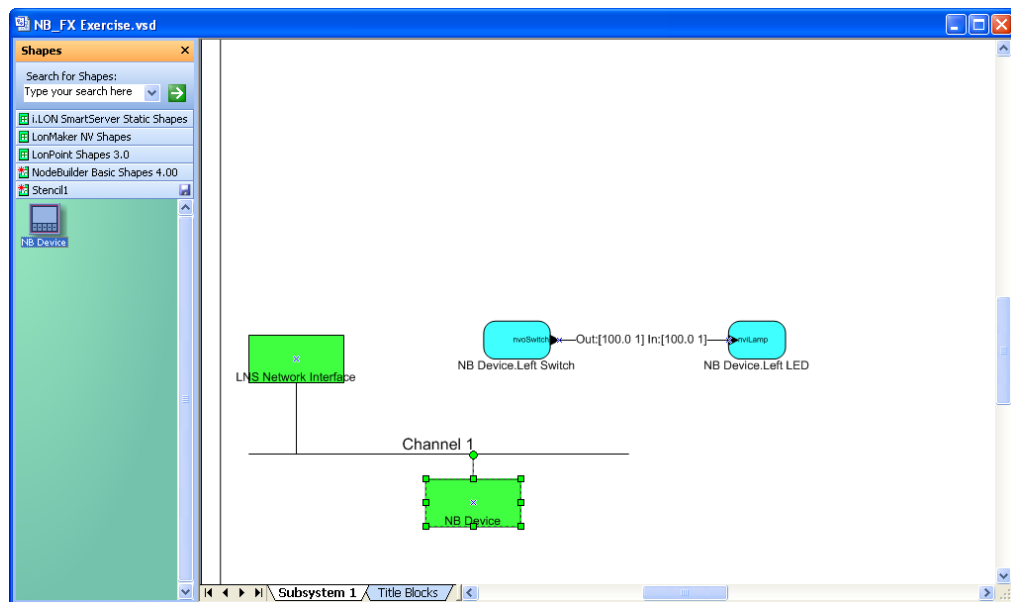
- c. In the **Device Name** property, enter the name to be shown for the custom device shape in your IzoT CT stencil.
- d. Click the **Basic Properties** tab.



- e. Set the **Location** and **Ping Interval** properties to the values to be saved with the custom device shape in your IzoT CT stencil. See the IzoT CT online help file for more information on these properties. Note that changes made to the **Description** are not saved in the custom device shape.
- f. Click the **Advanced Properties** tab.

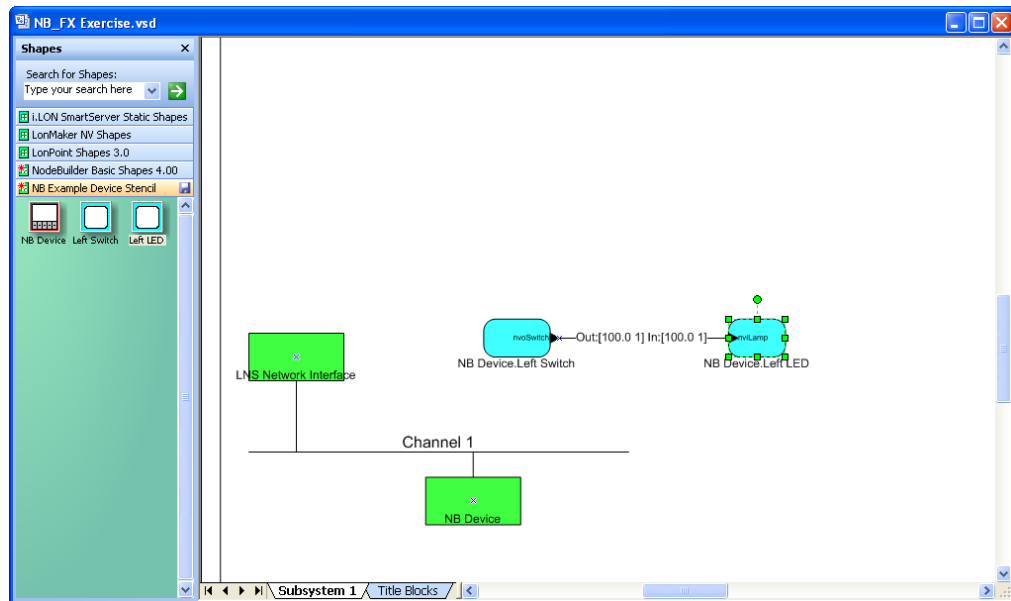


- g. Set the **Non-group Receive Timer** property to the value to be saved with the custom device shape in your IzoT CT stencil. See the IzoT CT online help file for more information on this property.
- h. Click **OK**.
- i. Drag your NodeBuilder device to your IzoT CT stencil. A new custom IzoT CT master shape with the device name specified in step c appears in the stencil.



- j. Click the disk icon (📁) on the stencil's title bar. Specify a name and location for your IzoT CT stencil file (.vss extension), and then save your IzoT CT stencil.

3. Create custom functional block shapes. Custom functional block shapes let you provide network integrators with functional block shapes that have built-in network variable shapes. To do this follow these steps:
 - a. Verify that functional block shapes for each functional block defined by the device interface have been added to the IzoT CT drawing. To create a functional block shape, drag a **Functional Block** shape from the **NodeBuilder Basic Shapes 4.00** stencil on the left of the IzoT CT window to the drawing, and then complete the Functional Block wizard.
 - b. Configure the default network variable and configuration property values for the custom functional block using the IzoT browser or an IzoT device plug-in (if you have created one for your device). You can create several versions of the same functional block for different configurations of that functional block.
 - c. Drag each functional block shape to your IzoT CT stencil. New custom IzoT CT master shapes with the functional block names specified in the Functional Block wizard appear in the stencil.



- d. Click the disk icon (📁) on the stencil's title bar to save your IzoT CT stencil.

Note: Custom IzoT CT shapes can contain multiple functional blocks, devices, and connections. For example, you can create custom IzoT CT shapes for two connected functional blocks, or for a device and all of its configured functional blocks. To do this, select multiple shapes and drag and drop them to a custom stencil. See the *IzoT Commissioning Tool User's Guide* for more information on creating complex custom IzoT CT shapes.

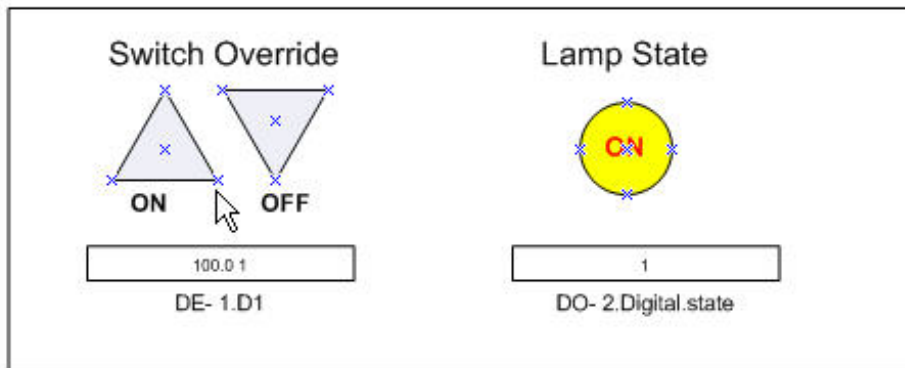
Creating an IzoT Device Plug-in

You can create an IzoT device plug-in to simplify and automate the installation of your devices for network integrators. An IzoTdevice plug-in is an application that implements the IzoTPlug-in API. IzoTdevice plug-ins are typically written in a .NET programming language such as C# or Visual Basic .NET, but you can write an IzoTdevice plug-in in any development environment that allows the creation of an (COM) automation server for Windows. For more information on writing IzoTdevice plug-ins and the IzoTPlug-in API, see the *OpenLNS Plug-in Programmer's Guide*.

Developing an HMI

You can create a human machine interface (HMI) for your device so that end users can monitor and control it. You will typically create an HMI if you are building a complete system that requires one; however, if your device is installed by integrators where each installation is unique, the integrators will typically develop the required HMIs.

You can use the IzoT Commissioning Tool to design a simple HMI for your device. With the IzoT Commissioning Tool, you use the data point shape in the IzoT CT Basic Shapes stencil and standard Visio shapes to create the HMI. For example, you can create an HMI that displays the current state of a lamp and provides override switches that let you manually turn the lamp on and off. For more information on creating HMIs with data point shapes, see Chapter 6 of the *IzoT Commissioning Tool User's Guide*.



You can use high-end HMI tools, such as Wonderware's InTouch or Intellution FIX, to represent 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.

Creating a Device Installation Application

You can create an installation executable that automatically installs all the files required by your device into the appropriate locations on your customers' computers. The files that your application should install include the device application (if your device uses downloadable application memory), the device interface file, user-defined resource files, the IzoT CT stencil, the IzoT device plug-in, and the HMI. Typically, the installation executable is created using an installation application such as the InstallShield® product.

If your device will be installed in a managed network (as opposed to a self-installed network), your customers must have an IzoT, OpenLNS, or an LNS network tool such as the IzoT Commissioning Tool already installed on their computers. Installing an IzoT, OpenLNS, or an LNS network tool creates a **LonWorks** folder that is stored by default in the root directory or program files directory on the user's computer (for example, **C:\Program Files (x86)\LonWorks**). The user, however, can change the location of the **LonWorks** folder when they are installing the IzoT, OpenLNS, or LNS tool. You can locate the **LonWorks** folder in the Windows registry at the following location:

HKEY_LOCAL_MACHINE\SOFTWARE\LonWorks\LonWorks Path

The following table lists and describes the files that your installation application should install:

*Programmable
Application Image
Files*
(.APB and .NXE)

The IzoT Commissioning Tool and other IzoT, OpenLNS, and LNS network tools use programmable application image files to download the compiled application image to a device. The programmable application image files have .APB, .NDL, and .NXE extensions.

On an IzoT NodeBuilder computer, the programmable application image files are stored in the **Development** or **Release** target folder within the device template folder. For example, the application image files for the development target in the quick-start exercise in this chapter are stored in the

C:\Users\Public\Documents\LonWorks\OpenLnsCt\Source\IzoT NB Exercise\IzoT NB Example Device\Development folder.

Your installation executable must install the .APB files. The .NDL file is used to support manufacture-time loading of devices and therefore does not need to be installed; the .NXE file is used to support legacy network tools and is usually not required. The .APB file should be installed in a folder where it can be found by the IzoT network tool on the target computer. For the IzoT Commissioning Tool, you can find this location in the Windows registry in the following location (by default, this location is **C:\Program Files (x86)\LonWorks\Import**):

```
HKEY_LOCAL_MACHINE\SOFTWARE\LonWorks\LonMaker for  
Windows\NxeSearchPath
```

Your installation executable should install your .APB file in a subdirectory labeled with your company name (**C:\Program Files (x86)\LonWorks\Import\YourCompany**, for example). Your installation should search for your company's folder and, if not found, it should create a folder with your company's name.

See *Building an Application Image* in Chapter 8 for more information on these programmable application image files.

Device Interface Files
(.XIF, .XFO*, and
.XFB)

* .XFO file is
optional.

The IzoT Commissioning Tool and other IzoT, OpenLNS, and LNS network tools use device interface files (also known as external interface files) to create IzoT device templates. Device interface files have .XIF, .XFO, and .XFB extensions.

On an IzoT NodeBuilder computer, the device interface files are stored in the same **Development** or **Release** target folder that contains the programmable application image files for the device.

Your installation executable must install the .XIF and .XFB files. Installing the .XFO file is optional; however, it speeds up device template importing for tools that support it such as the IzoT Commissioning Tool.

Your installation executable should install these device interface files in a folder where it can be found by the IzoT network tool on the target computer. For the IzoT Commissioning Tool, you can find this location in the Windows registry in the following location (by default, this location is **C:\Program Files (x86)\LonWorks\Import**):

```
HKEY_LOCAL_MACHINE\SOFTWARE\LonWorks\LonMaker for  
Windows\XifSearchPath
```


Your installation executable should install your device interface files in a subdirectory labeled with your company name (**C:\Program Files (x86)\LonWorks\Import\YourCompany**, for example). Your installation should search for your company's folder and, if not found, it should create a folder with your company's name.

See *Building an Application Image* in Chapter 8 for more information on these device interface files.

Device Resource Files
(**.TYP, .FMT, .FPT**)

Resource files are the files created by the NodeBuilder Resource Editor that contain network variable and configuration property type information and functional profile definitions. You must install all resource files that are used by your device.

The location of the resource files on the IzoT NodeBuilder computer can be found by starting the resource editor and finding the folder that contains the resource file set you want to include in the installation.

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 file (language resource file extensions vary by language as described in the *NodeBuilder Resource Editor User's Guide*. Uninstalling a device should not remove manufacturer resource files because they may be used by other devices from the manufacturer.

Resource files should be installed to the **LonWorks\Types** folder, in a subdirectory labeled with your company name (**C:\Program Files (x86)\LonWorks\Types\YourCompany**, for example).

OpenLNS Device Plug-in

If you have created an IzoT device plug-in, it should be installed and registered by your installation. See the *OpenLNS Plug-in Programmer's Guide* for more information.

IzoT CT Stencil

If you have created an IzoT CT stencil containing custom shapes for your device, it should be installed in the **LonWorks\LonMaker\Visio** folder in a subdirectory labeled with your company name (**C:\Program Files (x86)\LonWorks\LonMaker\Visio\YourCompany**, for example). See *Creating a LonMaker Stencil* earlier in this section for more information.

HMI Application

If you have created an HMI for your device, it should be installed and registered. See the documentation for your installation creation software and your HMI development tool for more information on the steps this entails.

Applying for LONMARK Certification

LONMARK International is an independent, non-profit organization that oversees LONWORKS technology and related standards. 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 LONMARK standard and can be easily integrated into LONWORKS networks with other LONWORKS devices from multiple vendors. For information on having your device LONMARK certified, see the LONMARK Web site at www.lonmark.org.

Creating and Opening IzoT NodeBuilder Projects

This chapter describes how to create, open, and copy IzoT NodeBuilder projects, and how to copy NodeBuilder projects and NodeBuilder device templates to another computer.

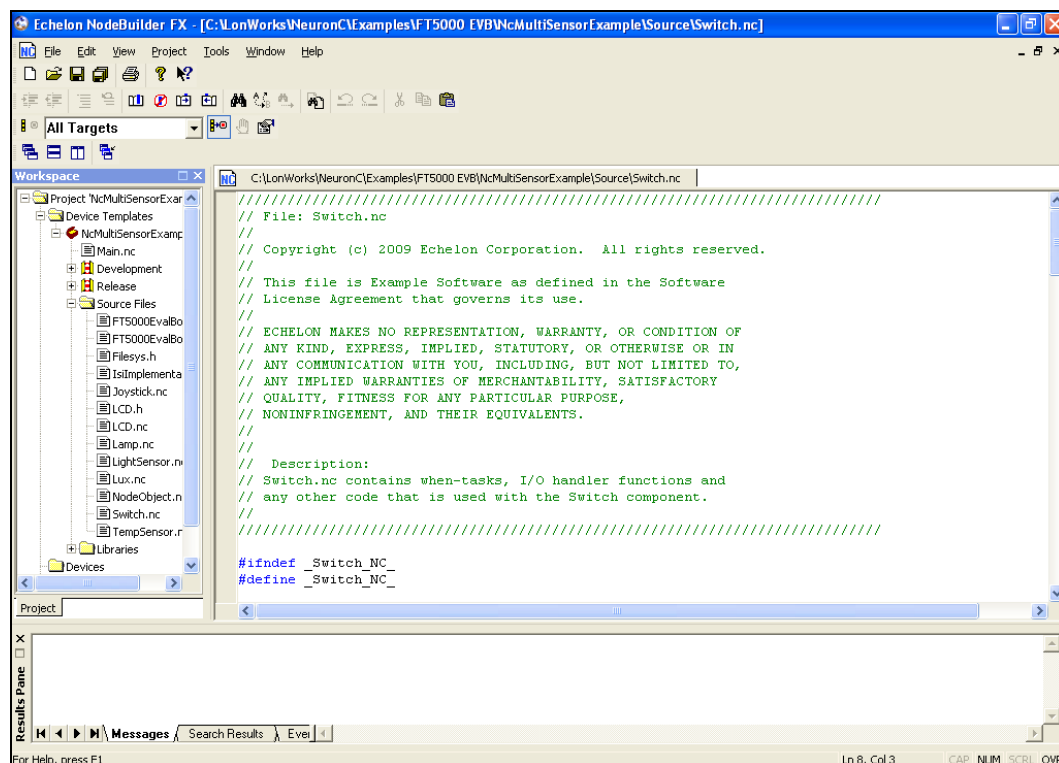
Introduction to the NodeBuilder Project Manager

A *NodeBuilder project* collects all the information about a set of devices that you are developing. 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, you start the NodeBuilder Project Manager from the IzoT Commissioning Tool or directly from the NodeBuilder program folder. You will typically start the project manager from the IzoT Commissioning Tool because it simplifies the process of associating the NodeBuilder project with the IzoT CT network.

You can use the same NodeBuilder project with multiple IzoT CT networks, and you can use a IzoT CT network with multiple NodeBuilder projects; however, you can only use a IzoT CT network with one NodeBuilder project at a time.

The NodeBuilder Project Manager initially contains three panes: the Project pane (left), the Edit pane (right), and the Results pane (bottom). These panes can all be moved and resized, and the Project and Results panes can be closed; however, the NodeBuilder Project Manager displays all the three panes by default.



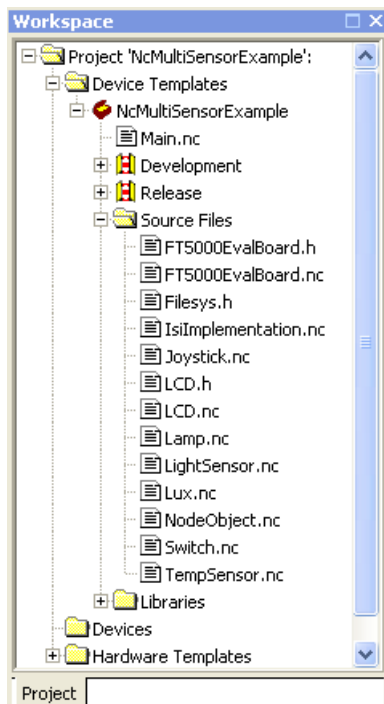
The following table describes the three panes in the NodeBuilder Project Manager:

Pane	Description
<i>Project</i>	Provides a hierarchical view of all the components in the NodeBuilder project. The Project pane lets you browse the files used in the NodeBuilder Project. See the following section for further description of the Project pane.

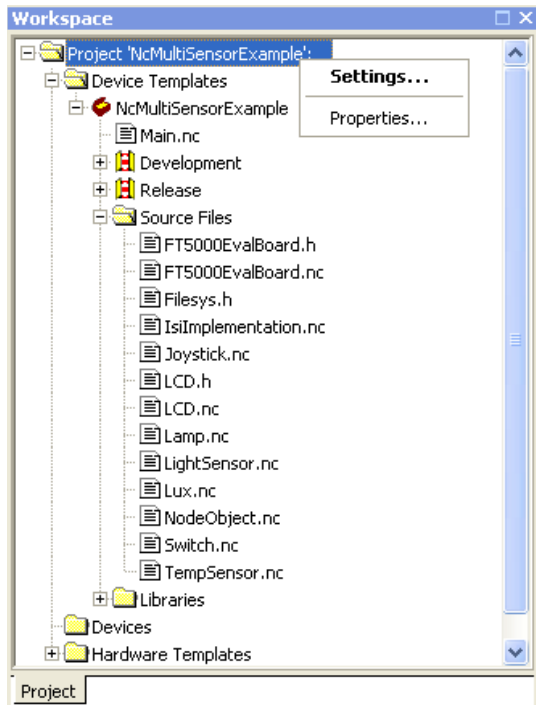
- Edit* Lets you to edit any of the Neuron C source files or header files that are used in the project. See Chapter 7, *Developing Device Applications*, for more information on using the Edit pane.
- Results* The Results pane contains three tabs: **Messages**, **Search Results**, and **Event Log**.
- The **Messages** tab displays compiler and other messages generated when you build the application image for a NodeBuilder device template. 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. See *Building an Application Image* in Chapter 8 for more information on using the Messages tab in the Results pane.
 - 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 Chapter 7 for more information on using the Search Results tab in the Results pane.
 - The **Event Log** contains debugger event messages. See Chapter 10, *Debugging a Neuron C Application*, for more information on using the **Event Log** tab in the Results pane.

Using the Project Pane

The Project pane appears on the left side of the NodeBuilder Project Manager by default. The Project pane provides a hierarchical view of all the components in the NodeBuilder project. You can use the Project pane to browse and open the files in your NodeBuilder Project.



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



Settings

Opens the **NodeBuilder Project Properties** dialog with the **Project** tab selected. The **Project** tab displays the project settings.

Properties

Displays file properties of the NodeBuilder project file (.NbPrj extension). The properties include the file name, location, size, and the dates on which the file was created, last modified, and last accessed.

The Project folder may also contain the following three folders: **Device Templates**, **Devices**, and **Hardware Templates**.

- The **Device Templates** folder contains all of the device templates that have been created in this NodeBuilder project. See *Creating Device Templates* in Chapter 5 for more information on device templates.
- The **Devices** folder contains a list all devices in IzoT CT drawings that have been associated with device templates in this NodeBuilder project. See *Building an Application Image* in Chapter 8 for more information. Note that the **Devices** folder will not appear if the NodeBuilder project is not associated with an IzoT CT network.
- The **Hardware Templates** folder contains a list of the hardware templates available in this NodeBuilder project. See *Using Hardware Templates* in Chapter 5 for more information on hardware templates.

Creating a NodeBuilder Project

To create a NodeBuilder project, you must first start the NodeBuilder Project Manager. You can start the NodeBuilder Project Manager from the IzoT Commissioning Tool, or you can start it standalone directly from the NodeBuilder program folder. You will typically start the project manager from the IzoT Commissioning Tool because it simplifies the process of associating the NodeBuilder project with the IzoT CT network.

Creating a NodeBuilder Project from IzoT CT

You can create a NodeBuilder project by starting the NodeBuilder Project Manager from the IzoT Commissioning Tool. To do this, follow these steps:

1. Create or open an IzoT CT drawing. See the *IzoT Commissioning User's Guide* for more information on creating and opening IzoT CT drawings. If you will want to load the application you develop into a device, make sure the IzoT CT computer is attached to the network.
2. Click **Add-Ins**, click **OpenLNS CT**, and 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.

Note: If you have previously created a project for this network and you want to create a new project, click **File** and then click **Create Project**.

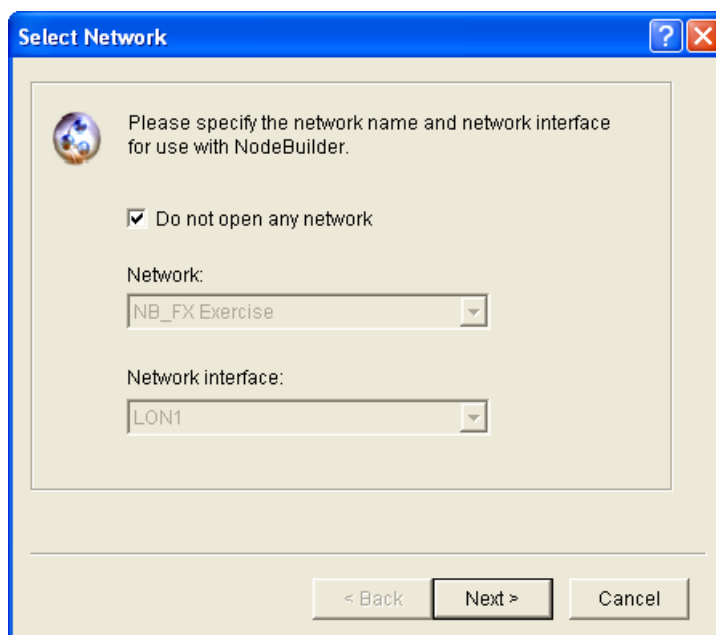
3. Enter project information into the wizard as described in steps 5–9 in the next section, *Creating a NodeBuilder Project from the NodeBuilder Project Manager*.

Note: You can also start the IzoT NodeBuilder tool from the IzoT Commissioning tool's New Device Wizard. See *Starting the IzoT NodeBuilder tool from the New Device Wizard* later in this chapter for more information on how to do this.

Creating a NodeBuilder Project from the NodeBuilder Project Manager

You can create a NodeBuilder project by starting the NodeBuilder Project Manager standalone. To do this, follow these steps:

1. Open the NodeBuilder Project Manager. To do this, click **Start** on the taskbar, point to **Programs**, point to **Echelon NodeBuilder**, and then click **NodeBuilder Development Tool**. The NodeBuilder Project Manager starts.
2. Click **File** and then click **Create Project**. The New Project wizard starts with the **Select Network** dialog.

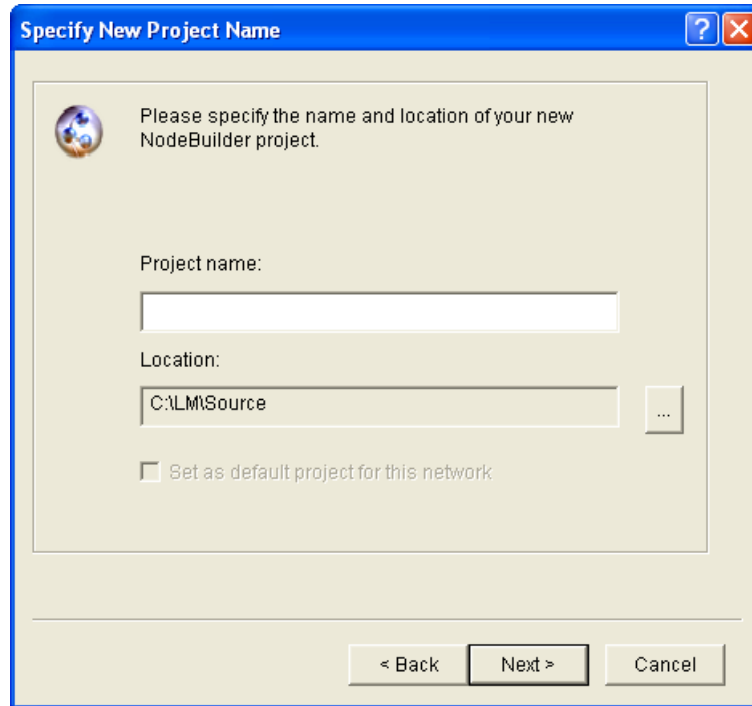


3. To associate an existing IzoT CT network with your NodeBuilder project, clear the **Do Not Open Any Network** check box if it is selected, select an existing IzoT CT network in the **Network**

property, and then select the IzoT network interface to be used for communication between the IzoT CT network and your NodeBuilder device in the **Network Interface** property.

Alternatively, you can select the **Do Not Open Any Network** check box to create a new project that is not associated with a IzoT CT network, and disable automatic IzoT device template creation and automatic load after build.

4. Click **Next**.
5. The **Specify New Project Name** dialog opens.

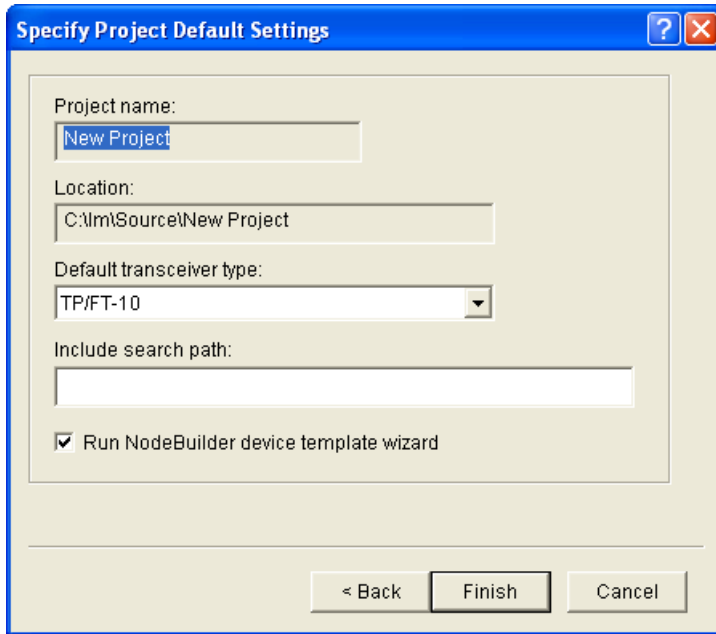


6. In the **Project Name** property, enter the name of your new NodeBuilder project. If you specified a IzoT CT network to be associated with the NodeBuilder project in the **Select Network** dialog, the default **Project Name** is that of the selected IzoT CT network. You can accept this default name or enter a new one.

Project files with this name and **.NbPrj**, **.NbOpt**, and **.NbWsp** extensions will be created in the project folder specified in the **Location** property. The project folder is stored in the **C:\Users\Public\Documents\LonWorks\OpenLnsCt \Source\<Project name>** folder by default. You can click the button to the right of the **Location** property to specify a different location.

If you specified a LonMaker network to be associated with the NodeBuilder project in the **Select Network** dialog, the **Set as Default Project** check box is selected. This means that this NodeBuilder project is automatically opened when the IzoT NodeBuilder tool is started from the selected IzoT CT network. If you selected the **Do Not Open Any Network** check box in the **Select Network** dialog, the **Set as Default Project** check box is unavailable.

7. Click **Next**. The **Specify Project Default Settings** dialog opens.



- Specify the following properties:

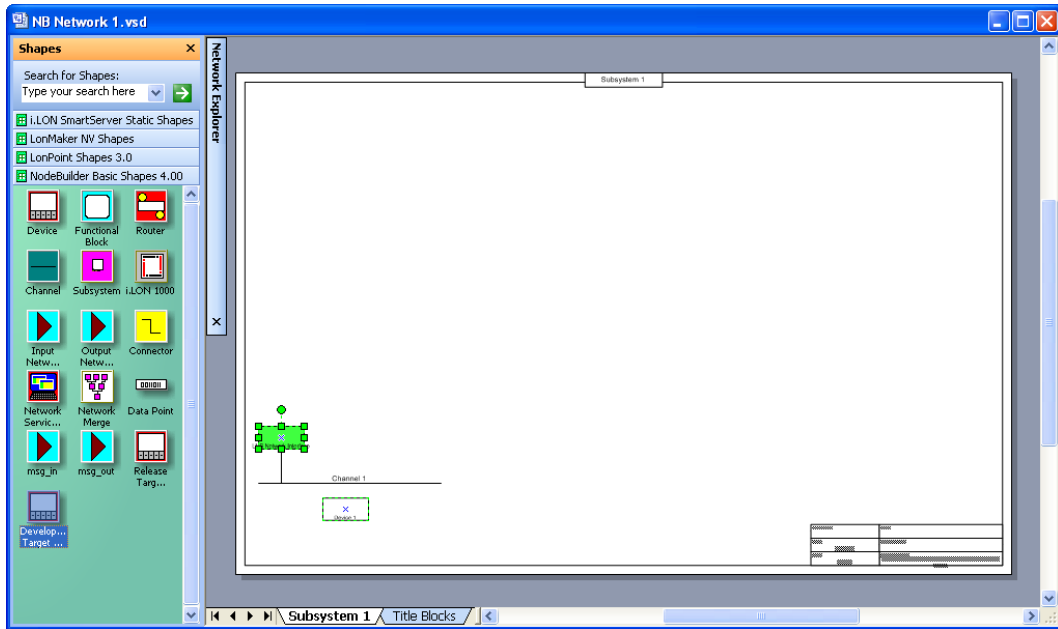
<i>Project Name</i>	The name of the project as specified in the Specify New Project Name dialog. This is a read-only field.
<i>Location</i>	The location of the project folder as specified in the Specify New Project Name dialog. This is a read-only field.
<i>Default Transceiver Type</i>	The transceiver type to be used for Hardware Templates that specify “default” for the transceiver type. The default transceiver type is TP/FT-10 . See the <i>Using Hardware Templates</i> section in Chapter 5 for more information on hardware templates.
<i>Include Search Path</i>	An optional semi-colon separated list of directories to be searched for include files when a NodeBuilder project is compiled. By default, only the device template source file and the Neuron C standard include 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 .NbPrj project file). Note that this list applies to the entire project. By default, this property is blank.
<i>Run Device Template Wizard</i>	Automatically opens the Device Template Wizard immediately after you click Finish . The Device Template Wizard guides you through the process of creating the first NodeBuilder device template for this project. See <i>Creating Device Templates</i> in Chapter 5 for more information. This option is selected by default.

- Click **Finish**. If you selected the **Run Device Template Wizard** check box in the **Specify Project Default Settings** dialog, the Device Template Wizard opens. Proceed to the *Specifying the Device Template Name* section in Chapter 5 to create a device template.

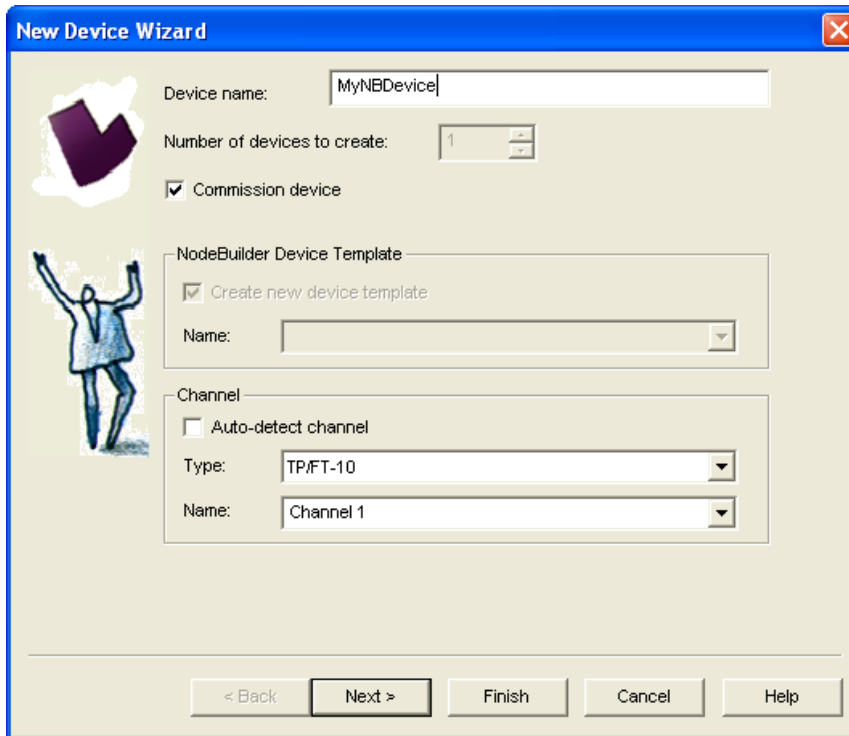
Creating a NodeBuilder Project from the New Device Wizard

You can create a NodeBuilder project from the New Device Wizard in the IzoT Commissioning Tool. To do this, follow these steps:

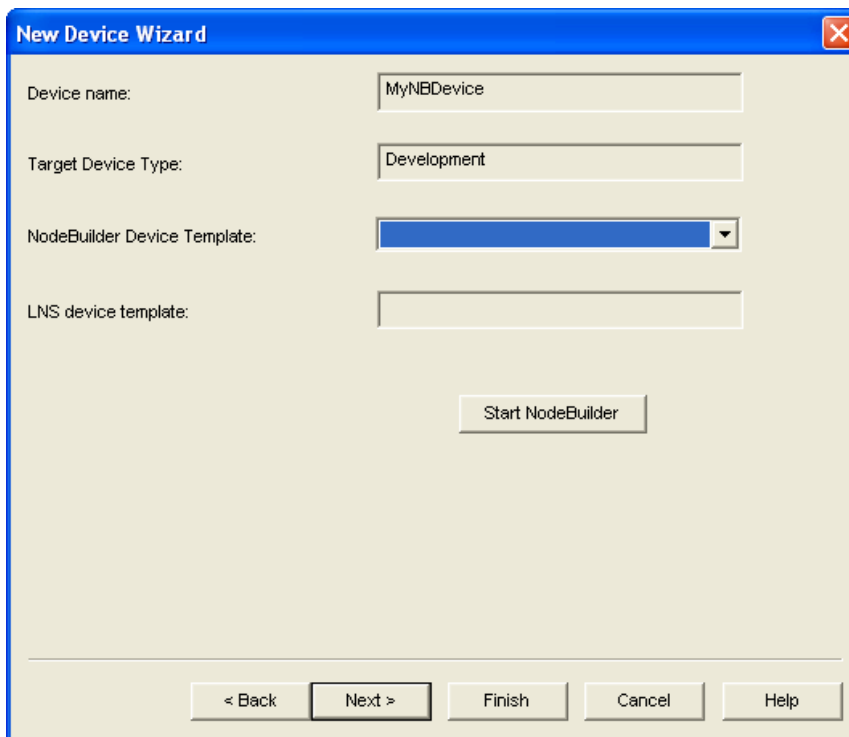
1. Create or open an IzoT CT network. See the *IzoT Commissioning User's Guide* for more information on creating and opening IzoT CT networks. If you plan on downloading your device application to a device, make sure that the IzoT CT computer is attached to the network.
2. Drag a **Development Target Device** or a **Release Target Device** shape from the **NodeBuilder Basic Shapes 4.00** stencil to your network drawing. Use a **Development Target Device** if you are building to a NodeBuilder hardware platform; use a **Release Target Device** if you are building to the release hardware. You can drop the shape anywhere, but a good location is just below the Channel 1 shape on your drawing.



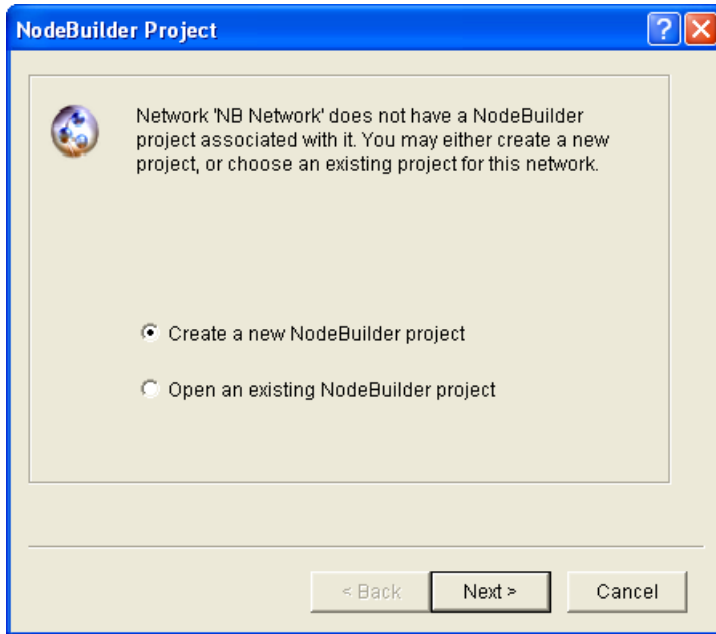
3. The New Device Wizard opens. In the **Device Name** property, enter the device name, select the **Commission Device** check box, and then select the **Create New Device Template** check box under **NodeBuilder Device Template**.



4. Click **Next**. The next page in the New Device Wizard lets you select the NodeBuilder device template.



5. Click **Start NodeBuilder** to create a new NodeBuilder project. The IzoT NodeBuilder tool starts automatically.
6. The New Project wizard opens.



7. Accept the default **Create a New NodeBuilder Project** option, and then click **Next**.
8. Accept the default NodeBuilder **Project Name**, which is the same name as the IzoT CT network, and then click **Next**.
9. Accept the defaults in the **Specify Default Project Settings** dialog, and then click **Finish**.
10. The NodeBuilder New Device Template wizard starts. Proceed to the *Specifying the Device Template Name* section in Chapter 5 to create a device template.

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 IzoT Commissioning Tool, or directly from the NodeBuilder program folder. You will typically start the project manager from the IzoT Commissioning Tool since that simplifies associating the NodeBuilder project with the IzoT CT network.

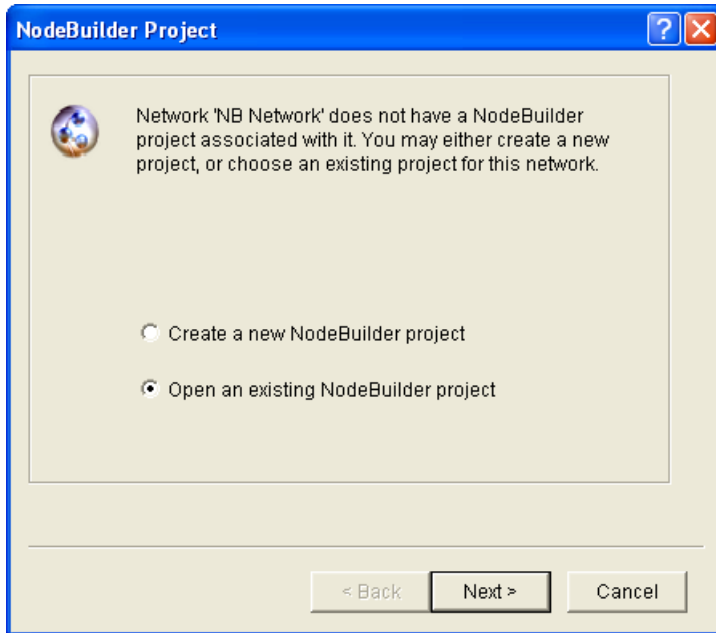
Opening a NodeBuilder Project from the IzoT Commissioning Tool

You can open a NodeBuilder project by starting the NodeBuilder Project Manager from the IzoT Commissioning Tool. To do this, follow these steps:

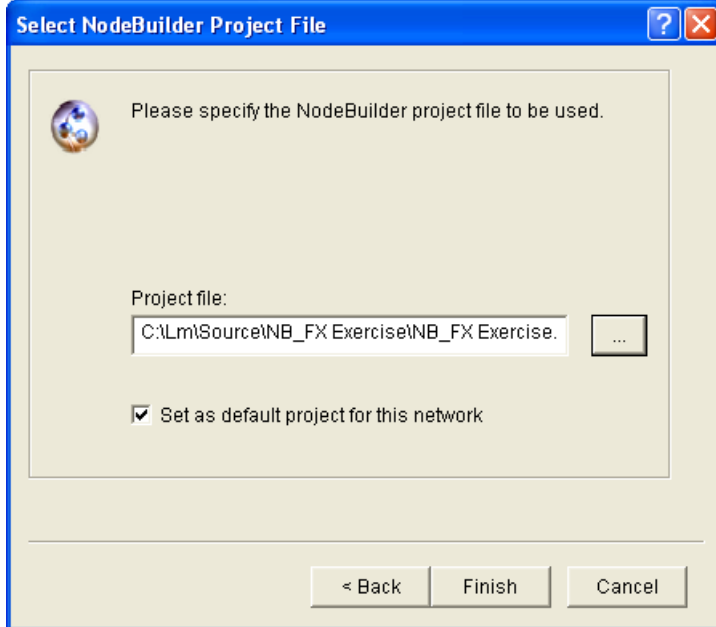
1. Create or open an IzoT CT drawing. See the *IzoT Commissioning User's Guide* for more information on creating and opening IzoT CT drawings. If you plan on downloading your device application to your device, make sure the IzoT CT computer is attached to the network.
2. Click **Add-Ins**, click **OpenLNS CT**, and 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 with the **NodeBuilder Project** dialog displayed.

Note: If you have previously created a NodeBuilder project for this network, the default project for the network opens. To open a different project, click **File**, click **Open Project**, and then skip to step 4.

3. In the **NodeBuilder Project** dialog, select the **Open an Existing NodeBuilder Project** option and then click **Next**.



4. The **Select NodeBuilder Project File** dialog opens. Click the button to the right of the **Project File** property, browse to and select the desired project folder (**C:\Users\Public\Documents\LonWorks\OpenLnsCt\Source\<Project Folder>** by default), and then select the project file (**.NbPrj** extension) in the project folder.



5. Click **Finish**.

Notes:

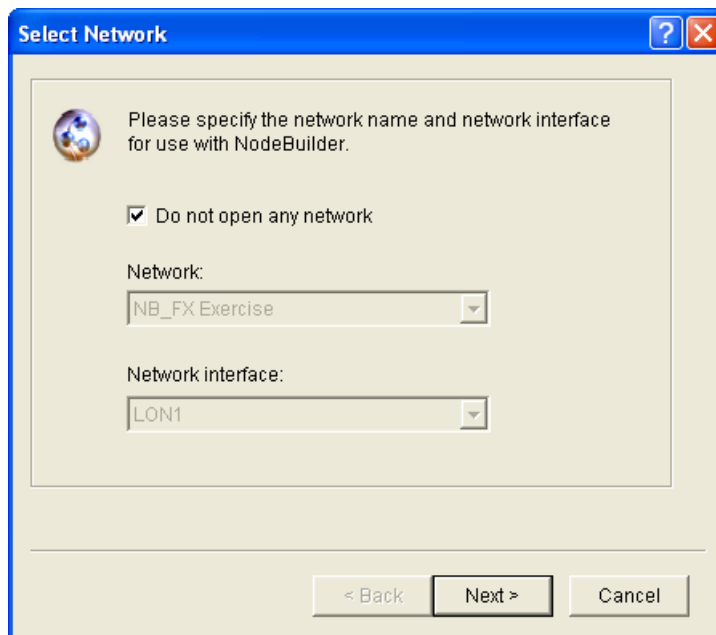
- You can open a project and start the New Device Template wizard at the same time by dragging a **Development Target** or **Release Target** device shape from the **NodeBuilder Basic Shapes 4.00** stencil to your network drawing.

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

Opening a NodeBuilder Project from the NodeBuilder Project Manager

You can open a NodeBuilder project by starting the NodeBuilder Project Manager standalone. To do this, follow these steps:

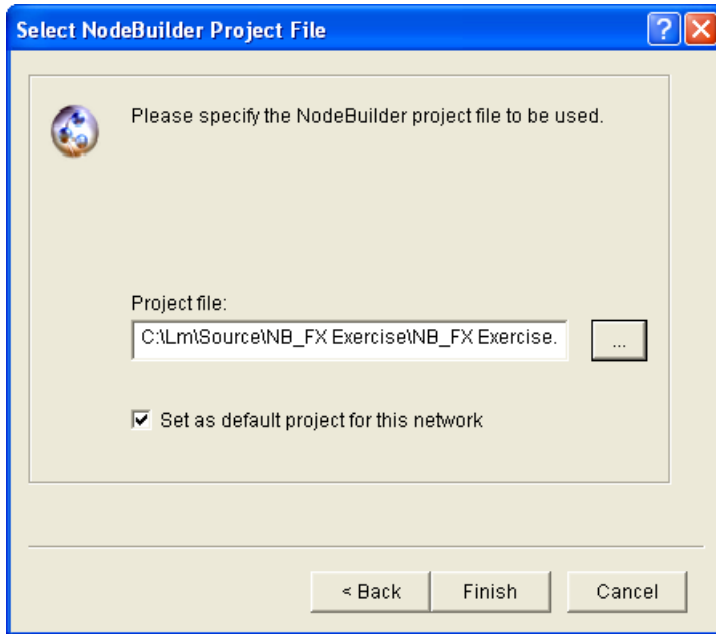
1. Open the NodeBuilder Project Manager. To do this, click **Start** on the taskbar, point to **Programs**, point to **Echelon NodeBuilder**, and then click **NodeBuilder Development Tool**. The NodeBuilder Project Manager starts.
2. Click **File** and then click **Open Project**. The New Project wizard starts with the **Select Network** dialog.



3. To associate an existing IzoT CT network with your existing NodeBuilder project, clear the **Do Not Open Any Network** check box if it is selected, select an existing IzoT CT network in the **Network** property, and then select the IzoT network interface to be used for communication between the IzoT CT network and your NodeBuilder device in the **Network Interface** property. Click **Next**.

Alternatively, you can select the **Do Not Open Any Network** check box to open a NodeBuilder project but not associate it with an IzoT CT network, and disable automatic IzoT device template creation and automatic load after build. Click **Next**.

4. The **Select NodeBuilder Project File** opens.



5. If you have previously associated a IzoT CT network with this NodeBuilder project, it appears in the **Project File** property.
6. To select a different NodeBuilder project, click the button to the right of the **Project File** property, browse to and select your project folder (**C:\Users\Public\Documents\LonWorks\OpenLnsCt\Source\<Project Folder>** by default), and then select the project file (.NbPrj extension) in the project folder.
7. Optionally, you can select the **Set as Default Project** check box to specify this NodeBuilder project as the default when the IzoT NodeBuilder tool is started from the IzoT Commissioning tool.
8. Click **Finish**.

Copying NodeBuilder Projects

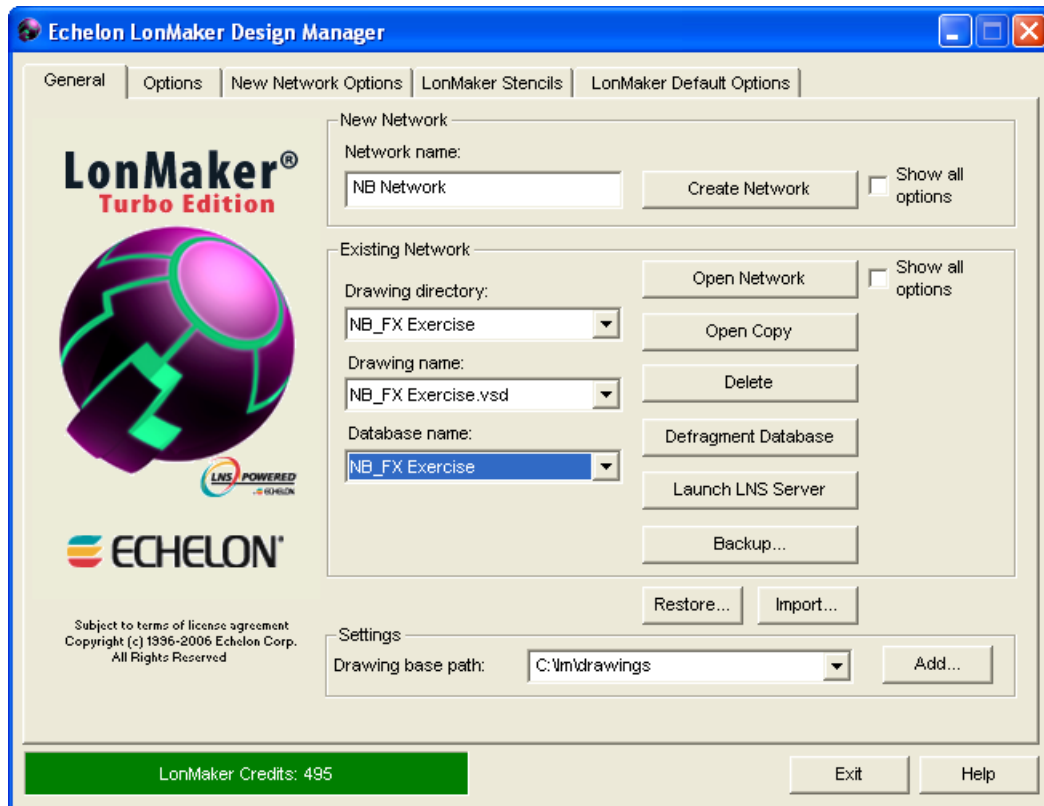
You can copy a NodeBuilder project to another computer using IzoT CT, or by manually copying the NodeBuilder project files. After you copy a NodeBuilder project, you must also copy any user-defined resource files used by the device template in the project from the source computer to the target computer, and then install and register your user-defined resource files on the target computer. See *Copying User-Defined Resource Files* for more information on how to do this.

Using the IzoT Commissioning Tool to Backup and Restore a NodeBuilder Project

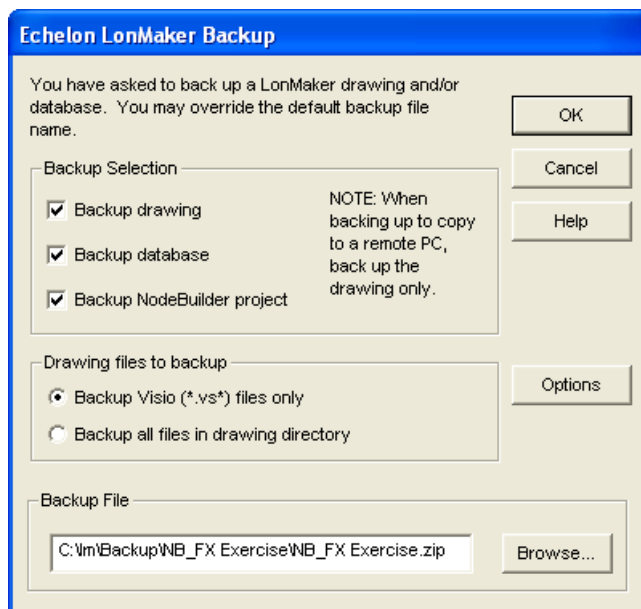
You can copy a NodeBuilder project to another computer by backing up the project files on the source computer and restoring them on the target computer with the IzoT Commissioning Tool. To do this, follow these steps:

1. Ensure that the source and target computers have the same versions of the IzoT NodeBuilder tool and the IzoT Commissioning Tool.
2. On the source computer, start the IzoT Commissioning Tool. To do this, click **Start** on the taskbar, point to **Programs**, point to **Echelon OpenLNS CT**, and then select **OpenLNS Commissioning Tool**. The **OpenLNS CT Design Manager** opens.

3. In the **Database Name** property under **Existing Network**, select the IzoT CT network design associated with the NodeBuilder project to be copied and then click **Backup**.

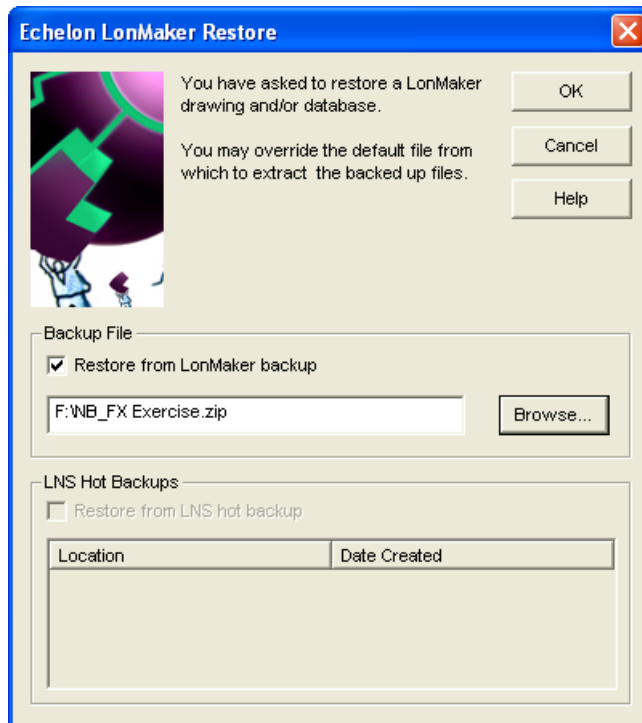


4. The **OpenLNS CT Backup** dialog opens.

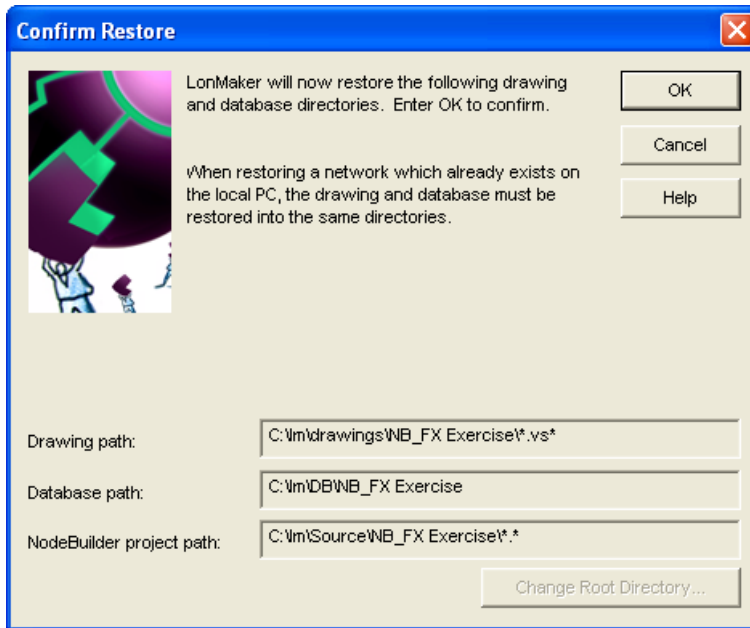


5. Select the **Backup Drawing**, **Backup Database**, **Backup NodeBuilder Project** check boxes under **Backup Selection** (the **Backup Drawing** and **Backup Database** check boxes are selected by default), and then click **OK**.

6. The IzoT CT drawing, IzoT network database, and the NodeBuilder project are all stored in a single IzoT CT backup file (.zip extension) that is specified in the **Backup File** property (C:\Users\Public\Documents\LonWorks\OpenLnsCt \Backup\<IzoT CT network>\<IzoT CT network>_<index>.zip by default).
7. After the backup has been created, copy the IzoT CT backup file from the source computer to a USB drive, another removable media, or a shared network drive with read/write permissions.
8. On the target computer, start the IzoT Commissioning Tool and then
9. Click **Restore**. The **OpenLNS CT Restore** dialog opens.



10. Click **Browse** to specify the location of the IzoT CT backup file, and then click **OK**. The **Confirm Restore** dialog opens.



11. Click **OK**. The IzoT CT drawing, IzoT network database, and the NodeBuilder project are copied to the target computer. The NodeBuilder project is associated with the IzoT CT network.
12. A message appears informing you that the network restore operation has been completed, and prompting you to select whether to open the IzoT CT network in order to recommission devices that have changed since the network was backed up.
 - Click **Yes** if you made *any* changes to the network since it was backed up. This prevents the network from behaving unpredictably if the IzoT CT network design is not in sync with the physical devices. Proceed to recommission and resynchronize the network.
 - Click **No** only if changes have not been made to the configuration of the existing physical devices on the network since it was backed up. This happens if the IzoT Commissioning Tool was OffNet the entire time, or if you added new devices and functional blocks but did not modify any existing devices or functional blocks. The IzoT CT drawing will not be opened.

See the *IzoT Commissioning User's Guide* for more information on backing up and restoring a LONWORKS Network Design.

Manually Copying NodeBuilder Project Files

You can manually copy the entire NodeBuilder project. To do this, follow these steps:

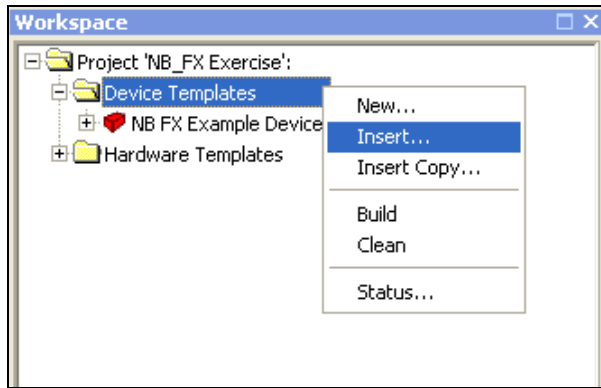
1. Ensure that the source and target computers have the same versions of the IzoT NodeBuilder tool and the IzoT Commissioning Tool.
2. On the source computer, copy the entire NodeBuilder Project folder to a USB drive, another removable media, or a shared network drive with read/write permissions. By default, the NodeBuilder Project folder is stored in the **C:\Users\Public\Documents\LonWorks\OpenLnsCt\Source** directory and has the same name as the NodeBuilder project. The NodeBuilder Project folder contains subdirectories for each device template in the NodeBuilder project.
3. On the source computer, copy any user-defined hardware templates and custom libraries to the USB drive, another removable media, or a shared network drive with read/write permissions. By default, user-defined hardware templates are stored in the **C:\Program Files (x86)\LonWorks\NodeBuilder\Templates\Hardware\User** directory.
4. Copy the NodeBuilder Project backup to the **C:\Users\Public\Documents\LonWorks\OpenLnsCt\Source** directory on the target computer.

5. Copy the user-defined hardware template backup to the **C:\Program Files (x86)\LonWorks\nodebuilder\templates\hardware\User** directory on the target computer. You need to create a **User** folder in the **Hardware** directory if one does not already exist.
6. Copy the library backup to the same folder as they were located on the source computer. If this is not possible, you can re-add them to the project as described in *Inserting a Library into a NodeBuilder Device Template*.
7. Start the IzoT NodeBuilder tool as described in *Opening a NodeBuilder Project* earlier in this chapter and browse to and open the NodeBuilder Project file (.NbPrj extension).

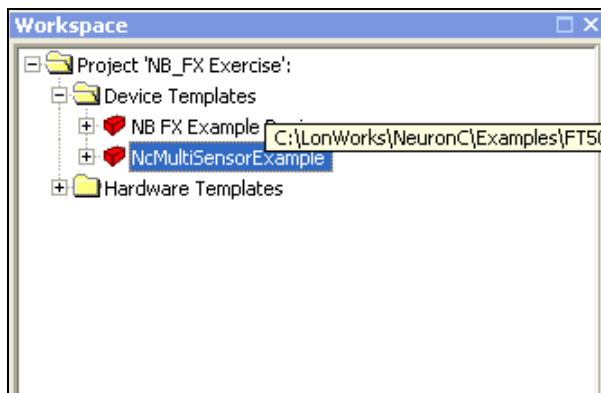
Copying NodeBuilder Device Templates

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

1. Ensure that the source and target computers have the same versions of the IzoT NodeBuilder tool.
2. If the NodeBuilder project that will contain the device templates has not been created on the target computer, create it as described in *Creating a NodeBuilder Project* earlier in this chapter
3. On the source computer, copy the device template folders to a USB drive, another removable media, or a shared network drive with read/write permissions. By default, the device templates within a given project are stored in individual folders in the **C:\Users\Public\Documents\LonWorks\OpenLnsCt\Source\<NodeBuilder Project>** directory that have names corresponding to their respective NodeBuilder device templates.
4. On the source computer, copy any user-defined hardware templates and custom libraries to the USB drive, another removable media, or a shared network drive with read/write permissions. By default, user-defined hardware templates are stored in the **C:\Program Files (x86)\LonWorks\nodebuilder\templates\hardware\User** directory.
5. Copy the device template backups to the **C:\Users\Public\Documents\LonWorks\OpenLnsCt\Source\<NodeBuilder Project>** directory of the target NodeBuilder project on the target computer.
6. Copy the user-defined hardware template backup to the **C:\Program Files (x86)\LonWorks\nodebuilder\templates\hardware\User** directory on the target computer. You need to create a **User** folder in the **Hardware** directory if one does not already exist.
7. Copy the library backup to the same folder as they were located on the source computer. If this is not possible, you can re-add them to the project as described in *Inserting a Library into a NodeBuilder Device Template*.
8. Copy any user-defined resource files from the source computer to the target computer, and then install and register the resource files on the target computer. See *Copying User-Defined Resource Files* for more information on how to do this.
9. On the target computer, open the IzoT NodeBuilder tool.
10. Right-click the **Device Templates** folder in the Project Pane on the left side of the NodeBuilder Project Manager, and then click **Insert** on the shortcut menu.



11. Browse to and open the device template folder backed up in step 3, and then select the NodeBuilder device template file (.NbdT extension). The device template is added to the NodeBuilder project under the **Device Templates** folder in the Project Pane.



Copying User-Defined Resource Files

After you copy a NodeBuilder project or a NodeBuilder device template to another computer, you must also copy any user-defined resource files on the source computer to the target computer, and then install and register the resource files on the target computer. User-defined resource files include the network variable types, configuration property types, functional profiles, enumerations, languages, and formats that you have created in your resource file set. To copy resource files to another computer, follow these steps:

1. On the source computer, copy the resource folder containing your user-defined resource files to a USB drive, another removable media, or a shared network drive with read/write permissions. By default, your resource folder is in the **C:\Program Files (x86)\LonWorks\types\user** directory on your computer.
2. Copy the user-defined resource file backup to the **C:\Program Files (x86)\LonWorks\types\user** directory on the target computer.

See *Using the Resource Pane* in Chapter 6 for more information on resource folders, resource file sets, and resources.

Viewing and Printing NodeBuilder XML Files

Many of the files created by the IzoT NodeBuilder tool are XML files. These files can be viewed and printed using a variety of tools including Internet Explorer or Microsoft Excel. 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, or another XML browsing tool.

The following XML files are created and maintained by the IzoT NodeBuilder tool:

Project File (*. NbPrj)	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. This file is stored in the project folder (C:\Users\Public\Documents\LonWorks\OpenLnsCt\Source\<NodeBuilder Project>).
Options File (*. NbOpt)	Contains the NodeBuilder project options for a project. There is one options file per project. This file is stored in the project folder C:\Users\Public\Documents\LonWorks\OpenLnsCt\Source\<NodeBuilder Project>.
Device Template File (*. NbDt)	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. This file is stored in the project folder (C:\Users\Public\Documents\LonWorks\OpenLnsCt\Source\<NodeBuilder Project>\<NodeBuilder 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 stored in the C:\Program Files (x86)\LonWorks\NodeBuilder\Templates\Hardware\Standard folder. User-defined hardware template files are stored in the C:\Program Files (x86)\LonWorks\NodeBuilder\Templates\Hardware\User folder. Hardware templates specific to the project can also be contained in the project folder.

Creating and Using Device Templates

This chapter describes how to use the New Device Template wizard in the NodeBuilder Project Manager to create, manage, and edit NodeBuilder device templates. It explains how to manage development and release targets and insert libraries into a device template. It describes how to use the Hardware Template Editor to create and edit hardware templates.

Introduction to Device Templates

Each type of device that you develop with the IzoT NodeBuilder tool is defined by a pair of device templates: a *NodeBuilder device template* and an *LNS device template*.

The *NodeBuilder device template* is an XML file with a **.NbDt** extension that specifies the information required for the IzoT NodeBuilder tool to build the device application. The NodeBuilder device template includes a list of Neuron C source code files and the hardware template name.

When you build the device application, the IzoT NodeBuilder tool automatically produces an *LNS device template*. The LNS device template defines the external interface to the device, and is used by the IzoT Commissioning tool and other LNS network tools to configure and bind the device.

Creating Device Templates

You can create device templates using the New Device Template wizard in the NodeBuilder Project Manager. The New Device Template wizard guides you through the process of creating a new NodeBuilder device template. In the NodeBuilder device template, you will specify a device template name, working directories, a Program ID, and hardware templates.

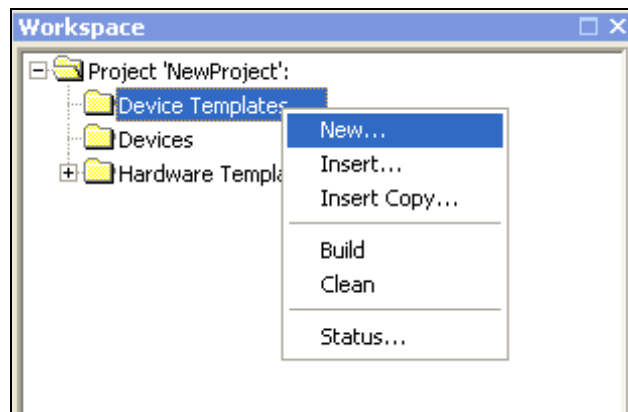
To create a device template, you do the following:

1. Start the New Device Template wizard.
2. Specify the device template name.
3. Specify the program ID.
4. Select the target hardware platform.

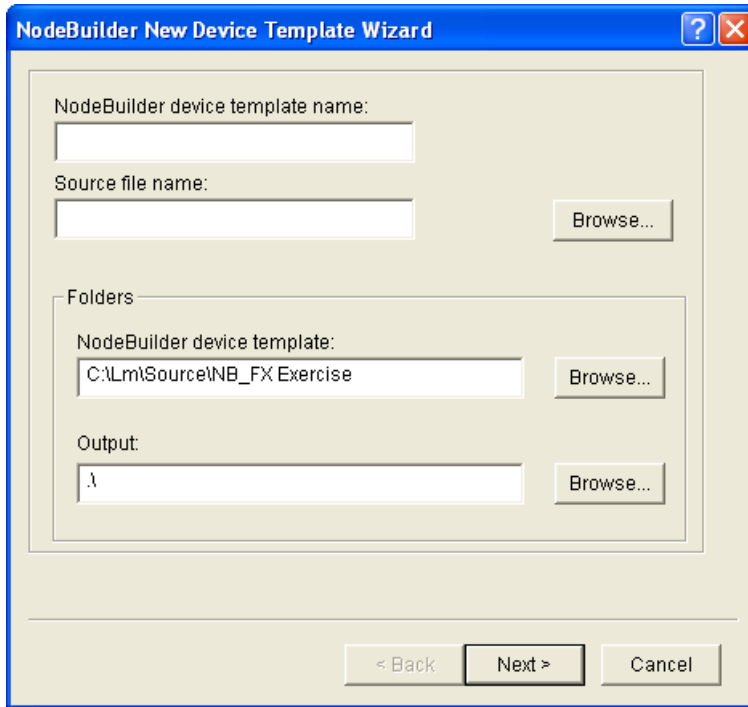
Starting the New Device Template Wizard

To start the new device template wizard follow these steps:

1. You can start the New Device Template wizard automatically after you finish creating a new NodeBuilder project or manually from the Project pane.
 - To automatically start the New Device Template wizard after you finish creating a new NodeBuilder project, select the **Run NodeBuilder Device Template Wizard** check box in the **Specify Project Default Settings** dialog at the end of the New Project Wizard. See *Creating a NodeBuilder Project* in Chapter 4 for more information on creating a NodeBuilder project and setting this option.
 - To manually start the New Device Template wizard, right-click the **Device Templates** folder in the Project pane and then select **New** on the shortcut menu.



2. The **NodeBuilder New Device Template Wizard** opens.

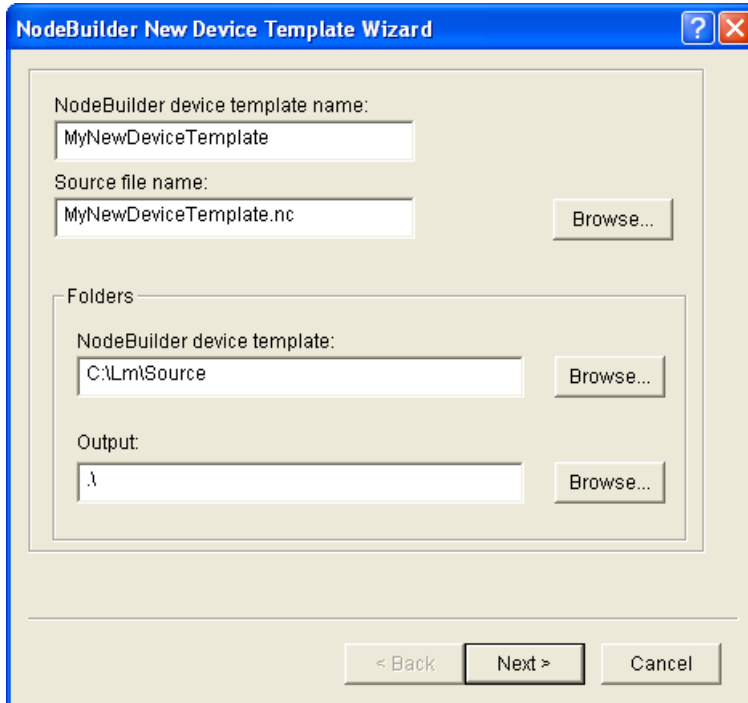


3. Proceed to the next section to specify the device template name.

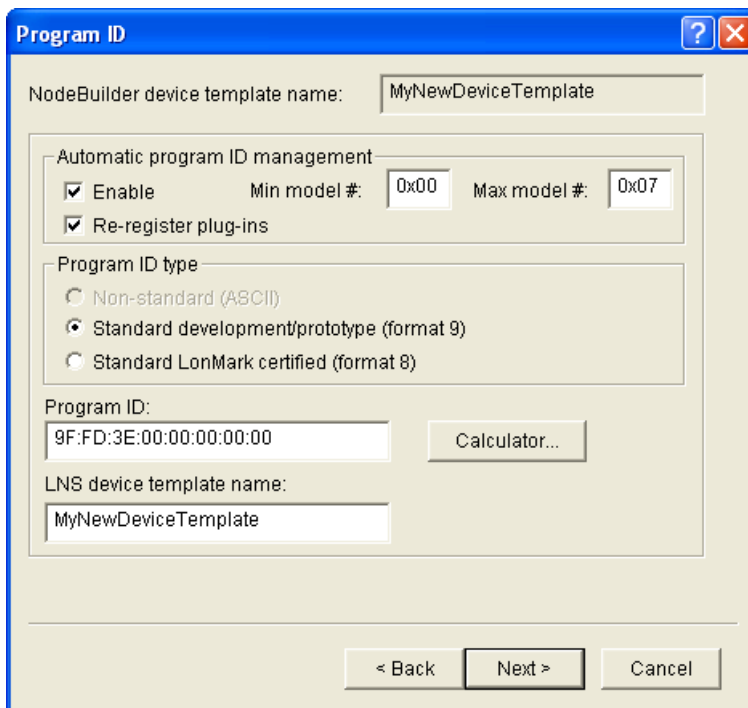
Specifying the Device Template Name

To specify the device template name, follow these steps:

1. In the **NodeBuilder Device Template Name** property, enter a valid Windows file name for the device template. A NodeBuilder device template file (.NbDt extension) with this name will be created in the folder specified in the **NodeBuilder Device Template** property under **Folders**.



2. Optionally, in the **Source File Name** property, you can 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** property under **Folders**. To select an existing source file, click **Browse**.
3. Optionally, in the **NodeBuilder Device Template** property under **Folders**, you can enter the *device template folder* where the device template file will be stored. By default, the name of the device template folder is the same as the device template that it contains (for example, the device template folder containing the **NB FX Example Device.NbDt** device template file is C:\Lm\Source\NB_FX Exercise\NB FX Example Device). To select a different folder, click **Browse** and then browse to and choose a different folder.
4. Optionally, in the **Output** property under **Folders**, you can enter the root folder for output files generated by the build process. You can specify either an absolute or relative path name. Relative paths are based on the device template folder. The default value is the build target folder (.).
5. Click **Next**. The **Program ID** dialog opens.



6. Proceed to the next section to specify the program ID.

Specifying the Program ID

The program ID is a 16-hex-digit number that uniquely identifies the device interface for a device. The program ID may be formatted as a standard or non-standard program ID. When formatted as a standard program ID, the 16 hex digits are organized as six fields that identify the manufacturer, classification, usage, channel type, and model number of the device.

To specify the program ID, follow these steps:

1. Click **Calculator**. The **Standard Program ID Calculator** dialog opens.

The **Standard Program ID Calculator** helps you select the appropriate values for the program ID fields. It lets you select the values from lists contained in a program ID definition file distributed by LONMARK International. The current file (**spidData.xml**) is available at <http://www.lonmark.org/spid>. This file is updated as LONMARK International adds new manufacturer IDs, device classes, usage values, and channel types.

The **Program ID** box at the bottom of this dialog is automatically updated as you enter the program ID fields. You can manually enter some or all of the program ID fields directly into this box. If you enter values directly in this box, the calculator updates the properties above in the dialog with those values.

2. Enter the following values for the program ID fields:
 - a. In the **Manufacturer ID (M:MM:MM)** property, either select your company from the list, enter your 5 hex-digit standard manufacturer ID or temporary manufacturer ID in the box to the right in decimal format (the calculator will convert it to hex format), or select the **Examples** manufacturer ID. By default, the manufacturer ID that you entered during of the IzoT NodeBuilder tool installation is shown by default.

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.

If you do not have a standard manufacturer ID, you can request a temporary manufacturer ID by filling out a simple form at www.lonmark.org/mid.
 - b. In the **Category** property, select the general purpose or industry of the device. The **Category** determines the device classes that will be available in **Device Class** property. Alternatively, you can select one of the following options to determine and organize the device classes shown in the **Device Class** property:
 - **ALL**. Show all the existing device classes.
 - **Profiles By Name**. Show an alphabetical list of all device classes with a profile.
 - **Profiles By Number**. Show a numeric list (sorted by device class number) of all device classes with a profile.

- c. In the **Device Class (CC:CC)** property, select the primary function of the device. To enter a device class value that has not yet been added to the standard list, select **<Enter Number[Decimal]>**, and then enter decimal values from 0 to 255 in the boxes to the right (the calculator will convert the values to hex format).
- d. In the **Usage (UU)** property, select the intended use of the device. The most significant two bits are determined by the **Has Changeable Interface** and **Use Field Valued Defined By Functional Profile** check boxes below the **Usage** property.

If you are using a standard usage value, select the **Use Field Defined By Functional Profile** check box below the **Usage** property, and select a standard usage value from the list.

If the primary functional profile implemented by your device specifies custom usage values, clear the **Use Field Defined By Functional Profile** check box below the **Usage** property, select **<Enter Number[Decimal]>** from the list, and then enter a decimal value from 0-255 in the box to the right (the calculator will convert the value to hex format).

- e. In the **Channel Type (TT)** property, select the channel type supported by the device's transceiver.

If you are using a transceiver that is not compatible with any of channel types in the list, select **Custom**.

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 (the calculator will convert the value to hex format).

- f. In the **Model Number (NN)** property, enter the specific product model within the range specified by the **Min Model #** and **Max Model #** properties in the **Program ID** dialog. You can assign a unique model number for the specified manufacturer, device class, usage, and channel type. 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 your published model number. You can have this property updated automatically by selecting the **Automatic Program ID Management** check box in the **Program ID** dialog.
- g. In the **Standard Development Program ID** property, identify your device as a standard development/prototype device or as a LONMARK certified device. If your device is a development or prototype device that is not yet LONMARK certified, select the **Standard Development Program ID** check box (the calculator sets the **F** field of the program ID to **9**). Clear this checkbox if your prototype is LONMARK certified (the calculator sets the **F** field of the program ID to **8**). This check box is selected by default.
- h. If your device has a changeable interface (it has *changeable-type network variables*, or the device supports *dynamic network variables*), select the **Has Changeable Interface** check box. This check box is cleared by default.

Integrators can use a network tool to change the types of *changeable-type network variables* when installing a network. You can implement changeable-type network variables on any type of device.

Dynamic network variables are network variables that are during installation time by a network tool. Network variables with changeable types may be implemented by any device; dynamic network variables may only be implemented by host-based devices. For more information on changeable-type network variables and dynamic network variables, see the *Application Laye0072 Interoperability Guidelines*.

LonMark Standard Program ID Calculator

Manufacturer (M:MM:MM) :
 <Enter Number [Decimal]> 1047870

Category:
 HVAC

Device class (CC:CC) :
 Thermostat (80.60)

Usage (UU) :
 Network Management

Channel type (TT) :
 TP/FT-10

Model number (NN) :
 x 00

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:FD:3E:50:3C:00:04:00

3. Click **OK** to return to the **Program ID** dialog in the New Device Template wizard. The **Program ID** property contains the program ID you specified in the **Standard Program ID Calculator** dialog.

Program ID

NodeBuilder device template name: MyNewDeviceTemplate

Automatic program ID management
 Enable Min model #: 0x00 Max model #: 0x07
 Re-register plug-ins

Program ID type
 Non-standard (ASCII)
 Standard development/prototype (format 9)
 Standard LonMark certified (format 8)

Program ID:
 9F:FD:3E:50:3C:00:04:00 Calculator...

LNS device template name:
 MyNewDeviceTemplate

< Back Next > Cancel

4. Verify that the **Enable** check box under **Automatic Program ID Management** is selected. This enables the Model number (NN) field in the program ID to be incremented automatically when the external interface of the device is changed. 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 model numbers specified by the **Min Model #** and **Max Model #** properties to avoid two devices having the same program ID but different external interfaces. When the **Max model #** value is reached, the model number field of the Program ID will be reset to the **Min model #** value. When this check box is selected, the **Min model #** and **Max model #** properties are enabled, and the **Nonstandard (ASCII)** Program ID Type is disabled.

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

You should clear this check box only 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. If you clear this check box, you must manually manage the program ID and device template name to ensure they are unique for each unique device interface.

The IzoT 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 IzoT NodeBuilder tool reports an error.

5. Optionally, you can select the **Re-register Plug-ins** property so that the IzoT NodeBuilder tool automatically re-registers LNS device plug-ins whenever the program ID changes. This option is only available if **Automatic Program ID Management** is enabled. This check box is selected by default.

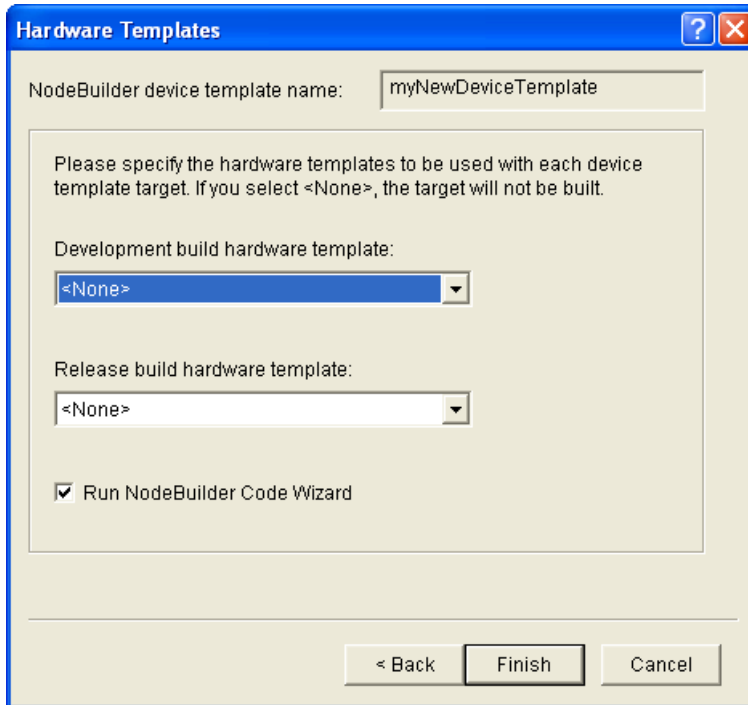
Only LNS device plug-ins that were registered for the most recent previous device template will be registered (for example, if you turn this option off for several program ID changes, then turn it back on, you will need to manually re-register LNS device plug-ins for the newest version of the device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

Note: You can access the **Program ID** settings after the device template has been created. To do this, right-click the device template in the Project pane, select **Settings** from the shortcut menu, and then select the **Program ID** tab.

6. The **LNS Device Template Name** property displays the name that the device template will be referred to by LNS tools such as the IzoT Commissioning tool. The default LNS device template name is the same as the NodeBuilder device template name. If you change the program ID of a NodeBuilder device template, you must change the **LNS Device Template Name** property. This is because each LNS device template has a unique program ID and multiple LNS device templates cannot have the same name.

If **Automatic Program ID Management** is enabled, the LNS Device Template name is automatically updated in the following 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 IzoT Commissioning tool).

7. Click **Next**. The **Target Platforms** dialog opens.



8. Proceed to the next section to specify the hardware templates used by development and release devices.

Specifying Target Platforms

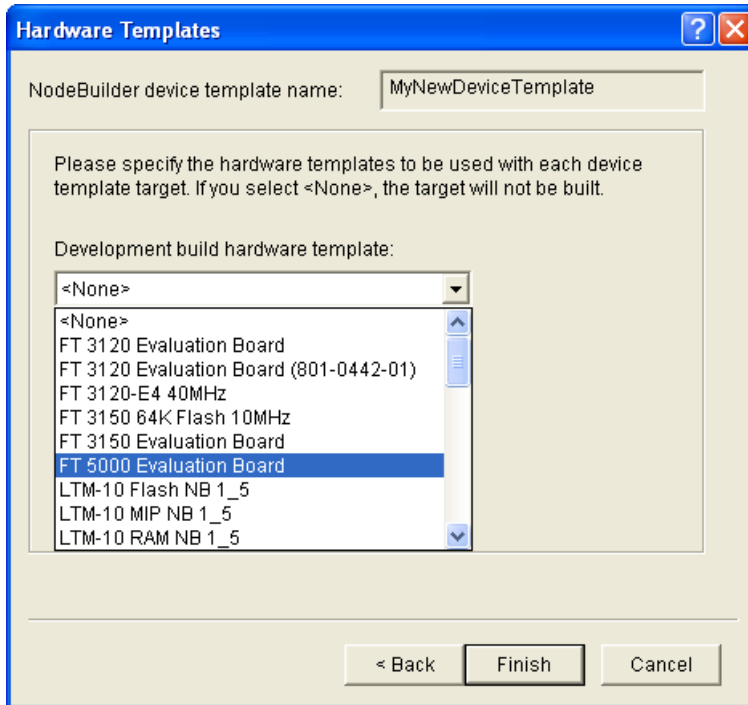
You can specify the hardware templates used for targets. A *hardware template* is a file that defines the hardware configuration for a device. It specifies hardware attributes including platform, transceiver type, Neuron Chip or Smart Transceiver model, clock speed, system image, and memory configuration.

A *target* is a LONWORKS device whose application is built by the IzoT 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.

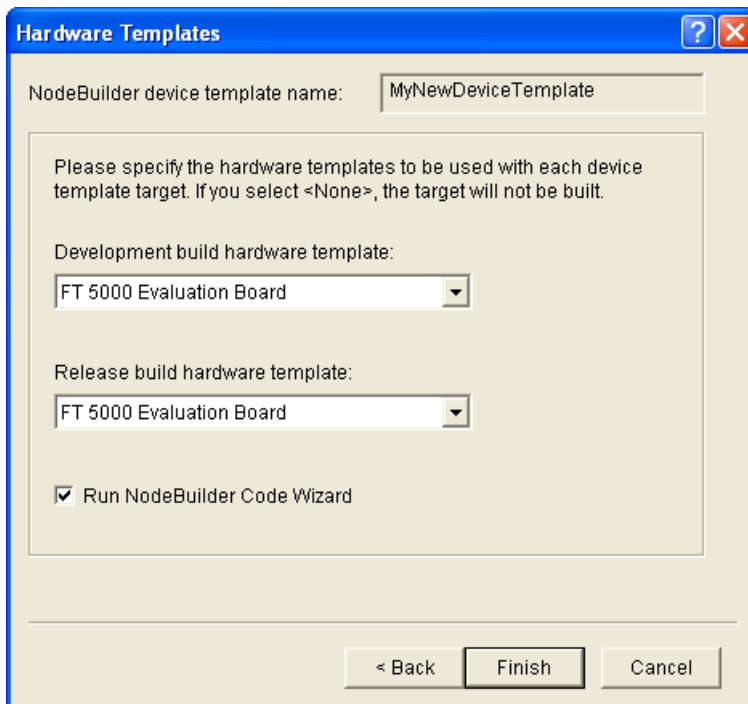
Note: You can skip this step now, but you must specify the hardware templates before you can build the device template.

To specify the target platforms, follow these steps:

1. In the **Development Build Hardware Template** property, select the hardware template to be used for development targets. The list contains all the hardware templates in the **Hardware Templates** folder in the Project pane.



2. In the **Release Build Hardware Template** property, select the hardware template to be used for release targets. The list contains all the hardware templates in the **Hardware Templates** folder in the Project pane.



3. Select the **Run NodeBuilder Code Wizard** check box to run the NodeBuilder Code Wizard immediately after clicking **Finish**. This check box is selected by default.

Note: You can change the default setting of this option. To do this, click **Project** and then click **Setting**, or right-click the Project folder in the Project pane and click **Settings** on the shortcut

menu. The **NodeBuilder Project Properties** dialog opens. Click the **Options** tab, change the setting, and then click **OK**.

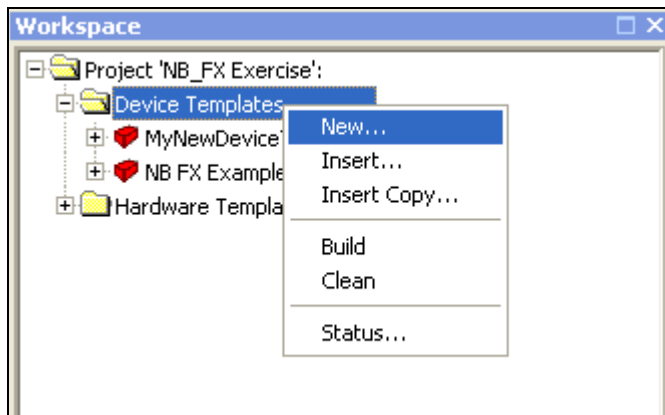
4. Click **Finish**. If you selected the **Run NodeBuilder Code Wizard** check box, the NodeBuilder Code Wizard starts. See Chapter 6, *Defining Device Interfaces and Creating their Neuron C Application Framework*, for more information about the NodeBuilder Code Wizard.

Managing and Editing Device Templates

You can manage and edit the device templates in a NodeBuilder project from the Project pane in the NodeBuilder Project Manager.

Managing Device Templates

The **Device Templates** folder in the Project pane of the project manager lists all the device templates that are defined as part of the current NodeBuilder project. You can right-click the **Device Templates** folder to open a shortcut menu with the following options:



- | | |
|--------------------|---|
| <i>New</i> | Creates a new device template in the currently open NodeBuilder project. This opens the New Device Template Wizard as described in <i>Starting the New Device Template Wizard</i> earlier in this chapter. |
| <i>Insert</i> | 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 device templates to be reused in multiple projects, and allows multiple projects to share a single device template. |
| <i>Insert Copy</i> | Creates a copy of an existing NodeBuilder device template and inserts it into the currently open NodeBuilder project.

When you select this option, a dialog opens allowing you to browse to and select a NodeBuilder device template file (.NbdT or .dev extension). After you select an existing device template, the New Device Template Wizard opens. Complete the New Device Template Wizard as described in <i>Creating Device Templates</i> .

All files associated with the device template (for example, all files in the Source Files subdirectory) will be copied to the new device template. |
| <i>Build</i> | Builds the application images for all qualifying targets. See <i>Building an Application Image</i> in Chapter 8 for more information on building device applications. |
| <i>Clean</i> | Deletes all output files created when building the currently open NodeBuilder project for all qualifying targets. See <i>Cleaning Build</i> |

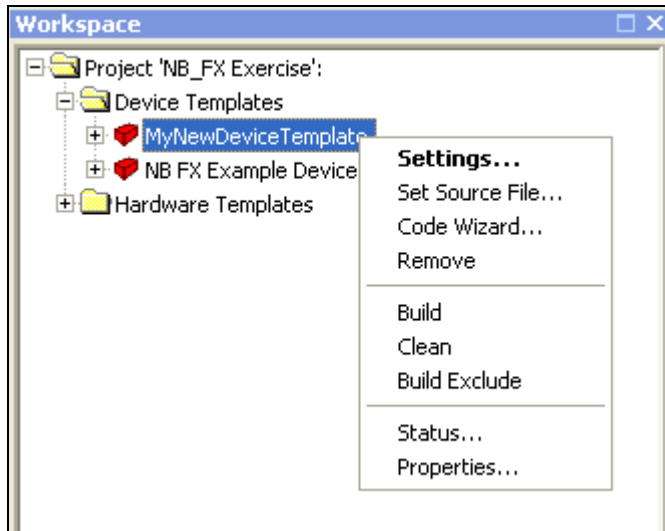
Output Files in Chapter 8 for more information on removing the files and folders produced by a build.

Status

Displays the build status for all device templates. See *Viewing Build Status* in Chapter 8 for more information on viewing the build status of NodeBuilder device templates and targets.

Viewing and Editing Device Templates

After you create a device template (♥), you can view and edit its properties. To do this, right-click the device template under the **Device Templates** folder in the Project pane to open a shortcut menu with the following options:



Settings

Opens the **NodeBuilder Device Template Properties** dialog. This dialog allows you to change the properties you set for the selected device template in the New Device Template Wizard.

Set Source File

Sets the main source file (.nc extension) for this device template. By default the main source file is <Device Template Name>.nc.

Code Wizard

Starts the NodeBuilder Code Wizard for this device template. See Chapter 6, *Defining Device Interfaces and Creating their Neuron C Application Framework*, for more information about using the NodeBuilder Code Wizard.

Remove

Removes this device template from the currently open NodeBuilder project. Note that this does not permanently delete the device template file or source files.

Build

Build the application image specified by this device template for all qualifying targets. See *Setting Build Options* in Chapter 8 for more information about setting build properties that control the build process.

Clean

Deletes all output files created when building this device template for all qualifying targets. See *Cleaning Build Output Files* in Chapter 8 for more information on removing the files and folders produced by a build.

Build Exclude

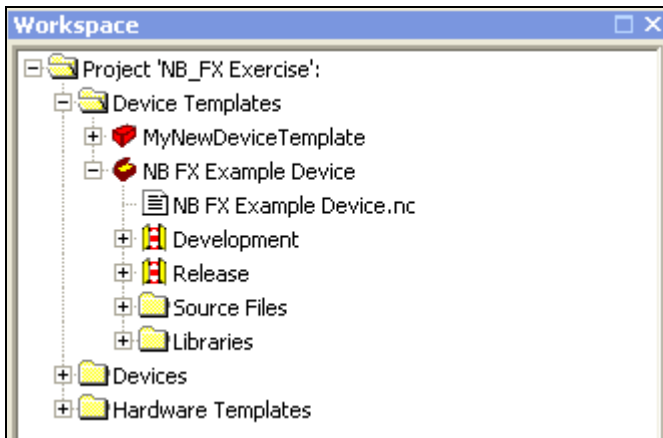
Determines if this device template will be included or excluded when you click the **Build** command for the **Device Templates** folder. When this option is enabled, the device template will be excluded from a device templates build, the device template name is dimmed, and a checkmark will appear next to the **Build Exclude** option on the shortcut

menu. When a device template is excluded, you can still explicitly build the device template by right-clicking the device template and selecting **Build** from the shortcut menu.

<i>Status</i>	Displays the build status for this device template. See <i>Viewing Build Status</i> in Chapter 8 for more information on viewing the build status of NodeBuilder device templates and targets.
<i>Properties</i>	Displays the name, location, size, and the dates on which the file was created, last modified, and last accessed.

Viewing Device Template Components

After you create a device template (🔴), you can view and edit its components, which include the main source file (📄), development and release targets (🏗️), source files (📁), and libraries (📁). To do this, expand the device template under the **Device Templates** folder in the Project pane to display the components, which are described as follows:



<i>Main Source File</i>	The main Neuron C source file (.nc extension) for this device template. This file may include other source files by using Neuron C #include statements. By default, this file is named <Device Template Name>.nc.
-------------------------	---

Editing the Main Source File

You can double-click the main source file to edit it. See Chapter 7, *Developing Device Applications*, for more information on editing the main source file.

Viewing the Main Source File Properties

To view the location, size, and date stamps of a source file, right-click the source file and then click **Properties** on the shortcut menu.

<i>Development/Release</i>	The development and release targets contain information specific to building application images for development and release targets, respectively. See the next section, <i>Managing Development and Release Targets</i> , for more information.
----------------------------	--

Source Files

This folder contains all the source files associated with this device template except for the main source file. When you add source files to the NodeBuilder project directly or using the NodeBuilder Code Wizard, they are added to this folder.

Adding Source Files

You can add other files to this folder, including Neuron C source files (.nc extension), header files (.h extension), C files (.c extension), text files (.txt extension), or other specification or documentation files. To do this, right-click the **Source Files** folder and click **Insert** on the shortcut menu.

Note: Adding files to this folder does not automatically include them when you build the application image. You must insert `#include` statements in your Neuron C code to explicitly include these files in the build.

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

Editing Source Files

You can double-click any source file to edit it. See Chapter 7, *Developing Device Applications*, for more information on editing source files.

Removing Source Files

To remove a source file from the device template, right-click the source file and then click **Remove** on the shortcut menu.

Viewing Source File Properties

To view the location, size, and date stamps of a source file, right-click the source file and then click **Properties** on the shortcut menu.

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.

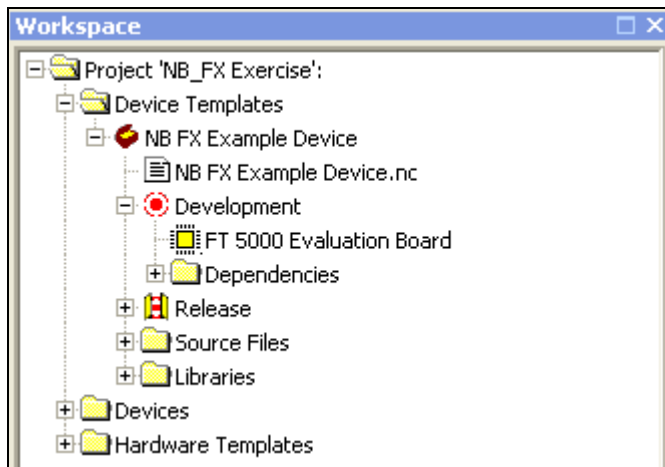
You can add a library in the **Specify Library Type** dialog. To access this dialog, right-click the **Libraries** folder and then click **Insert** on the shortcut menu. See *Inserting Libraries into a NodeBuilder Project* later in this chapter for more information on adding libraries to a project.

Managing Development and Release Targets

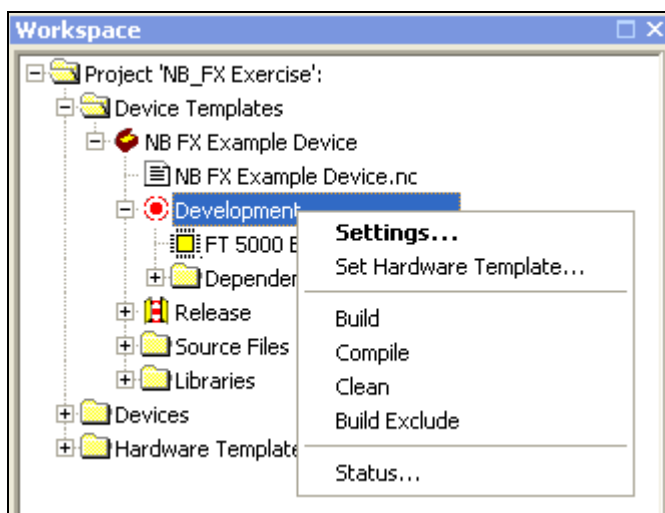
Each NodeBuilder device template in the Project pane contains **Development** and **Release** targets (🔴) that can be built. These targets are defined by their *hardware templates* and *dependencies*.

A *hardware template* (🔧) is 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. The hardware template is listed directly under its target in the Project pane. You can edit the properties of a hardware template in the **Hardware Template Editor** dialog. To access this dialog, double-click the hardware template or right-click the hardware template and click **Settings** on the shortcut menu (see *Creating Hardware Templates* for more information).

Dependencies are the files required to build the application image for a target. A list of dependencies is automatically created when you build the application image for a target. These files are listed in the **Dependencies** folder (📁) under the target in the Project pane. The list is empty until you successfully build an application image for a target



You can view and edit the properties of a target. To do this, right-click the target under its parent device template in the Project pane to open a shortcut menu with the following options:

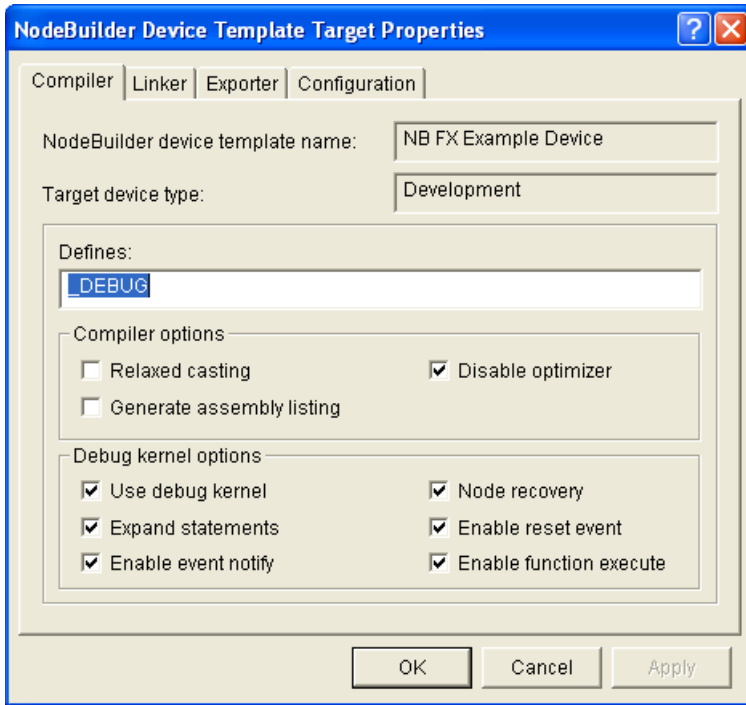


<i>Settings</i>	Opens the NodeBuilder Device Template Target Properties dialog, which includes compiling, linking, exporting, and configuration options for the target. See the following subsections for more information on the target properties you can set in this dialog.
<i>Set Hardware Template</i>	<p>Opens the Set Target Device Hardware Template dialog, where you can select the hardware template to be used for this target. You can select from all hardware templates contained in the Hardware Templates folder. Alternatively, you can drag a hardware template from the Hardware Templates folder to the Development or Release target. Note that a target cannot be built or cleaned until it has a hardware template.</p> <p>Note: If your NodeBuilder project is not associated with an IzoT CT network and you change the hardware template for a device template that uses a 6000 Series chip, you must associate the NodeBuilder project with an IzoT CT network and then re-build the device application to implement the clock speed associated with the selected hardware template. If you load the device application with the IzoT Commissioning Tool without using the IzoT NodeBuilder tool's automatic load after build feature, the device may not use the correct clock speed.</p>
<i>Build</i>	Builds the application image for this target only. See <i>Building an Application Image</i> in Chapter 8 for more information.
<i>Compile</i>	Compiles the application for this target only.
	Note: Only the compilation step of the build process is completed when you select this option. The application is not linked and the application image is not created.
<i>Clean</i>	Deletes all output files created when building this target. See <i>Cleaning Build Output Files</i> in Chapter 8 for more information on removing the files and folders produced by a build.
<i>Build Exclude</i>	Determines if this target will be included or excluded when you click the Build command for the device template or the Device Template folder. When this option is enabled, the target will be excluded from a device template build, the target name is dimmed, and a checkmark will appear next to the Build Exclude option on the target's shortcut menu. When a target is excluded, you can still explicitly build the target by right-clicking the target and selecting Build from the shortcut menu. See <i>Excluding Targets from a Build</i> in Chapter 8 for more information.
<i>Status</i>	Displays the build status for this target. See <i>Viewing Build Status</i> in Chapter 8 for more information on viewing the build status of NodeBuilder device templates and targets.

Setting Device Template Target Properties: Compiler

The Neuron C Compiler (NCC) is a Neuron C tool that is used to produce Neuron assembly source files from Neuron C source code.

You can modify the compiler options for a target. To do this, right click the target, click **Settings** on the shortcut menu, then select the **Compiler** tab in the **NodeBuilder Device Template Target Properties** dialog.



You can set the following properties:

Defines

You can define a symbol, which can then be tested from the program using the **ifdef** or **ifndef** directive. The default is **_DEBUG** for development targets. This field is blank by default for release targets.

The NodeBuilder FX tool pre-defines several preprocessor symbols, including: **_ECHELON**, **_NEURONC**, **_NODEBUILDER**, and **_NCC6**. See the *Neuron C Reference Guide* for a complete list and more information on these symbols.

Compiler Options

Relaxed Casting

Allows the **const** attribute to be removed from a variable without generating an error (a warning will still be generated by default). This check box is cleared by default.

Generate Assembly Listing

Generates an assembly listing when the Neuron C application is compiled and stores it in your working directory. This listing will have the same name as the Neuron C source file with a **.NL** extension. This check box is cleared by default.

Assembly listings are generated by the Neuron C compiler and are useful for analyzing the timing and memory efficiency of Neuron C application code. See the *Neuron Chip Data Book* for the timings of the Neuron Chip instructions. These listings are also useful in understanding how the code generated by the Neuron C compiler is affected by the use of various programming constructs and optimizations in the source file.

Debug Kernel Options

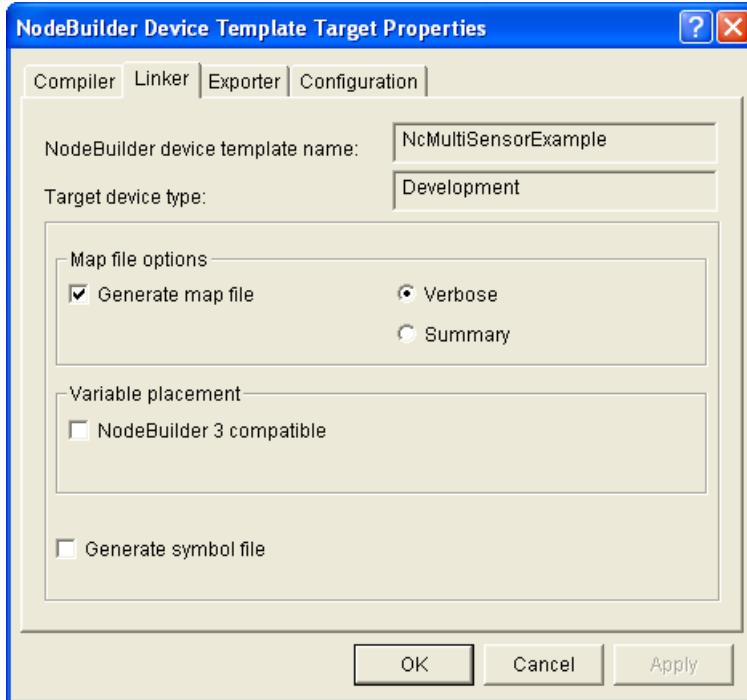
- Disable optimizer* Disables the compiler's code optimizer. Optimization typically generates smaller and faster code, and is typically enabled for release targets. However, optimization can severely change the code initially generated by the compiler, which can make it difficult to place and use breakpoints in an application that is being debugged. This check box is selected by default.
- Note:** Debugging an optimized application is not supported.
- See the *Neuron C Reference Guide* for information about the pragma optimization directive, which provides detailed control over the effective optimization level.
- Use Debug Kernel* Uses the debug kernel when compiling and linking the application. This check box is selected for **Development** targets by default.
- If this check box is cleared at compile time, you will not be able to debug the application. Clearing this check box also disables the **Expand statements**, **Enable Event Notify**, **Enable Event Reset**, and **Enable Functional Execute** options.
- Expand Statements* Expands all Neuron C statements to at least 2 bytes of machine code. A single Neuron C statement must correspond to machine code that is at least 2 bytes long in order to place a breakpoint, and the optimizer can reduce statements to less than 2 bytes. This check box is selected by default.
- If you are debugging a 3100 Series device, you must select this check box. Compiling such a debug target with this option cleared generates a linker error (NLD#515), which states that the **Expand Statements** option should be enabled for such targets.
- If you are debugging a 6000 Series device, you can clear this check box to reduce the size of your application's debug image so it is closer in size to the release image (typically, the debug and release images will still vary in size due to different optimization settings). Compiling such a debug target with this option selected generates a linker warning (NLD#516), which states that the **Expand Statements** option is not required for such targets.
- Enable Event Notify* Enables the debug kernel to communicate debug events from the device back to the IzoT NodeBuilder tool. This check box is selected by default. Clearing this check box also disables the **Enable Event Reset** option.
- Node Recovery* Enables the user to make the device applicationless by pressing the service pin for 3 seconds during a power cycle. This check box is selected by default.
- Enable Reset Event* Enables the debug kernel to notify the device when a reset event occurs. This option is useful when you are debugging, but it should be disabled if you are using debug targets in larger networks because the reset notifications can consume network bandwidth (for example, if an entire site is powered at once). This check box is selected by default. Note that the **Enable Event Notify** check box must be selected in order to set this option.

Enable Function Execute Enables the debugger to get and update the values of system timers and to update the values of network variables in the watch list when suspended at a breakpoint. This check box is selected by default.

Setting Device Template Target Properties: Linker

The Neuron Linker (NLD) is a Neuron C tool that is used to produce Neuron executable files. It links the application image, user-libraries, system libraries, and the Neuron firmware.

You can modify the linker options for a target. To do this, right click the target, click **Settings** on the shortcut menu, then select the **Linker** tab in the **NodeBuilder Device Template Target Properties** dialog.



You can set the following properties:

Map File Options

Generate Map File Generates a link map file in your working directory. This link map file will have the same name as the device template, but with the **.MAP** extension. This check box is selected by default.

If you select this check box, select whether to generate a **Verbose** or a **Summary** link map.

- A **Summary** link map summarizes the memory usage of your application image to determine how much margin is available in each memory device.
- A **Verbose** link map contains a report showing the location of every code and data segment; this report is useful for a detailed understanding of the memory usage of each type of memory in your target device. This is the default link map.

Variable Placement

NodeBuilder 3 Allocates application EEPROM variables prior to allocating on-chip

Compatible

EEPROM to meet system requirements.

In NodeBuilder 3.0 and prior releases, the IzoT NodeBuilder tool allocated system on-chip EEPROM after placement of explicit and implicit on-chip EEPROM variables. This could cause link failure in applications that declare a large amount of implicit on-chip EEPROM variables.

Implicit on-chip EEPROM variables are those application EEPROM variables declared without use of the explicit **onchip** or **offchip** keyword. These variables are placed in on-chip EEPROM when possible, or in off-chip EEPROM when necessary.

Do not select this option unless your application code makes assumptions about order or location of EEPROM variables, and it requires backwards-compatible variable placement. This check box is cleared by default.

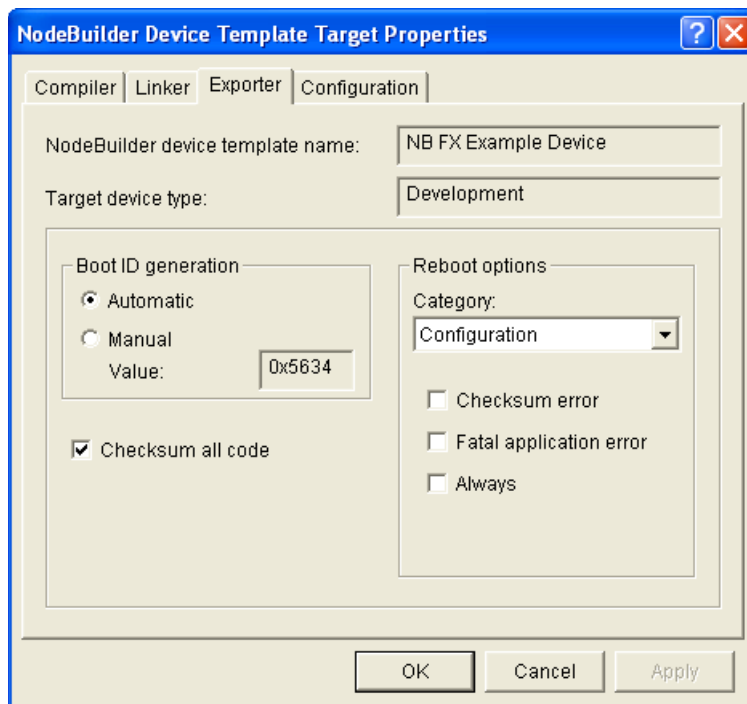
Generate Symbol File

Generates a symbol file when the project is built. Symbol files are only required if you are creating ShortStack 2.1 MicroServer images or custom firmware images.

Setting Device Template Target Properties: Exporter

The Neuron Exporter (NEX) is a Neuron C tool that takes input from the compiler and the linker and produces downloadable application image files (.APB, .NDL, and .NXE extensions), programmable application image files (.NRI, .NFI, .NEI, .NME, and .NMF, extensions), and device interface files (.XIF and .XFB extensions).

You can modify the exporter options for a target. To do this, right click the target, click **Settings** on the shortcut menu, then select the **Exporter** tab in the **NodeBuilder Device Template Target Properties** dialog.



You can set the following properties:

<i>Boot ID Generation</i>	<p>Select whether the boot ID is generated automatically or manually.</p> <p>Note: This option is intended for Neuron 3150 Chips and 3150 Smart Transceivers.</p> <ul style="list-style-type: none">• Automatic. Allocates a new boot ID each time the application image is built. This causes the on-chip EEPROM to be rebooted from the system image if the external memory device has been updated. This is the default. You should select this option unless you are trying to rebuild an application image from archived source files.• Manual Value. If you are trying to rebuild an application image from archived source files, specify the 16-bit hexadecimal boot ID. To archive source files that will be rebuilt later, store the device template, Neuron C source files, and device files in the archive. Make sure you select the Manual Value option in the archived image prior to archiving. The NodeBuilder software will update the Boot ID value to the last automatically assigned value.
<i>Checksum All Code</i>	<p>Computes the application checksum over the part of the application image and system image that is in ROM and in writable memory. Selecting this check box increases the time it takes the Neuron Chip to complete its reset processing. This check box is selected by default.</p> <p>If you clear this check box, the application checksum is computed only over the part of the application image that is in writable memory. See Reboot Options for more information.</p>
Reboot Options	<p>Specifies when the device should reboot various parts of its on-chip EEPROM memory. The device does this by copying its initial state from off-chip ROM or flash memory.</p> <p>Some hardware designs can cause corruption of the contents of the Neuron Chip's on-chip EEPROM (for example, designs with inadequate power supply noise decoupling for the Neuron Chip). The Neuron firmware can detect this because it maintains several 8-bit checksums, including ones for the configuration image, application image, and system image.</p> <p>Note: These options are intended for Neuron 3150 Chips and 3150 Smart Transceivers, and they are available if the target hardware uses custom Neuron firmware that is based on version 6 standard firmware (or later).</p> <p>You can select reboot options for Configuration, Application, and Communication Parameters categories. You can select the desired category from the Categories list.</p>
<i>Configuration</i>	<p>Specify when to copy the network image from the ROM into on-chip EEPROM. This includes the address assignment and binding information in the device, but it does not include the communications parameters.</p> <ul style="list-style-type: none">• Checksum error. Reboot whenever there is a configuration checksum error.• Fatal application error. Reboot whenever there is an application checksum error, illegal device state, memory allocation failure, or

application image inconsistency, or other fatal application error.

Application

- **Always.** Reboot every time the Neuron Chip is reset.

Specify when to copy the on-chip part of the application image from ROM into the on-chip EEPROM. All applications have at least part of their image in on-chip EEPROM in the Read-Only data structure. See Appendix A of the *Neuron Chip Data Book* for a description of this structure.

Note: You cannot use this option to recover from corruption of any part of the application image that is in off-chip memory. If off-chip memory gets corrupted, recovery will fail and the device will be in the applicationless state. You will then need to re-program the memory chip, or re-download the application over the network

- **Fatal Application Error.** Reboot whenever there is an application checksum error, illegal device state, memory allocation failure, or application image inconsistency, or other fatal application error.
- **Always.** Reboot every time the Neuron Chip is reset.
- **App'less is Fatal.** Classifies the applicationless state as a fatal application error. If you select this option, you should also select the **Fatal Application Error** check box or else recovery will not occur.

Note: If you are downloading the device application over the network, do not select this check box. This is because 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 management tool, the application, or another device.

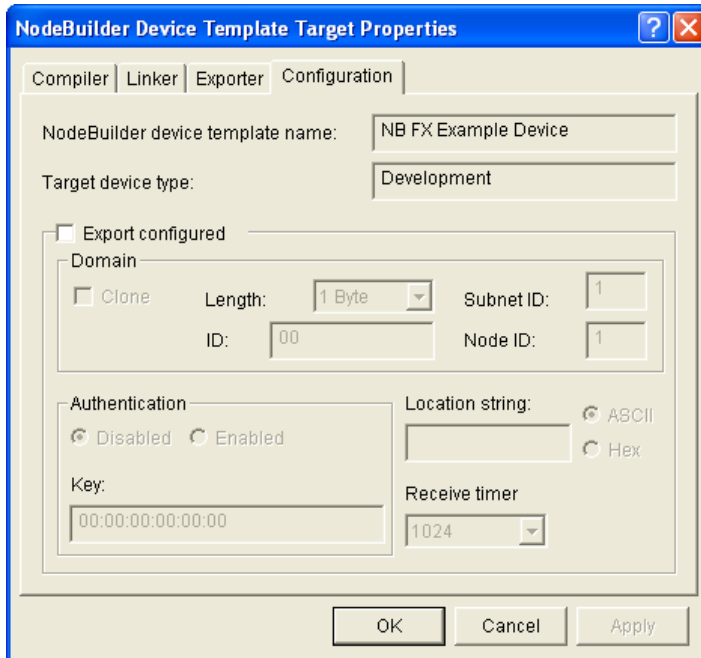
Communication Parameters

Specify when to re-initialize the communications parameters from the ROM copy. Note that re-initializing the communications parameters will cause the device to lose its priority assignment if it previously had one.

- **Checksum Error.** Reboot when a checksum error is detected.
- **Type/rate Mismatch.** Reboot if 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 every time the Neuron Chip is reset.

Setting Device Template Target Properties: Configuration

You can modify the configuration options for a target. To do this, right click the target, click **Settings** on the shortcut menu, then select the **Configuration** tab in the **NodeBuilder Device Template Target Properties** dialog.



You can select the **Export Configured** check box to enable the IzoT NodeBuilder tool build a configured device application for the target. The target will have the properties set in this dialog set when the application is downloaded to the device. This information is used to set the fields of the system domain table and other configuration data structures (see the *Neuron C Reference Guide* for more information). This check box is cleared by default.

Tip: Most device applications should be exported as unconfigured; therefore, you will typically leave the **Export Configured** check box cleared. Creating a configured device application is recommended for advanced users only.

If you select the **Export Configured** check box, you can set options for the following properties (see the *Neuron C Programmers Guide* and *Neuron C Reference Guide* for more information on these properties):

Domain

Clone

Uses a clone domain, which is a domain address within a device that specifies that the device can receive messages from other devices with the same network address. If you are exporting your application image as a configured image, you can configure the domain as a clone domain. A clone domain is typically only used in self-installed networks where multiple devices within a network may have the same address.

Limitations

Devices using a clone domain have the following limitations:

- Devices 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.
- Devices cannot receive acknowledgements and responses. The device will, however, continue to send acknowledgements and responses with proper subnet/node information.
- Devices cannot use Authentication 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.

<i>Length</i>	Specify the length of the domain that will be loaded into the device. This can be set to one of the following values: <ul style="list-style-type: none">• <None>. The device will not include a pre-configured domain record. If you want to use authentication, the firmware must support open media authentication.• 0 bytes. The device uses the 0 length domain. If you want to use authentication, the firmware must support open media authentication.• 1, 3, 6 bytes. The domain length. This value determines the format of the Domain ID field.
<i>ID</i>	The domain ID that will be loaded into the device with the application, as a hexadecimal value. You can only set this property if you set the Domain Length property to 1 byte or greater.
<i>Subnet ID</i>	The subnet ID that will be loaded into the device.
<i>Node ID</i>	The node ID that will be loaded into the device.
<i>Authentication</i>	The authentication key that will be loaded into the device.
<i>Location String</i>	The location string that will be loaded into the device. Select whether the location string is set in ASCII or Hex format.
<i>Receive Timer</i>	The receive timer value that will be loaded into the device.

Inserting a Library into a NodeBuilder Device Template

You can add a library to a NodeBuilder device template, or reference a required library from your Neuron C source code. 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.

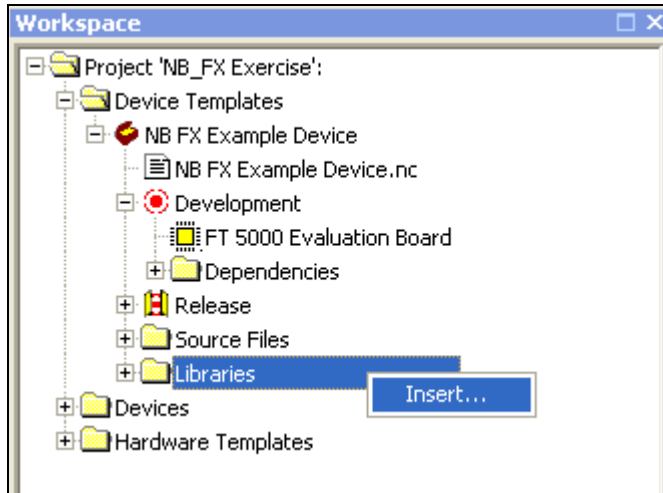
There are two types of libraries: *standard* and *custom*. The standard libraries are included with the IzoT NodeBuilder tool. When you build a device template, some standard libraries are automatically linked in your Neuron C code such as the **CodeWizard-3.lib** library (if you are using version 3 code templates), and the **gen.lib**, **psg.lib** and **extarith.lib** libraries. You may explicitly include standard libraries in a NodeBuilder project for documentation purposes. Note that some libraries provided by Echelon must be explicitly included as custom libraries such as the ISI and CCL libraries.

Custom libraries are any libraries that you or a third party creates. Custom libraries must be explicitly included in a NodeBuilder project. You can create your own custom libraries (see the *Neuron C Programmer's Guide* for more information on how to do this). You can use the *pragma library* directive with your source code. This directive lets you specify a library and library location from your Neuron C source file (see the *Neuron C Reference Guide* for more information). Alternatively, you can insert a library into a NodeBuilder device template using the NodeBuilder project manager, follow the steps outlined below.

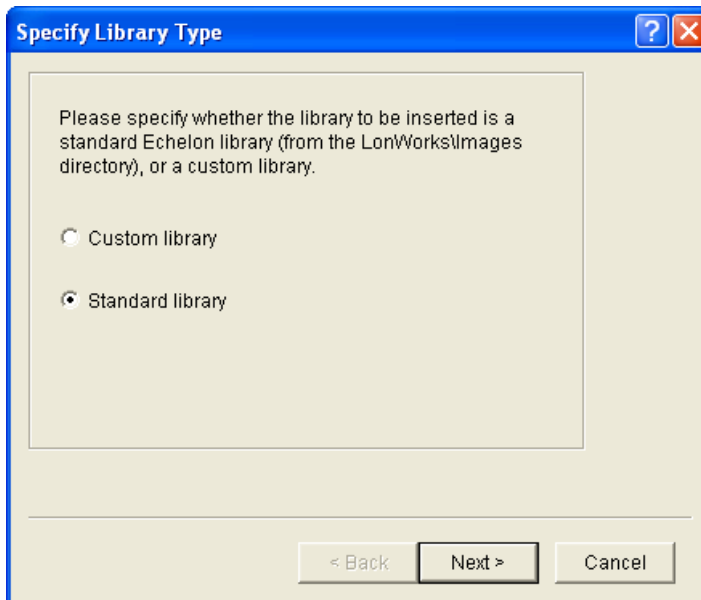
Because the *pragma library* directive supports location-independent references to your library, this is the recommended method.

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

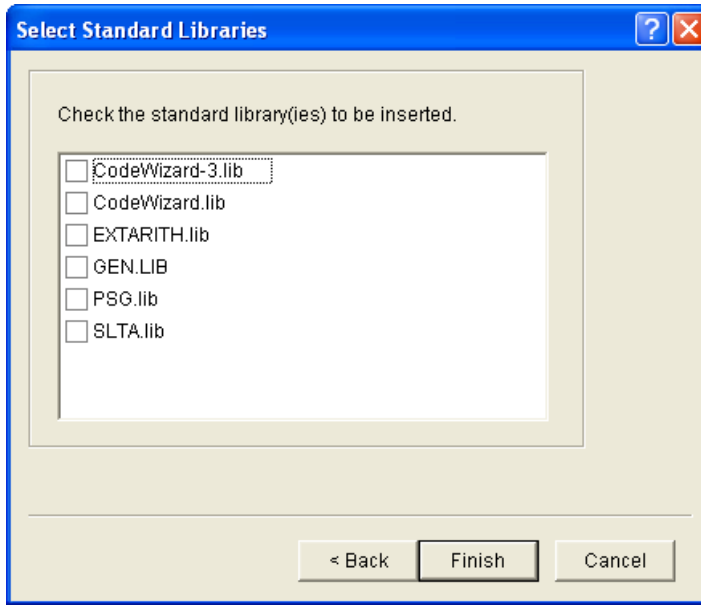
1. Expand the device template in the Project pane of the NodeBuilder project manager.
2. Right-click the **Libraries** folder and then click **Insert** on the shortcut menu.



3. The **Specify Library Type** dialog opens.



4. Select a **Custom Library** or a **Standard library** to add to the device template and then click **Next**. Standard libraries (.lib extension) are stored in the C:\LonWorks\Images folder; custom libraries can be stored anywhere.
5. If you selected **Standard Library** in step 4, the **Select Standard Libraries** dialog opens.



Select one or more of the following standard libraries in the C:\LonWorks\Images folder to be explicitly included in the project (for documentation purposes only) and then click **Finish**.

CodeWizard-3.lib The NodeBuilder Code Wizard library used by the version 3 code template. The Code Wizard library supplies most of the utility functions defined in the **CodeWizard.h** file. See *Version 3 Templates* in Chapter 6 for more information on the version 3 code templates.

Note: You do not need to add a reference to the **CodeWizard-3.lib** file to your NodeBuilder Device Template. Version 3 of the **CodeWizard.h** file automatically links with this library through the **#pragma library** directive.

CodeWizard.lib The NodeBuilder Code Wizard library used by the version 2 template. Existing applications that use the version 2 code templates already contain a reference to this library; however new applications that use the version 2 code templates need to reference this library. See *Version 2 Templates* in Chapter 6 for more information on the version 2 code templates.

Note: You should use the version 3 code templates for all new device development. The version 1 code templates do not use or require a special function library.

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*.

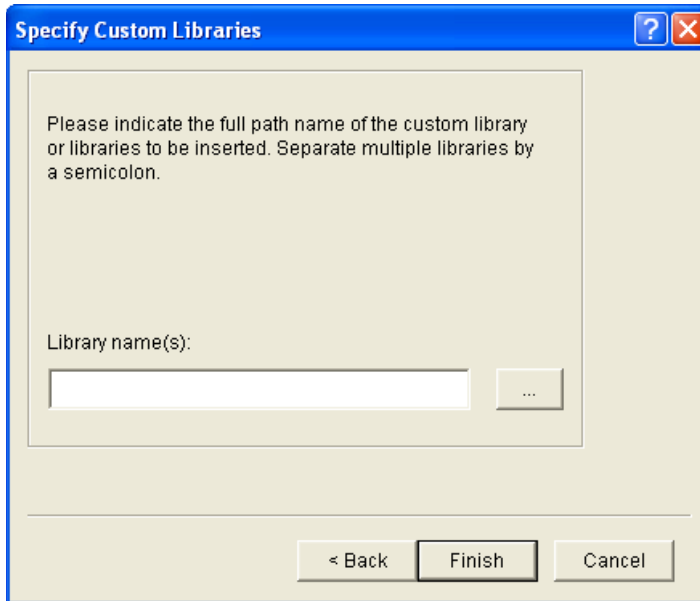
Psg.lib The programmable serial gateway library. Provides serial I/O functions for the PSG/3 and PSG-20 programmable serial gateways. For more information, see the programmable serial gateway documentation.

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

SLTA.lib The SLTA-10 Serial LonTalk Adapter support library.

When you build the application image, the IzoT NodeBuilder tool first searches for the selected libraries in the folder within the Images folder that contains the system image for the target (for example, C:\LonWorks\Images\Ver18). If the libraries are not in the version folder, the libraries in the parent C:\LonWorks\Images folder are used.

6. If you selected **Custom Library** in step 4, the **Specify Custom Libraries** dialog opens. Enter the full path of the library or libraries to be added to the device template. You can enter multiple library files by separating the paths with a semi-colon. To browse to a library file, click the button to the right of the **Library Names** property and then browse to any file with the **.lib** extension. When you have finished specifying the custom libraries, click **Finish**.



Note: You can view a summary of the contents of any library file using the Neuron Librarian standalone tool. To do this, open a command prompt and enter the following command:

```
nlib -r <library file name>
```

To save the summary, redirect the output to a file using the following command:

```
nlib -r <library file name> > <text file name>
```

For more information on using the Neuron Librarian tool and other standard Neuron C tools that can be run standalone, see Appendix A of the *Neuron C Programmer's Guide*.

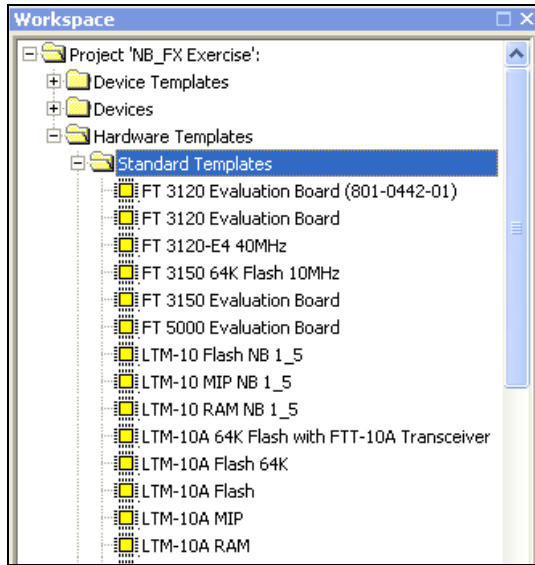
Using Hardware Templates

You can create new hardware templates or copies of existing ones and then configure them 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 IzoT 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 pane of the NodeBuilder Project Manager. 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 IzoT NodeBuilder tool. The Standard hardware templates are read-only;

however, you can use the **Insert Copy** feature to create your own custom hardware template based on a Standard template and then edit your custom template.



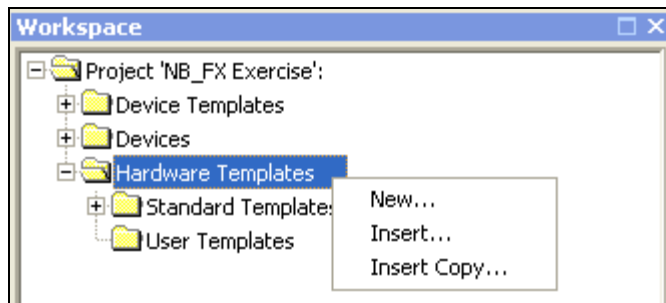
- The **User Templates** folder contains your custom hardware templates that can be used by all NodeBuilder projects on this computer. Any hardware templates unique to this project are located in the **Hardware Templates** folder, and are not contained in the **Standard Templates** or **User Templates** folders.

To create hardware templates, you do the following:

1. Create a new hardware template either using the **New** shortcut command on the **Hardware Templates** or **User Templates** folder, duplicating an existing hardware template using the **Insert** shortcut command on the **Hardware Templates** folder, or creating a copy of an existing hardware template using the **Insert Copy** shortcut command on the **Hardware Templates** or **User Templates** folder.
2. Set the hardware template properties for the new hardware template in the **Hardware Template Editor** dialog. You can also use this dialog to edit existing hardware templates or edit the hardware template being used in an existing device template.

Creating Hardware Templates

You can create new hardware template and add existing ones into your project from the Project pane. To do this, right-click the **Hardware Templates** or **User Templates** folders to open a shortcut menu that has the following options:



New

Creates a new hardware template to be added to the selected folder. Selecting this option opens the **Hardware Template Editor** dialog where you can create a new hardware template.

- If you are creating a hardware template in the **Hardware Templates** folder, it will be placed in the NodeBuilder project folder (for example, C:\Lm\Source\NB_FX Exercise).
- If you are creating a hardware template in the **User Templates** folder, the new User hardware template will be placed in the User hardware templates folder, which is C:\Lm\Source\Templates\Hardware by default. If this folder does not already exist on your computer, you will be prompted to create it.

You can set the default User hardware templates folder in the **Options** tab of the **NodeBuilder Project Properties** dialog. To access this dialog, click **Project** and then click **Settings**, or right-click the Project folder in the Project pane and click **Settings** on the shortcut menu.

You can create folders in the User hardware templates folder, but the IzoT NodeBuilder tool will only show them if they contain at least one hardware template.

Proceed to the next section, *Editing Hardware Templates*, to configure the hardware, memory, and description of the new hardware template.

Insert

References an existing hardware template and inserts it into the currently open NodeBuilder project. After you select this option, browse to and select an existing NodeBuilder device template file (**.NbHwt** extension) to be inserted into your current NodeBuilder project.

Note: This command is only available for the **Hardware Templates** folder.

Insert Copy

Creates a copy of an existing NodeBuilder hardware template, lets you modify the hardware template properties, and inserts the modified hardware template into the currently open NodeBuilder project.

After you select this option, browse to and select an existing NodeBuilder device template file (**.NbDt** extension) to be inserted into your current NodeBuilder project. After you select an existing hardware template, the **Hardware Template Editor** dialog opens.

Proceed to the next section, *Editing Hardware Templates*, to configure the hardware, memory, and description of the hardware template copy.

Note: You can also create a copy of an existing NodeBuilder hardware template by dragging a standard or user hardware template to the **Hardware Templates** folder.

Notes:

- 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 new 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, which opens the **Hardware Template Editor** dialog.

- Do not modify hardware templates in the **Standard Templates** folder because any changes that you make will be overwritten by future NodeBuilder updates. To modify a standard template, first insert a copy in the **User Templates** folder, and then edit the resulting custom template. Future upgrades of the IzoT NodeBuilder tool will not modify any user templates.
- You can remove project-specific hardware templates in the **Hardware Templates** folder. To do this, right-click the template and then click **Remove** on the shortcut menu. Note that removing a hardware template only removes the hardware template from the project; it does not delete the hardware template file.
- You cannot remove hardware templates in the **Standard Templates** and **User Templates** folders because they may be used by other NodeBuilder projects.

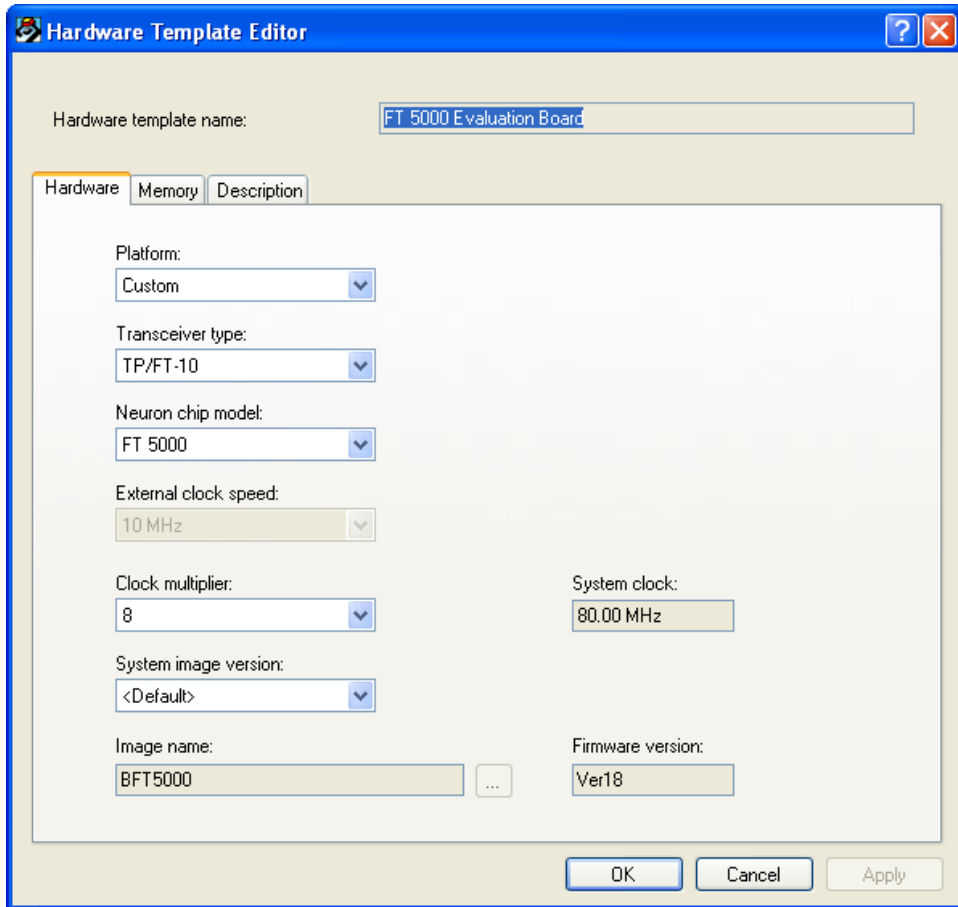
Editing Hardware Templates

When you create a new hardware template or create a copy of an existing one, you can configure the hardware, memory, and description properties of the new hardware template. You can also edit these properties for an existing hardware template or for a hardware template being used in an existing device template. The following sections describe how to set these properties.

Note: If you are editing a hardware template that is associated with a development or release target, the changes you make are also saved to the original hardware template in the **Hardware Templates** folder.

Setting Hardware Properties

You can set hardware properties for a hardware template on the **Hardware** tab of the NodeBuilder **Hardware Template Properties** dialog. If you open an existing template or create a new hardware template using **Insert Copy**, this tab will show the properties of the selected hardware template. If you create a new hardware template, it will contain the default values shown in the following image:



You can set the following properties on the **Hardware** tab:

Hardware Template Name

Enter the name of the hardware template. By default, new hardware templates are named Custom 1, Custom 2, and so on. The hardware template 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

A platform is a category of hardware implementations. Most hardware templates, including standard and user-defined hardware templates, are implemented using the **Custom** platform. The **Custom** platform is suitable for all user-defined hardware.

Other platform types are used for unique hardware implementations. For example, the **LTM-10** platform does not have a fixed transceiver type, and such flexibility may complicate the implementation.

Select one of the following hardware platforms:

- **Custom.** Select this if you are not using an LTM-10, LTM-10A, or LonBuilder Emulator. This is the default.
- **LTM-10.**
- **LTM-10A.**
- **LonBuilder Emulator 3150.**

<i>Transceiver Type</i>	<p>Select the transceiver type supported by the Neuron Chip or Smart Transceiver model selected in the Neuron Chip Model property. Each transceiver type identifies a unique set of transceiver parameters that are included in the application image. The default transceiver type is TP/FT-10.</p> <p>Select <Default> to use the project default transceiver. You can set the default transceiver in the Project tab of the NodeBuilder Project Properties dialog. To access this tab, click Project, click Settings, and then click the Project tab, or right-click the Project folder in the Project pane, click Settings on the shortcut menu, and then click the Project tab.</p>
<i>Neuron Chip Model</i>	<p>Select the Neuron Chip or Smart Transceiver model supported by the hardware platform selected in the Platform property. The default Neuron Chip model is FT 6000</p>
<i>External Clock Speed</i>	<p>Displays the frequency of the external crystal used for the Neuron Chip or Smart Transceiver model selected in the Neuron Chip Model property.</p> <p>For 6000 Series chips, the external crystal has a frequency of 10MHz; however, you can change the system's internal clock speed from 5MHz to 80MHz. To do this, you change the frequency at which the Neuron Chip or Smart Transceiver runs in the Clock Multiplier property.</p> <p>For 3100 Series chips, you can select a different clock speed from the list of those available for the selected Neuron Chip and transceiver type, or for the selected Smart Transceiver. This property is unavailable for those Neuron Chip or Smart Transceiver models that support only one external clock speed. See your Neuron Chip or Smart Transceiver data book for more information.</p>
<i>Clock Multiplier</i>	<p>For 6000 Series chips, you can select the frequency at which the Neuron Chip runs to modify the system clock speed. You can select multipliers of ½, 1, 2, 4, and 8. The default multiplier is 8.</p> <p>This property is fixed at ½ for the 3100 chip series.</p> <p>Note: If you modify this property and your NodeBuilder project is not associated with a LonMaker network, you must associate the NodeBuilder project with a LonMaker network and then load the device application with the IzoT NodeBuilder tool to implement the change. If you load the device application with the IzoT Commissioning tool without using the IzoT NodeBuilder tool's automatic load after build feature, the device may not use the correct clock speed.</p>
<i>System Clock</i>	<p>The effective clock speed of the internal system. For 6000 Series chips, this is the product of the External Clock Speed and the Clock Multiplier. The default internal system clock speed is 80.00 MHz (the crystal's speed external clock speed of 10MHz multiplied by the default clock multiplier of 8), and it may be as low as 5 MHz (10MHz * ½).</p> <p>Note: The 5.00 MHz system clock setting is intended only to facilitate backward compatibility with older designs that cannot scale to higher clock rates. There is no power consumption advantage to using 5.00 MHz over 10.00 MHz.</p> <p>For 3100 Series chips, this is the same value as the External Clock Speed multiplied by ½.</p>

System Image Version

Select the system image version for the selected Neuron Chip or Smart Transceiver model. See your Neuron Chip or Smart Transceiver data book for more information.

Select **<Default>** to use the default system image for the chosen chip. The default system image is the most current system image version included with this version of the IzoT NodeBuilder tool and any applied service packs.

Select **<Custom>** to specify your own custom system image in the Image Name property. See the *Neuron C Programmer's Guide* for information on creating custom system images.

Image Name

Displays the file name of the system image. If **<Custom>** is selected in the **System Image Version** property, you can enter a system image file name or click the button to the right and browse to a system image symbol file (.sym extension).

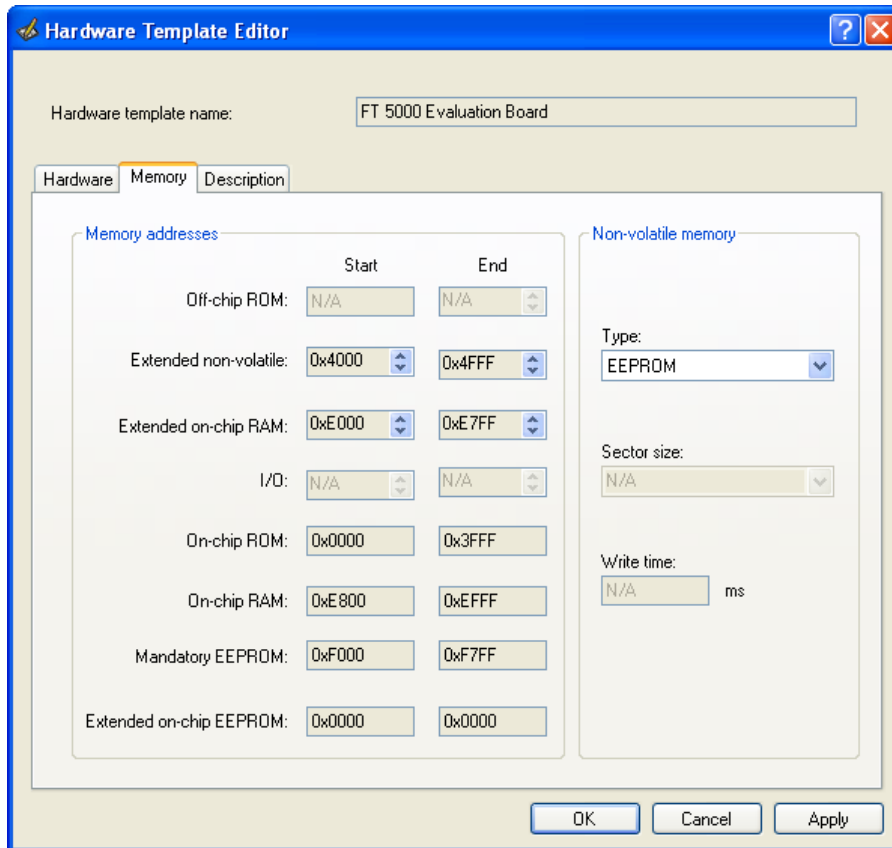
For 6000 Series chips, the name of the default system image is **BFT6000**.

Firmware Version

Displays the firmware version used by the selected system image if the **System Image Version** property is set to **<Default>**; otherwise N/A is displayed.

Setting Memory Properties

You can view and set the on-chip and off-chip memory properties for a hardware template on the **Memory** tab of the **NodeBuilder Hardware Template Editor** dialog.



The **Memory Addresses** box details how on chip and off-chip memory is organized on the selected Neuron Chip or Smart Transceiver model. These values are dependent on the chip type and may be modified depending on the Neuron chip model and available memory. You can modify the **Start** and **End** locations for available memory by clicking the arrows. A value of **0x0000** is displayed for any memory location that has not been set; **N/A** is displayed for any memory location that is not available.

The **Non-Volatile Memory** box specifies the type of external non-volatile memory (EEPROM, FLASH, and NVRAM) used, if any. If **EEPROM** is selected, the **Write Time** field specifies the EEPROM write time. If **Flash** is selected, the **Sector Size** field specifies the flash memory sector size.

The following sections describe the memory properties of the 5000 Series chips, 3150 Neuron core, and 3120 and 3170 Neuron core.

5000 Series Chips

The address ranges and consumption for the on-chip and off-chip memory of the 5000 Series chips are as follows:

<i>Off-Chip ROM</i>	The 5000 Series chips do not support off-chip memory; therefore, this property is set to N/A .
<i>Extended Non-Volatile</i>	<p>The 5000 Series chips use a serial memory interface for external non-volatile memory devices (EEPROM or flash). The application code and configuration data are stored in the external non-volatile memory device and then copied into the internal RAM when the device is reset. The device application is then executed from the internal RAM.</p> <p>The Extended Non-Volatile memory always starts at 0x4000 and can extend to a configurable address of less than 0xE7FF (a maximum of 42KB).</p> <p>Echelon currently supports and provides drivers for the following flash devices, which you can select from the Type property in the Non-Volatile Memory box: Atmel AT25F512AN, ST M25P05-AVMN6T, and SST25VF512A. See the Neuron Chip or Smart Transceiver data book for more information.</p> <p>Note: The drivers for different flash devices consume varying amounts of EEPROM code space because of the different programming algorithms required for the different flash devices. For example, the SST driver takes 40 bytes more of EEPROM than the other two supported flash devices.</p>
<i>Extended On-chip RAM</i>	The Extended On-chip RAM can start at a configurable address at or above 0x4000 or at the end of any extended non-volatile memory and must end at 0xE7FF .
<i>On-chip ROM</i>	The On-chip ROM is set from 0x0000 to 0x3FFF .
<i>On-chip RAM</i>	The On-chip RAM is set from 0xE800 to 0xEFFF .
<i>Mandatory EEPROM</i>	The On-chip EEPROM is set from 0xF000 to 0xF7FF . This reflects the fact that a minimum of 2K of external serial EEPROM is required for the 5000 Series chips.
<i>Extended On-chip EEPROM</i>	The 5000 Series chips do not use Extended On-chip EEPROM; therefore, this property is set from 0x0000 to 0x0000 .

6000 Series Chips

The address ranges and consumption for the on-chip and off-chip memory of the 6000 Series chips are as follows:

<i>Off-Chip ROM</i>	The 6000 Series chips do not support off-chip memory; therefore, this property is set to N/A .
<i>Extended Non-Volatile</i>	<p>The 6000 Series chips use a serial memory interface for external non-volatile flash memory devices. The application code and configuration data are stored in the external non-volatile memory device and then copied into the internal RAM when the device is reset. The device application is then executed from the internal RAM.</p> <p>Echelon supports a number of compatible serial flash memory parts. You must select the part from the <i>Type</i> list to match your device hardware.</p> <p>You do not need to need to specify memory boundaries for a device based on a Series 6000 chip, because the Neuron Linker automatically configures the memory map to meet the combined requirements of the chip, the Neuron firmware and your application.</p>
<i>Extended On-chip RAM</i>	
<i>On-chip ROM</i>	
<i>On-chip RAM</i>	
<i>Mandatory EEPROM</i>	You do not need to need to specify any of these memory boundaries for a device based on a Series 6000 chip, because the Neuron Linker automatically configures the memory map to meet the combined requirements of the chip, the Neuron firmware and your application.
<i>Extended On-chip EEPROM</i>	The 6000 Series chips do not use Extended On-chip EEPROM; therefore, this property is set from 0x0000 to 0x0000 .

3150 Neuron Core

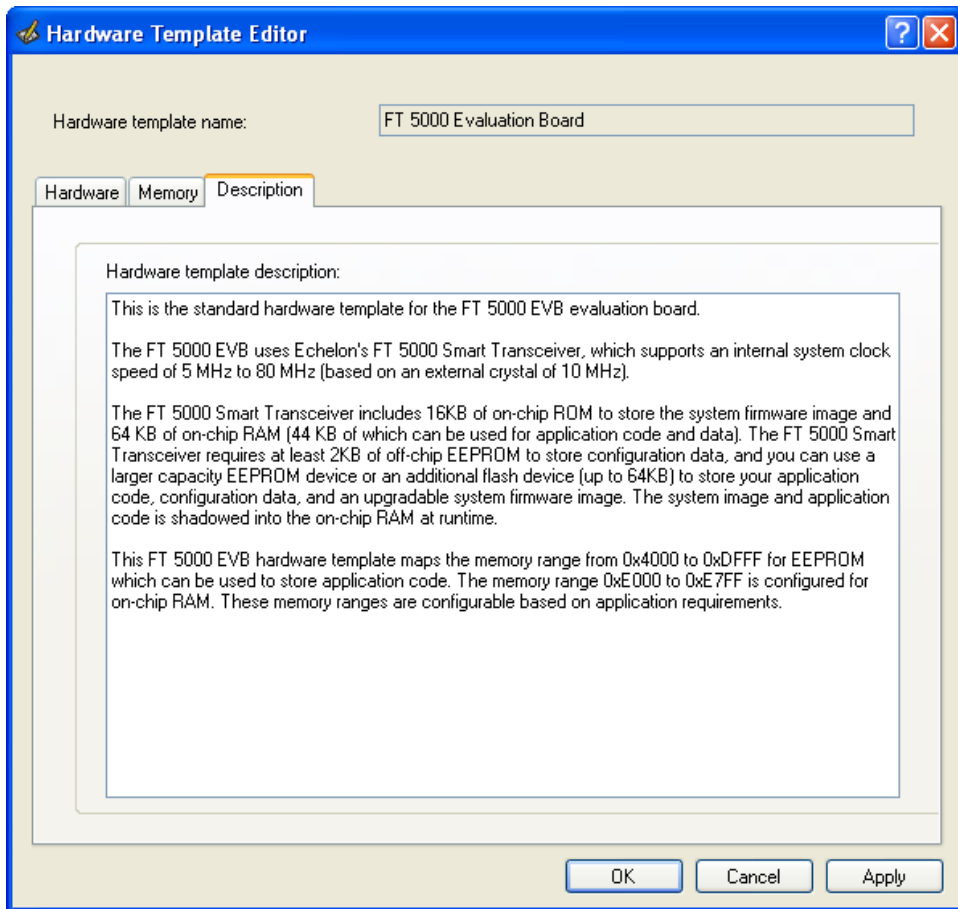
For Neuron 3150 Chips, 3150 FT Smart Transceivers, and 3150 PL Smart Transceivers, the on-chip memory values are dependent on the chip type and may not be modified with the exception of the **Extended On-chip RAM**. The **Type** property in the **Non-Volatile Memory** box specifies the type of non-volatile memory (EEPROM, FLASH, and NVRAM) used, if any. For devices where the system image is kept in non-volatile memory, select either **flash** or **NVRAM**. EEPROM is not supported for this configuration.

3120 and 3170 Neuron Core

For the Neuron 3120 Chips, 3120 FT Smart Transceivers, 3120 PL Smart Transceivers, and 3170 PL Smart Transceivers, the on-chip memory values are dependent on the chip type and may not be modified with the exception of the **Extended On-chip RAM**. These chips do not support off-chip memory, therefore, the **Off-Chip ROM**, **Off-Chip RAM**, **Off-Chip Non-Volatile** and **I/O** properties are set to **N/A**.

Setting the Hardware Template Description

You can enter an optional description for a hardware template in the **Description** tab of the NodeBuilder Hardware Template Properties dialog. This description will be saved in the hardware template file and will be available if this hardware template is used in other NodeBuilder projects.



Defining Device Interfaces and Creating their Neuron C Application Framework

This chapter describes how to use the NodeBuilder Code Wizard to define your device interface and generate Neuron C code that implements it. It explains how to start the NodeBuilder Code Wizard, how to add functional blocks, network variables, and configuration properties to your device template, and how to create the Neuron C framework for your device application.

Introduction to Device Interfaces

The NodeBuilder Code Wizard generates Neuron C source code that implements your device interface and creates the Neuron C framework for your device application. The device interface defines the functional blocks, network variables, and configuration properties implemented by your device. The framework created by the Code Wizard implements the most common device and functional block management tasks that are used in interoperable networks, and are required for certification of interoperable devices.

Functional blocks, network variables, and configuration properties are described as follows:

- Functional blocks group network variables and configuration properties into functional units that define desired system functionalities. Functional blocks define standard formats and semantics for how information is exchanged between devices on a network.
- Network variables allow devices to send and receive data over the network. Network variables are data items (such as temperature, the state of a switch, or actuator position setting) that a particular device shares with other devices on the network.
- Configuration properties define device behavior by determining how data is manipulated and when data it is transmitted, for example. Configuration properties control the application's algorithms, while network variables provide input and output to the algorithms. For example, a configuration property may specify a minimum change that must occur on a physical input to a device before the corresponding output network variable is updated. Configuration properties can be applied at the device, functional block, or network variable level. Configuration properties may be set during device installation, operation, and maintenance.

To create a device interface and the Neuron C framework for the device application, you do the following:

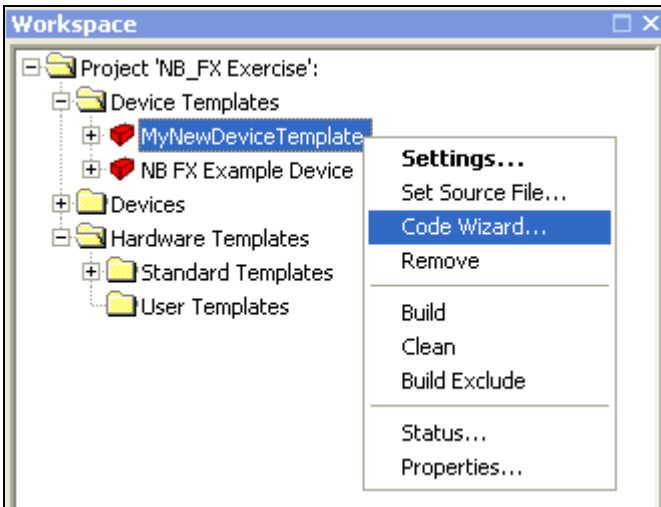
1. Start the NodeBuilder Code Wizard.
2. Define the device interface.
3. Generate the Neuron C code.

Starting the Code Wizard

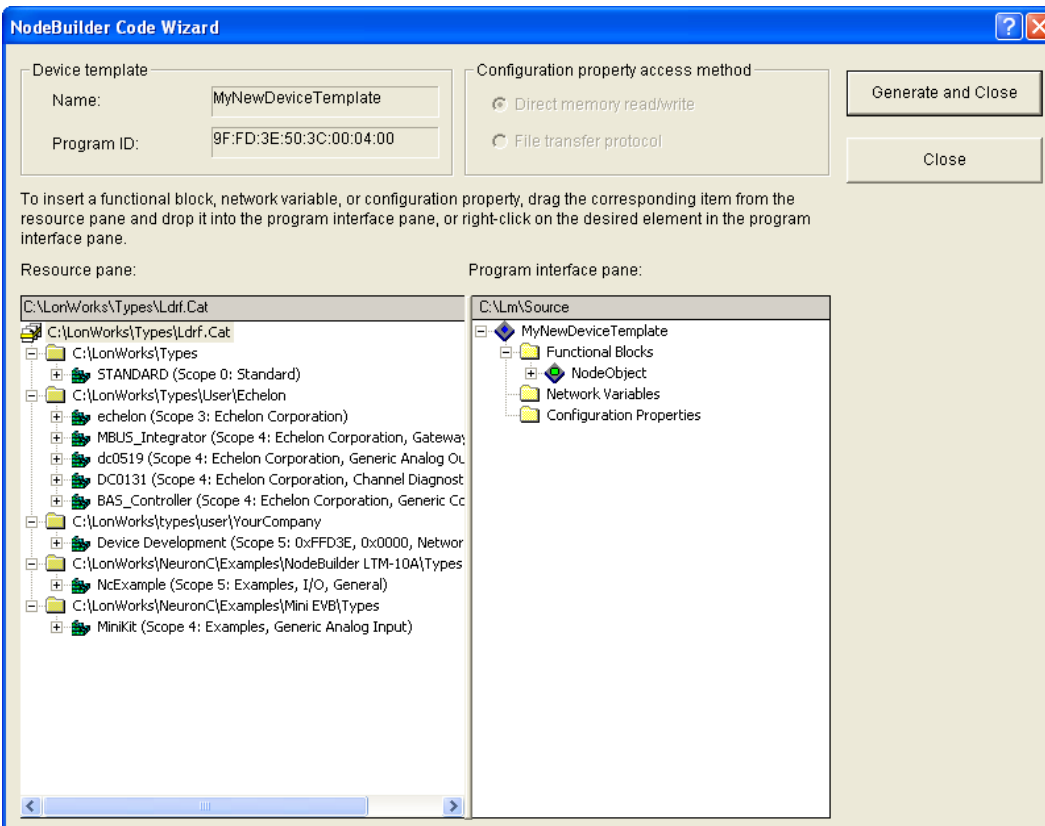
You can start the NodeBuilder Code Wizard when you are creating a new device template in the New Device Template wizard or any time from the NodeBuilder Project Manager.

To start the NodeBuilder Code Wizard when you are creating a new device template, select the **Run NodeBuilder Code Wizard** check box in the **Target Platforms** dialog of the New Device Template wizard. See *Creating Device Templates* in Chapter 5 for more information on setting this option in the NodeBuilder Code Wizard.

To start the Code Wizard from the NodeBuilder Project Manager, right-click a device template in the Project pane and click **Code Wizard** on the shortcut menu.



The NodeBuilder Code Wizard opens.



The NodeBuilder Code Wizard user interface is essentially divided into two panes: the Resource pane and the Program Interface pane. The following sections describe how to use these panes.

Using the Resource Pane

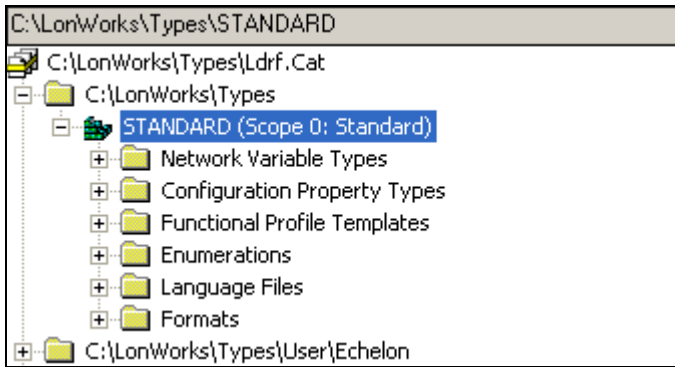
The Resource pane provides the full functionality of the NodeBuilder Resource Editor. It lists functional profiles, network variable types, and configuration property types, which are collectively referred to as *resources*. To define your device interface, you drag these resources from the Resource pane to the Program Interface pane as described in *Using the Program Interface Pane* later in this

section. For more information on creating and editing resource file sets and resources, see the *NodeBuilder Resource Editor User's Guide*.

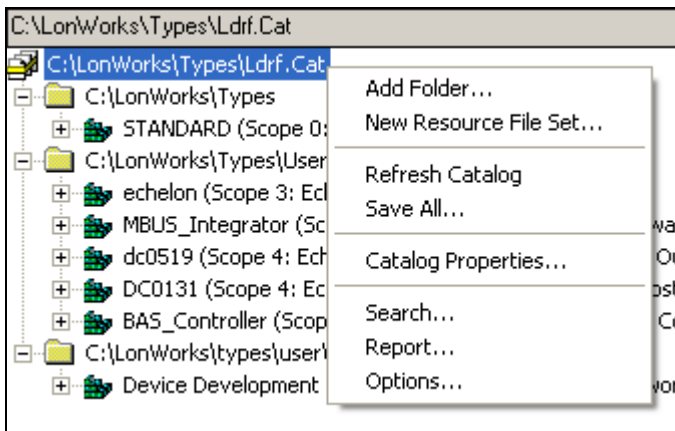
The Resource pane displays a hierarchical view of your resource catalog. The *resource catalog file* (📁) is at the top of the hierarchy. The resource catalog file is used to identify all the directories containing resource file sets. The resource catalog file has a **.Cat** extension. The default resource catalog file is **Ldrf.cat**, and it is stored in the **C:\LonWorks\Types** folder.

Below the resource catalog files are entries for each *resource folder* contained in the resource catalog. A resource folder may contain one or more *resource file sets*. By default, there is one resource folder (**C:\LonWorks\Types**) that contains the **STANDARD** resource file set. There may be a **C:\LonWorks\Types\User\Echelon** folder if you installed the LonPoint plug-in when you installed the IzoT Commissioning tool, or you may have additional resource folders if you have installed any other plug-ins, or already created your own resource files. You can add and remove resource folders in the resource catalog file from the Resource pane.

Each resource file set (📁) includes individual folders containing functional profile, network variable type, configuration property type, format, and language file resources.



Note: The Resource pane does not include the menu and toolbar displayed at the top of the NodeBuilder Resource Editor; however, you can access the commands provided by these components by right-clicking the Resource Catalog in the Resource pane.



Introduction to Resource File Sets

Resources are grouped into *resource file sets*, which apply 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. The scope 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 serves as a filter, indicating the relevant parts of the program ID.

The scope may be one of the following values:

Scope	Program ID Fields Used
0	Standard
1	Device Class
2	Device Class and Usage
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 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 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 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 0–2 are reserved for standard resource definitions published by Echelon and distributed by LONMARK International. Scope 0 applies to all devices; therefore, there is a single scope 0 resource file set called the *standard resource file set*. A standard resource file set is included with the IzoT NodeBuilder tool. You can download updated standard resource files from the LONMARK Web site at www.lonmark.org/technical_resources/resource_files.

You can define your own functional profiles, network variable and configuration property types, and formats in scope 3–6 resource files.

Introduction to Resources

Each resource file set may contain definitions for the following resources:

Resource	Description
<i>Network Variable Types</i>	Type information for network variables. This information includes the size, units, scaling factors, and type category (float, integer, signed, and so on) for each type. Network variable types can contain a single scalar value, a structure containing multiple fields (for example, the SNVT_switch network variable contains 2 fields for the value and state), or enumerated values that allow the network variable to be set to one of a discrete number of values. Network variables types are defined in a resource file with a .typ extension.
<i>Configuration Property Types</i>	Type information for configuration properties. This information includes the size, units, scaling factors, and type category (float, integer, signed, and so on) for each type. Like network variable types, configuration property types can contain scalar, structured, or enumerated values. Configuration property types are defined in a resource file with a .typ extension (this is the same file used for network variable types).

Resource	Description
<i>Functional Profiles</i>	<p>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.</p> <p>Each functional profile can define mandatory and optional network variables and configuration properties, which are collectively known as mandatory and optional member network variables and configuration properties. When a functional block implements a functional profile, it must implement all mandatory member network variables and member configuration properties defined by the functional profile, and it may implement some, all, or none of the optional member network variables and member configuration properties.</p> <p>Functional profiles are defined in a resource file with a .fpt extension. Functional profiles are also called <i>functional profile templates</i>. Functional blocks are implementations of functional profiles, and are formerly known as <i>LonMark objects</i>.</p>
<i>Enumerations</i>	<p>An enumeration type is a list of integral constants that are 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.</p> <p>Enumeration types are defined in a resource file with a .typ extension (along with network variable and configuration property types). C-language definitions of enumerations are also automatically generated in C-language header files (.h extension), which can be used to publish the enumeration type to the Neuron C compiler.</p>
<i>Language Files</i>	<p>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 <i>language files</i>. 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 values:</p>

Language	File Extension
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”

Resource	Description
	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"

Formats Each network variable and configuration property type must have at least one format defined. This format describes how the value will appear when using text-oriented visualization tools such as the LonMaker Browser. 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 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.

Using the NodeBuilder Resource Editor

You can use the NodeBuilder Resource Editor to create, modify, and view resources. The resource editor is a standalone application that you can start from the **NodeBuilder Project Manager**, or start independently from the **Echelon NodeBuilder** program folder. You can start the NodeBuilder Resource Editor by one of the following methods:

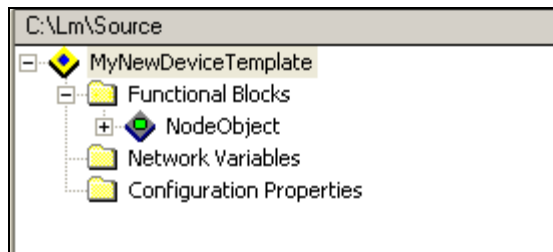
- Click **Start** on the taskbar, point to **Programs**, point to **Echelon NodeBuilder**, and then select **NodeBuilder Resource Editor**.
- From the **NodeBuilder Project Manager**, click **Tools** and then click **NodeBuilder Resource Editor**.

Note: If you are running the **NodeBuilder Code Wizard**, the Resource pane provides the full functionality of the NodeBuilder Resource Editor.

For more information on using the NodeBuilder Resource Editor, see the *NodeBuilder Resource Editor User's Guide*.

Using the Program Interface Pane

The Program Interface pane lists all the functional blocks, network variables, and configuration properties currently in the device interface. After you create a new device template, the Program Interface pane includes a tree view that has a device template object (📁) with three folders listed underneath it: **Functional Blocks**, **Network Variables**, and **Configuration Properties**.

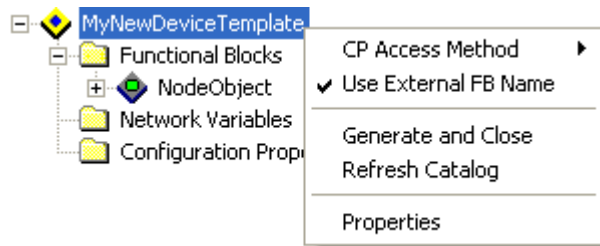


- The **Functional Blocks** folder contains all the functional blocks contained in this device interface.
- The **Network Variables** folder contains all the *device network variables* for this device interface. Device network variables belong to the device and therefore are not contained in any functional block. You can use device network variables to create a portion of your device interface for proprietary or legacy information.
- The **Configuration Properties** folder contains all *device configuration properties* for this device interface. Device configuration properties belong to the device and therefore are not contained in any functional block. You can use device configuration properties to create a portion of your device interface for proprietary or legacy information.

If you use device network variables or device configuration properties in your device interface, your device will not comply with interoperability guidelines version 3.4 (or better) and therefore cannot be certified by LONMARK.

A better alternative for adding members to a functional profile is to create a user-defined functional profile template (UFPT) that inherits from an existing standard functional profile template (SFPT), and then add new mandatory or optional member network variables to the UFPT. This method results in a new functional profile that you can easily reuse in new devices. See the *NodeBuilder Resource Editor User's Guide* for more information on creating new functional profiles.

You can right-click the device template to open a shortcut menu that has the following commands:



CP Access Method

Configuration properties may be accessed using read and write network management commands, or they be accessed using the LONWORKS File Transfer Protocol (FTP). Select this option to choose the configuration property access method: **Direct Memory Read/Write** (recommended) or **File Transfer Protocol**.

- **Direct Memory Read/Write.** This method requires less space and code on the target device. This is the recommended option and the default. When this option is selected, the Code Wizard automatically implements the Node Object functional block's optional **nvoFileDirectory** network variable. The optional **nviFileReq**, **nviFilePos**, and **nvoFileStat** network variables may not be in the Node Object functional block when this option is selected.
- **File Transfer Protocol.** When this option is selected, the Code Wizard automatically implements the Node Object functional block's optional **nviFileReq**, **nviFilePos**, and **nvoFileStat** network variables. The optional **nvoFileDirectory** network variable may not be in the Node Object functional block when this option is selected.

You can also select one of these options in the **Configuration Property Access Method** box at the top of the user interface.

The NodeBuilder 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 a standard functional block that is used by network management tools to test and manage the other functional blocks on your device and is also used to report alarms generated by your device.

If you remove the Node Object functional block, the Code Wizard cannot generate code for your device interface. See the *LonMark Application Interoperability Guidelines* for more information about the Node Object.

Use External FB Name If this option is enabled, the IzoT Commissioning tool and other network management tools use the functional block name set in the Code Wizard (for example, Switch or LED). This option is enabled by default.

If this option is disabled, network management tools use the functional profile name (for example, **SFPTopenLoopSensor** or **SFPTopenLoopActuator**).

Generate and Close Creates the Neuron C code framework for your device interface and closes the NodeBuilder Code Wizard. You can also create the Neuron C code by clicking the **Generate and Close** option in the upper right-hand corner of the user interface.

To close the NodeBuilder Code Wizard without generating any code, click the **Close** option in the upper right-hand corner of the user interface.

Refresh Catalog Updates the **Program Interface pane** with any changes made to the network variable types, configuration property types, or functional profiles used by the device template that are listed in the Resource pane.

If you change the name of a network variable type, configuration property type, or functional profile, it will be removed from the device interface when the NodeBuilder Code Wizard is refreshed and must be re-added.

You can also refresh an individual functional block or the device's **Network Variables** and **Configuration Properties** folders by right-clicking on them and then clicking **Refresh** on the shortcut menu.

Properties Select this option to open the **Device Template Properties** dialog. You can use this dialog to view the name, code template, and program ID of the device template; view the number of functional blocks, network variables, and configuration properties (both CPNVs and file CPs) in the device interface; and view the configuration property access method.

For new device interfaces created with the NodeBuilder FX tool, you can change the code template used for the device application in the **Framework Version** property. See *Using Code Wizard Templates* later in this chapter for more information.

You can add text to be included in the device's self-documentation string in the **Self-Documentation** box in this dialog. .

Defining the Device Interface

The *device interface* consists of the functional blocks, network variables, and configuration properties that let your device communicate with other LONWORKS devices and allow it to be configured by network tools.

A network variable defines an operational input or output for the device. The structure, range, units, and format of the network variable are defined by a *network variable type*.

A configuration property specifies a configuration option for a network variable, a functional block, or the entire device. The structure, range, units, and format of a configuration property are defined by a *configuration property type*.

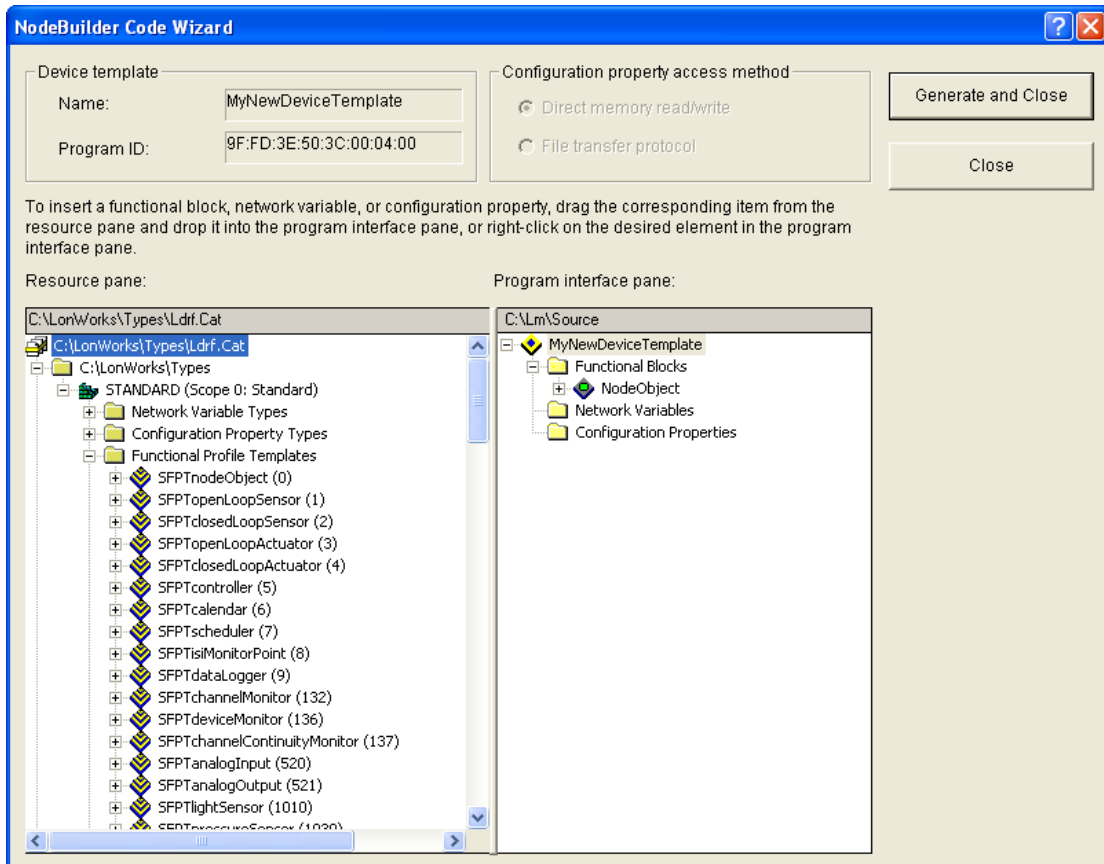
A functional block groups network variables and configuration properties that are related to a particular function for the device. Each functional block is defined by a *functional profile* that specifies the mandatory network variables and configuration properties that the functional block must implement, and the optional network variables and configuration properties that the functional block may implement. The functional profile also defines the behavior that the functional block must implement.

Functional profiles, network variable types, and configuration property types are defined in *resource files*. 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. The IzoT NodeBuilder tool includes a Standard resource file set, which defines many standard functional profile templates (SFPTs), standard network variable types (SNVTs), and standard configuration property types (SCPTs) that you can use for your device interface.

If you need additional functional profiles or types that are not defined in the standard resource file set, you can create your own user-defined functional profiles (UFPTs), user-defined network variable types (UNVTs), and user-defined configuration property types (UCPTs). For more information on resource files, including how to create user-defined functional profiles and types, see the *NodeBuilder Resource Editor User's Guide*.

To define your device interface, you first determine the functional profiles to be implemented by your device. To select the functional profiles to be implemented by your device, you first browse the SFPTs in the standard resource file set in the Resource pane. LONMARK International publishes documentation for some standard functional profiles, which details the behavior expected from each functional block that implements a given functional profile. You can view these functional profile documents on the LONMARK Web site at http://www.lonmark.org/technical_resources/guidelines/functional_profiles.

Note: The following graphic shows the functional profiles sorted by index; by default, they are sorted by name. To change how the functional profiles are sorted, right-click the resource catalog in the Resource Pane, click **Options** on the shortcut menu, select **By Index** in the **Sort By** box, and then click **OK**.



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 key* or *FPT key*. If your device is a simple sensor or actuator, you can use functional profiles 1–4: **SFPTopenLoopSensor (1)**, **SFPTclosedLoopSensor (2)**, **SFPTopenLoopActuator (3)**, or **SFPTclosedLoopActuator (4)**. If your device is more complex, you can browse the other SFPTs in the Standard resource file set for any suitable standard profiles have been defined.

If you cannot find an SFPT that fits your device, you can define a user-defined functional profile template (UFPT). You can create a UFPT from scratch, or you can create a UFPT that inherits from a SFPT and then add network variables and configuration properties to the UFPT. See the *NodeBuilder Resource Editor User's Guide* for more information on creating UFPTs.

After you determine the functional profiles that your device interface needs to implement, you do the following to finish defining your device interface:

1. Add functional blocks.
2. Edit mandatory network variables.
3. Implement desired optional network variables.
4. Implement desired optional configuration properties.

If your device needs network variables or configuration properties that are not included in any functional profile, you can create a UFPT that inherits from a SFPT as described in the *NodeBuilder Resource Editor User's Guide*. Alternatively, you can add implementation-specific network variables and configuration properties to the device interface; however this is not recommended because your device will not pass LONMARK certification.

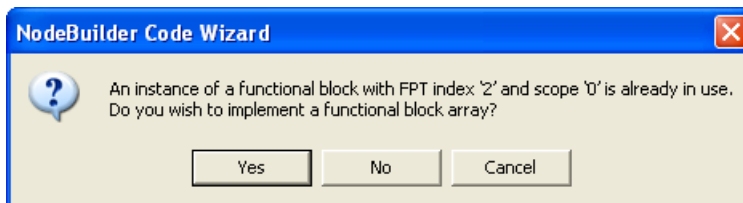
Adding Functional Blocks

Functional blocks represent specific device functions. For example, a device could have four hardware digital inputs, and digital would have its own functional block. To add a functional block to a device template, follow these steps:

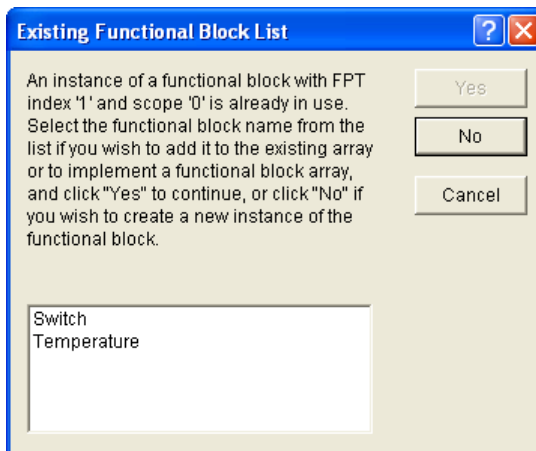
1. Drag the desired functional profile from the Resource Pane to the **Functional Blocks** folder in the Program Interface pane. A new functional block with the same name as the functional profile (without the SFPT or UFPT prefix, and truncated to 16 characters or less) is added to the device interface.

For example, dragging a **SFPTsccChilledCeiling** functional profile to the Program Interface pane creates a functional block named **sccChilledCeilin**. If you add more functional blocks from the same functional profile without changing the default functional block name, an index is appended to the name in order to maintain unique functional block names. The functional blocks are sorted by name.

2. If you added a functional profile of the same type and scope as an existing one, a dialog opens and prompts you whether you want to create a functional block array. Click **Yes** to create an array of functional blocks. Click **No** to create a new functional block using the same functional profile.



If you have added a functional profile that has the same type and scope as two or more existing functional profiles in the device template, the **Existing Functional Block List** dialog opens. To create a functional block array or add the functional block to an existing array, select an existing functional block for which an array is to be created or select an existing functional block array and then click **Yes**. To create a new functional block using the same functional profile, click **No**.

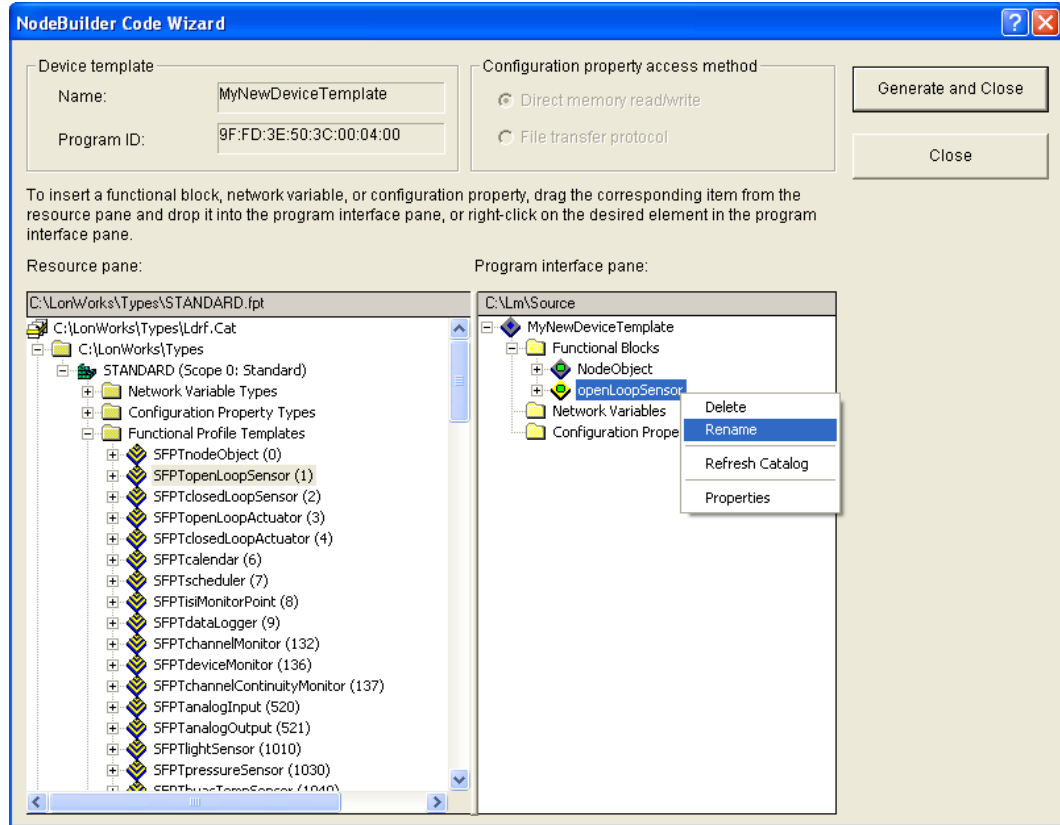


A functional block array is useful if your device contains two or more identical switches, lights, dials, controllers, or other I/O components that will each have an identical external interface. In addition, a functional block array saves code space and reduces the number of **when** clauses in your code. See *Using Large Functional Block Arrays* for how to manage your device application's memory when you are implementing large functional block arrays.

Note: You can still create a functional block array after adding the functional block to the device interface. To do this, right-click the new functional block in the Program Interface pane, and then click **Properties** on the shortcut menu. The **Functional Block Properties** dialog opens. Select

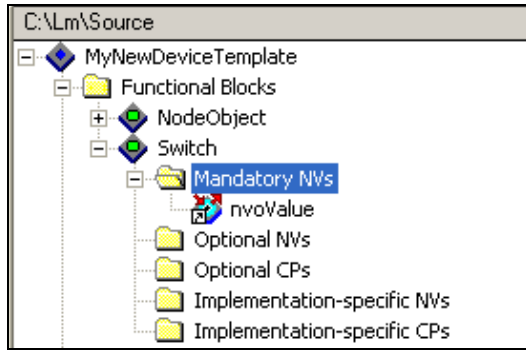
the **Use Array** checkbox, enter the number of functional blocks in the array in the **Size** box, and then click **OK**.

3. In the Program Interface pane, right-click the new functional block and then select **Rename** on the shortcut menu to change the name of the functional block. LNS network tools use this name is used to identify the functional block. This name is not case sensitive; however, creating a functional block, removing it, and then creating another functional block with different capitalization can cause compilation problems, and is therefore not recommended.



Note: You can have the IzoT Commissioning tool and other network management tools identify the functional block by its functional profile name instead of the user-specified name. To do this, right-click the device template in the Program Interface pane, and disable the **Use External FB Name** option.

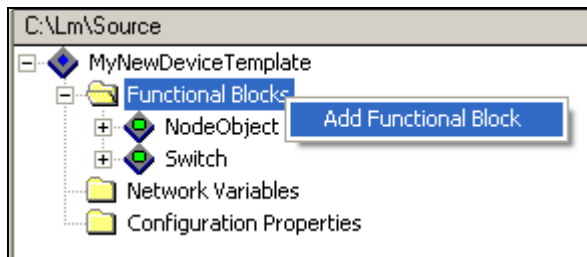
4. All the mandatory network variables and configuration properties specified by the functional profile are automatically added to the **Mandatory NVs** and **Mandatory CPs** folders under the functional block. These folders only exist only if the functional profile contains mandatory network variables or configuration properties. Mandatory items can not be deleted from the functional block. You can expand these folders to display the mandatory members in the functional profile.



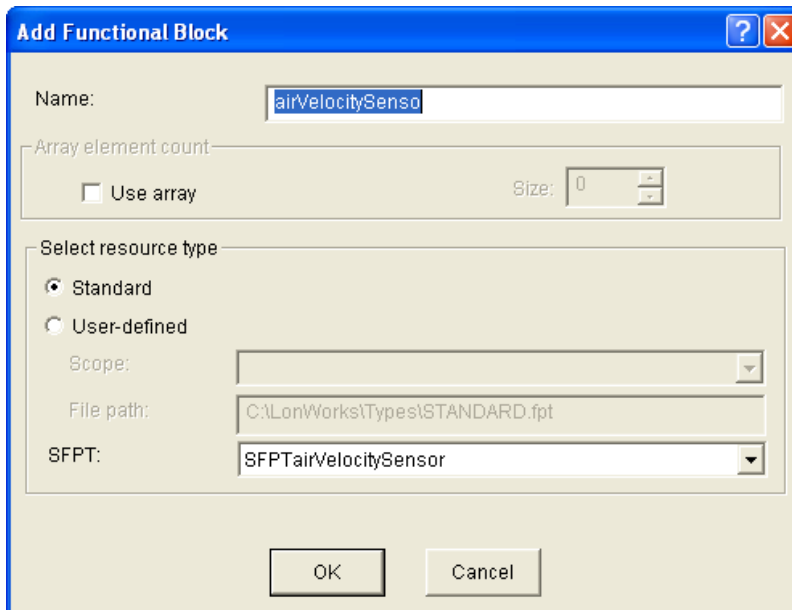
5. If any of the mandatory network variables do not have a default type set by the functional profile (for example, the **nvoValue** network variable in the **openLoopSensor** profile), you need to set the network variable type. See *Editing Mandatory Network Variables* for more information on how to do this and edit other network variable properties.

Alternatively, you can add a functional block directly from the Program Interface pane following these steps:

1. Right-click the **Functional Blocks** folder in the Program Interface pane and click **Add Functional Block** in the shortcut menu.



2. The **Add Functional Block** dialog opens.



3. In the **Select Resource Type** box, select whether the functional block you are adding is based on a **Standard** or **User-Defined** profile. If you select a **User-Defined** profile, select the **Scope** of the functional profile.

4. In the **SFPT** or **UFPT** property, select the desired functional profile template.
5. In the **Name** property, enter a name for your functional block.
6. To create a functional block array, select the **Use Array** checkbox, and then enter the number of functional blocks in the array in the **Size** box. See *Using Large Functional Block Arrays* for how to manage your device application's memory when you are implementing large functional block arrays.
7. Click **OK**.
8. In the Program Interface pane, set the network variable type for any mandatory network variables that do not have a default type set by the functional profile.

Using Large Functional Block Arrays

Implementing member network variables that apply to a functional block array of x elements requires one network variable per functional block array element for each member network variable. Implementing a single member network variable will therefore require x network variables, implemented as an array of x network variables. Most functional profiles specify more than one mandatory network variable (m), and will require $m*x$ network variables.

Devices based on Neuron chips that use version 16 firmware or greater (for example, the 5000 or 6000 Series chips) support up to 254 static network variables (this limit is subject to available system resources and application requirements). Storage for network variable values is by default allocated to the NEAR RAM segment. The NEAR RAM segment allows accessing these variables with the most efficient code (smaller, faster) compared to linking those network variable values into on-chip or off-chip FAR RAM segments. However, the Neuron Chip's hardware architecture limits the NEAR RAM segment to 256 bytes in total, shared among global application and system variables and network variables.

Implementing very large functional block arrays, implementing smaller functional block arrays with large numbers of member network variables, or generally implementing large numbers of functional blocks (or network variables or application variables in general) will eventually exhaust the NEAR RAM segment, and it will cause compiler or linker errors.

You will need to select variables for FAR RAM segments and NEAR RAM segments, respectively. You will typically try to allocate the most frequently accessed and most time-critical ones into NEAR RAM, permitting inherent limitations.

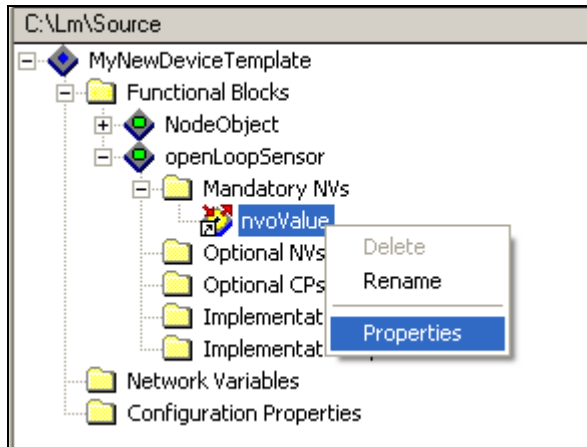
To change the allocation rules, double click a member network variable, then click **Advanced** in its properties dialog, and select **far** to force a variable out of the NEAR RAM segment. See Chapter 8 of the *Neuron C Reference Guide* for more information about using RAM in your Neuron C application.

Editing Mandatory Network Variables

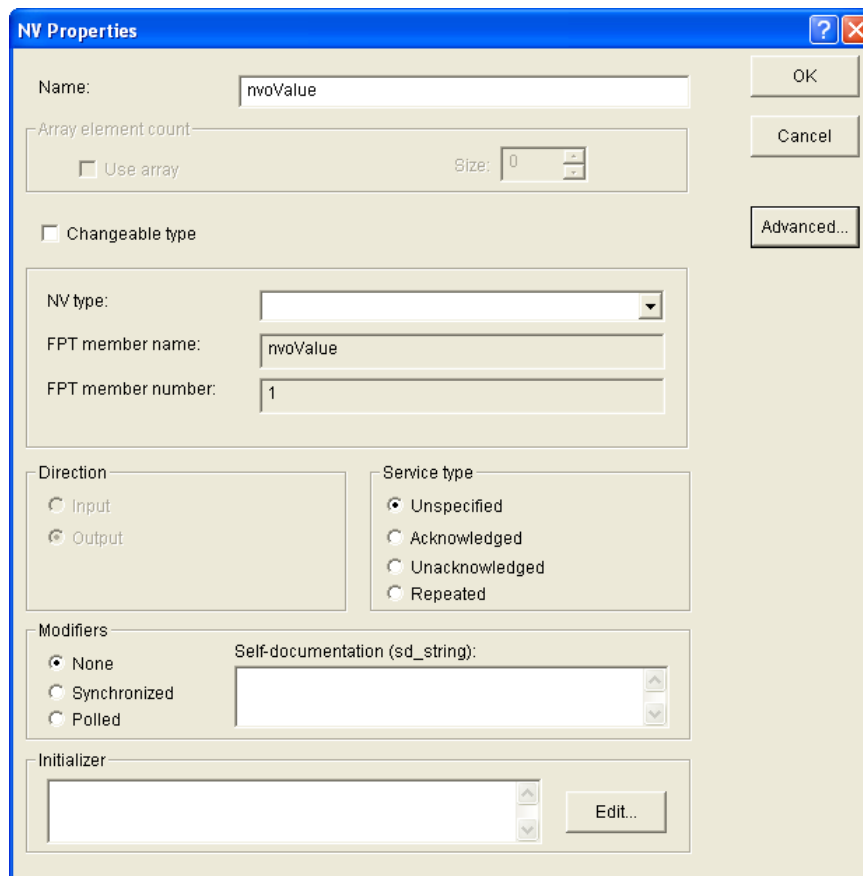
When you add a functional block to your device interface, all the mandatory network variables specified by the functional profile are automatically added to the **Mandatory NVs** folder under the functional block. The functional profile provides defaults for all the properties of the network variables; however, you can edit some of the properties. For example, you can set the network variable type if the network variable does not have one set for it by the functional profile (for example, the **nvoSwitch** member network variable in the **openLoopSensor** profile is defined using the placeholder type **SNVT_xxx**), change the modifiers and messaging service used if the network variable is an output network variable, set the initial value for the network variable when the device is reset, and set the storage classes used by the network variable.

To edit a mandatory network variable, as well as optional and implementation-specific network variables, follow these steps:

1. Double-click the network variable or right-click the network variable and select **Properties** from the shortcut menu.



2. The NV **Properties** dialog opens.



3. Edit the following properties:

<i>Name</i>	<p>Displays the name of the network variable that will be used in the IzoT Commissioning tool and other network management tools. The default name is the functional profile name.</p> <p>You can change the name of the network variable. The name must be unique to the device, can contain up to 16 alphanumeric characters, and must start with a letter. The name cannot contain spaces or the following characters: \ / : * ? “ < > .</p> <p>You can also rename the network variable by right-clicking it in the Program Interface pane and then clicking Rename on the shortcut menu.</p>
<i>Array Element Count</i>	<p>The Use Array check box indicates whether the functional block containing the network variable is an array (selected if the functional block has been implemented as an array; cleared otherwise). If the functional block has been implemented as an array, the Size box displays how many functional blocks are in the array.</p> <p>This network variable will be implemented in each functional block in the array. This information can be useful when determining how many network variables have been created on the device. This field is read-only.</p>
<i>Changeable Type</i>	<p>Enables network integrators to change the type of this network variable. This lets you 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 selecting this check box enables a network integrator to change this network variable type to a range of other types to allow the PID controller to control, light, pressure, or other types.</p> <p>This option is only available if the Has Changeable Interface option was selected in the Standard Program ID Calculator, and if the functional profile defines no specific type for the network variable. See <i>Specifying the Program ID</i> in Chapter 5 for more information on setting this option.</p> <p>For more information on implementing changeable-type network variables in your Neuron C code, see <i>Using Changeable-Type Network Variables</i> in Chapter 7 and the <i>Neuron C Programmer’s Guide</i>.</p>
<i>NV Type</i>	<p>Displays the standard or user-defined type of the network variable. For implementation-specific network variables, you can change the network variable type; otherwise, this field is read-only.</p>
<i>FPT Member Name</i>	<p>Displays the name of the network variable as specified in the functional profile. For implementation-specific network variables, you can change the member name; otherwise, this field is read-only.</p>
<i>FPT Member Number</i>	<p>Displays the member number of the network variable as specified in the functional profile. For implementation-specific network variables, you can change the member number; otherwise, this field is read-only.</p>
<i>Direction</i>	<p>Displays the direction of the network variable as specified in the functional profile (Input or Output). For implementation-specific network variables, you can change the direction; otherwise, this field is read-only.</p>

Service Type

Displays the service type used by the network variable to send updates as specified in the functional profile (**Unspecified**, **Acknowledged**, **Repeated**, or **Unacknowledged**). This property is only available for output network variables.

You can change the service type for mandatory and optional output network variables if the functional profile has not specified one, and you can change the service type for implementation-specific output network variables. The service types vary in reliability and resources consumed:

- **Unspecified.** The network management tool or integrator will determine which service type is used.
- **Acknowledged.** The sending device expects to receive confirmation from the receiving device or devices that a network variable update was delivered. The sending application is notified when an update fails, but it is up to the developer of the sending device to handle the notification in the device application. While acknowledged service is very reliable, it can create excessive message traffic, especially for large fan-out or polled fan-in connections. When acknowledged messaging is used, every receiving device has to return an acknowledgment. Acknowledged messaging can be used with up to 63 receiving devices, but an acknowledged message to 63 devices generates at least 63 acknowledgements—more if any retries are required due to lost acknowledgements.

Acknowledged service is the best choice for most network variable connections due to its superior reliability and performance.

- **Unacknowledged.** The sending device sends out the network variable update only once and does not expect any confirmation from the receiving device. This message service type consumes the least amount of resources, but is the least reliable.

Unacknowledged service is often used with data that is frequently repeated as part of the application's algorithm, where the occasional loss of an update might not be critical.

- **Repeated.** The sending device sends out a series of network variable updates, but does not expect any confirmation from the receiving device. Repeated service with three repeats has a 99.999% success rate in delivering messages. Repeated service provides the same probability of message delivery as acknowledged messaging with the same number of retries, with significantly lower network overhead for large multicast fan-out connections. For example, a repeated message with three retries to 64 devices generates four packets on the network, whereas an acknowledged message requires at least 64 packets. However, the repeated message in this case does not allow for the backlog estimation that an acknowledged message does.

Modifiers

Indicates whether the network variable uses the **Synchronized** or **Polled** modifiers. This property is only available for output network variables.

You can change the modifiers for mandatory and optional output network variables if the functional profile has not specified them, and you can change the modifiers for implementation-specific output network variables. This property may be one of the following values.

- **None.** The network variable is neither synchronous nor polled.
- **Synchronized.** The device sends all output network variable updates, and queues and processes 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 you select this option.

If the network variable is not synchronized, the device sends only the most recent output network variable update if the device application updates the output network variable multiple times before the application leaves the current *when*-task. Similarly, the device processes only the most recent input network variable update if the device receives multiple updates before the device application can process them.

Note that most network variables are not synchronized.

- **Polled.** Output network variable updates are sent only in response to a poll request from a device that reads this network variable.

Self-document (sd_string)

Optionally, you can enter comments to be appended to the self-documentation string for this network variable. This text can provide additional notes that can be accessed from a network tool.

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.

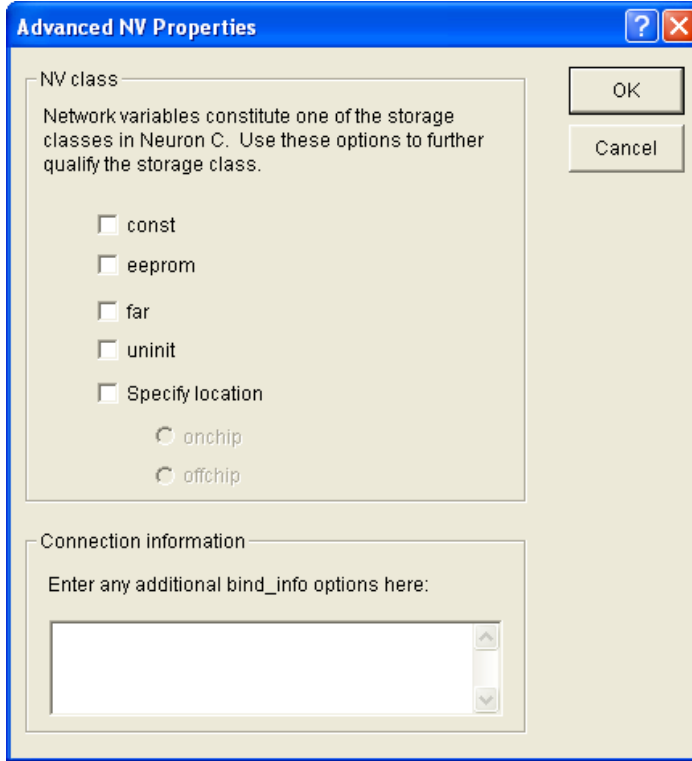
The total length of the self-documentation string can be up to 1024 characters, including the characters automatically generated by the Neuron C Compiler, any external functional block names, a semicolon to separate comments (if you enter comments in this box), your comments themselves (and possibly including formatting characters), and a terminating zero byte.

Initializer

Optionally, you can set the value for the network variable when the device is reset. If this network variable is a structure, union, float, or enumeration, click **Edit** to open the **Edit Initializer** dialog and enter the value or values. See *Setting Initial Values for Network Variables and Configuration Properties* later in this chapter for more information.

Note: Network variables are automatically reset to 0 during reset processing (except for those declared with the optional **eprom** modifier and those implementing configuration properties); therefore, they do not need to be explicitly initialized to 0.

- Optionally, you can click **Advanced** to open the **Advanced NV Properties** dialog and further specify the storage class used by the network variable. Consider a scenario where you declare a large network variable array that exceeds the limits of the near RAM segment, which is 256 bytes. In this case, you need to move the network variable array to the far RAM segment by selecting the **far** check box in this dialog. For more information on managing memory resources, see Chapter 8 of the *Neuron C Programmer's Guide*.



You can set the following properties and then click **OK** to return to the **NV Properties** dialog:

NV Class	You can specify the following storage classes for the network variable.
<i>const</i>	<p>The network variable is of const type. The Neuron C compiler will not allow modifications of const type variables by the device's program. However, a const network input variable will still be placed in modifiable memory and the value may change as a result of a network variable update from another device. Selecting this class disables the uninit keyword option.</p> <p>When used with the declaration of a configuration network variable (CPNV), the const storage class prevents both the Neuron C application and network tools from writing to the CPNV. The application may cast away the affects of the const type property to implement device-specific configuration properties as configuration network variables. However, the network variable will be placed in modifiable memory; therefore, network variable connections may still cause changes to such a configuration network variable.</p>
<i>eeprom</i>	<p>The network variable is placed in EEPROM or flash memory instead of RAM. All variables are placed in RAM by default. EEPROM and flash memory is only appropriate for variables which change infrequently because of the overhead and execution delays inherent in writing such memory, and the limited number of writes for such</p>

memory devices.

far

The network variable is placed in the far section of the variable space. By default, Neuron C variables are placed in the near RAM segment; however, the near RAM segment is limited to 256 bytes.

Accessing data in near RAM is faster and requires less code space than accessing data in far RAM. As a result, you may need to move some application variables or network variables into far RAM to make enough space in the near RAM segment for those variables that are either frequently accessed in a time-critical section of code, or accessed from many locations in your source code

The maximum size of near memory areas is 256 bytes of RAM and 255 bytes of EEPROM, but this may be less in some scenarios.

uninit

Prevents compile-time initialization of network variables. This is useful for **eprom** variables that do not or should not be written by program load or reload. Selecting this class disables the **const** option.

A different mechanism, subject to your network management tool, is used to determine whether configuration properties (including configuration network variables), will be initialized after loading or commissioning the device. The **uninit** keyword cannot be used to prevent configuration network variables from being initialized by the network management tool. See your network tool's documentation for details.

Specify Location

Select this check box to place the network variable in the off-chip portion or on-chip portion of the variable space, and then select one of the following options:

- **Offchip.** This keyword places the variable in the off-chip portion of the variable space. By default, the linker places variables in either space as it chooses, depending on availability. If the requested memory is not available, the link fails.
- **Onchip.** This keyword places the variable in the on-chip portion of the variable space. By default, the linker places variables in either space as it chooses, depending on availability. If the requested memory is not available, the link fails.

Connection Information

You can specify the Neuron C **bind_info** options. These options allow you to specify default connection information for this network variable (priority, authentication, service type, rate). For example, entering **authenticated(config) priority(config)** in this box generates the following line of code:

```
network <direction> <NV type> bind_info  
(authenticated(config) priority(config))
```

This is only required for **bind_info** options that are not handled by the **NV Properties** dialog. For example **ackd**, **unackd**, and **unackd_rpt** options are already handled by the **NV Properties** dialog.

Network management tools such as the IzoT Commissioning tool can override these settings. See the *Neuron C Reference Guide* for more information.

If you define a network variable to use priority, the device containing

the network variable must have priority enabled when it is installed. To enable priority on a device installed in a LonMaker network, right click the device, click **Properties** on the shortcut menu, select the **Advanced Properties** tab and then set the priority to **Enable - Automatic** or **Enable - Manual**. If you set the priority to **Enable - Manual**, you must also set the priority slot. Setting the priority to **Disable** disables priority.

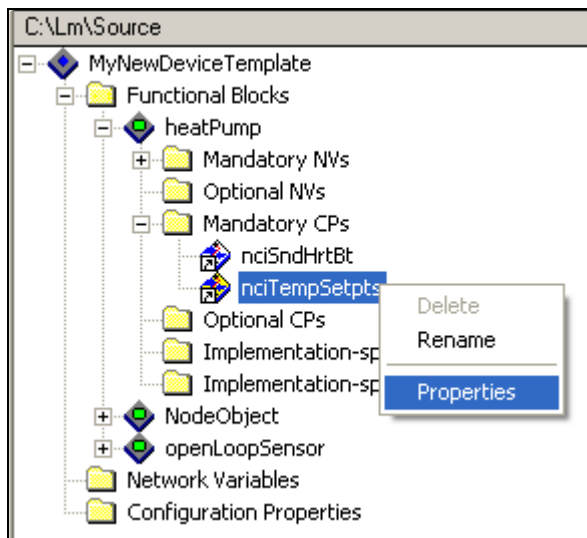
5. Click **OK**.

Editing Mandatory Configuration Properties

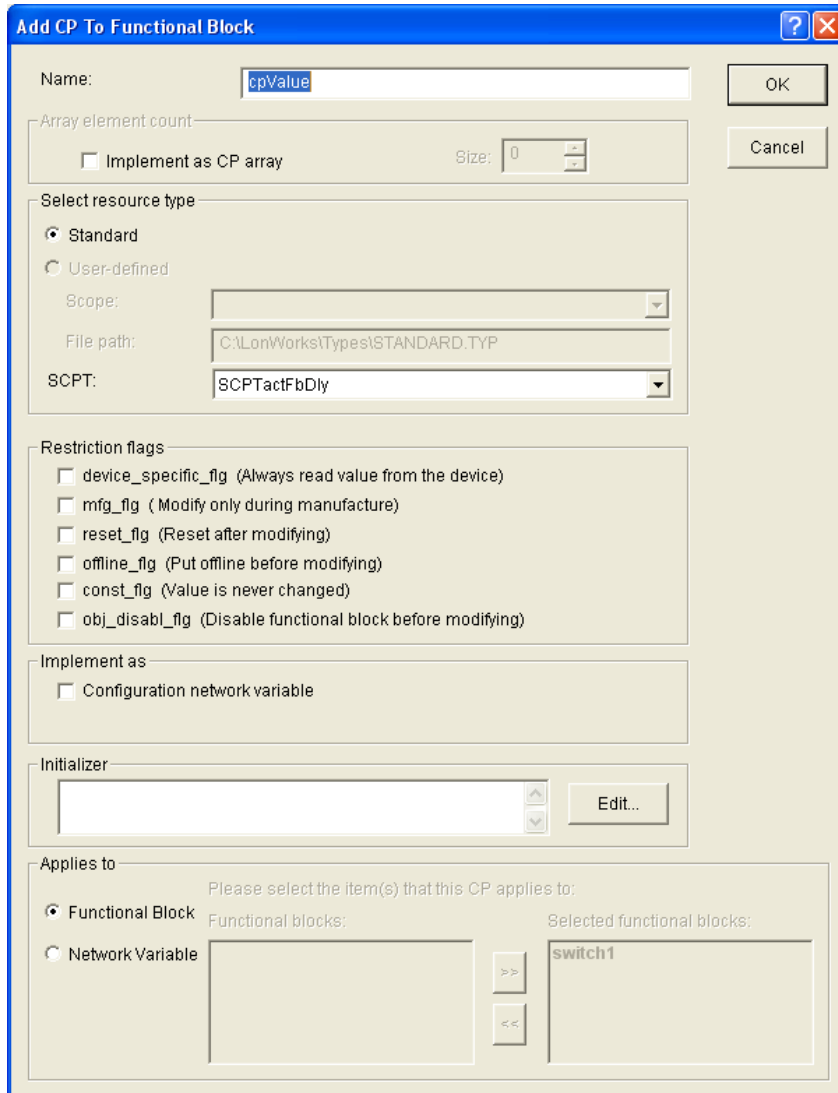
When you add a functional block to your device interface, all the mandatory configuration properties specified by the functional profile are automatically added to the **Mandatory CPs** folder under the functional block. The functional profile provides defaults for all the properties of the configuration properties; however, you can edit some of the properties. For example, you can implement the configuration property as an array (if allowed by the functional profile), set the configuration property flags, change how the configuration property is implemented (configuration network variable [CPNV] or configuration file [file CP]), and set the initial value for the configuration property after the device application has been downloaded to the device and the device has been reset.

To edit a mandatory configuration property, as well as optional and implementation-specific configuration properties, follow these steps:

1. Double-click the configuration property or right-click the configuration property and select **Properties** from the shortcut menu.



2. The **CP Properties** dialog opens.



3. Edit the following properties:

Name Displays the name of the configuration property that will be used in the IzoT Commissioning tool and other network management tools. The default name is the functional profile name.

You can change the name of the configuration property. The name must be unique to the device, can contain up to 16 alphanumeric characters, and must start with a letter. The name cannot contain spaces or the following characters: \ / : * ? " < > |.

You can also rename the configuration property by right-clicking it in the Program Interface pane and then clicking **Rename** on the shortcut menu.

Array Element Count A functional profile may require a configuration property to be implemented as an array and may enforce a minimum and maximum array size, or a functional profile may give you the option to implement the configuration property as an array and let you set the number of configuration properties in the array. If the functional profile template defines whether this configuration property must be implemented as an array or as a single configuration property, the

Implement as CP Array check box is set appropriately and unavailable.

If the functional profile template does not define how this configuration property must be implemented or if this is an implementation-specific configuration property, you have the option to configure the configuration property as an array or as a single configuration property.

To implement this configuration property as an array, select the **Implement as CP Array** check box (for configuration properties implemented as configuration files) or **Use Array** check box (for configuration properties implemented as configuration network variables [CPNVs]), and then specify the number of elements in the array in the **Size** box.

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, modifiable, memory in the device. A linker error will occur if the specified array size exceeds the device's resources.

Note: Configuration property arrays implemented as configuration network variables (CPNVs) are subject to the same limitations as network variables. Specifically, Neuron C applications are limited to 62 or 254 static network variables. In the case of a configuration property array implemented as CPNVs, each element in the array counts as one network variable.

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

<i>CP Type</i>	Displays the standard or user-defined type of the configuration property. This field is read-only.
<i>FPT Member Name</i>	Displays the name of the configuration property as specified in the functional profile. This field is read-only. For implementation-specific network variables, this field is empty.
<i>Restriction Flags</i>	<p>You can set the one or more of the following configuration property flags. Network tools are responsible for checking these flags and handling configuration properties appropriately.</p> <ul style="list-style-type: none">• device_specific_flg. Specifies that the configuration property should always be read from the device, never from the LNS database. An example use of this flag is for the device's minor version number. The minor version would be part of the application image download so network tools could check the version that was loaded into the device.• mfg_flg. Specifies that the configuration property should be modified only at manufacture time. Installation tools should not modify this configuration property unless they are being used for manufacturing. An example is a configuration property used to hold calibration data.• reset_flg. Specifies that the device should be reset after the configuration property is modified in order for the application to work properly.• offline_flg. Specifies that the configuration property should be

modified only when the device has been set offline by a network tool such as the IzoT Commissioning tool. Do not set this option if you are using FTP to transfer configuration properties because devices must be online to run the file transfer protocol.

- **const_flg**. Specifies that the configuration property should be considered read-only. It must not be written. It might be stored in ROM. An example of this kind of property is the device major version number.
- **obj_disabl_flg**. Specifies that the configuration property should only be modified when the functional block has been disabled by a network tool such as the IzoT Commissioning tool. The application will have a chance to initialize the functional block when the network management tool enables it.

Note: The **reset_flg**, **offline_flg**, and **obj_disabl_flg** flags comprise a hierarchy, where the **reset_flg** has the highest precedence and the **obj_disabl_flg** flag has the lowest.

For example, if you specify the **offline_flg** flag for a configuration property, the device will be set offline, but other steps or configuration property updates occurring at the same time might also require that the device be reset.

Similarly, the **offline_flg** flag, which applies to devices, has a higher precedence than the **obj_disabl_flg**, which applies to functional blocks. As a result, if you specify the **offline_flg** flag, it is expected that the device is offline or the functional block is disabled.

Implement As

You can specify the following implementation options for the configuration property.

Configuration Network Variable

Enables you to read, write, and bind the configuration property like a network variable. If this check box is cleared, the configuration property is implemented as a configuration file (file CP). This check box is cleared by default.

Note that CPNVs are have the following limitations: (1) they must be based on network variable types and must meet the target platform's network variable size limit (228 bytes starting with version 21 firmware, 31 bytes with earlier versions); and (2) if the CPNV is implemented as an array that applies to multiple functional blocks or network variables, the CPNV array must always be shared statically or globally. It is therefore recommended that you only use CPNVs if your application requires configuration properties that must be bound or if you are adding a **SCPTnwrkCnfg** configuration property.

Static CP

Creates a single configuration property that is shared by all the functional blocks in a functional block array. 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.

This property is only available if the configuration property is in a functional block array. Modifying the value of the configuration property on any functional block in the array modifies it for all of them (only one variable is allocated).

If this check box is cleared, a separate configuration property is created for each functional block in the array. This check box is

cleared by default.

Initializer

Optionally, you can set the value for the network variable when the device is reset. If this network variable is a structure, union, float, or enumeration, click **Edit** to open the **Edit Initializer** dialog and enter the value or values. See *Setting Initial Values for Network Variables and Configuration Properties* later in this chapter for more information.

Note: Configuration properties have default values that are defined in resource files. Default values are included in the definition of the configuration property type, in the definition of the functional profile's member configuration property (an optional initial value override), and possibly in the definition of an inherited functional profile. The Neuron C compiler will automatically initialize the configuration property to its defined default value.

Therefore, you can explicitly set the initial value or the configuration property; however, it is recommended that you use the default values defined in the resource file, if possible.

Applies to

Select whether the configuration property is applied to a network variable, a functional block, or the device as a whole. This is called *global configuration property sharing*.

- If this is an implementation-specific configuration property, you can apply it to the functional block or to any of the network variables on the functional block (any network variable in any of the Mandatory NVs, Optional NVs, or Implementation-specific NVs folders).
- If this is a device configuration property (a configuration property that you added to the Configuration Properties folder of a device), the configuration property can be applied to the device or to any of the network variables in the Network Variables folder.

Adding a Shared Configuration Property

To apply a configuration property to a functional block or network variable, follow these steps:

1. If another configuration property with the same type, array size, **Implement As** setting, and **Applies To** setting exists on the device, it will appear in **Applies To** property under **Network Variables** or **Functional Blocks**.
2. Select the **Functional Block** or **Network Variable** option.
3. Select one or more of the items from the **Network Variables** or **Functional Blocks** list and click the right arrow button to move the selected functional block or network variable to the **Selected Network Variables** or **Selected Functional Blocks** list.

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.

4. 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 Program Interface pane with the same configuration property name in their respective folders.

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.

For example, if you are creating a configuration property that applies to both a **nvoCO2ppm1** network variable on a **co2Sensor1** functional block and a **nvoCO2ppm2** network variable on a **co2Sensor2 functional block**; the Neuron C expression `co2Sensor1::nvoCO2ppm1::cpValue == co2Sensor2::nvoCO2ppm2::cpValue` will always be true because 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 shared configuration property, select the configuration property to be removed, and then click the left arrow button. 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 using global configuration property sharing:

- 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.
- All configuration property types can be shared.
- A configuration property that applies to the entire device cannot be shared.
- 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.

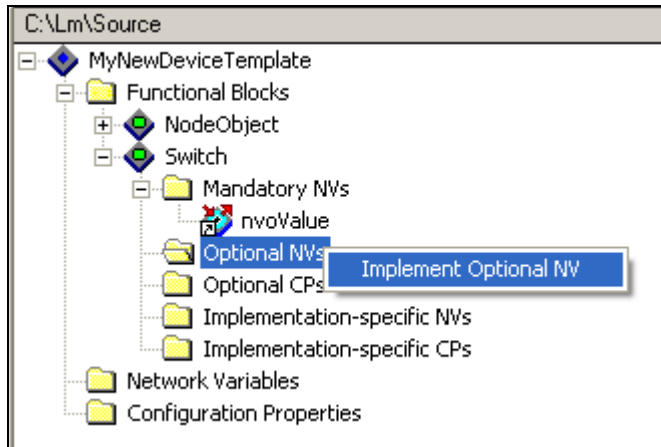
- A configuration property that is shared among the members of a functional block or network variable array must always be shared among all members of that array.
- A configuration property can be shared between network variables on different functional blocks.
- A configuration property that inherits 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.
- 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 (for example, same type, same array size, and so on).
- A single configuration property that inherits its type from a network variable cannot be shared simultaneously by both changeable and non-changeable network variables.
- 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.

5. Click **OK**.

Implementing Optional Network Variables

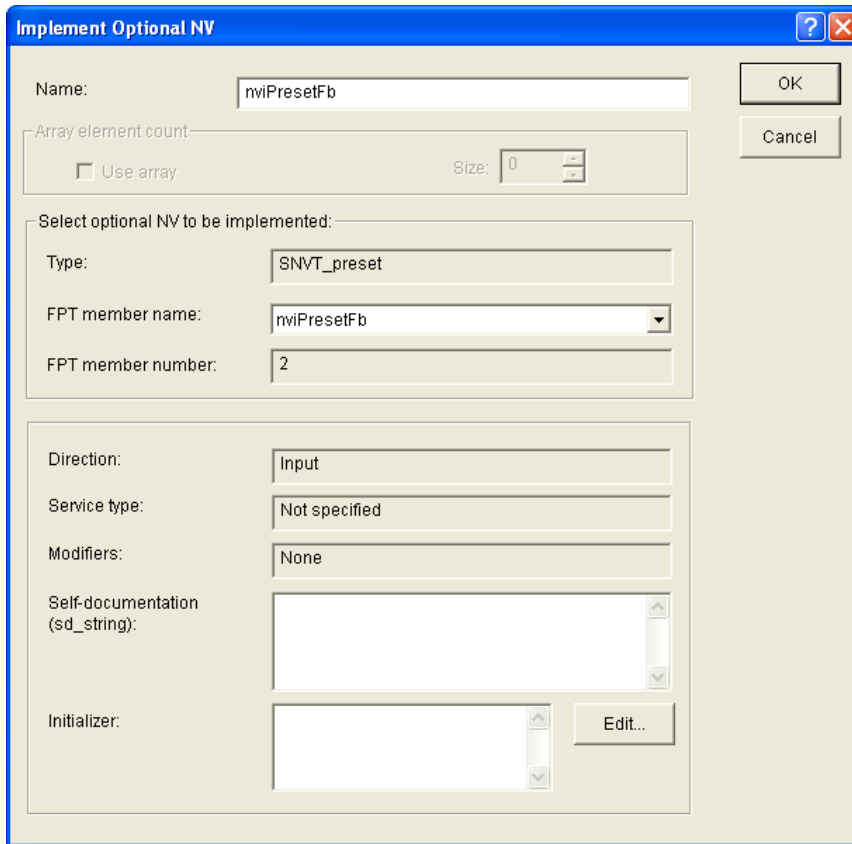
Functional profiles specify mandatory network variables that must be implemented by any implementation of the profile, and they also specify optional network variables that may be implemented but are not required. When a functional profile is added to the device interface in the NodeBuilder Code Wizard, the wizard adds all the mandatory members of the functional profile to the device interface, but it 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 Program Interface pane and select **Implement Optional NV** from the shortcut menu.



Alternatively, 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 Program Interface pane. If a functional profile does not have any optional network variables defined, it does not have an **Optional NVs** folder.

2. The **Implement Optional NV** dialog opens.



3. In the **FPT Member Name** property, select the optional network variable from the list of those that have not yet been implemented in this functional block.
4. In the **Name** property, enter the name of the optional network variable that will be displayed in the IzoT Commissioning tool and other network management tools. The default name is the functional profile name. This name must be unique to the device, can contain up to 16 alphanumeric characters, and must start with a letter.

The name cannot contain spaces or the following characters: \ / : * ? “ < > |.

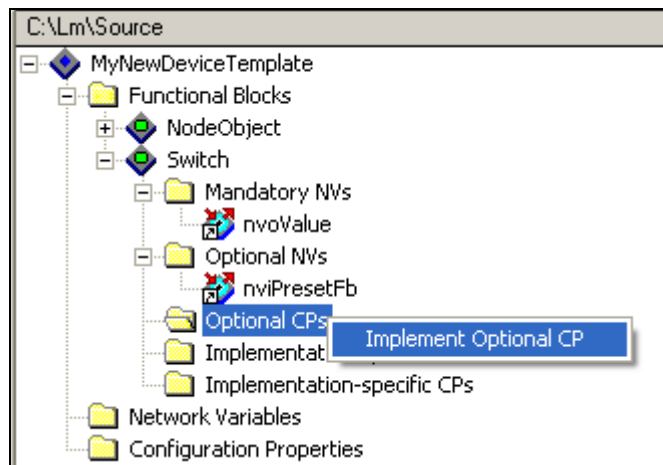
5. The **Use Array** check box in the **Array Element Count** box is a read-only property that indicates whether the optional network variable is implemented as a single network variable (the check box is cleared), or as an array of network variables that applies to an array of functional blocks (the check box is selected). If the optional network variable is implemented as an array of network variables, the **Size** box displays the number of elements in the functional block array, which is also the same number of elements that are in the network variable array (this enables one network variable to be allocated to each member of the functional block array).
6. Optionally, in the **Self-Documentation (sd_string)** property, you can enter comments to be appended to the self-documentation string for this network variable. This text can provide additional notes that can be accessed from a network tool.
7. Optionally, in the **Initializer** property, you can set the value for the network variable when the device is reset. If this network variable is a structure, union, float, or enumeration, click the box to the right to open the **Edit Initializer** dialog and enter the value or values. See *Setting Initial Values for Network Variables and Configuration Properties* later in this chapter for more information.
8. Click **OK**. The optional network variable is added to the **Optional NVs** folder.

Note: After you create the optional network variable, you can edit its properties following the steps described in *Editing Mandatory Network Variables* earlier in this chapter. For example, you may want to change the modifiers and messaging service used if the network variable is an output network variable, or you may want to set the storage classes used by the network variable.

Implementing Optional Configuration Properties

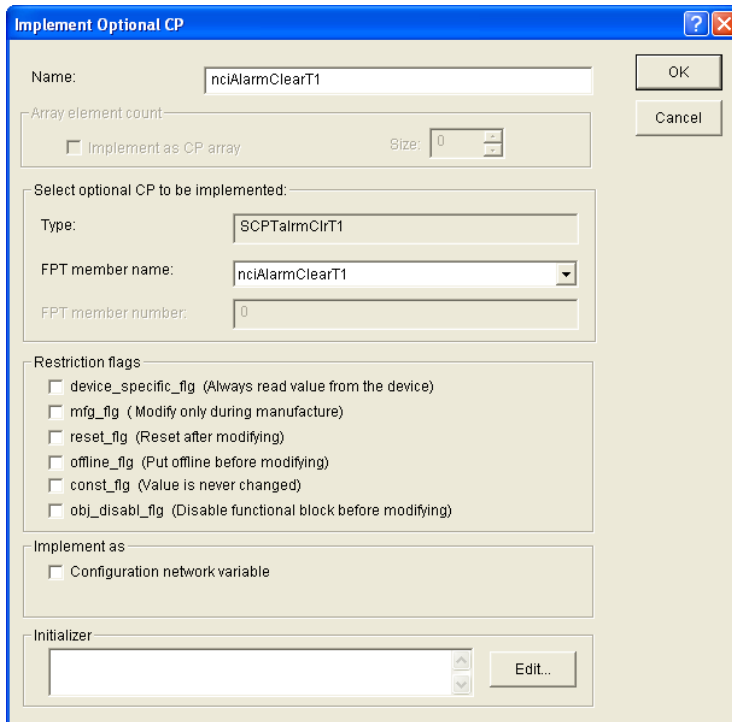
A functional profile specifies mandatory configuration properties that must be implemented by any implementation of the profile, and they may also specify optional configuration properties that may be implemented but are not required. When a functional profile is added to the device interface in the 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 Program Interface pane and select **Implement Optional CP** from the shortcut menu.



Alternatively, 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 Program Interface pane. If a functional profile does not have any optional configuration properties defined, it does not have an **Optional CPs** folder.

2. The **Implement Optional CP** dialog opens.



3. In the **FPT Member Name** property, select the optional configuration property from the list of those that have not yet been implemented in this functional block.
4. In the **Name** property, enter the name of the optional configuration property that will be displayed in the IzoT Commissioning tool and other network management tools. The default name is the functional profile name. This name must be unique to the device, can contain up to 16 alphanumeric characters, and must start with a letter. The name cannot contain spaces or the following characters: \ / : * ? " < > |.
5. To implement this configuration property as an array, select the **Implement as CP Array** check box (for configuration properties implemented as configuration files) or **Use Array** check box (for configuration properties implemented as configuration network variables [CPNVs]), and then specify the number of elements in the array in the **Size** box.

A functional profile may require a configuration property to be implemented as an array and may enforce a minimum and maximum array size, or a functional profile may give you the option to implement the configuration property as an array and let you set the number of configuration properties in the array. If the functional profile template defines whether this configuration property must be implemented as an array or as a single configuration property, the **Implement as CP Array** check box is set appropriately and unavailable.

If the functional profile template does not define how this configuration property must be implemented, you have the option to configure the configuration property as an array or as a single configuration property.

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, modifiable, memory in the device. A linker error will occur if the specified array size exceeds the device's resources.

Note: Configuration property arrays implemented as configuration network variables (CPNVs) are subject to the same limitations as network variables. Specifically, Neuron C applications are limited to 62 or 254 static network variables. In the case of a configuration property array implemented as CPNVs, each element in the array counts as one network variable.

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

6. In the **Restriction Flags** box, you can set configuration property flags that network tools must check in order to handle the configuration property appropriately. See *Editing Mandatory Configuration Properties* earlier in this chapter for more information on these flags.
7. To implement the configuration property as a configuration network variable (CPNV), select the **Configuration Network Variable** check box. This enables you to read, write, and bind the configuration property like a network variable. If this check box is cleared, the configuration property is implemented as a configuration file. This check box is cleared by default.
Note: CPNVs have the following limitations: (1) they must be based on network variable types and must meet the target platform's network variable size limit (228 bytes starting with version 21 firmware, 31 bytes with earlier versions); and (2) if the CPNV is implemented as an array that applies to multiple functional blocks or network variables, the CPNV array must always be shared statically or globally. It is therefore recommended that you only use CPNVs if your application requires configuration properties that must be bound or if you are adding a **SCPTnwrkCnfg** configuration property.
8. Optionally, in the **Initializer** property, you can set the value for the configuration property that will be stored in the LNS network database and set the first time the device is reset after the device application has been downloaded to the device. If this configuration property is a structure, union, float, or enumeration, click the box to the right to open the **Edit Initializer** dialog and enter the value or values. See *Setting Initial Values for Network Variables and Configuration Properties* later in this chapter for more information.
9. If the configuration property is member of a functional block array, you can select the **Static CP** check box to create a single configuration property that is shared by all the functional blocks in the array (this is called static configuration property sharing). Modifying the value of the configuration property on any functional block in the array modifies it for all of them (only one variable is allocated). If this check box is cleared, a separate configuration property is created for each functional block in the array. This check box is cleared by default.
10. Click **OK**. The optional configuration property is added to the **Optional CPs** folder.

Note: After you create the optional configuration property, you can edit its properties following the steps described in *Editing Mandatory Configuration Properties* earlier in this chapter. For example, you may want to change the configuration property flags, or change how the configuration property is implemented (configuration network variable [CPNV] or configuration file [file CP]).

Adding Implementation-specific Network Variables

You can add a network variable member to a functional block or device that is not defined by any 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 because they are not documented by a functional profile.

WARNING: If you use implementation-specific network variables in your device interface, your device will not comply with interoperability guidelines version 3.4 (or better) and therefore cannot be certified by LONMARK.

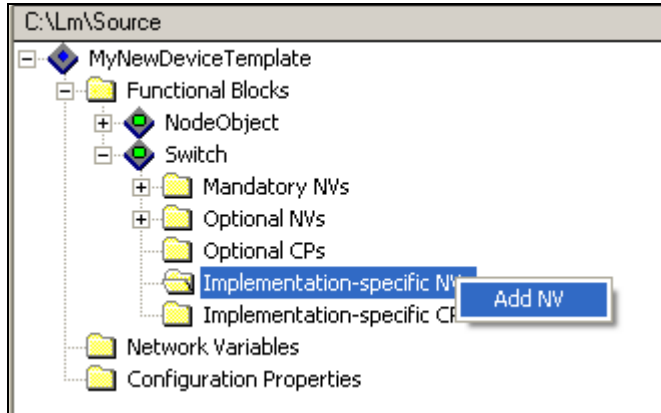
A better alternative for adding members to a functional profile is to create a user-defined functional profile template (UFPT) that inherits from an existing standard functional profile template (SFPT), and then add new mandatory or optional member network variables to the UFPT. This method results in a new functional profile that you can easily reuse in new devices. See the *NodeBuilder Resource Editor User's Guide* for more information on creating new functional profiles.

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 SFPT, but a SNVT may be added to a UFPT.

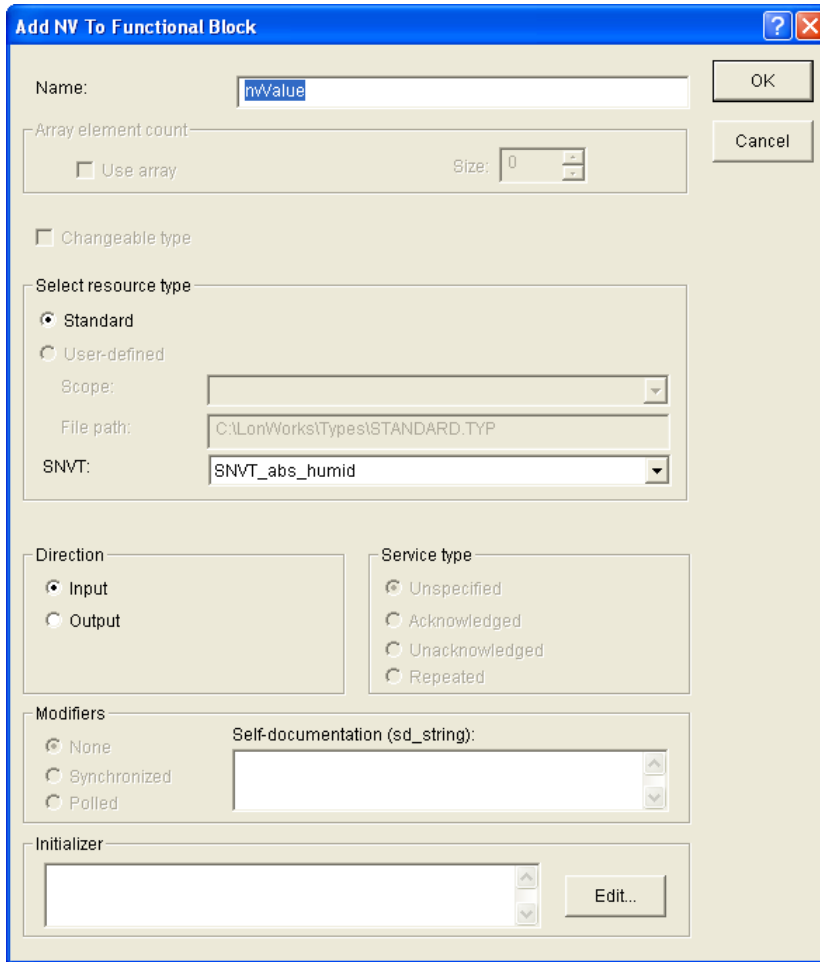
To add an implementation-specific network variable to a functional block, follow these steps:

1. Right-click the **Implementation-specific NVs** folder for the functional block in the Program Interface pane and then click **Add NV** on the shortcut menu.



Alternatively, you can right-click the **Network Variables** folder in the Program Interface pane and select **Add NV** from the shortcut menu, or you can drag a network variable from a **Network Variables** folder in the Resource pane to the functional block's **Implementation-specific NVs** folder or the **Network Variables** folder in the Program Interface pane.

2. The **Add NV to Functional Block** dialog opens (or **Add NV to Device** dialog if you are adding the network variable to the device folder).



3. In the **Name** property, enter a name for the network variable as it will appear in the IzoT Commissioning tool and other network management tools. This name must be unique to the device, can contain up to 16 alphanumeric characters, and must start with a letter. The name cannot contain spaces or the following characters: \ / : * ? " < > |. The default name is **nvValue**.
4. If your device has a changeable interface (it has network variables with changeable types, or it supports dynamic network variables), you can select the **Has Changeable Interface** check box so that network integrators can change the network variable's type. This option is only available if you selected the **Has Changeable Interface** check box when defining the device's program ID in the **Standard Program ID Calculator** (see *Specifying the Program ID* in Chapter 5 for more information). This check box is cleared by default.

Selecting this option lets you create a network variable that can send or receive different kinds of information, depending on how the device is used. For example, you can implement a generic PID controller device using a **SNVT_temp_f** as the initial type, and then let a network integrator change the **SNVT_temp_f** network variable to a range of other types so that the PID controller can control light, pressure, or other types.

5. In the **Select Resource Type** box, select whether the network variable you are adding is based on a **Standard** or **User-Defined** type. If you select a **User-Defined** type, select the **Scope** of the functional profile containing the network variable type. To use a **User-Defined** type, you must first add the resource file containing the UNVT to the resource catalog as described in the *NodeBuilder Resource Editor User's Guide*.
6. In the **SNVT** or **UNVT** property, select the network variable to be added to the functional block or device from the list.

If you are selecting a UNVT, the list contains all the **UNVTs** in resource files of the scope specified in the **Scope** field that match the program ID template to the degree specified by the scope. The 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.

7. In the **Direction** property, select whether you are adding an **Input** or **Output** network variable.
8. If you are creating an **Output** network variable, select the messaging service type to be used for transmitting updates for this output network variable in the **Service Type** box. You have four choices: **Unspecified**, **Acknowledged**, **Unacknowledged**, or **Repeated**. See *Editing Mandatory Network Variables* earlier in this chapter for more information about these different service types.
9. If you are creating an **Output** network variable, you can make the output network variable **Synchronized** or **Polled** in the **Modifiers** box. See *Editing Mandatory Network Variables* earlier in this chapter for more information about these modifiers.
10. Optionally, in the **Self-Documentation (sd_string)** property, you can enter comments 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 property can be used to provide additional notes that can be accessed from a network tool.
11. Optionally, in the **Initializer** property, you can set the value for the network variable when the device is reset. If this network variable is a structure, enumeration, or float, click the box to the right to open the **Edit Initializer** dialog and enter the value or values. See *Setting Initial Values for Network Variables and Configuration Properties* later in this chapter for more information.
12. Click **OK**. The implementation-specific network variable is added to the **Implementation-specific NVs** folder.

Adding Implementation-specific Configuration Properties

You can add a configuration property to a functional block or device 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.

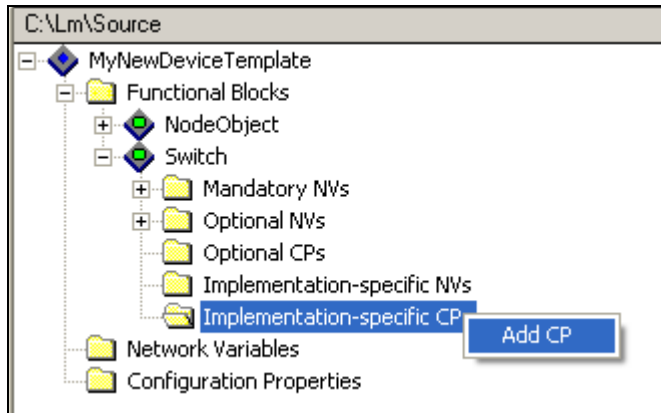
WARNING: If you use implementation-specific configuration properties in your device interface, your device will not comply with interoperability guidelines version 3.4 (or better) and therefore cannot be certified by LONMARK.

A better alternative for adding members to a functional profile is to create a user-defined functional profile template (UFPT) that inherits from an existing standard functional profile template (SFPT), and then add new mandatory or optional member configuration properties to the UFPT. This method results in a new functional profile that you can easily reuse in new devices. See the *NodeBuilder Resource Editor User's Guide* for more information on creating new functional profiles.

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, a UCPT could not be added to a SFPT, but a SCPT may be added to a UFPT.

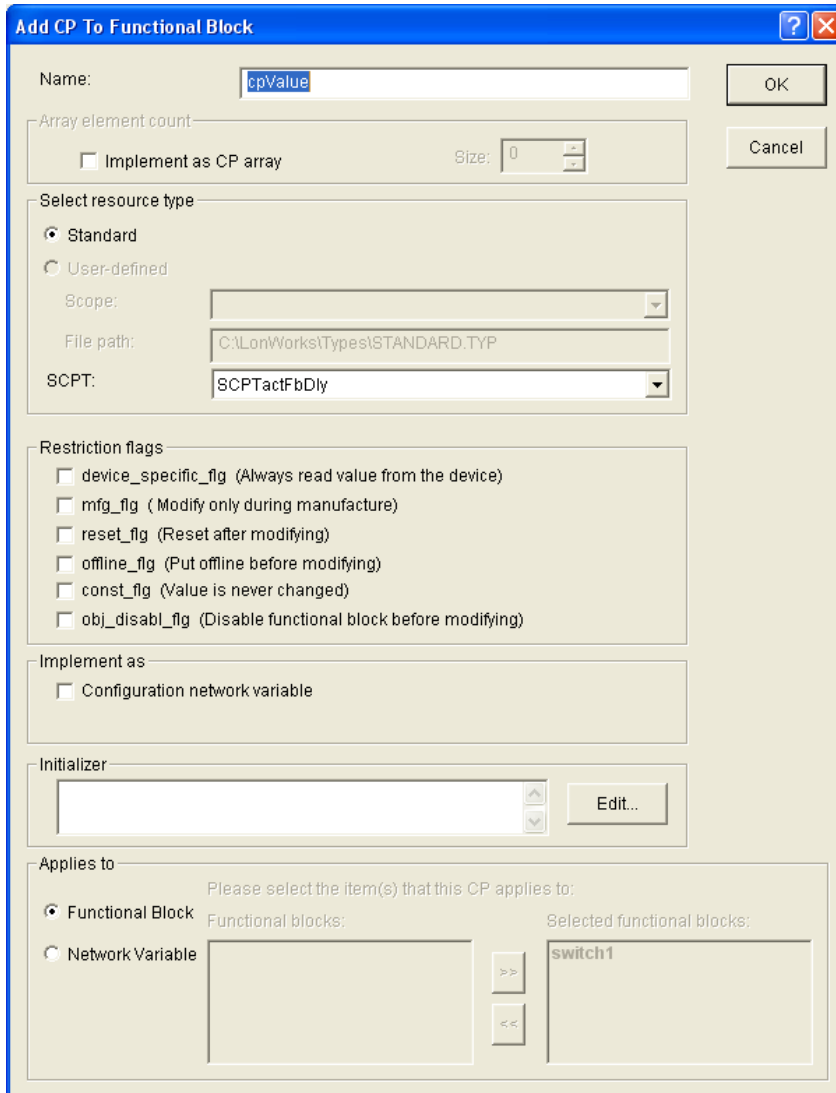
To add an implementation-specific configuration property to a functional block or device, follow these steps:

1. Right-click the **Implementation-specific CPs** folder for the functional block in the Program Interface pane and then click **Add CP** on the shortcut menu.



Alternatively, you can right-click the **Configuration properties** folder in the Program Interface pane and select **Add CP** from the shortcut menu, or you can drag a configuration property from a **Configuration properties** folder in the Resource pane to the functional block's **Implementation-specific CPs** folder or the **Configuration properties** folder in the Program Interface pane.

2. The **Add CP to Functional Block** dialog opens (or **Add CP to Device** dialog if you are adding the configuration property to the device folder).



3. In the **Name** property, enter a name for the configuration property as it will appear in the IzoT Commissioning tool and other network management tools. This name must be unique to the device, can contain up to 16 alphanumeric characters, and must start with a letter. The name cannot contain spaces or the following characters: \ / : * ? " < > |. The default name is **cpValue**.
4. To implement this configuration property as an array, select the **Implement as CP Array** check box (for configuration properties implemented as configuration files [file CPs]) or **Use Array** check box (for configuration properties implemented as configuration network variables [CPNVs]), and then specify the number of elements in the array in the **Size** box.

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, modifiable, memory in the device. A compiler or linker error will occur if the specified array size exceeds the device's resources.

Note: Configuration property arrays implemented as configuration network variables (CPNVs) are subject to the same limitations as network variables. Specifically, Neuron C applications are limited to 62 or 254 static network variables. In the case of a configuration property array implemented as CPNVs, each element in the array counts as one network variable.

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

5. In the **Select Resource Type** box, select whether the configuration property you are adding is based on a **Standard** or **User-Defined** type. If you select a **User-Defined** type, select the **Scope** of the functional profile containing the configuration property type. To use a **User-Defined** type, you must first add the resource file containing the UCPT to the resource catalog as described in the *NodeBuilder Resource Editor User's Guide*.
6. In the **SCPT** or **UCPT** property, select the configuration property to be added to the functional block or device from the list.

If you are selecting a UCPT, the list contains all the **UCPTs** in resource files of the scope specified in the **Scope** field that match the program ID template to the degree specified by the scope. The configuration property'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.
7. To implement the configuration property as a configuration network variable (CPNV), select the **Configuration Network Variable** check box. This enables the configuration property to be read, written, and bound just like a network variable; however, this consumes limited network variable resources on the device. If this check box is cleared, the configuration property is implemented as a configuration file. This check box is cleared by default.
8. If the configuration property is in a functional block array, you can select the **Static CP** check box to create a single configuration property that is shared by all the functional blocks in the array. Modifying the value of the configuration property on any functional block in the array modifies it for all of them (only one variable is allocated). If this check box is cleared, a separate configuration property is created for each functional block in the array. This check box is cleared by default.
9. In the **Restriction Flags** box, you can set configuration property flags that network tools must check in order to handle the configuration property appropriately. See *Editing Mandatory Configuration Properties* earlier in this chapter for more information on these flags.
10. Optionally, in the **Initializer** property, you can set the value for the configuration property when the device is reset. If this configuration property is a structure, enumeration, or float, click the box to the right to open the **Edit Initializer** dialog and enter the value or values. See *Setting Initial Values for Configuration Properties and Configuration Properties* later in this chapter for more information.
11. In the **Applies To** box, select whether the configuration property is applied to a network variable, a functional block, or the device as a whole. See *Editing Mandatory Configuration Properties* earlier in this chapter for more information on how to do this.
12. Click **OK**. The configuration property is added to the **Implementation-specific CPs** folder.

Note: After you create the implementation-specific configuration property, you can edit its properties following the steps described in *Editing Mandatory Configuration Properties* earlier in this chapter. For example, you may want to change the configuration property flags, or change how the configuration property is implemented (configuration network variable [CPNV] or configuration file).

Setting Initial Values for Network Variables and Configuration Properties

You can set the initial value for any network variable or configuration property. For network variables, this value will be set when the device is reset. For configuration properties, this value will be stored in the LNS network database, and it will be set the first time the device is reset after the device application has been downloaded to the device.

Each network variable and configuration property creation, implementation, and editing dialog has an **Initializer** property. You can enter a valid Neuron C initializer statement in the **Initializer** property. The following examples demonstrate valid Neuron C initializer statements:

Data Type	Example SNVT	Example Initializer
Integral	SNVT_temp	0

Float	SNVT_volt_f	{0, 0x42, 1, 0x7c, 0x6666}
Structure	SNVT_switch	{200, 0}
Enumeration	SNVT_hvac_mode	HVAC_AUTO

If you need help entering a valid initializer value, you can click the button to the right of the **Initializer** property to open the **Edit Initializer** dialog. This dialog provides information on the data type such as scaling, and minimum and maximum values. If the network variable or configuration property is a structure, enumeration, or float, this dialog is very useful:

- For structures, it lists the individual fields in the data type and lets you enter valid values for each field.
- For enumerations, it lists all the available enumerations for the data type.
- For floats and s32 type values, it lets you convert them to structures.

The following subsections describe how to set values for floats, structures, and enumerations in the **Edit Initializer** dialog.

Notes:

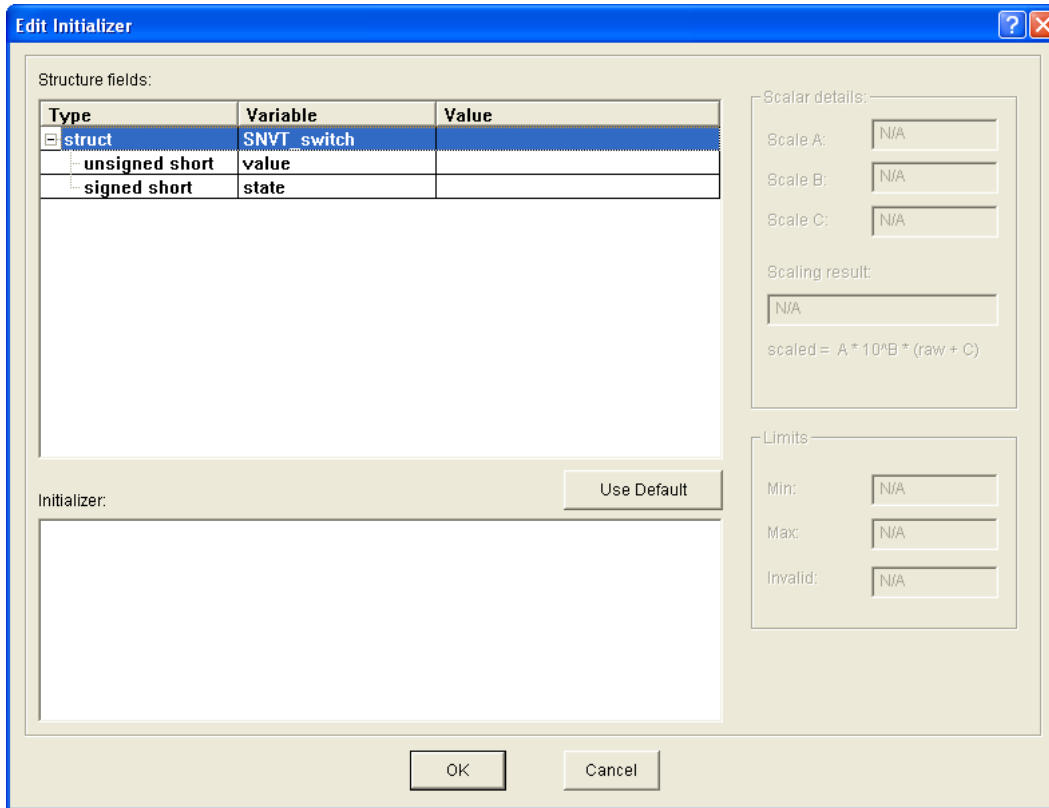
- Network variables are automatically reset to **0** during reset processing (except for those declared with the optional **eeprom** modifier); therefore, they do not need to be explicitly initialized to **0**.
- Configuration properties have default values that are defined in resource files. Default values are included in the definition of the configuration property type, in the definition of the functional profile's member configuration property (an optional initial value override), and possibly in the definition of an inherited functional profile. The Neuron C compiler will automatically initialize the configuration property to its defined default value.

Therefore, you can explicitly set the initial value or the configuration property; however, it is recommended that you use the default values defined in the resource file, if possible.

For more information on the initializer format, see Appendix A of the *Neuron C Reference Guide* and any C reference manual.

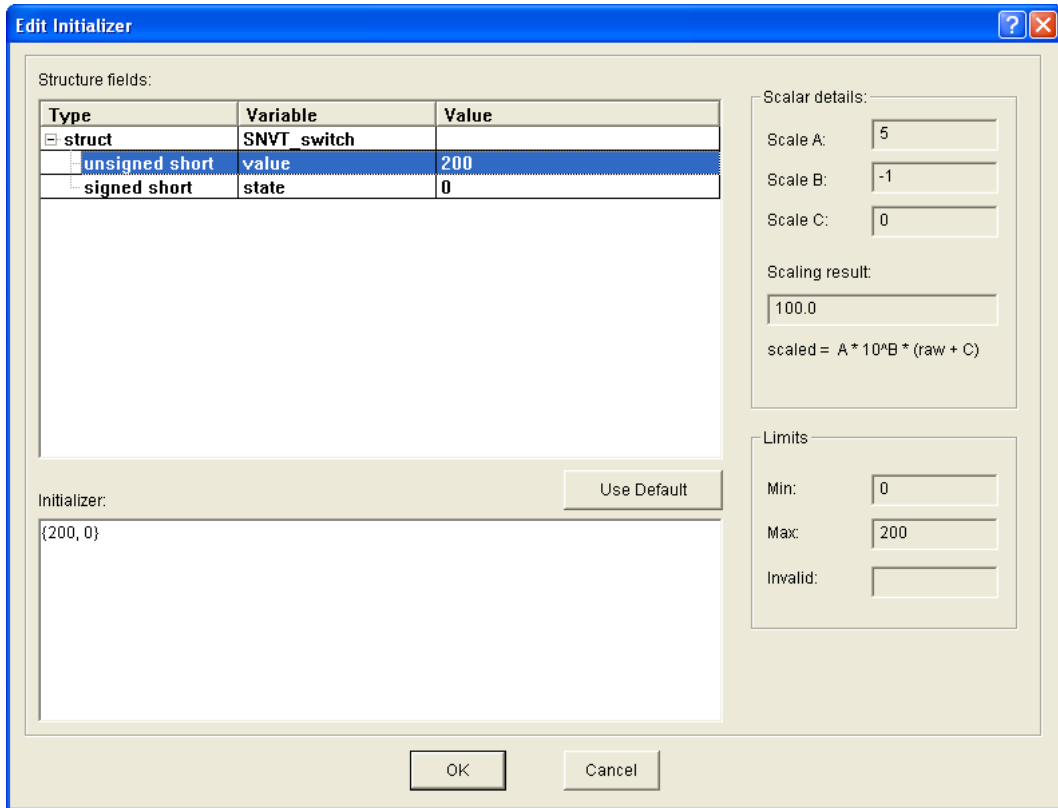
Setting Initial Values for Structured Data Types

For structured network variables and configuration properties, the **Edit Initializer** dialog displays the individual fields in the structure.



The **Structure Fields** box displays all the fields in the structured network variable or configuration property. If no initializer has previously been set, the **Value** boxes for each field and the **Initializer** box are empty. To enter values for the fields in the structure, follow these steps:

1. Click anywhere in the field's row. The scaling for the field is displayed in the **Scalar Details** box, and its minimum, maximum, and invalid (if any) values are displayed in the **Limits** box.
2. Click anywhere in the field's **Value** box and enter a valid value for the field. If the field is an enumeration, select the value from the list in the **Value** box. The **Scaling Result** box displays how the specified value will be scaled. For example, if you enter **200** for the **value** field of a **SNVT_switch** data point, the **Scaling Result** box displays **100.0**.



3. All other fields in the structure are automatically set to their default values, which are defined in resource files. If no default value is defined for the field, it is set to **0** or the minimum value allowed if **0** is out of range. You can set all fields to their default values by clicking **Use Default**.

The current initial value for the structure is displayed in the **Initializer** box. The values are enclosed in braces and are separated with commas (e.g., {SET_OFF, 0, 0} for a **SNVT_setting** data type).

4. Enter values for all other fields in the structure.

You can edit the values of a field by either selecting the field and clicking **Value** in the **Structure Fields** box or directly editing the value in the **Initializer** box. You can add comments or arrange the initializer value to be displayed in a separate line by editing the **Initializer** box directly. If you select a field in the **Structure Fields** box, the corresponding value in the **Initializer** box is highlighted and vice versa. For a union, you can only set the first member; all subsequent members are read-only.

You can use a preprocessor **#define** statement to define a string that can be used as a structure initializer. For example, you can enter the following: `#define myInit {FS_XFER_OK, 1, 2 {{{3}, {0x00, 0x00, 0x00, 0x000}}, 0}}`. If you do this, you can enter **myInit** directly in the **Initializer** box 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.

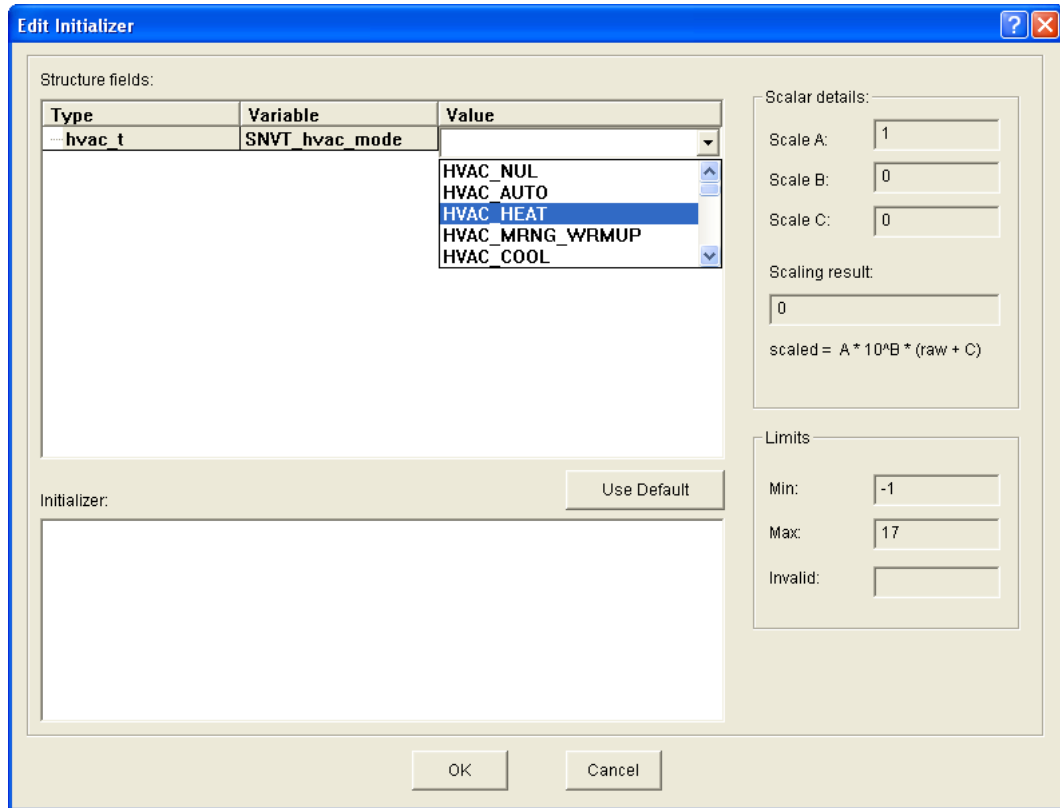
5. Click **OK** to save the changes. The value specified in the **Initializer** box will be transferred to the **Initializer** property of the respective network variable or configuration property dialog.

Setting Initial Values for Enumerations

For enumerated network variables and configuration properties, you can enter a value following these steps:

1. Click anywhere in the data type's row.

- Click anywhere in the **Value** box, and select a value from the list of possible enumeration values. You can set the enumeration to its default value by clicking **Use Default**.

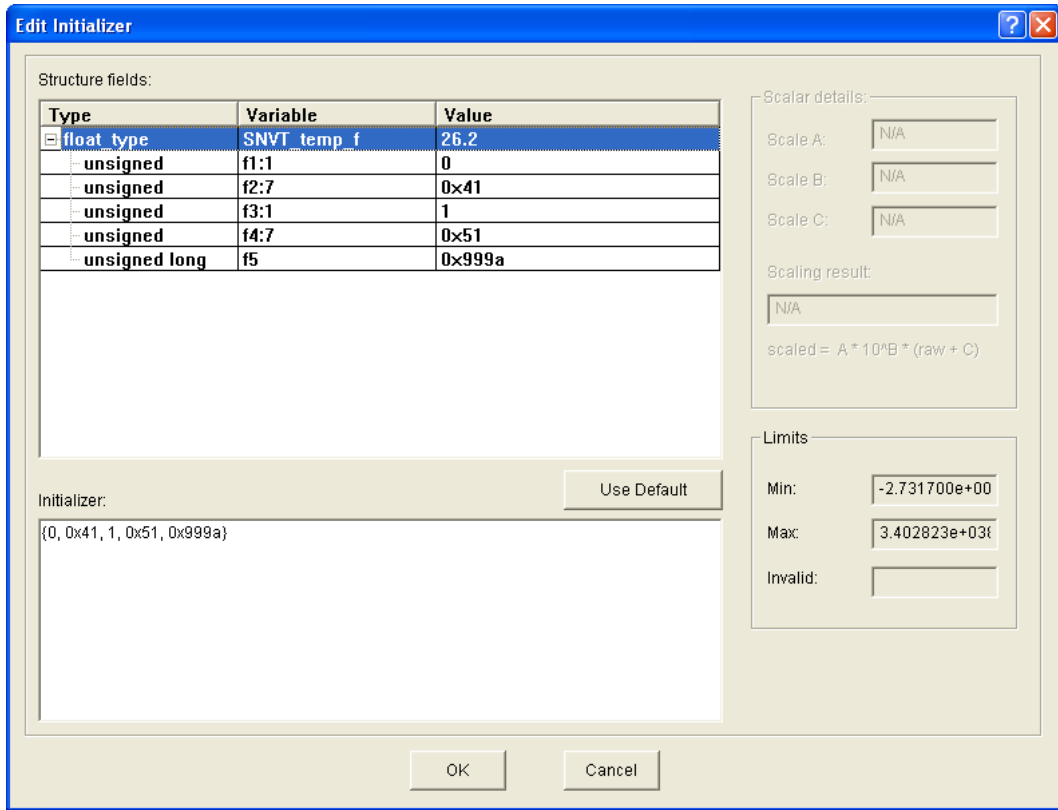


- The current initial value for the enumeration is displayed in the **Initializer** box. You can edit the value by clicking **Value** in the **Structure Fields** box or directly editing the value in the **Initializer** box.
- Click **OK** to save the changes. The value specified in the **Initializer** box will be transferred to the **Initializer** property of the respective network variable or configuration property dialog.

Setting Initial Values for Floating Point and s32 Data Types

For floating point and s32 data types, the **Edit Initializer** dialog lets you convert their values to the structures used by Neuron C to represent their values in memory. To do this, follow these steps:

- Click anywhere in the data type's row. The scaling for the data type is displayed in the **Scalar Details** box, and its minimum, maximum, and invalid (if any) values are displayed in the **Limits** box.
- Click anywhere in the float or 32 data type's **Value** box and enter a valid value. The value is automatically converted to the structure used by Neuron C to represent it in memory; the values of the fields in the structure appear below the data type. You can set the floating point or s32 data type to its default value by clicking **Use Default**.



3. The current initial value for the float or 32 data type is displayed in the **Initializer** box. You can edit the values of a field by either selecting the field and clicking **Value** in the **Structure Fields** box or directly editing the value in the **Initializer** box
4. Click **OK** to save the changes. The value specified in the **Initializer** box will be transferred to the **Initializer** property of the respective network variable or configuration property dialog.

Using Changeable-Type Network Variables

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.

The method used by the Neuron firmware to change the size of a network variable 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 method provides reliable updates to network variable sizes.

For more information on how to implement a changeable-type network variable in your device application, see *Implementing Changeable-Type Network Variables* in Chapter 7. For more information about changeable-type network variables and the NV length override system image extension, as well as a commented source code example that illustrates all aspects of creating an application that uses changeable-type network variables, see the *Neuron C Programmer's Guide*.

Generating Code with the Code Wizard

You can use the NodeBuilder Code Wizard to generate Neuron C source code that implements your device interface and creates the framework for your device application. To do this, click the **Generate and Close** option in the upper right-hand corner of the user interface. Alternatively, you can right-click the device template folder in the Program Interface pane and click **Generate and Close** on the shortcut menu.

The NodeBuilder Code Wizard checks whether the device template meets the following requirements:

- The device template has a Node Object functional block with an index of 0.
- The network variables required for the selected configuration property access method are in the Node Object functional block.
- The **Synchronized** option is set for the **nvoStatus** network variable in the Node Object functional block.
- The **Changeable Type** option is not set for any network variable if the device does not have a changeable interface (it has network variables with changeable types, or the device supports dynamic network variables). See *Specifying the Program ID* in Chapter 5 for more information on setting the **Has Changeable Interface** option in the **Standard Program ID Calculator** for a device template.
- A member name is defined for each implementation-specific network variable.
- All configuration property types, network variable types, and functional profiles have defined resources when code is being generated.
- All network variables have a distinct type. Some functional profiles contain network variables with no defined type (referred to as **SNVT_xxx**). The NodeBuilder Code Wizard forces a distinct and valid type to be assigned to these network variables.

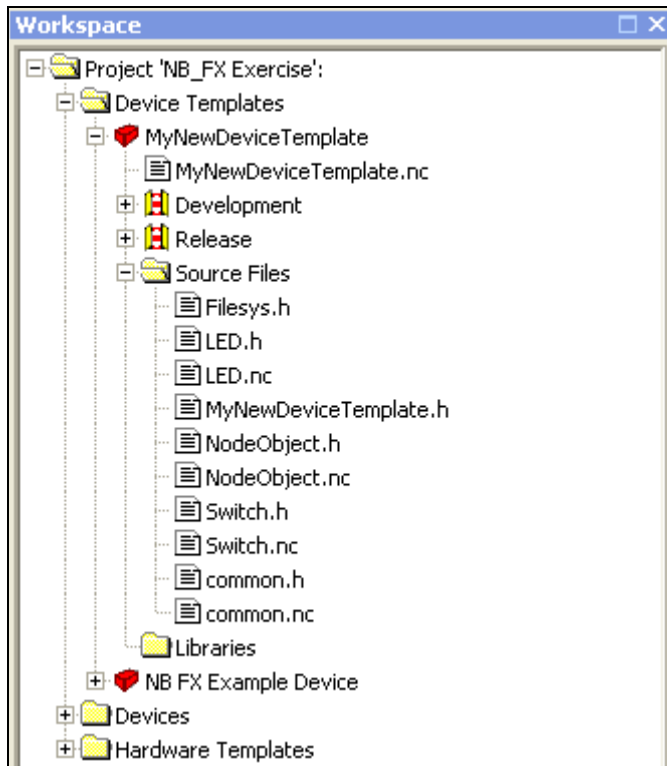
If your device interface includes any of these errors, a warning message appears explaining the error. Fix the error and then re-generate the code.

The NodeBuilder Code Wizard generates the Neuron C source files for your device interface (see *Files Created by the Code Wizard* for more information), the NodeBuilder Code Wizard closes, and you are returned to the NodeBuilder Project Manager. If any read-only files will be overwritten, a confirmation dialog opens.

See Chapter 7, *Developing Device Applications*, for information on using the Neuron C programming language to write your device application and editing the Neuron C source files created by the Code Wizard.

Files Created by the Code Wizard

When you generate Neuron C code, the NodeBuilder Code Wizard creates a series of header and Neuron C source files, which are listed in the Project pane of the NodeBuilder Project Manager. The main Neuron C source file, *<Device Template Name>.nc*, is listed underneath the device template. All the other files are shown under the **Source File** tree, but all the header and source files are stored in the same location on your computer.



The following table lists and describes the files created by the NodeBuilder Code Wizard.

Source File	Description
<i><Device Template Name>.nc</i>	The main Neuron C source file for the device application. All other files generated by the Code Wizard are included in this file using #include statements.
<i><Device Template Name>.h</i>	Contains header information and function declarations for the main source file. Defines a number of constants that are used in the application code.
<i><Functional Block Name>.nc</i>	Contains 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.
<i><Functional Block Name>.h</i>	Contains header information and function declarations for the corresponding functional block source file.
common.nc	Contains common definitions and some device management functions. Most of the utility functions contained in this file may remain unused because they are provided by the CodeWizard-3.lib and CodeWizard.lib library files. Functions and definitions this file provides should not be modified.
common.h	Contains header information and function declarations for common.nc .

Source File	Description
filesys.nc	<p>Contains functions used to facilitate transfer of configuration properties implemented as configuration files.</p> <p>This file is only generated if you selected the File Transfer Protocol configuration property access mode (for more information, see <i>Using the Program Interface Pane</i> earlier in this chapter).</p>
filesys.h	<p>Contains header information and function declarations for configuration properties implemented as configuration files.</p>
filexfer.nc	<p>Contains functions used to implement FTP transfer of configuration properties.</p> <p>This file is only generated if you selected the File Transfer Protocol configuration property access mode (for more information, see <i>Using the Program Interface Pane</i> earlier in this chapter).</p>
filexfer.h	<p>Contains header information and function declarations for filexfer.nc.</p> <p>This file is only generated if you selected the File Transfer Protocol configuration property access mode (for more information, see <i>Using the Program Interface Pane</i> earlier in this chapter).</p>
NodeObject.h	<p>Contains header information for node object declarations</p>
NodeObject.nc	<p>Contains the implementation of the node object functional block.</p>
CodeWizard-3.lib	<p>Library containing a number of utility functions used by the application framework. The CodeWizard-3 library is automatically linked with the applications based on version 3 Code Wizard Templates.</p> <p>The version 3 templates include improved code size, speed, and compliance with interoperability guidelines. The instructional comments have been revised and improved, and the templates have improved support for applications with a large number of network variables and functional blocks.</p> <p>The version 3 templates are architecturally identical to version 2 templates except that they use the CodeWizard-3.lib library. Version 3 templates also use the new #pragma library compiler directive to automatically link with this library, which means that you no longer have to specify the Code Wizard library in your project.</p>

Note: To share source files among multiple NodeBuilder device templates through a common folder, you need to specify the full path of the folder in the NodeBuilder project's **Include Search Path** property. To do this, click **Project** and then click **Setting**, or right click the **Project** folder in the Project pane and click **Settings** on the shortcut menu. The **NodeBuilder Project Properties** dialog opens. Click the **Project** tab, specify the full path of the shared folder in the **Include Search Path** property, and then click **OK**.

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

Using Code Wizard Templates

When you generate the Neuron C code for a device interface, the Code Wizard creates the source code file based on a code template. The code templates define the general infrastructure and layout of the generated application. In addition, the code templates supply many utility functions for managing device and functional block status, which you can use in your application, as needed. By default, the code template used for new device interfaces created with NodeBuilder FX tool is the version 3 template. Previous releases of the NodeBuilder tool used version 2 templates (NodeBuilder 3.1) and version 1 templates (NodeBuilder 3.0). The following sections describe the version 3, version 2, and version 1 templates and how to upgrade existing device applications to the new version 3 code templates.

Version 3 Templates

The version 3 templates include improved code size, speed, and compliance with interoperability guidelines. The instructional comments have been revised and improved, and the templates have improved support for applications with a large number of network variables and functional blocks. Version 3 templates are architecturally identical to version 2 templates except that they use the **CodeWizard-3.lib** library. Version 3 templates also use the new **#pragma library** compiler directive to automatically link with this library, which means that you no longer have to specify the Code Wizard library in your project.

You can upgrade device applications written for a 3100 Series chip to the new version 3 code templates when porting them to a 5000 or 6000 Series chip. To do this, follow these steps:

1. Create a new device template that includes a hardware template that uses the Neuron 5000 processor or FT 5000/FT 6000 Smart Transceiver. See *Creating Device Templates* in Chapter 5 for more information.
2. Create the device interface with the NodeBuilder Code Wizard based on the existing device. See *Defining the Device Interface* earlier in this chapter for more information.
3. Generate the Neuron C code for the device interface.
4. Manually copy your Neuron C code from the old application into the new application.

Note: The Neuron C Version 2.2 language includes the following new keywords: **interrupt**, **__lock**, **stretchedtriac**, **__slow**, **__fast**, and **__parity**. Some of these keywords use a double underscore prefix to avoid any naming collisions within existing device applications.

5. Remove any references to **CodeWizard.lib** library from your device template, as the version 3 templates automatically link with the revised **CodeWizard-3.lib** library.
6. Build your upgraded device application. See Chapter 8, *Building and Downloading Device Applications*, for more information.

Version 2 Templates

The version 2 templates moved most utility functions from the application space into the **CodeWizard.lib** library, and they included improvements to code size, speed, and compliance with interoperability guidelines. The Code Wizard still supports applications based on version 2 templates.

You should upgrade existing NodeBuilder projects to the Version 3 template; however, you can continue using the Code Wizard with version 2 templates. To continue using version 2 templates, verify that your device template references the standard **CodeWizard.lib** library. Typically, device templates created with earlier versions of the NodeBuilder tool already reference this library, but you may need to add an explicit reference to it in some cases (see *Inserting a Library into a NodeBuilder Device Template* in Chapter 5 for how to do this).

Version 1 Templates

The version **1** templates were the initial implementation of the Code Wizard templates, which are no longer supported by the Code Wizard.

You should upgrade existing NodeBuilder projects to the Version **3** template because it generally results in more compact and faster code, and better compliance with interoperability guidelines. You can, however, continue using the Code Wizard with the version **1** templates. To continue using version **1** templates, verify that your device template does not reference the standard **CodeWizard.lib** library.

Creating the Device Application

The code produced by the Code Wizard is skeleton code. It implements the device interface that you have defined, but it does not implement any device functionality. You will use the NodeBuilder Project Manager to edit the source files generated by the Code Wizard and implement your device's functionality. To create your device application, do the following:

- See the *Modifying Neuron C Code Generated by the Code Wizard* section in Chapter 7. This section describes the Neuron C code generated by the NodeBuilder Code Wizard and provides guidelines on how to modify it. In addition, it lists the Neuron C Version 2 features that are not supported by the NodeBuilder Code Wizard.
- See the *Neuron C Programmer's Guide*. This document details how to write device applications using the Neuron C Version 2.2 language. It also describes how to design and implement a device application.
- See the *Neuron C Reference Guide*. This document provides reference information for writing device applications using the Neuron C Version 2.2 language.

Developing Device Applications

This chapter provides an overview of the Neuron C Version 2.3 programming language. It describes how to edit the Neuron C source code generated by the NodeBuilder Code Wizard to implement your device functionality. It explains how to use the NodeBuilder Editor to edit, search, and bookmark Neuron C code.

Introduction to Neuron C

Neuron C Version 2.3 is a programming language based on ANSI C that you can use to develop applications for Neuron Chips and Smart Transceivers. It includes network communication, I/O, interrupt-handling, and event-handling extensions to ANSI C, which make it a powerful tool for the development of LONWORKS device applications. Following are a few of the extensions to the ANSI Standard C language:

- A network communication model based on *functional blocks* and *network variables* that simplifies and promotes data sharing between like or disparate devices.
- A network configuration model based on functional blocks and *configuration properties* that facilitates interoperable network configuration tools.
- A type model based on standard and user *resource files* expands the market for interoperable devices by simplifying integration of devices from multiple manufacturers.
- An extensive built-in set of *I/O objects* that supports the powerful I/O capabilities of Neuron Chips and Smart Transceivers. Powerful *event-driven programming* extensions based on *when-tasks* that provide easy handling of network, I/O, and timer events.
- Language extensions that define application interrupt handlers and use synchronization tools, where available.

Neuron C provides a rich set of language extensions to ANSI C tailored to the unique requirements of distributed control applications. Experienced C programmers will find Neuron C a natural extension to the familiar ANSI C paradigm. Neuron C offers built-in type checking and allows the programmer to generate highly efficient code for distributed LONWORKS applications.

Neuron C omits ANSI C features not required by the standard for free-standing implementations. For example, certain standard C libraries are not part of Neuron C. Other differences between Neuron C and ANSI C are detailed in the *Neuron C Programmer's Guide*.

This chapter provides an introduction to Neuron C. For more details on Neuron C, see the *Neuron C Programmer's Guide*.

Unique Aspects of Neuron C

Neuron C implements all the basic ANSI C types, and type conversions as necessary. In addition to the ANSI C data constructs, Neuron C provides some unique data elements.

Network variables are fundamental to Neuron C and LONWORKS applications. Network variables are data constructs that have language and Neuron firmware support to provide the look and feel of a regular global C variable, but with additional properties of communicating across a LONWORKS network, to or from one or more other devices on that network. The network variables make up part of the device interface for a LONWORKS device.

Configuration properties are Neuron C data constructs that are another part of the device interface. Configuration properties allow the device's behavior to be customized using a network tool such as the IzoT Commissioning tool or a customized plug-in created for the device. Configuration properties provide the look and feel of a normal variable to the C program, with the addition of controlled access by network configuration tools.

Neuron C also provides a way to organize the network variables and configuration properties in the device into functional blocks. Functional blocks provide a collection of network variables and configuration properties that are used together to perform one task. These network variables and configuration properties are called the functional block members.

Each network variable, configuration property, and functional block is defined by a type definition contained in a resource file. Network variables and configuration properties are defined by network variable types (NVTs) and configuration property types (CPTs). Functional blocks are defined by functional profile templates (FPTs).

Network variables, configuration properties, and functional blocks in Neuron C can use standardized, interoperable types. The use of standardized data types promotes the interconnection of disparate devices on a LONWORKS network. For network variables, the standard types are called standard network variable types (SNVTs). For configuration properties, the standard types are called standard configuration property types (SCPTs). For functional blocks, the standard types are called standard functional profile templates (SFPTs). If you cannot find standard types or profiles that meet your requirements, Neuron C also provides full support for user-defined network variable types (UNVTs), user-defined configuration property types (UCPTs), and user-defined functional profile templates (UFPTs).

A Neuron C application executes in the environment provided by the Neuron firmware. This firmware provides an event-driven scheduling system as part of the Neuron C language's run-time environment. Therefore, a Neuron C application does not use a single entry point, as is the case with ANSI C's **main()** function. Instead, a Neuron C application uses *when*-tasks and *interrupt*-tasks to specify application code to be executed in response to various system events or interrupt requests, much in the way of a .NET event handler.

The Neuron firmware contains a scheduler, which executes these *when*-tasks in an orderly and deterministic fashion as and if needed. Neuron C *when*-tasks can be triggered by system events (such as reset), network events (such as a network variable update or network error), I/O events (such as a new reading from an I/O input), timer events, or any arbitrary application-defined event.

Interrupt-tasks are activated as the interrupt request occurs, subject to interrupt prioritization rules. Neuron C interrupt-tasks can be triggered by edge or level conditions on any of the dedicated I/O pins, by events occurring in the embedded timer and counter units, or by a dedicated high-resolution system timer. *Interrupt*-tasks are only supported by 5000 or 6000 Series chips. Other interrupt sources, such as those related to sending or transmitting serial data over the embedded UART, are handled transparently by the Neuron firmware.

Neuron C also provides a lower-level application messaging service integrated into the language in addition to the network variable model. While the network variable model has the advantage of being a standardized method of information interchange that promotes interoperability between multiple devices from multiple vendors, application messaging is available for proprietary and standard special-purpose solutions. Application messages are used with the LONWORKS file transfer protocol, a standard mechanism for transfer of large amounts of data, and the ISI protocol, a standard mechanism to manage networks without intervention of a dedicated tool or specialist.

Another Neuron C data object is the application timer object. Timer objects can be declared and manipulated like variables. When a timer expires, the Neuron firmware automatically manages the timer events and notifies the program of those events. Timers may be automatically reloading (repeating), or one-shot timers, with a resolution ranging from 0.001–65,535 seconds.

Neuron C supports programmable hardware timer units through a variety of I/O library functions. These functions provide a resolution up to 1 MHz (1 μ s) or better, subject to the selected I/O model, Neuron Chip type, clock speed, and other factors (see the *I/O Model Reference* for more information). The 5000 and 6000 Series chips also support a configurable high-resolution system timer, which can be used to generate periodic interrupt requests.

Neuron C supports up to 35 different I/O models, ranging from simple bit Direct I/O models for typical input or output hardware to complex Timer/Counter models for triacs. Neuron C also includes Serial and Parallel I/O models for serial and parallel communication busses. These I/O models are standardized I/O “device drivers” for the Neuron Chip or Smart Transceiver I/O hardware. Each I/O model fits into the event-driven programming model. A function-call interface is provided to interact with each I/O object. The function-call interfaces are optimized for their respective I/O models, yet they are similar to each other so that they are easy to use.

Neuron C Variables

The following sections briefly discuss various aspects of Neuron C-specific variable declarations. Data types affect what sort of data a variable represents. Storage classes affect where the variable is stored, whether it can be modified (and if so, how often), and whether there are any device interface aspects to modifying the data.

Neuron C Variable Types

Neuron C supports the following C variable types. The keywords shown in square brackets below are optional. If omitted, they will be assumed by the Neuron C language, per the rules of the ANSI C standard:

- **[signed] long [int]** 16-bit quantity
- **unsigned long [int]** 16-bit quantity
- **signed char** 8-bit quantity
- **[unsigned] char** 8-bit quantity
- **[signed] [short][int]** 8-bit quantity
- **unsigned [short][int]** 8-bit quantity
- **enum** 8-bit quantity (**int** type)

Neuron C provides some predefined enum types. One example is shown below:

```
typedef enum {FALSE, TRUE} boolean;
```

You should use the **unsigned int** type whenever possible because it is the type best supported by the Neuron Chip and Smart Transceiver's hardware architecture. The **unsigned int** type is preferred over **signed int** type.

Neuron C also provides predefined objects that, in many ways, provide the look and feel of an ANSI C language variable. These objects include Neuron C timer and I/O objects. See Chapter 2 of the *Neuron C Programmer's Guide* for more details on I/O objects, and see Chapter 4 in the *Neuron C Reference Guide* for more details on timer objects.

The extended arithmetic library also defines **float_type** and **s32_type** for IEEE 754 and signed 32-bit integer data respectively. These types are detailed further in Chapter 3 of the *Neuron C Reference Guide*.

Neuron C Storage Classes

If no class is specified for a declaration at file scope, the data or function is global. *File scope* is that part of a Neuron C program that is not contained within a function, a *when*-task, or an *interrupt*-task. Global data (including all data declared with the **static** keyword) is present throughout the entire execution of the program, starting from the point where the symbol was declared. Declarations using **extern** references can be used to provide forward references to variables, and function prototypes must be declared to provide forward references to functions. In addition, **extern** references can be used to publish a symbol and allow for linking with other object files.

Upon power-up or reset of a Neuron Chip or Smart Transceiver, the global data in RAM is initialized to its initial-value expression, if present; otherwise, it is set to **0**.

Neuron C supports the following ANSI C storage classes and type qualifiers:

- **auto** declares a variable of local scope. Typically, this would be within a function body. This is the default storage class within a local scope and the keyword is normally not specified. Variables of auto scope that are not also static are not initialized upon entry to the local scope, unless you provide an explicit initializer. The value of the variable is not preserved once program execution leaves the scope.

- **const** declares a value that cannot be modified by the application program. Affects self-documentation (SD) data generated by the Neuron C compiler when used in conjunction with the declaration of CP families or configuration network variables. **extern** declares a data item or function that is defined in another module, in a library, or in the system image.
- **static** declares a data item or function which is *not* to be made available to other modules at link time. Furthermore, if the data **item is** local to a function or to a **when()**task, the data value is to be preserved between invocations, and is not made available to other functions at compile time.

In addition to the ANSI C storage classes, Neuron C provides the following classes and class modifiers:

- **network** begins a network variable declaration. See Chapter 3, *How Devices Communicate Using Network Variables*, of the *Neuron C Programmer's Guide* for more details.
- **uninit** when combined with the **eprom** keyword (see below), specifies that the EEPROM variable is not initialized or altered on program load or reload over the network.

The following Neuron C keywords allow you to direct portions of application code and data to specific memory sections.

- **eprom**
- **far**
- **offchip** (only on Neuron Chips and Smart Transceivers with external memory)
- **onchip**

These keywords are particularly useful on the Neuron 3150 Chip and 3150 Smart Transceivers, since a majority of the address space for these parts is mapped off chip. See *Using Neuron Chip Memory* in Chapter 8 of the *Neuron C Programmer's Guide* for a more detailed description of memory usage and the use of these keywords.

Variable Initialization

Initialization of variables occurs at different times for different classes. The **const** variables, except for network variables, *must* be initialized. Initialization of **const** variables occurs when the application image is first loaded into the Neuron Chip or Smart Transceiver. The **const ram** variables are placed in off-chip RAM that must be non-volatile. The **eprom** and **config** variables are also initialized at load time, except when the **uninit** class modifier is included in these variable definitions.

Global RAM variables are initialized at reset (specifically when the device is reset or powered up). By default, all global RAM variables (including **static** variables) are initialized to zero at this time.

Initialization of I/O objects, input network variables (except for **eprom**, **config**, **config_prop**, or **const** network variables), and timers also occurs at reset. Zero is the default initial value for network variables and timers.

Local variables (except **static** ones) are not automatically initialized unless you provide explicit initialization, nor are their values preserved when the program execution leaves the local scope.

Neuron C Declarations

The Neuron C Version 2.3 programming language and ANCI C both support the following declarations:

Declaration	Example
Simple data items	<code>int a, b, c;</code>
Data types	<code>typedef unsigned long ULONG;</code>
Enumerations	<code>enum hue {RED, GREEN, BLUE};</code>

Declaration	Example
Pointers	<code>char *p;</code>
Functions	<code>int f(int a, int b);</code>
Arrays	<code>int a[4];</code>
Structures and unions	<code>struct s { int field1; unsigned field2 : 3; unsigned field3 : 4; };</code>

The Neuron C Version 2.3 programming language also supports the following declarations:

Declaration	Example
I/O objects	<code>IO_0 output oneshot relay_trigger;</code>
Timers	<code>mtimer led_on_timer;</code>
Network variable	<code>network input SNVT_temp nviTemperature;</code>
Configuration Properties	<code>SCPTdefOutput cp_family cpDefaultOut;</code>
Functional Blocks	<code>fblock SFPTnodeObject { ... } myNode;</code>

Introduction to Neuron C Code Editing

The Neuron C source code generated by the NodeBuilder Code Wizard provides the framework for your device application. It implements the device interface that you have defined, but it only implements basic device functionality. The functionality supplied by Code Wizard includes the most common tasks required for interoperable device and functional block management, but it does not include any code implementing your application's core algorithms. You can implement your device's functionality by editing your device application's Neuron C source code in the Edit pane of the NodeBuilder Project Manager to.

In addition to network variable, configuration property, and functional block declarations that comprise the device interface, the Neuron C code generated with the NodeBuilder Code Wizard also contains the following features:

- **Skeleton when() task for functional blocks or functional block arrays.** The **when()** task provides notification upon incoming network variable updates for the functional block or functional block array. If the functional block has no input network variables, no **when()** task is generated.
- **Default implementation for handling of system events.** System events include 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.
- **Code handling 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**, **RQ_ENABLE**. Handling for other requests is partially implemented but must be completed by the developer. The C language comments supplied in the source files generated by the Code Wizard describe the aspects and ramifications of various interoperability procedures. For more information, see the *LonMark Application Layer Interoperability Guidelines*.

- **Default directors for functional blocks or functional block arrays.** The source code for each functional block or functional block array contains a default implementation of a director function
- **Utility functions to manage functional block state.** The Code Wizard generates **common.h** and **common.nc** files, which contain some utility functions. Most utility functions are delivered with the **CodeWizard-3.lib** library file (**CodeWizard.lib** for version 2 templates), and they are declared in the **CodeWizard.h** header file. See these files for more information on these functions.
- **File directory structure.** 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 are mutually exclusive.

If FTP is used to access configuration property template and value files, and at least one 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 both sequentially and random access. The FTP server supports configuration property files with up to the amount of available space on the Neuron Chip. This space is equal to 64 KB minus any address space used for code, data, or other 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 the **filexfer.h** file for more details.

Modifying Neuron C Code Generated by the Code Wizard

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 code 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.

Code Commands

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

```
//<Command>
```

These commands indicate where the NodeBuilder Code Wizard puts certain pieces of generated code. For example, the **//<Include Headers>** 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 a code outside of the Code Wizard start (//{{NodeBuilder Code Wizard Start), end (//{{NodeBuilder Code Wizard End), you will manage the section of the code on your own.

For example, one Neuron C feature that is not supported by the NodeBuilder Code Wizard is a single configuration property being applied to more than one network variable. The following example demonstrates this:

```
//{{NodeBuilder Code Wizard Start
:
:
//<Fblock Input NV Declarations>
network input SNVT_temp_p nviTempP
nv_properties {
    cpTransInMin = 0,
    cpTransInMax = 3000L
```

```

};
//
//<Fblock Output NV Declarations>
network output SNVT_lev_percent nvoPercentage;
:
:
//
//}}NodeBuilder Code Wizard End

```

You can override the code generated by the NodeBuilder Code Wizard by moving the **//<Fblock Output NV Declarations>** command out of the Code Wizard section, as shown below:

```

//{{NodeBuilder Code Wizard Start
:
//<Fblock Input NV Declarations>
network input SNVT_temp_p nviTempP
nv_properties {
    cpTransInMin = 0,
    cpTransInMax = 3000L
};
:
:
//
//}}NodeBuilder Code Wizard End
//
//<Fblock Output NV Declarations>
network output SNVT_lev_percent nvoPercentage
nv_properties {
    cpTransInMin = 0,
};

```

Once you take the **//<Fblock Input NV Declarations>** command out of the Code Wizard managed section of the code, the Code Wizard will no longer create input network variable declarations. If you want to add additional input network variables to the functional block, they must be added manually.

Code Guidelines

The following sections provide recommendations for modifying the code generated by the NodeBuilder Code Wizard. This is not a comprehensive list 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, variables, and the interrupt system in the **when (reset)** task within the main Neuron C file (**main.nc**). 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. The following example demonstrates this:

```

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

```

....

Add when-tasks Responding to I/O and Timer Events

You can 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 affect global I/O or timers, and add them to the individual functional block's source file if they affect functional block-specific items.

Add interrupt-tasks Responding to Interrupt Requests

You can add *interrupt*-tasks to respond to interrupt requests, as needed. Add these interrupt handlers to the main source, and enable interrupt processing. Interrupt processing is typically enabled in the reset task, but other tasks, such as the online and offline tasks, may also enable or disable the interrupt system.

Add Code to when(nv_update_occurs(<nv>)) when-task of Functional Blocks with Input NVs

For functional blocks that implement input network variables, add code to the **when(nv_update_occurs(<nv>))** *when*-task in the subject functional block or functional block array, where <nv> is the related input network variable or network variables. The Neuron C scheduler will raise this event and execute the related *when*-task when at least one of the associated input network variables has been updated. 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 *when*-task.

Code Wizard implements one input network variable *when*-task for each functional block or functional block array in order to achieve short scheduler-cycles and therefore a responsive device.

Share Code with filexfer.nc when Handling Explicit Messages on a Device Implementing FTP

When adding code that handles explicit messages and contains unqualified **when(msg_arrives)** event handlers on a device that implements FTP with the sender-capability enabled, the sender routine does itself implement such an event handler. There can only be one such event handler and this handler must be the last *when*-task in compilation order; therefore, you must share your code with the code provided in the **filexfer.nc** file. The FTP server implementation uses the **#pragma scheduler_reset** directive if the sender-capability is enabled (this is the default). See the **filexfer.nc** and **filexfer.h** files for more details.

Ignore NCC#310 and NC#463 Compiler Warnings

You may notice a few compiler warnings that appear when compiling unedited Code Wizard code, referring to items being declared but never used (warning **NCC#310**), or referring to the **const** attribute being casted away (warning **NCC#463**). The second warning, **NCC#463**, should only occur if support for the file transfer protocol has been requested. Both messages can be safely ignored in this case.

You can eliminate **NCC#310** warnings during a final clear-up phase during device development. This will reduce memory requirements and reduce the size of the application image, thus reducing download times. The items referred to by these **NCC#310** warnings are utility functions provided for your convenience. These functions are declared in **common.nc** and can safely be removed if not used.

Alternatively, you can use the **#pragma disable_warning** directive to disable selected warnings.

Implementing Changeable-Type Network Variables

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.

The method used by the Neuron firmware to change the size of a network variable 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 network variable length override system image extension to get it. This method provides reliable updates to network variable sizes.

For more information about changeable-type network variables and the NV length override system image extension, as well as a commented source code example that illustrates all aspects of creating an application that uses changeable-type network variables, see Chapter 3 of the *Neuron C Programmer's Guide*.

To implement a changeable-type network variable in your device application, follow these steps (see Chapter 3 of the *Neuron C Programmer's Guide* for a more detailed discussion of step 4):

1. Create a device template that has a changeable interface. See *Specifying the Program ID* in Chapter 5 for more information on how to do this.
2. In the Code Wizard, create a new network variable or edit an existing one and select the **Changeable Type** checkbox in the dialog for creating or editing the network variable. See *Editing Mandatory Network Variables*, *Implementing Optional Network Variables*, or *Adding Implementation-specific Network Variables* in Chapter 6 for more information on how to do this.
3. Generate Neuron C code for your device interface. See *Generating Code with the Code Wizard* in Chapter 6 for more information on how to do this

1. In the Neuron C code generated by the Code Wizard, do the following:
 - a. Complete the implementation of the **get_nv_length_override** function. The 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.
 - b. The Code Wizard uses the **#pragma unknown_system_image_extension_isa_warning** directive to generate Neuron C source code that will compile. The Code Wizard enables this directive in the device template's main header file. If you use a combination of Code Wizard generated code and your own code, you can edit the relevant portion of the main header file.

You should only use the older **nv_len()** function 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.

You can use the **get_current_nv_length()** function to determine the current length of a network variable at any time (see the *Neuron C Reference Guide* for more information about this method).

- c. 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*.

- d. 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*.

Neuron C Version 2 Features Not Supported by the Code Wizard

The following overview summarizes features of the Neuron C Version 2 language that are currently not used or not supported by the NodeBuilder Code Wizard. See the *Neuron C Programmer's Guide* and *Neuron C Reference Guide* for more information about Neuron C Version 2.3.

Message Tags

The generation of declarations or the use of message tags is not supported with the exception of automatically generated FTP server implementation that contains a message tag (**fx_explicit_tag**). Also see *when() clauses* later in this section.

I/O Models

The NodeBuilder Code Wizard does not generate or support the generation of declarations or use of I/O objects.

Network Variables

Network variable arrays. The NodeBuilder Code Wizard only generates declarations for a network variable array if it applies to a functional block array. The sizes of the two arrays will be the same (for example, one network variable per functional block). The NodeBuilder Code Wizard does not support declaring a network variable array and distributing the elements of this array among multiple functional blocks or functional block arrays.

Polled modifier for input network variables. The NodeBuilder Code Wizard supports the **polled** network variable modifier for output network variables, but it does not support the **polled** network variable modifier for input network variables. The **polled** modifier, combined with input network variables, is only used for host-based application development with a model file. This feature is not required or supported for development of Neuron-hosted applications because the Neuron C compiler automatically detects the polling inputs and generates the Neuron C code accordingly.

Configuration Properties

Network variable class config. The NodeBuilder Code Wizard does not support the network variable class **config** because this keyword is not recommended for use in new development. The NodeBuilder Code Wizard supports configuration network variables using the Neuron C network variable class **cp** instead.

cp_family re-use. The NodeBuilder Code Wizard code will declare one **cp_family** of a given type for each instance of a configuration property, unless the configuration property it references is a functional block array. Specifically, if the complete device requires two (or more) configuration properties of type **T**, the declaration of a single **cp_family** of type **T** is technically sufficient in many cases; however, the NodeBuilder Code Wizard will generate two (or more) **cp_families** of type **T**.

This means that a **cp_family** generated by the NodeBuilder Code Wizard will always have a single member unless the configuration properties applies to a functional block array. In this case, the size of that array equals the size of the **cp_family**.

Global configuration properties. The NodeBuilder Code Wizard does not currently support the **global** CP modifier, but it does support sharing a configuration property through the **static** CP modifier.

The NodeBuilder Code Wizard does not support the generation of configuration properties that apply to multiple disjointed functional block (for example, not be members of the same functional block array).

The NodeBuilder Code Wizard does not support sharing a configuration property among the members of a network variable array that applies to the entire device (for example, it is not part of a functional block or functional block array). This restriction applies to both the **static** and **global** configuration property sharing scopes.

range_mode_string. The NodeBuilder Code Wizard does not support the **range_mode_string** option, which supports the setting of LONMARK range modification for a configuration property.

when() clauses

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

If your device application processes incoming messages and includes the pre-defined FTP server, you must use the existing implementation and start your own handler code from there. For more information about removing parts of the code generated by the NodeBuilder Code Wizard, see *Modifying Neuron C Code Generated by the Code Wizard* earlier in this chapter.

when(nv_update_occurs(nv1..nvX)). For functional blocks or functional block arrays that contain input network variables, the NodeBuilder Code Wizard always generates a single **when()** task to handle incoming network variable updates, using the Neuron C construct **when(nv_update_occurs(nv1..nvX))**.

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. See the *Neuron C Programmer's Guide* for more details about the use of NV range specifications as arguments to the **nv_update_occurs** function.

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

#pragma scheduler_reset. The implementation of the FTP server requires the presence of **#pragma scheduler_reset**. This is automatically inserted as needed by the NodeBuilder Code Wizard (see the **filexfer.nc** file). You may not remove this pragma.

LONMARK Style

The NodeBuilder Code Wizard can only generate code for a device template that includes a valid Node Object functional block. The Node Object functional block must be the first object in the device's list of objects. The functional profile key for the Node Objects functional profile is **0** at present scope; therefore, you can create own Node Object functional profile with a key of 0 that inherits from the scope 0 functional profile.

Director Functions

The NodeBuilder Code Wizard always creates one director per functional block or functional block array. It does not currently support functional blocks without director functions, and it does not support the sharing of one director function among multiple functional blocks (except for functional block arrays).

Interrupt Tasks

The NodeBuilder Code Wizard does not currently generate code for application-specific interrupt-tasks.

Using the NodeBuilder Editor

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 of the NodeBuilder Project Manager. 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.

This section describes the following:

1. The color-coding scheme used to highlight source code based on Neuron C syntax.
2. How to search for a text string in a single source file or in all source files in the project.
3. How to use bookmarks to return to frequently used parts of your code.
4. How to set the options in the **Editor** tab of the **NodeBuilder Project Properties** dialog that control syntax highlighting, tab settings, auto indent, font settings, and automatic loading.

Using Syntax Highlighting

If you are editing a Neuron C file (**.nc** extension), header file (**.h** extension), or C file (**.c** extension), the Edit pane in the NodeBuilder Project automatically color-codes text based on Neuron C syntax. This color-coding is designed to make your Neuron C code more easily readable. You can change these colors using the editor options (for more information, see *Setting Editor Options* later in this chapter).

The following table lists the default colors and their corresponding Neuron C syntax:

Green	Neuron C comment. Commented text is not compiled during a build.
Blue	Neuron C language specific keyword or function
Pink	String or number. This includes the arguments to #include statements, and numerical values assigned to variables.
Dark Blue	Constant or preprocessor directive.
Grey	Code generated and updated by the NodeBuilder Code Wizard.
Black	All other text.

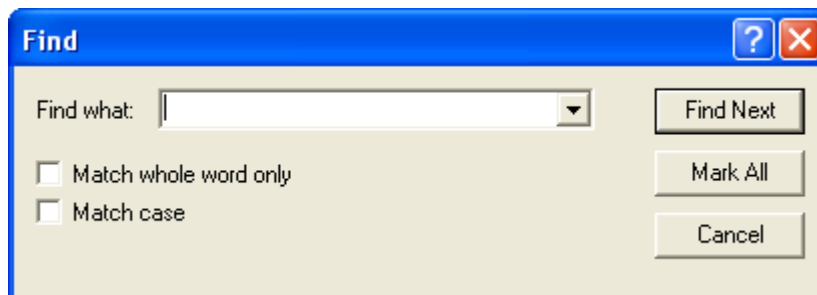
Searching Source Files

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. Click **Edit** and then click **Find** (or press CTRL+F). The **Find** dialog opens.



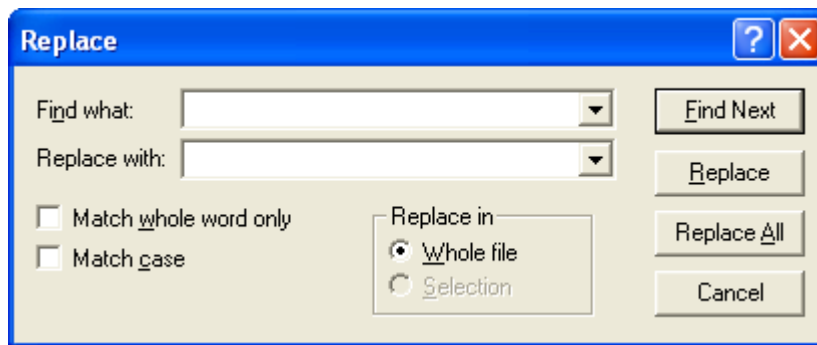
3. Enter the text string to search for in **Find what**.

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 have every line in the file containing the string bookmarked (for more information, see *Using Bookmarks* later in this chapter.).

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. Click **Edit** and then click **Replace** (or press CTRL+H). The **Replace** dialog opens.

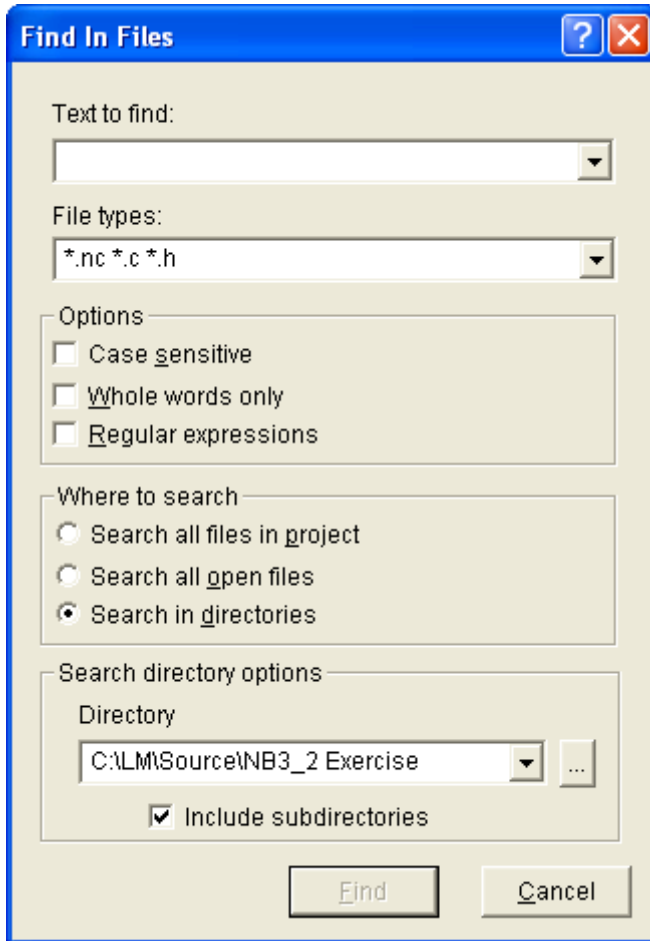


3. Enter the text string to search for in **Find what**.
4. Enter and the text string that you want to replace it with in **Replace with**.
5. Set **Match Whole Word Only** to find only whole words that match the string. Set **Match case** to make the search case sensitive.
6. If you selected text prior 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. Click **Edit** and then click **Find in Files** (or press CTRL+SHIFT+F). The **Find in Files** dialog opens.



2. In the **Text to Find** property, enter the text string to be found.
3. In the **File Types** property, select the file types to be searched. By default, the search will look in Neuron C files (**.nc** extension), header files (**.h** extension), and C files (**.c** 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.
4. In the **Options** box, select one or more of following check boxes to modify the search (all of the check boxes are cleared by default):
 - **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:

Expression	Description
*	An asterisk in the search string replaces zero or more characters. An asterisk must be accompanied by at least two other characters (for example, 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 (for example, if you search for zoo+ , it will return zoot and zoom , but not zoo). Use \+ to represent a plus character.


Expression	Description
?	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 (for example, re). This re that 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 (for example, bo{2,}k{2,}e{2,}per would find instances of bookkeeper , bookkeeeper , and so on. 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 form feed.
\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.

5. In the **Where** property, select which files to search. You have the following three choices:
 - **Search all Files in Project.** Searches all files in the current NodeBuilder project. This is the default.
 - **Search all Open Files.** Searches all currently open files. Open the Window menu to see which files are currently open.
 - **Search in Directories.** Search all files in a specific directory.
 6. If you selected the **Search in Directories** option in step 5, enter the directory to be searched in the **Directory** property. The NodeBuilder project directory will be selected by default. Click the button to the right to browse to a different directory. To search all the subfolders in the **Directory** property, select the **Include Subdirectories** check box. This check box is cleared by default.
 7. Click **Find**.
 8. The **Search Results** tab of the Results pane will display the results of the search. Each instance of the string found is displayed in a line in the Results pane. The line includes 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 your 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 to be bookmarked, or on the line containing the bookmark to be removed.
3. Click **Edit**, 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.

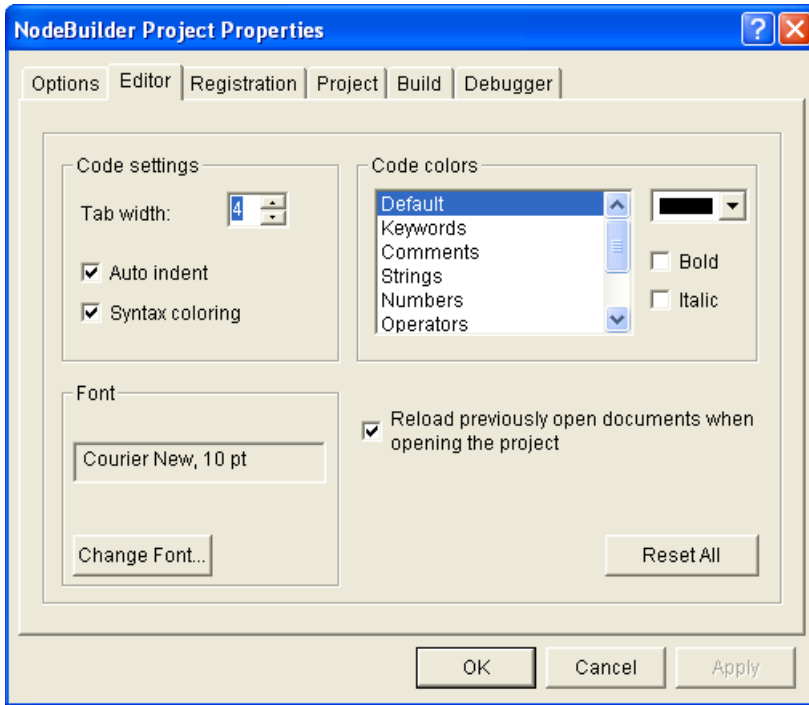
Once you have set any bookmarks in a file, you can go to the next bookmark in the file. To go to the next or previous bookmark, click **Edit**, point to **Bookmarks**, and then click **Next Bookmark** or **Previous Bookmark**.

To remove all bookmarks from the current source file, click **Edit**, 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 for the current NodeBuilder project. To set editor options, follow these steps:

1. Click **Tools** and then click **Options**. The **NodeBuilder Project Properties** dialog opens with the **Editor** tab selected.



Alternatively, you can access this tab by clicking **Project**, clicking **Settings**, and then clicking the **Editor** tab, or by clicking the **Project Settings** button (🔧) on the **Editor** toolbar, and then clicking the **Editor** tab.

2. Set the following properties:

Code Settings

<i>Tab Width</i>	Determines the tab size. By default, the tab size is 4 .
<i>Auto Indent</i>	Automatically indents code inside a function or conditional statement. This check box is selected by default.
<i>Syntax Coloring</i>	Enables syntax highlighting. You can specify colors in the Code Colors property. This check box is selected by default.
<i>Font</i>	Sets the font and font size used to display text in the editor. Click Change Font to choose a new font or font size. You may choose only from fixed width fonts.
<i>Code Colors</i>	Sets the colors used by the editor when the Syntax Coloring check box is selected. 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 the Bold or Italic check boxes. These check boxes are cleared by default.
<i>Reload Previously Open Documents</i>	Opens all documents that were open the last time you closed the project when you open a project.
<i>Reset All</i>	Resets all options on this tab to their defaults.

3. Click **OK** to save the changes.

Building and Downloading Device Applications

This chapter describes how to compile Neuron C source code, build an application image, and download the application image to a device. It explains how to add target devices to a NodeBuilder project and how to manage them.

Introduction to Building and Downloading Applications

You can build an application image for one or more development or release targets in a NodeBuilder project. After you build the application image, you can download it to a development platform such as an FT 6000 EVB or an LTM-10A Platform, a custom device that you have manufactured, or a third-party device. You can add target devices to your NodeBuilder project using the IzoT Commissioning tool or the NodeBuilder Project Manager, and then manage them and edit their settings.

The following sections describe how to do the following:

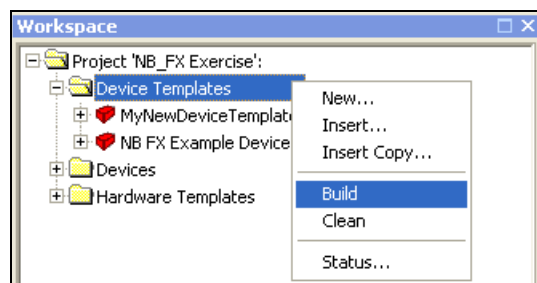
1. Build an application image with the IzoT NodeBuilder tool.
2. Download the application image to a target with the IzoT NodeBuilder tool.
3. Add target devices to a NodeBuilder project using the IzoT Commissioning tool and the IzoT NodeBuilder tool, manage target devices, and edit target device settings.

Building an Application Image

You can build an application image for one or more development or release targets in a NodeBuilder project. When you build an application image, the IzoT NodeBuilder tool compiles the source code specified by the device template, links the compiled code with the standard libraries and any user-specified libraries in the device template, creates downloadable application image files, creates a ROM image, and creates device interface files that are required by the IzoT Commissioning tool and other network tools.

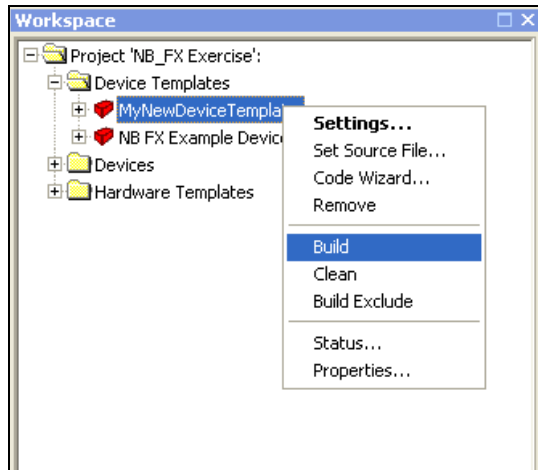
To build an application image for one or more targets, follow these steps:

1. Close the LonMaker Browser if it is open.
2. Open the project in the NodeBuilder Project Manager. For more information on how to do this, see *Opening a NodeBuilder Project* in Chapter 4.
3. Build the application image for all the targets in the project, all the targets in a device template, or one or more targets in the project.
 - To build all the targets in the current NodeBuilder project, click **Project** and then click **Build All**, or right-click the **Device Templates** folder in the Project pane and click **Build** on the shortcut menu. This builds all non-excluded targets in the project. For more information on excluding targets, see *Excluding Targets from a Build* later in this chapter.

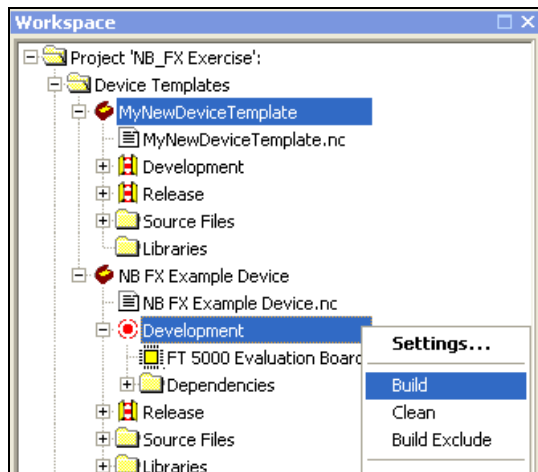


You can clean all targets automatically before building them. To do this, click **Project** and then click **Build All Unconditionally**. For more information on cleaning targets, see *Cleaning Build Output Files* later in this chapter.

- To build all the targets in a device template, right-click a device template in the Project pane and click **Build** on the shortcut menu.



- To build one or more targets in the current NodeBuilder project, click one target device template, optionally, hold down CTRL and click the other targets or device templates containing the targets to be built, right-click one of the selected items, and then click **Build** on the shortcut menu.




4. The IzoT NodeBuilder tool automatically saves all unsaved project files when you start a build. If there are any unsaved changes and the **Prompt before Saving Files** check box in the **Options** tab of the **NodeBuilder Project Properties** dialog is selected, you will be prompted to save the changes or cancel the build.
5. 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 the build status. You can double-click an error or warning to go 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.

```

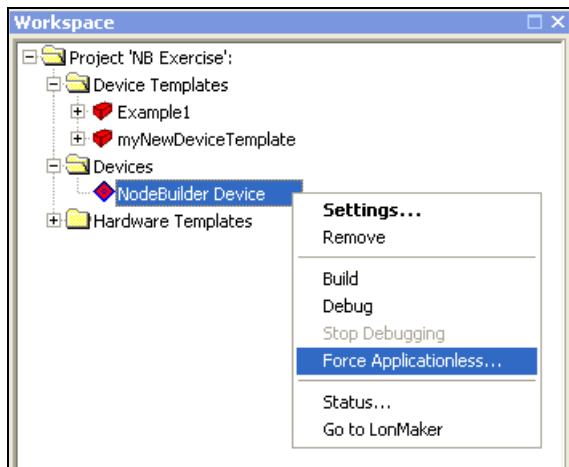
x >>>> Build for project 'NB FX Exercise' <<<<
----- Building 'NB FX Example Device': 'Development' target -----
Resolving CodeWizard-3.lib as C:\LonWorks\Images\CodeWizard-3.lib
Resolving GEN.LIB as C:\LonWorks\Images\GEN.LIB
Resolving EXTARITH.LIB as C:\LonWorks\Images\EXTARITH.LIB
Resolving PSG.LIB as C:\LonWorks\Images\PSG.LIB
Exporter driver: The boot ID has been updated to 0x0AF4
Project Make: Updating device template file succeeded
Importing device template 'C:\lm\Source\NB FX Exercise\NB FX Example Device\Development\NB FX Example Device.xif' as 'NB F:
Registering plugins for new LNS device template...
----- Building 'NB FX Example Device': 'Release' target -----
Build due for component Neuron (R) C Compiler, reason: File C:\LM\SOURCE\NB FX EXERCISE\NB FX EXAMPLE DEVICE\NB FX EXAMPLE
Cleaned file C:\lm\Source\NB FX Exercise\NB FX Example Device\Release\NB FX Example Device.nxe
Cleaned file C:\lm\Source\NB FX Exercise\NB FX Example Device\Release\NB FX Example Device.apb
Cleaned file C:\lm\Source\NB FX Exercise\NB FX Example Device\Release\NB FX Example Device.xif
Cleaned file C:\lm\Source\NB FX Exercise\NB FX Example Device\Release\NB FX Example Device.xfb
Cleaned file C:\lm\Source\NB FX Exercise\NB FX Example Device\Release\NB FX Example Device.ndl
Cleaned file C:\lm\Source\NB FX Exercise\NB FX Example Device\Release\NB FX Example Device.map
Cleaned file C:\lm\Source\NB FX Exercise\NB FX Example Device\Release\NB FX Example Device.nme
Requesting build from Neuron (R) C Compiler
Compiling...
Compiler driver: Attached to LonUCL32-3
Set Neuron (R) C Compiler command --defloc=C:\lm\Source\NB FX Exercise\NB FX Example Device
Messages Search Results Eval

```

Note: To stop a build in progress, open the **Project** menu and then select **Stop Build**.

6. If the **Load After Build** option () in the IzoT NodeBuilder toolbar is set or if the **Load after Build** check box in the **Build** tab of the **NodeBuilder Project Properties** dialog is selected, all commissioned devices that use one of the applications produced by the build are automatically downloaded to the devices. If there are any uncommissioned devices associated with the NodeBuilder project, you need to replace them with the IzoT Commissioning tool when the build is complete (for more information, see *Replacing a Device in a LonMaker Network* in Chapter 7 of the *IzoT Commissioning User's Guide*). The status of this operation will be shown in the NodeBuilder Results pane.
7. Each device is assigned the LNS Device Template specified by its **LNS Device Template Name** property in the **Program ID** tab of the **NodeBuilder Device Template Properties** dialog. If you change a device's program ID, the device template name must also be changed. This is handled automatically if **Automatic Program ID Management** is enabled for the NodeBuilder device template in the **Program ID** tab of the **NodeBuilder Device Template Properties** dialog (it is enabled by default).

If you are unable to load a previously-built device because of a program ID conflict, you can set the device applicationless by expanding the Devices folder in the Project pane, right-clicking the device, and then clicking **Force Applicationless** on the shortcut menu.



8. The IzoT NodeBuilder tool generates *downloadable application image files*, *programmable application image files*, and *device interface files*. The following table describes these files:

Downloadable Application Image Files

(.APB, .NDL, and NXE,)

These files contain the application image used by the IzoT Commissioning tool and other network tools to download the compiled application image to a device.

There are three types of downloadable application image files: the binary application image file (.APB extension), the .NDL file, and the text application image file (.NXE extension). These files are described as follows:

- The **.APB** file is used by the IzoT Commissioning tool and other LNS network tools to download an application images to a device over the network. The **.APB** files can be used to upgrade the device application for a previously installed device.

Note: The **.APB** files cannot be used to update a device's communication parameters or the clock multiplier for a 5000 or 6000 Series chip. If you change these properties, you must associate the NodeBuilder project with a LonMaker network and then load the device application with the IzoT NodeBuilder tool to implement the change.

- The **.NDL** file is used to support manufacture-time loading of devices with the NodeLoad utility. For more information on the NodeLoad utility, see the *NodeLoad Utility User's Guide*.
- The **.NXE** file is supplied to support some legacy network tools, but it is not normally required.

Programmable Application Image Files

(.NRI, .NEI, .NFI, .NME, and .NMF)

These files contain an application image that is used by a programming tool to program an application image into a memory chip. Programming tools include generic device programmers and specialized Neuron 3120 programmers.

The **.NMF** and **.NME** types are used with the 5000 or 6000 Series chips to program serial EEPROM (**.NME**) and flash (**.NMF**) memory parts. Image files intended for serial memory parts may also be programmed in-circuit, subject to the availability of suitable hardware.

There are five types of programmable application image files: ROM application image file (**.NRI** extension), the EEPROM and flash application image file (**.NEI** extension), the off-chip serial EEPROM application image file (**.NME** extension), the Neuron flash image (**.NFI** extension), and the off-chip flash application image file (**.NMF** extension).

- **ROM.** The ROM application image file (**.NRI** extension) 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 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.
- **EEPROM and flash memory.** The EEPROM application image file (**.NEI** extension) contains a EEPROM application image that is used for programming an external 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 device 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.

- **Off-chip serial EEPROM.** For the 5000 or 6000 Series chips, the **.NME** application image file is supplied and supports programming the serial EEPROM memory part.
- **Neuron flash image.** For a Neuron 3120E4 Chip or an FT 3120 Smart Transceiver, the **.NFI** file contains an EEPROM application image that is used for programming the on-chip EEPROM. 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.
- **Off-chip serial flash.** For the 5000 or 6000 Series chips, the **.NMF** application image file is supplied and supports programming the optional serial flash memory part.

Device Interface Files

(**.XIF**, **.XFB**, and **.xfo**)

These files contain a definition of the device interface that is used by the IzoT Commissioning tool and other LNS network tools to learn the interface to a device, without requiring the device to be physically attached to the network.

There are three types of device interface files: the text device interface file (**.XIF** extension), the binary device interface file (**.XFB** extension), and the optimized device interface file (**.xfo** extension).

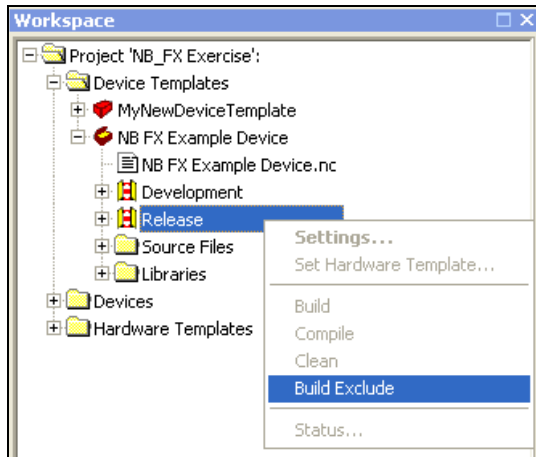
- **XIF.** The text device interface file is a text description of the device interface. The format of this file is detailed in the *LONMARK External Interface File Reference Guide*, which is available on LONMARK Web site at www.lonmark.org/technical_resources/guidelines/developer.shtml
- **XFB** and **XFO.** The binary device interface file and optimized device interface file are used by the IzoT Commissioning tool and other LNS tools to create LNS device templates, which define the device interface to LNS tools.

Device manufacturers should distribute the binary application image file (**.APB**) and text device interface file (**.XIF**) files to customers to support their devices. The **.NDL** file may also be distributed to support loading the devices in the field with the NodeLoad utility. This is useful for systems where an LNS network tool is not available to download device applications.

Note: If you provide **.NDL** files for upgrading device applications, do not change the device's communication parameters or change the clock multiplier on a 5000 or 6000 Series chip. Changing the communication parameters may cause communication with the device to be lost permanently. Changing the clock multiplier on a 5000 or 6000 Series chip may affect the device's power consumption and EMC performance, and it may affect the peripheral circuitry attached to the Neuron 5000 Processor or FT 5000 or FT 6000 Smart Transceiver.

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 dimmed.

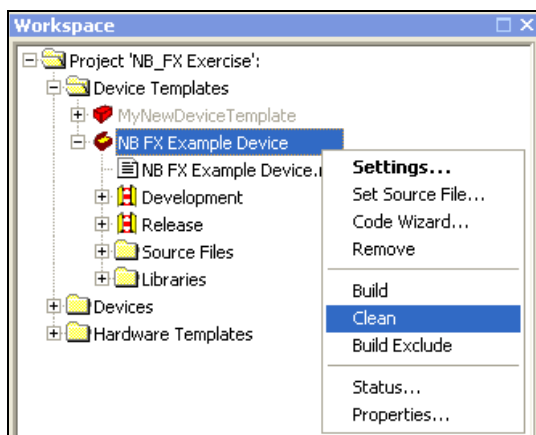


To include the device template or target in the build after you have excluded it, right-click it and select **Build Exclude** again.

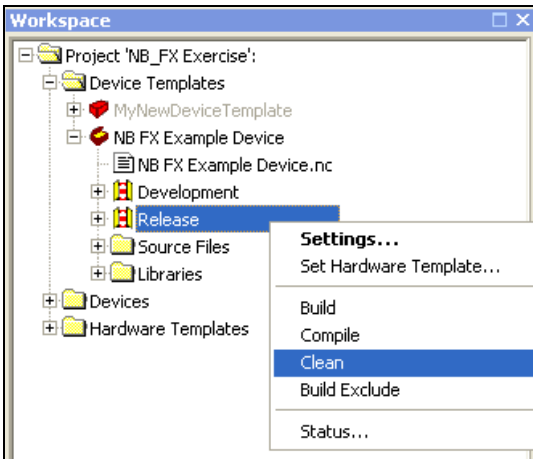
You can also choose to build to only development or release targets in the entire project. To do this, select **Development Targets** or **Release Targets** in **Build Type** in the NodeBuilder toolbar. To build all targets, select **All Targets**.

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.

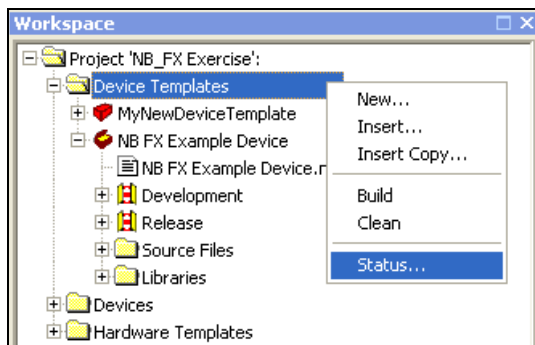


Note: The **Clean** command only removes files and folders produced by the IzoT NodeBuilder tool. It does not remove any files that you have generated with the IzoT NodeBuilder tool.

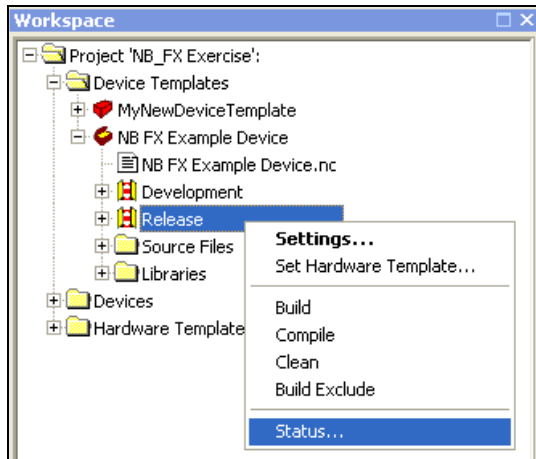
Viewing Build Status

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 view the build status for the entire project, a specific device template, a specific device template target, or a specific target. To do this, follow these steps:

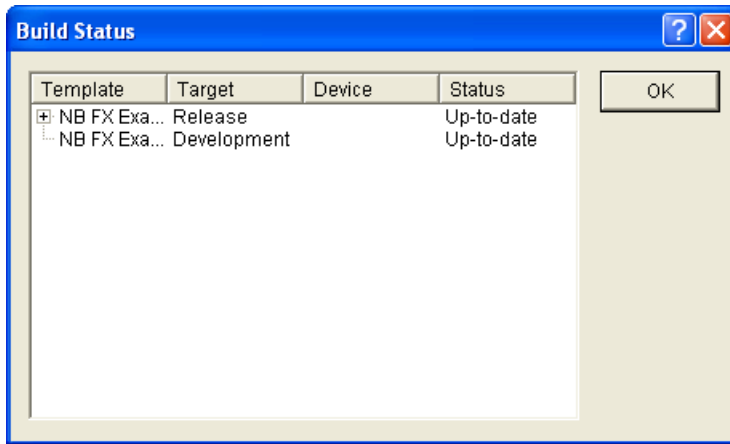
1. Select whether to view the build status for the entire project, a specific device template, a specific device template target, or a specific target.
 - To see the build status for the entire project, right-click the **Device Templates** folder and then click **Status** on the shortcut menu.



- To see the build status for a specific device template, target, or device, right-click it and then click **Status** on the shortcut menu.



2. The **Build Status** dialog opens.



3. Each row in this dialog represents a device template target or a target. Targets are listed beneath their associated device template target. The dialog 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 contains status for a device template target, this column is empty.
- Status* The target status. This may be 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 it 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.

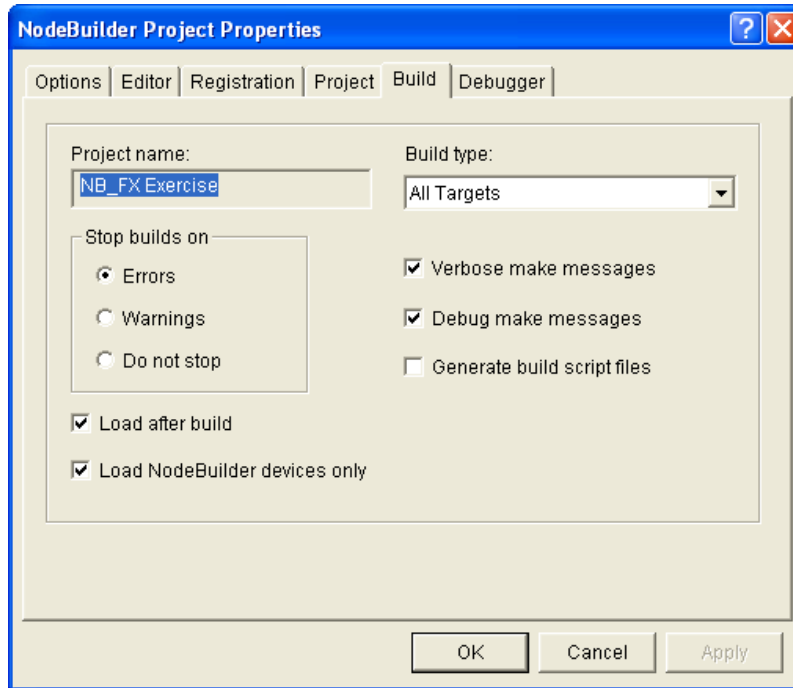
Export required. Applies to device template targets only. Indicates that a property has been changed that requires the device to be exported.

Load required. Applies to targets only. Indicates that the application image has been modified since the target was last loaded. The IzoT NodeBuilder tool will only be aware of loads performed by the IzoT NodeBuilder or IzoT Commissioning tools. If you load the application with another tool, the IzoT NodeBuilder tool will not update the status until the application is built and loaded using the IzoT NodeBuilder or IzoT Commissioning tools. The Load required status is undetermined when you use the IzoT NodeBuilder tool as a standalone application.

Setting Build Options

You can set build properties that control the build process. To set build properties, follow these steps:


1. Click **Project** and then click **Settings**, or click the **Project Settings** button (🔧) in the Project toolbar. The **NodeBuilder Project Properties** dialog opens with the **Build** tab selected.



Alternatively, you can right-click the **Project** folder at the top of the Project pane, click **Settings** on the shortcut menu, and then click the **Build** tab; or click **Tools**, click **Options**, and then click the **Build** tab.

2. Set the following options for building device applications:

Stop Builds on Determines when a build is stopped. A build may be stopped when an error or 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.
<i>Load after Build</i>	Loads the application into a device immediately after the application image is built. The devices must be commissioned with the IzoT Commissioning 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. This check box is selected by default.
<i>Load NodeBuilder Devices only</i>	Limits loads to the targets listed in the Devices folder in the Project pane. This check box is selected by default.
<i>Build Type</i>	Determines whether All Targets (the default), Development Targets , or Release Targets will be built when you build a project or device template. You can also view and change the build type from the NodeBuilder toolbar.
<i>Verbose Make messages</i>	Displays more descriptive messages in the Results pane when you build a device template. This check box is selected by default.
<i>Debug make Messages</i>	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. This check box is selected by default.
<i>Generate Build Script Files</i>	Generates build script files when you build a device template. This check box is cleared by default. Build scripts are described in Appendix A, <i>Using the NodeBuilder Command Line Project Make Facility</i> .

3. Click **OK**.

Downloading an Application Image

You can download an application image that you have built with the IzoT NodeBuilder tool to a LONWORKS device. The device may be a development platform such as the FT 6000 EVB or an LTM-10A Platform, a custom device that you have manufactured, or a third-party device. Typically, you will 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 FT 6000 EVB and the LTM-10A Platform include Neuron firmware that is preloaded into the device. The Neuron firmware allows these devices to be downloaded over a LONWORKS network so that you do not have to use any special 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 *Adding and Managing Target Devices* later in this chapter.

If you are using a custom device that does not have an on-chip Neuron firmware image (similar to a 3150 Neuron Chip or 3150 Smart Transceiver), you must program the Neuron firmware image into the external memory (EEPROM, flash, PROM, ROM, and so on) before you can use the device as a target.

Once you have completed development, you will load your application image into the device as part of your manufacturing process. The files containing the application image are described in *Building 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.

Processor	System Image Memory Type	Application Memory Type	Application Image File Extension	Application Image Programming Tool
Neuron 5000 Processor FT 5000 Smart Transceiver	On-chip ROM	Off-chip serial EEPROM	.NME	Compatible Device Programmer
		Off-chip serial flash	.NMF	
		Off-chip serial EEPROM or flash	.NDL	NodeLoad Utility
FT 6000 Smart Transceiver	On-chip ROM	Off-chip serial EEPROM	.NMF	Compatible Device Programmer
		Off-chip serial EEPROM or flash	.NDL	NodeLoad Utility
Neuron 3150 Chip FT 3150 Smart Transceiver PL 3150 Smart Transceiver	Off-chip flash	Off-chip flash	.NEI	Device programmer
			.NDL	NodeLoad Utility
Neuron 3120xx Chip	On-chip EEPROM	On-chip EEPROM	.NEI	Neuron 3120 Programmer
			.APB and .NXE (TP/XF-1250 devices only)	Network Tool
			.NDL	NodeLoad Utility
Neuron 3120E4 Chip FT 3120 Smart Transceiver PL 3120 Smart Transceiver PL 3170 Smart Transceiver	On-chip EEPROM	On-chip EEPROM	.NFI	Compatible Programmer
			.APB and .NXE (for initial load, TP/XF-1250 devices only)	Network tool
			.NDL	NodeLoad Utility

The procedure that you will use to program the application image depends on whether you are programming off-chip memory for a device based on a Neuron 5000 or 6000 core; the off-chip or on-chip memory for a device based on a Neuron 3150 core; or the on-chip memory for a device based on a Neuron 3120 core. These procedures are described in the following sections. See the Smart Transceiver databook for more information.

Programming 5000 and 6000 Off-chip Memory

A 5000 or 6000 Series device requires at least 2K of external serial EEPROM, and it can optionally contain external serial flash memory. There is no on-chip non-volatile memory provided for the application. Many types of EEPROM devices are supported; however, Echelon currently supports and provides drivers for only the following three external flash devices: **Atmel AT25F512AN**, **ST M25P05-AVMN6T**, and **SST25VF512A**.

Note: The drivers for different flash devices consume varying amounts of EEPROM code space because of the different programming algorithms required for the different flash devices. For example, the SST driver takes 40 bytes more of EEPROM than the other two supported flash devices.

The system image resides in on-chip ROM. The application image and the system image are copied from the external non-volatile memory into the on-chip RAM at chip startup and reset. The Neuron firmware is responsible for copying any writes that are directed towards external non-volatile memory. See the Neuron Chip or Smart Transceiver data book for more information.

The build process produces an **.NME** file for application code and data designated for external EEPROM and an optional **.NMF** file for application code for external flash memory, if it is available.

You can download the device application over the network, or you can transfer the device application over an I2C or SPI interface using the application images files generated by the build process if the device has not been installed on the network. Using the I2C or SPI interface is ideal for the ex-circuit programming of serial flash and EEPROM devices. In addition, you can use any compatible device programmer with the I2C or SPI interface to program these memory devices in-circuit, which helps with the development and mass-production of generic device hardware, and lowers production costs. See the next section, *Programming 5000 and 6000 Series Chips In-Circuit*, for more information on preparing your device hardware for in-circuit programming.

You can load an alternate system image from external EEPROM or external flash, if required. This feature may be required if a newer firmware image becomes available at a later date. In such a case, the system image will always start at address 0xC000 in the external part. In the case of external EEPROM, the part has to be at least 32K in order to support alternate system images.

The 5000 and 6000 Series chips contain the version 18 Neuron firmware in their on-chip ROM; therefore, you do not need to program the memory parts before the device is first used. When the Neuron firmware initializes during power-up, it can detect empty memory parts, and then boot into the applicationless state with communication parameters set for a TP/FT-10 channel at a clock multiplier setting of 1. You can then load your application image using the IzoT NodeBuilder tool, IzoT Commissioning tool, NodeLoad Utility, or other network tool.

You need to pre-program the serial memory parts if you want the device to start with a different version of the Neuron firmware, or if you want to increase application loading speed during production.

Programming 5000 and 6000 Series Chips In-Circuit

You can use the I²C or SPI interface on the 5000 and 6000 Series chips for the in-circuit programming of your external non-volatile memory EEPROM and flash devices. This lets you pre-produce generic hardware and load one of several application images into the board at production time, without the need for costly sockets or re-soldering.

To perform in-circuit programming, you need a method to connect your external serial EEPROM or flash memory device to a compatible device programmer, while disconnecting these signal lines from the 5000 and 6000 Series chip. Echelon has tested the Aardvark™ I2C/SPI USB Host Adapter from TotalPhase™ (Part No. TP240141), with the 10-pin split cable from TotalPhase (Part No. TP240212), as one method for creating this connection (for more information on this adapter, go to the TotalPhase Web site at www.totalphase.com/products/aardvark_i2cspi/). The Aardvark has six signal lines: two for the I²C interface (SDA and SCL), and four for the SPI interface (MOSI, MISO, SCL, and SS). The I²C/SPI interface used by the Neuron 5000 Processor or FT 5000 and 6000 Smart Transceiver has

some pins that are multifunctional; therefore you must program each external non-volatile memory device individually.

After you connect the I²C or SPI interface to the Aardvark programmer or other compatible in-circuit device programmer, you can use a program such as the Flash Center Memory Programmer from TotalPhase to program your external serial EEPROM or flash memory device. You can download the Flash Center Memory Programmer for free from the TotalPhase Web site at www.totalphase.com/products/flash_center/#downloads. If you use the Flash Center Memory Programmer software, you also need to change the extension of the **.NME** and **.NMF** application image files generated by the NodeBuilder tool to **.HEX**. This is because the Flash Center Memory Programmer requires hex files that have **.HEX** extensions.

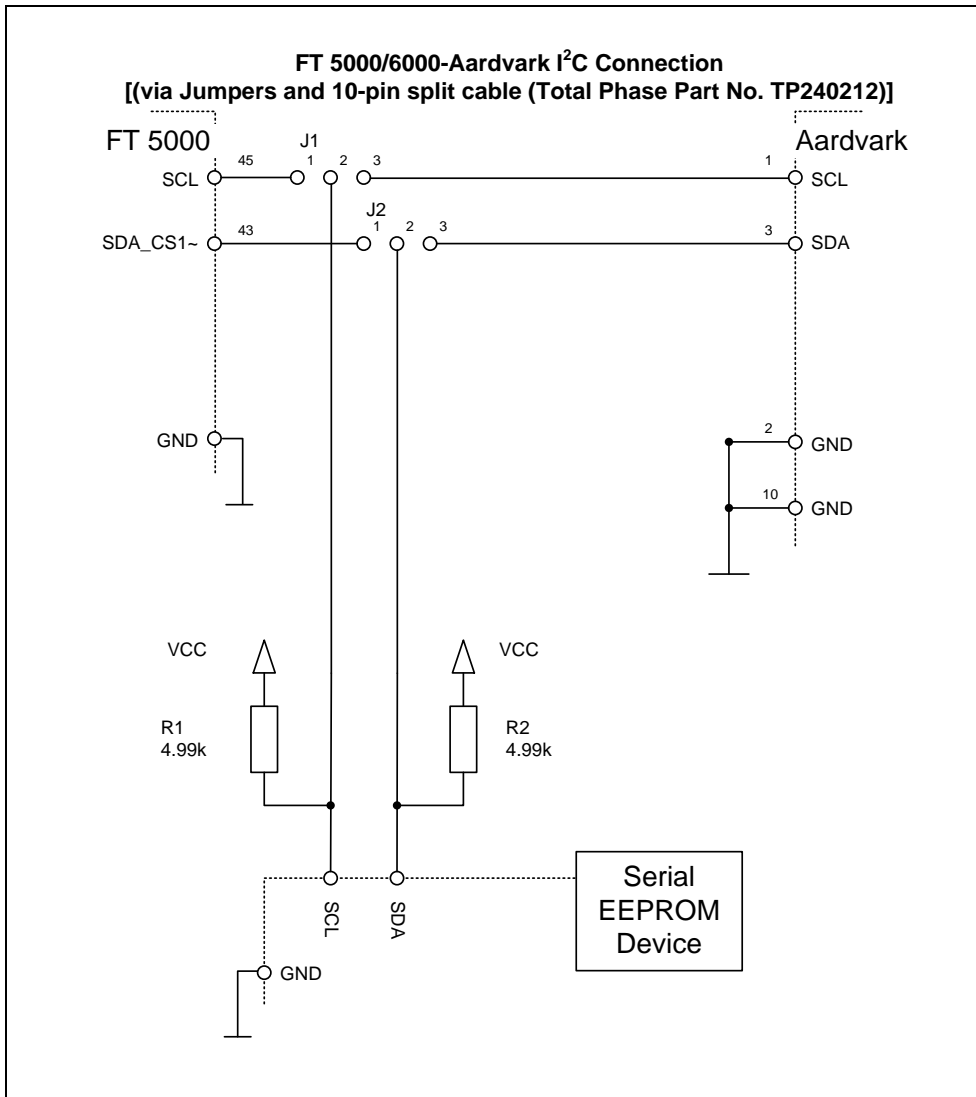
The following sections provide two sets of diagrams illustrating connection schemes that you could use for connecting external serial memory devices to the Aardvark programmer over the I²C and SPI interfaces in the Neuron core of the 5000 and 6000 Series chips.

The first diagram in each section illustrates how to connect the external serial memory device to the Aardvark programmer by connecting the TotalPhase 10-pin split cable to the Aardvark programmer and then inserting the flying leads on the 10-pin split cable to the jumpers on your device's board. This is ideal for scenarios where you want to physically disconnect the external serial memory device from the Neuron chip. Note that instead of using flying leads, you could use one or more custom cable adapters that are individually wired or switched-in to match the configuration for each external serial memory device to be programmed.

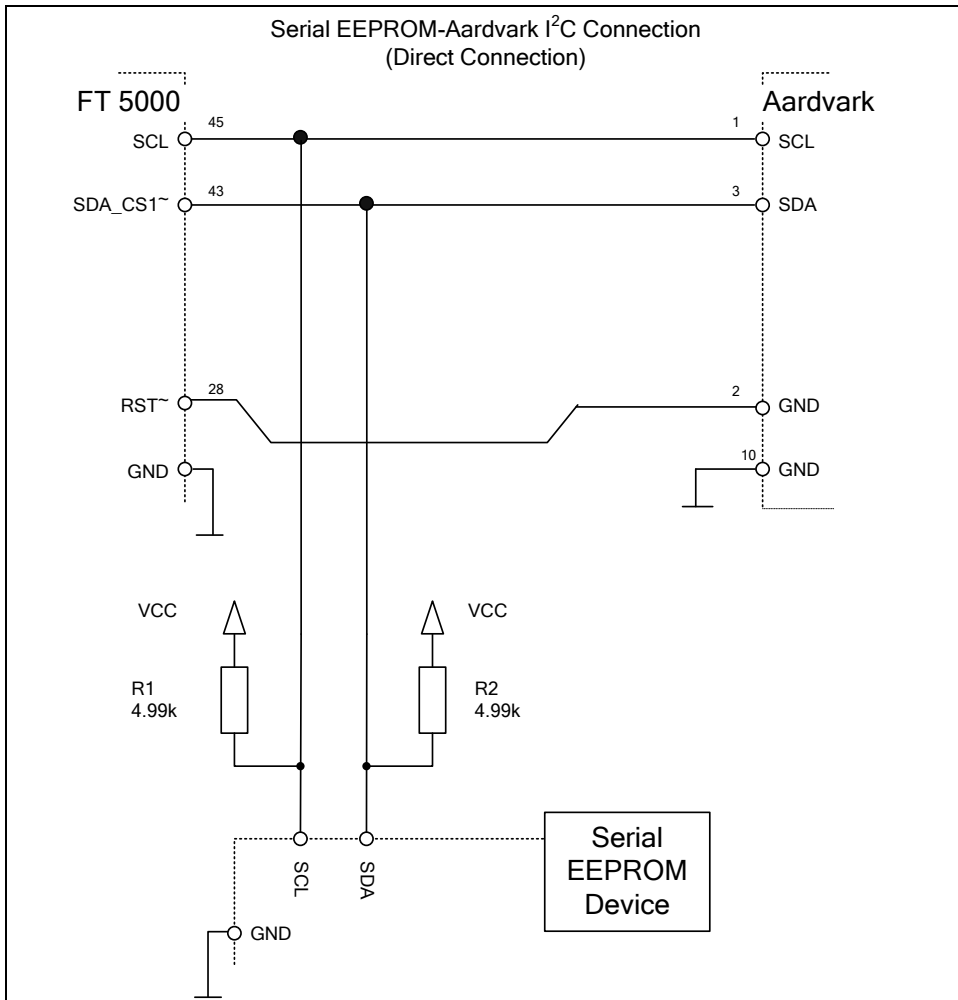
The second diagram in each section illustrates how to directly connect the external serial memory device to the Aardvark programmer. This is ideal for small devices where there may be insufficient space for jumpers on the board, or simple devices where jumpers are not desired. Note that in this scenario, the external serial memory devices are still connected to the 5000 or 6000 Series chip. You therefore must connect the RST~ pin on the Neuron chip to GND on the Aardvark programmer. This holds the RST~ line low, places the I²C and SPI interfaces in a high impedance state, and idles the Neuron chip. This eliminates the possibility of the Aardvark programmer conflicting with the Neuron chip when the Aardvark is accessing the I²C and SPI interfaces.

Serial Memory Device-Aardvark Connection Scheme for I²C Interface

To connect an external serial EEPROM device to the Aardvark programmer and perform in-circuit programming over the I²C interface, you could use the following schemes:



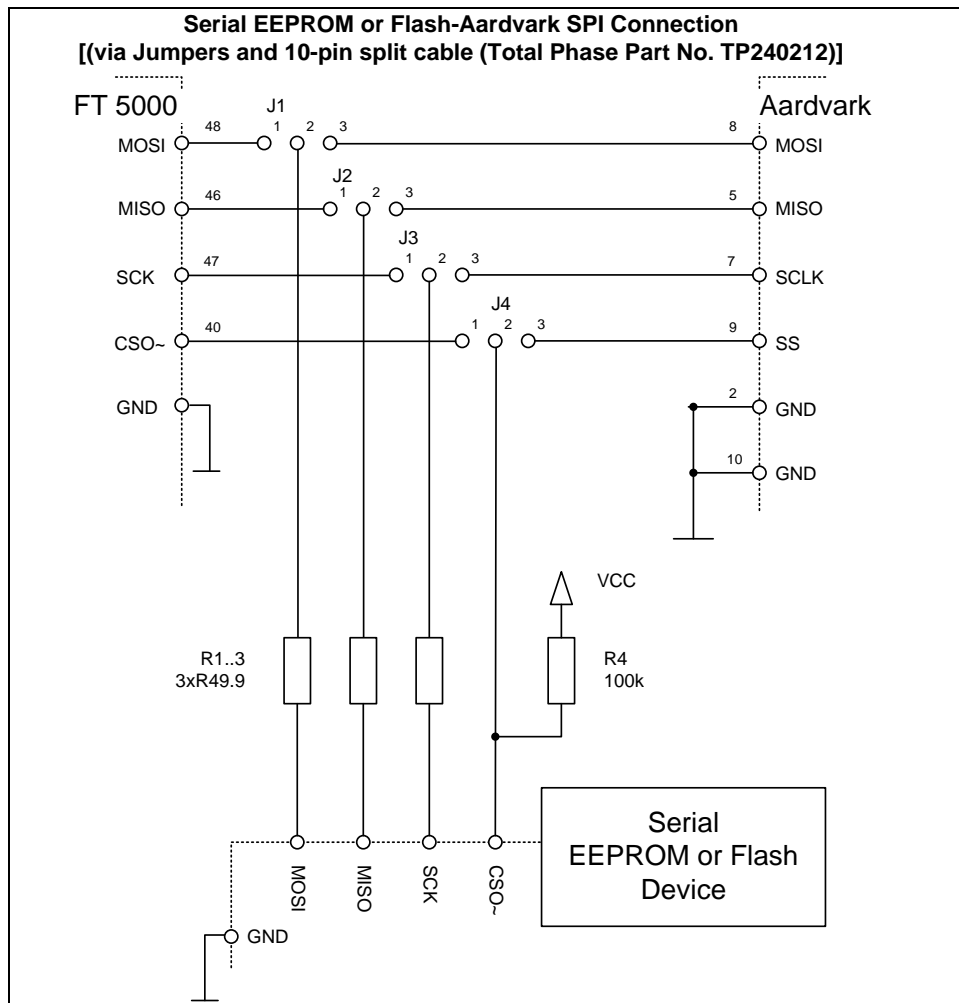
Note: In this diagram, all jumpers are set into position 1-2 for normal operation, and they are set into position 2-3 for in-circuit programming. You must always power off your device before changing the jumper settings.



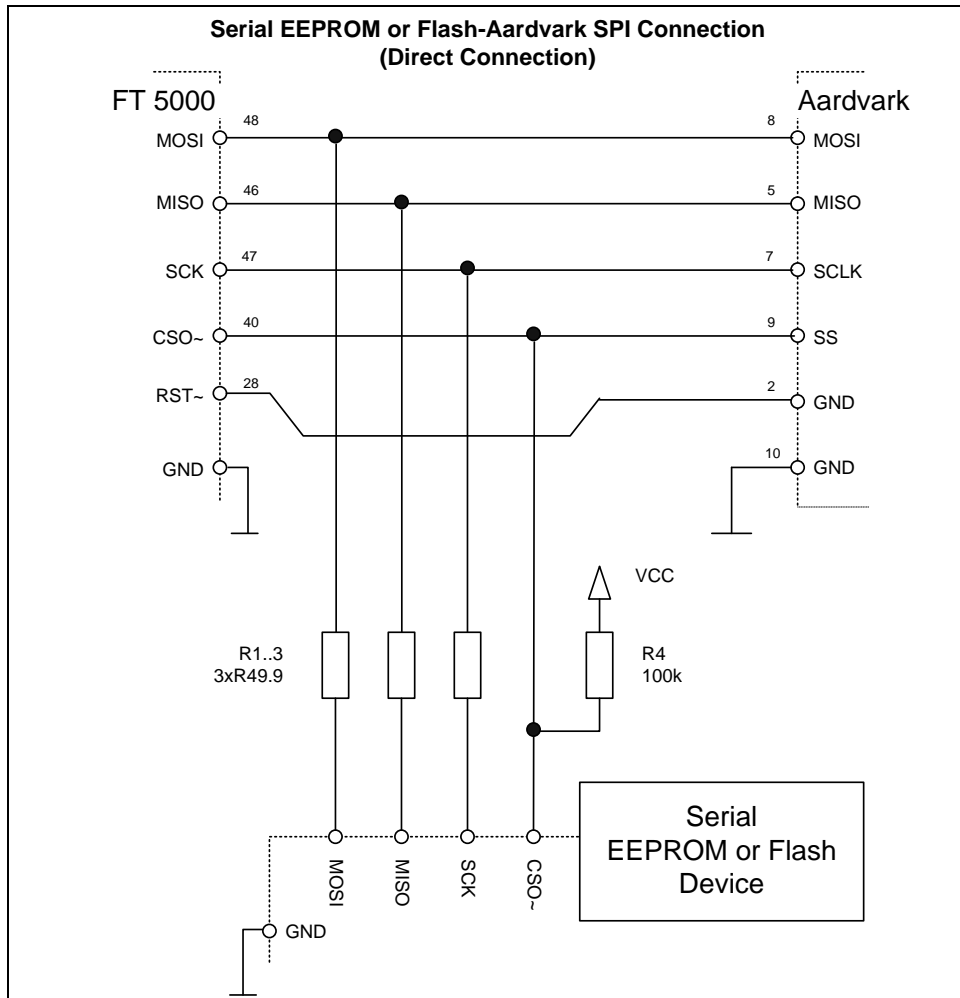
Note: Pins 2 and 10 on the Aardvark are both connected to ground inside the Aardvark. As a result, when the Aardvark is connected to the device board, it finds a reference ground at pin 10 and it takes the RST~ line to ground on pin 2 at the same time. This means that once the Aardvark connection is removed, the RST~ line is released and the 5000 or 6000 Series chip resumes normal operation.

Serial Memory Device-Aardvark Connection Scheme for SPI Interface

To connect an external serial EEPROM or flash device to the Aardvark programmer and perform in-circuit programming over the SPI interface, you could use the following schemes:



Notes: In this diagram, all jumpers are set into position 1-2 for normal operation, and they are set into position 2-3 for in-circuit programming. You must always power off your device before changing the jumper settings.



Note: Pins 2 and 10 on the Aardvark are both connected to ground inside the Aardvark. As a result, when the Aardvark is connected to the device board, it finds a reference ground at pin 10 and it takes the RST~ line to ground on pin 2 at the same time. This means that once the Aardvark connection is removed, the RST~ line is released and the 5000 or 6000 Series chip resumes normal operation.

Programming 3150 Off-chip Memory

A device based on a Neuron 3150 Chip, FT 3150 Smart Transceiver, or PL 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. Typical configurations use a 64KB flash memory part, and sometimes a RAM device. The application code may reside in any combination of the off-chip memory types, and the on-chip EEPROM. For information on the placement of application code in the various memory types, see *Using Memory* in the *Neuron C Programmer's Guide*.

You can program the Neuron firmware and your application image into a PROM or flash memory device using a compatible device 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 *Building 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 because 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, FT 3150 Smart Transceiver, or PL 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, FT 3150 Smart Transceiver, or PL 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 Device Template Target Properties: Configuration* in Chapter 5. 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 in *Setting Device Template Target Properties: Configuration* in Chapter 5.

Because 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, you can force this process to occur again with the same firmware image by resetting the Neuron 3150 Chip, FT 3150 Smart Transceiver, or PL 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 C:\LonWorks\Images folder, 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.

Programming 3120 and 3170 On-chip Memory

A Neuron 3120xx Chip, FT 3120 Smart Transceiver, PL 3120 Smart Transceiver, or PL 3170 Smart Transceiver does not support external memory; therefore, the only memory to program is on-chip EEPROM, which must be programmed over the network or with a 3120 or 3170 programmer.

A blank Neuron 3120xx Chip, FT 3120 Smart Transceiver, PL 3120 Smart Transceiver, or PL 3170 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 IzoT Commissioning tool.

To pre-program a Neuron 3120xx Chip, FT 3120 Smart Transceiver, PL 3120 Smart Transceiver, or PL 3170 Smart Transceiver with an application or network configuration other than the default, you must program it in a Neuron 3120 Chip or Neuron 3170 Chip programmer. Refer to the documentation supplied with the particular programmer for details.

Programming PL 3120 and PL 3170 Smart Transceiver Parameters

The PL 3120 and PL 3170 Smart Transceivers ship with an initial set of transceiver parameters pre-loaded for programming purposes. To ensure optimal operation, you must re-program the transceiver parameters for all PL 3120 and PL 3170 chips using the NodeLoad utility.

- For a device based on a PL 3120 Smart Transceiver, you can use the NodeLoad utility with the `-X` option to change the transceiver parameters from the factory default parameters to any of the supported parameters.
- For a device based on a PL 3170 Smart Transceiver, you can use the NodeLoad utility to change the parameters to any of the various C-band types (the PL 3170 Smart Transceiver does *not* support A-band operation).

To load transceiver parameters using the NodeLoad utility, you must use the `.NDL` or `.NEI` image because the `.NXE` image does not contain transceiver parameter values. You can also use a universal programmer, such as BP Microsystems' programmer or HiLo System's programmer, to change the parameters prior to soldering the chip onto your PCB board. All valid transceiver parameters included in the application image files generated by the IzoT NodeBuilder tool are supported.

If you reboot a PL 3120 or PL 3170 Smart Transceiver, the smart transceiver will restore the factory default parameters and go back to the initial state. Rebooting in this case refers to any of the following operations:

Software	Action
OpenLNS application	Invoking the Reboot() method for AppDevice or Router object.
NodeUtil utility (version older than 1.96)	Sending the "Reboot" command for the device
Network management command with the appl_reset option	Writing a value of zero to the second byte of the transceiver parameters on the Smart Transceiver and resetting the device with the Set Node Mode.

Note: If you simply power cycle or reset your device, it will maintain the programmed change; it will not restore the factory default.

Upgrading Device Applications

The 5000 and 6000 Series chips are compatible with device applications written for 3150 and 3120 Neuron Chips and Smart Transceivers. You can use the IzoT NodeBuilder tool to port your old application to a 5000 or 6000 Series chip. To do this, you open the device's NodeBuilder project, update the Neuron Chip model used by the hardware template to the Neuron 5000 processor or FT 5000 or FT 6000 Smart Transceiver, and then re-build the device application. See *Editing Hardware Templates* in Chapter 5 for more information on using the Hardware Template Editor.

Note: The Neuron C Version 2.2 language includes the following new keywords: **interrupt**, **__lock**, **stretchedtriac**, **__slow**, **__fast**, and **__parity**. Some of these keywords use a double underscore prefix to avoid any naming collisions within existing device applications. The Neuron C Version 2.3 language adds the following new keywords: **__resident**, **__type_scope** and **__type_index**.

You can also use the IzoT NodeBuilder tool to upgrade your existing device applications to the new Version 3 code templates when porting them to a 5000 or 6000 Series chip. The Version 3 code templates include improved code size, speed, and compliance with interoperability guidelines. To upgrade existing device applications to the Version 3 template, see *Using Code Wizard Templates* in Chapter 6.

Adding and Managing Target Devices

A *target device* is a LONWORKS device application that is built by the IzoT 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 and resource files, but they may 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 device is defined by a LonMaker 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 device to a NodeBuilder project using the IzoT Commissioning tool or the NodeBuilder Project Manager (you should use the IzoT Commissioning tool because it is typically faster and easier). After you add a target device, you can use the NodeBuilder Project Manager to re-build and debug it and to view and change its NodeBuilder device template and target type.

Adding a Target Device with the IzoT Commissioning Tool

You can add a target device to a NodeBuilder project using the IzoT Commissioning tool. To add a target device with the IzoT Commissioning 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.
3. Create a new a LonMaker network or open an existing one. See the *IzoT CommissioningTool User's Guide* for more information on creating and opening LonMaker drawings.

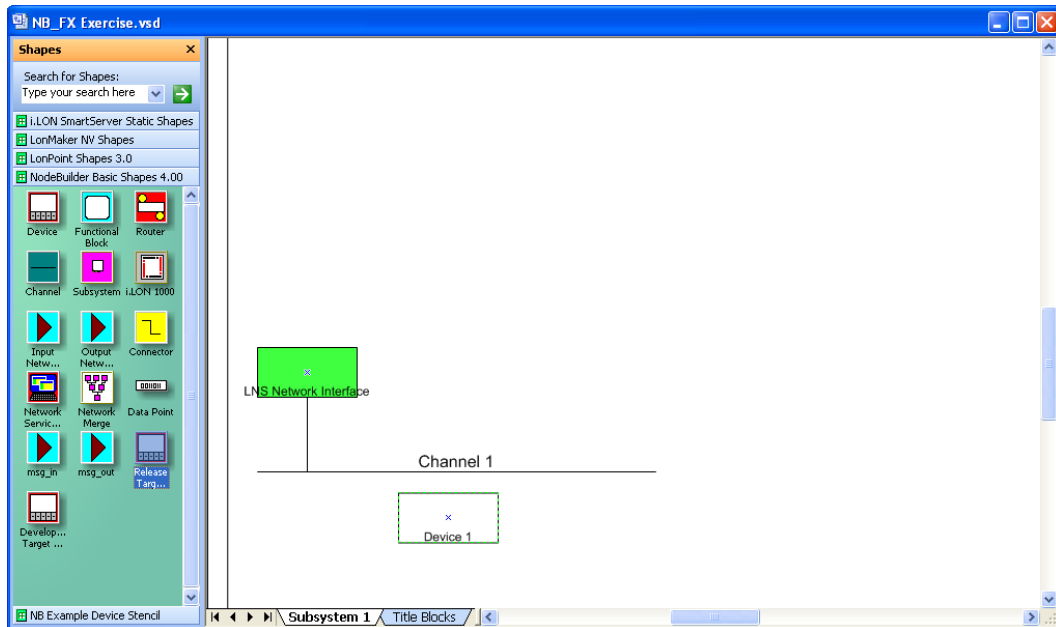
You will use the IzoT Commissioning tool to install, bind, configure, and test the targets in your project. The IzoT Commissioning tool displays a network drawing that shows the devices, functional blocks, and connections in your network.

The IzoT Commissioning tool also displays *stencils* that contain shapes that you can drag to your LonMaker drawing. The IzoT Commissioning tool includes a **NodeBuilder Basic Shapes 4.00** stencil with shapes that you can use to add new devices, functional blocks, and connections to your network drawing. The **NodeBuilder Basic Shapes 4.00** stencil contains shapes that can be

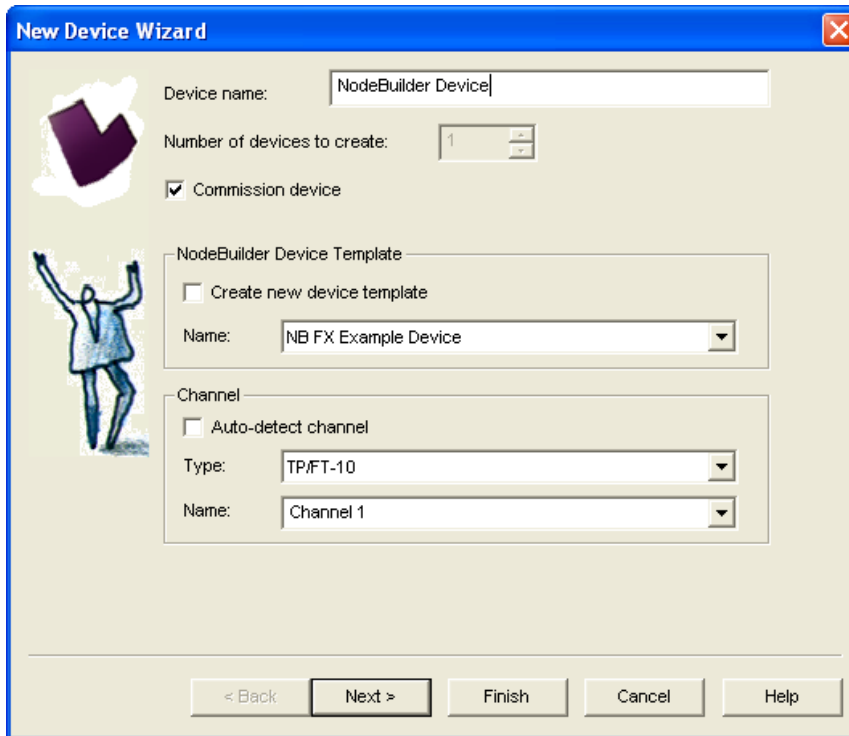
used with any device. You can also create custom stencils with shapes customized for your devices and networks.

The **NodeBuilder Basic Shapes 4.00** stencil contains a **Development Target Device** shape and a **Release Target Device** shape. These special device types help distinguish between other devices on the network and the target devices used by the IzoT NodeBuilder tool. The IzoT NodeBuilder tool lets you create a mixed network of development hardware (such as the FT 5000 EVB, FT 6000 EVB, or the LTM-10A Platform), release hardware (your own hardware), and other devices.

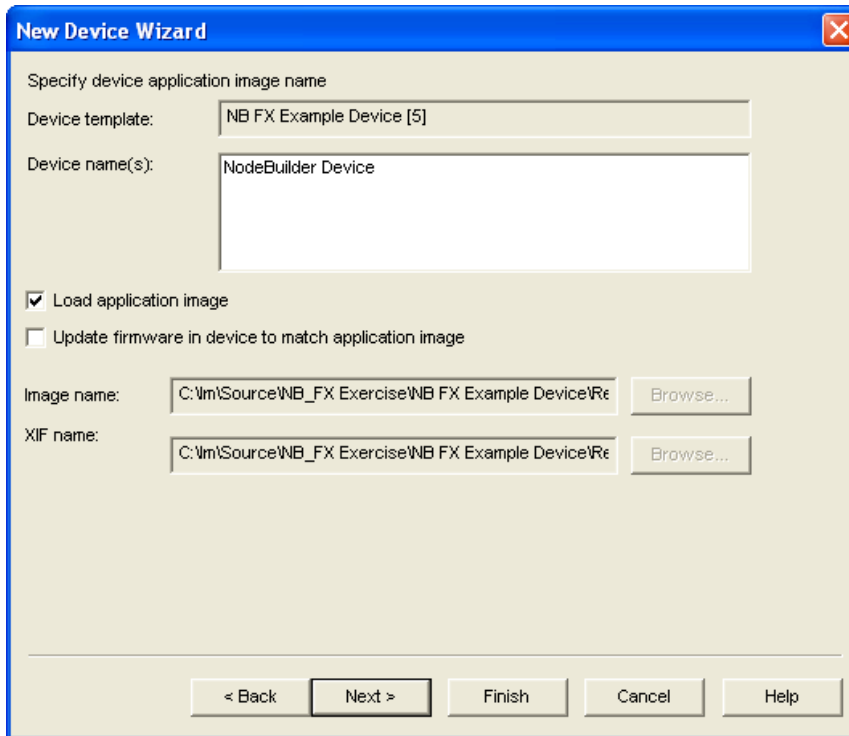
4. Drag a **Development Target Device** shape or **Release Target Device** shape from the **NodeBuilder Basic Shapes 4.00** stencil to your network drawing. You can drop the shape anywhere, but a good location is just below the Channel 1, near the LNS network interface shape on your drawing.



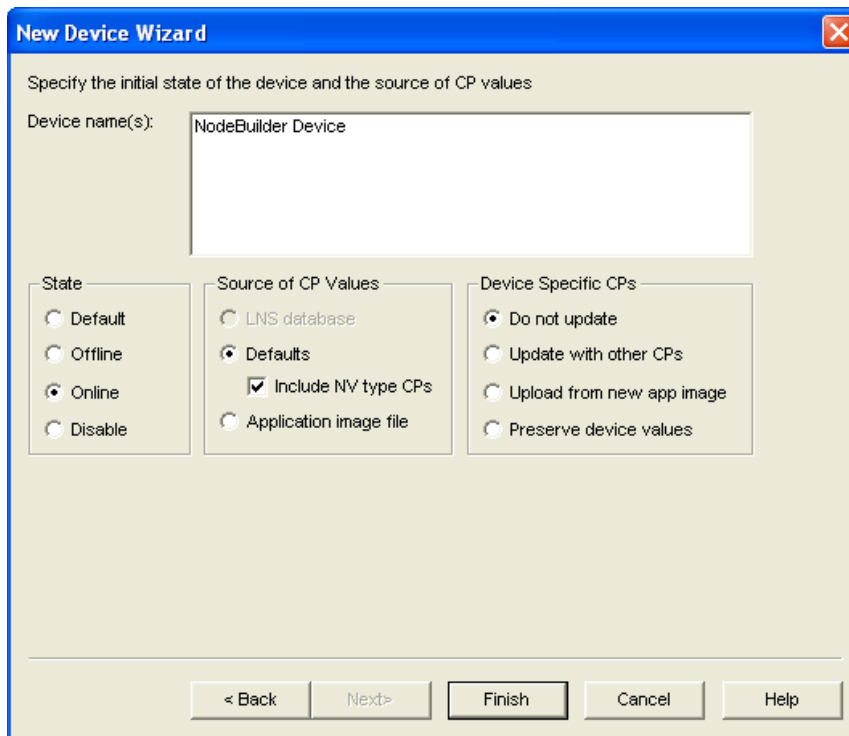
5. The New Device Wizard opens.
6. In the **Device Name** property, enter a name for the target. This name must be unique for all the devices and targets within the current page (subsystem). The default name is the Device followed by an integer (e.g. Device 1). The device name may be up to 85 alphanumeric characters and include embedded spaces; the name may not include the period, backslash, colon, forward slash, or double quote characters.
7. Select the **Commission Device** check box.



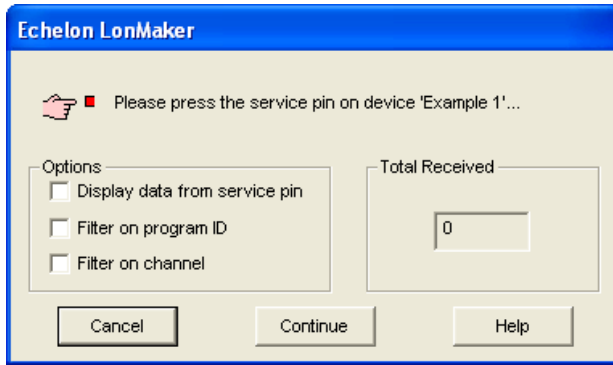
8. Click **Next** three times. The window in the New Device Wizard lets you select the application image to be downloaded to your device.
9. Select the **Load Application Image** check box and then click **Next**. This specifies that you will download the binary application image file (.APB extension) built for the device application to the device. The binary application image files for your device applications are stored in the `C:\Lm\Source\<NodeBuilder Project>\<NodeBuilder Device Template>\<Release || Development>` folder.



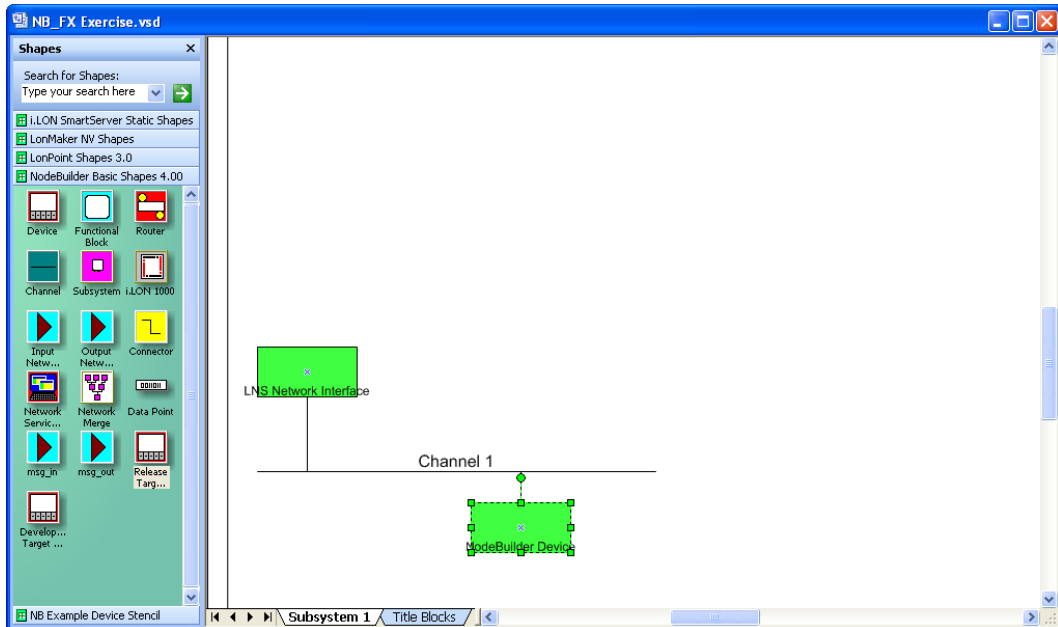
10. The next window lets you set the initial device state and the source of configuration property values when your device is commissioned.
11. Select the **Online** option under **State**. This means that your device will run its application after it has been commissioned.



12. Click **Finish**. The Press Service Pin window appears.



13. Press the service pin on the development platform you to be loaded and commissioned. The IzoT Commissioning tool loads the application image for your device application to the device and makes it operational. When the IzoT Commissioning tool is done commissioning, it will return to the LonMaker drawing. The device shape will be solid green indicating that the device has been commissioned and is online. The device application will not do anything until you test the device or connect it to other devices.

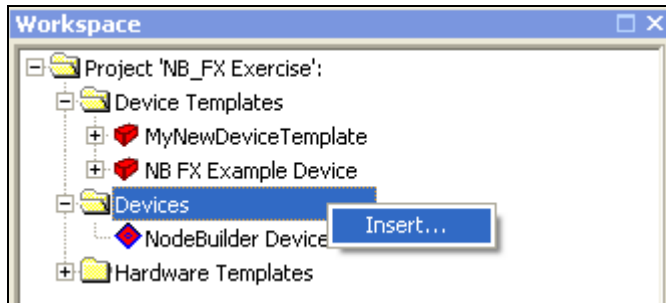


14. Test your device's interface using the IzoT Commissioning tool. See Chapter 9, *Testing a NodeBuilder Device Using the IzoT Commissioning Tool*, for more information.
15. Debug your device application Debugging a Neuron C Application. See Chapter 10, *Debugging a Neuron C Application*, for more information.

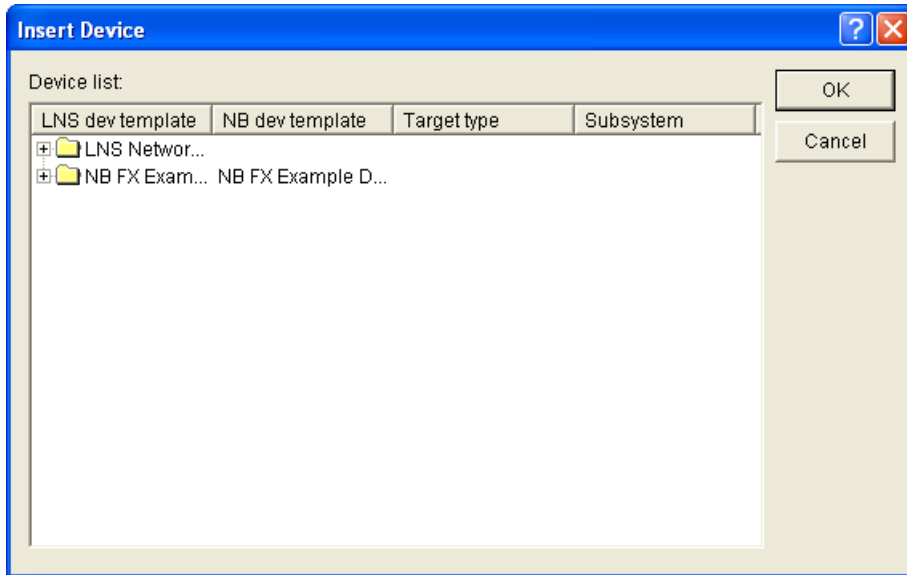
Adding a Target Device with the NodeBuilder Project Manager

You can use the NodeBuilder Project Manager to add the devices in any open LonMaker network to your current NodeBuilder project. To do this, follow these steps:

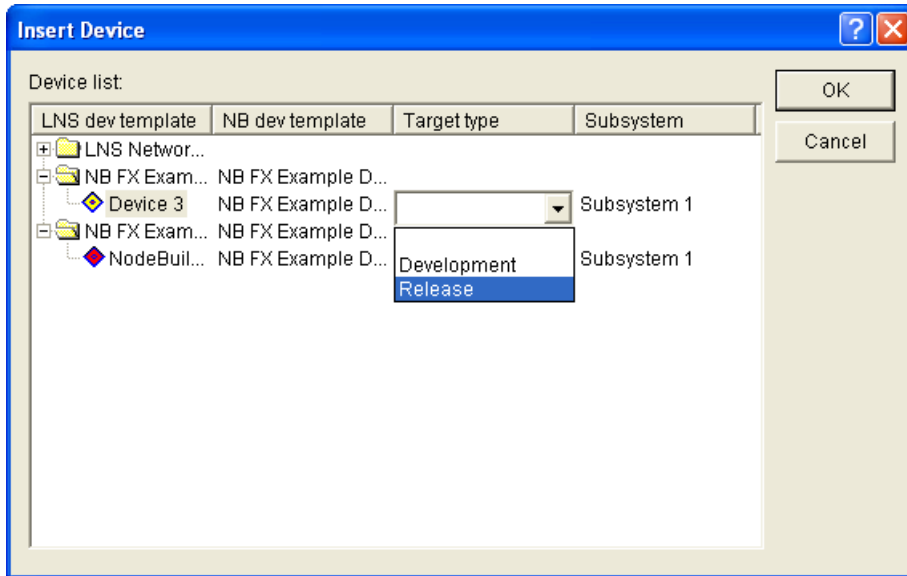
1. Right-click the **Devices** folder in the Project pane and click **Insert** on the shortcut menu.



2. The **Insert Device** dialog opens.



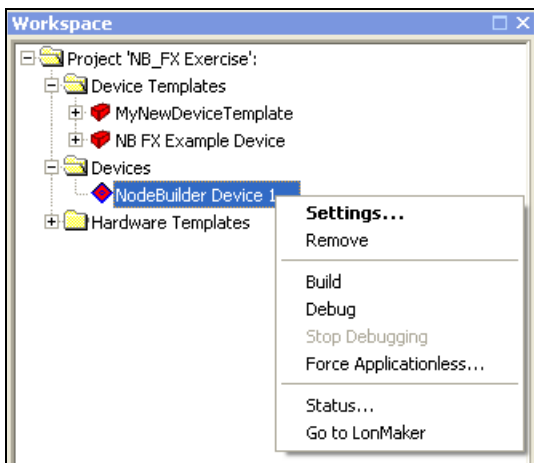
3. This dialog organizes the devices in currently open LonMaker networks by LNS device template name. If the LNS device template used by the device is based on a NodeBuilder device template, the NodeBuilder device template name is displayed in the **NB Dev Template** column. These devices cannot be added to your NodeBuilder project.
4. Expand the folder containing the desired device template and then select the device to be added.
5. Click anywhere under the **NB Dev Template** column, and then select a NodeBuilder device template in the current project that is currently not associated with an LNS device template in the project.
6. Click anywhere under the **Target Type** column, and then select either a **Development** or **Release** target type.



7. Click **OK** to add the target to the **Devices** folder in the NodeBuilder Project pane. If this device is commissioned, the IzoT NodeBuilder tool will download the application to the device the next time you build it.

Managing Target Devices

You can build, debug, and edit target devices from the Project pane in the NodeBuilder Project Manager. The **Devices** folder in the Project pane contains all the targets defined in the current NodeBuilder project that you have created in a LonMaker network. You can right-click a device to open a shortcut menu with the following options:



Settings

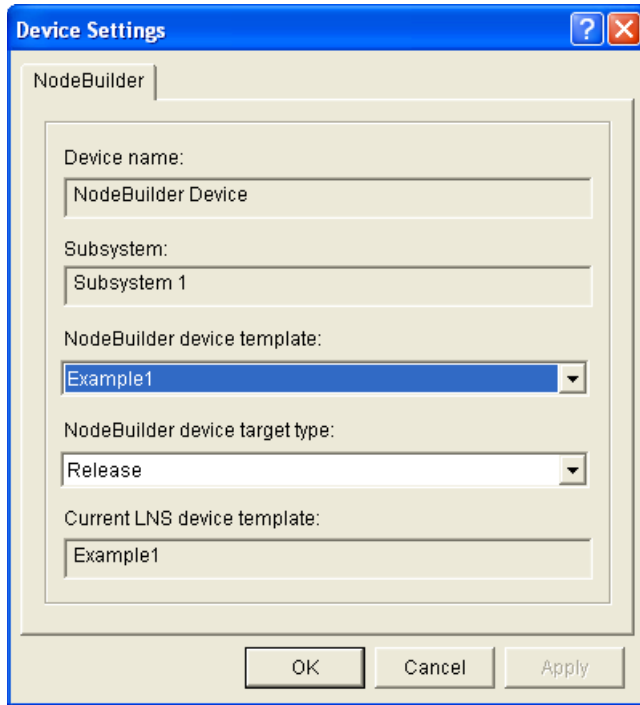
Opens the **Device Settings** dialog, which lets you view and configure device settings including the NodeBuilder device template and target type.

<i>Remove</i>	Removes the device as a target for future builds. The device is removed from the current NodeBuilder project, but it is not removed from the LonMaker drawing or network, and none of the device files are deleted. To replace the LonMaker shape in your LonMaker drawing, drag the Device shape in the LonMaker Basic Shapes stencil over the Development or Release Target shape, select the Replace the Existing Device Shape with the Shaped just Dropped check box in the New Device Wizard, and then click OK .
<i>Build</i>	Builds the application image for the device template assigned to this device. For more information, see <i>Building an Application Image</i> earlier in this chapter.
<i>Debug</i>	Debugs the device. For more information, see Chapter 10, <i>Debugging a Neuron C Application</i> . This command is unavailable if the application image has not been built. This command is not displayed if the device is already being debugged.
<i>Stop Debugging</i>	Stops debugging the device. This command is not displayed if the device is not being debugged.
<i>Force Applicationless</i>	Forces the selected device to the applicationless state by clearing its program ID. To use the device, you must reload the application, or load a new application.
<i>Status</i>	Displays the build status for this device and its device template.
<i>Go to LonMaker</i>	Switches focus to the LonMaker drawing with the device shape selected. The LonMaker drawing must be open for this command to work.

Editing Target Device Settings

You can edit the device settings for a target device. The device settings include the NodeBuilder device template and NodeBuilder target type for the target. To edit the target device settings, follow these steps:

1. Right-click the target in the **Devices** folder in the Project pane and then click **Settings** on the shortcut menu.
2. The **Device Settings** dialog opens.



3. You can view and set the following properties:

<i>Device Name</i>	Displays the name of the device specified in the LonMaker drawing. This field is read-only.
<i>Subsystem</i>	Displays the subsystem (drawing page) in the LonMaker drawing where the device is located. This field is read-only.
<i>NodeBuilder Device Template</i>	<p>Displays the name of the current NodeBuilder device template used by the target. You can change the NodeBuilder device template used by the target by selecting a different one from the list of those in the current NodeBuilder project. If you change the NodeBuilder device template, the change is not implemented until you build the device template and load the target.</p> <p>When you load the target with the new device template, the IzoT Commissioning tool will preserve any functional blocks and connections that are compatible between the old device template and the new device templates. Incompatible functional blocks and connections 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, drag the new shape on top of the old shape in your LonMaker drawing, select the Replace the Existing Device Shape with the Shaped just Dropped check box in the New Device Wizard, and then click OK.</p>
<i>NodeBuilder Device Target Type</i>	<p>Displays the device target type, which may be Development or Release. You can change the target type. The device shape in the LonMaker drawing will not change when you change the target type. If you change the target type, you should replace the shape by dragging the new shape on top of the old shape.</p>

Current LNS Device Template Displays the name of the LNS device template associated with the target. This field is read-only and is automatically updated if you build the target with a new NodeBuilder device template.

4. Click **OK** to save the settings.

Testing a NodeBuilder Device Using the IzoT Commissioning Tool

This chapter describes how to use the Data Point shape and LonMaker Browser in the IzoT Commissioning tool to monitor and control your device. It explains how to use the IzoT Commissioning tool to connect your NodeBuilder device to other LONWORKS devices in a network.

Introduction to Testing NodeBuilder Devices

You can use the IzoT Commissioning tool to test your NodeBuilder device. You can press the hardware inputs on your device and use the IzoT Commissioning tool to monitor changes to the values of the network variables in the device interface. You can also use the IzoT Commissioning tool to control the values of the input network variables and observe whether the hardware outputs function as designed and output network variable values change accordingly. After you determine that your device is functioning as designed, you can use the IzoT Commissioning tool to connect your development devices to other devices and verify their operation within a network.

Monitoring and Controlling NodeBuilder Devices

You can monitor and control your device with the IzoT Commissioning tool using the Data Point shape in the **LonMaker Basic Shapes** stencil or the LonMaker Browser.

The Data Point shape lets you monitor and control a single network variable or configuration property value from the current drawing page. It is ideal for testing smaller device interfaces with few network variables and configuration properties. You can also use the Data Point shape to create simple HMIs in your LonMaker drawing.

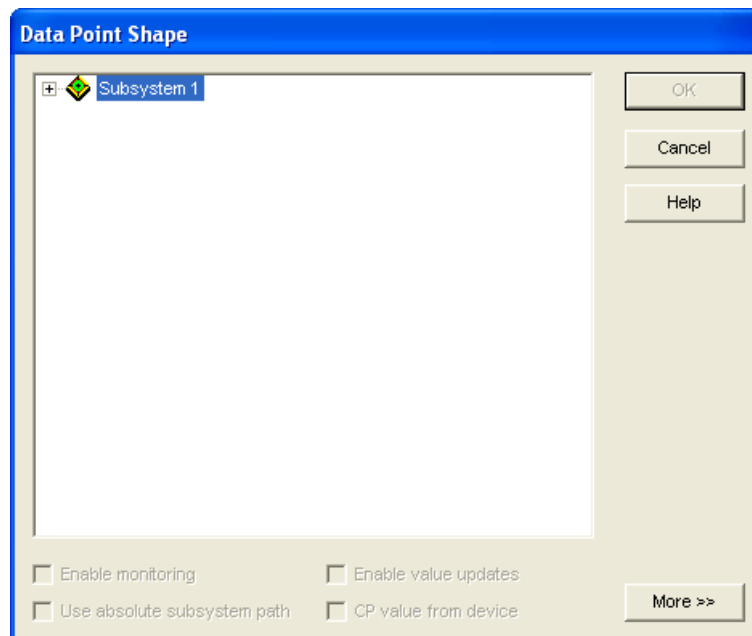
The LonMaker Browser can display the values for all the input and output network variables and configuration properties in your device interface. It is ideal for testing devices with larger external interfaces.

The following sections describe how to monitor and control your device using each of these methods.

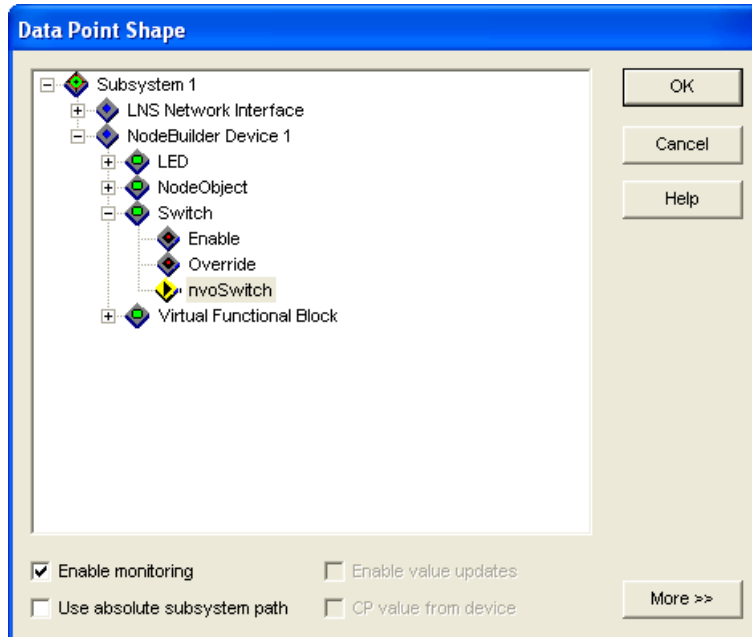
Using the Data Point Shape

To test your device's interface with the Data Point shape, follow these steps:

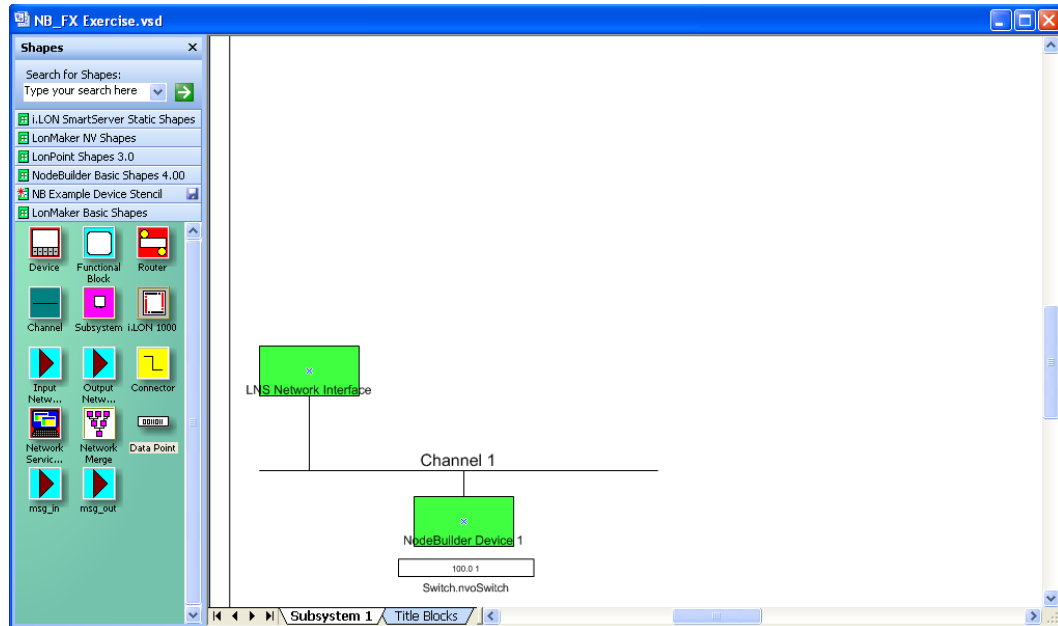
1. Open the LonMaker drawing containing your device. See the *IzoT Commissioning Tool User's Guide* for more information on opening LonMaker drawings.
2. Drag a **Data Point** shape from the **LonMaker Basic Shapes** stencil on the left of the LonMaker window to the drawing. You can place the Data Point shape anywhere, but a good place is directly above or below the device or functional block containing the data point to be monitored and controlled. The **Data Point Shape** dialog opens.



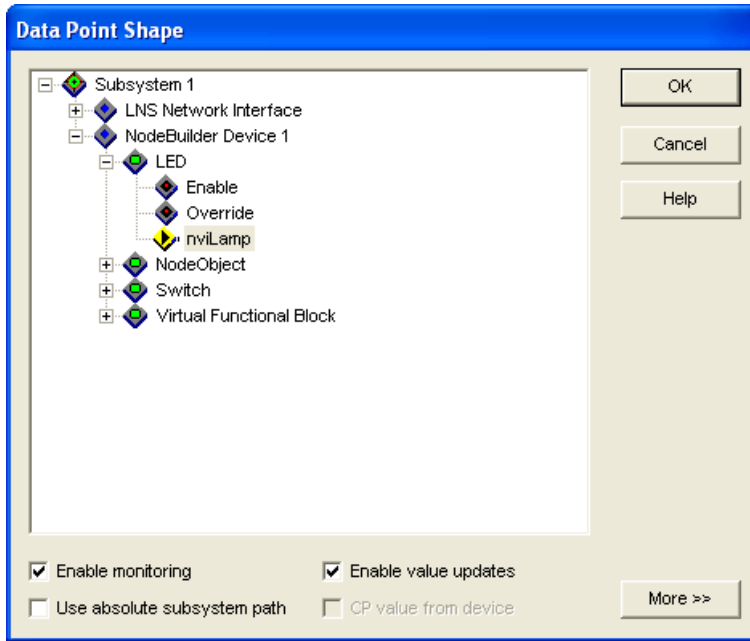
- Expand the **Subsystem** icon, expand your NodeBuilder device icon, expand a functional block in the device interface corresponding to a hardware input, and then select an output network variable in the functional block; select the **Enable Monitoring** check box; and then click **OK**.



- The Data Point shape is added to your LonMaker drawing.



- Toggle the hardware input and observe the value of the corresponding output network variable change in the Data Point shape.
- Repeat steps 2–4 to add a Data Point shape that monitors and controls an input network variable in a functional block corresponding to a hardware output. In the **Data Point Shape** dialog, select the **Enable Value Updates** check box.



7. Double-click the Data Point shape for the input network variable, enter a different value, and then click anywhere outside the Data Point shape. Observe the hardware output change based on the value you entered for the input network variable.

If the data point has a structured value, you can also set the value by right-clicking the data point shape and selecting **Value Details** on the shortcut menu. The **Set Network Variable Value** dialog opens. You can set the values for the individual fields in the structure, and then click **OK** to save the changes.

If the data point has an enumerated value, you can set the value by right-clicking the data point shape, pointing to **Set Details** on the shortcut menu, and then selecting an enumeration from the list that appears.

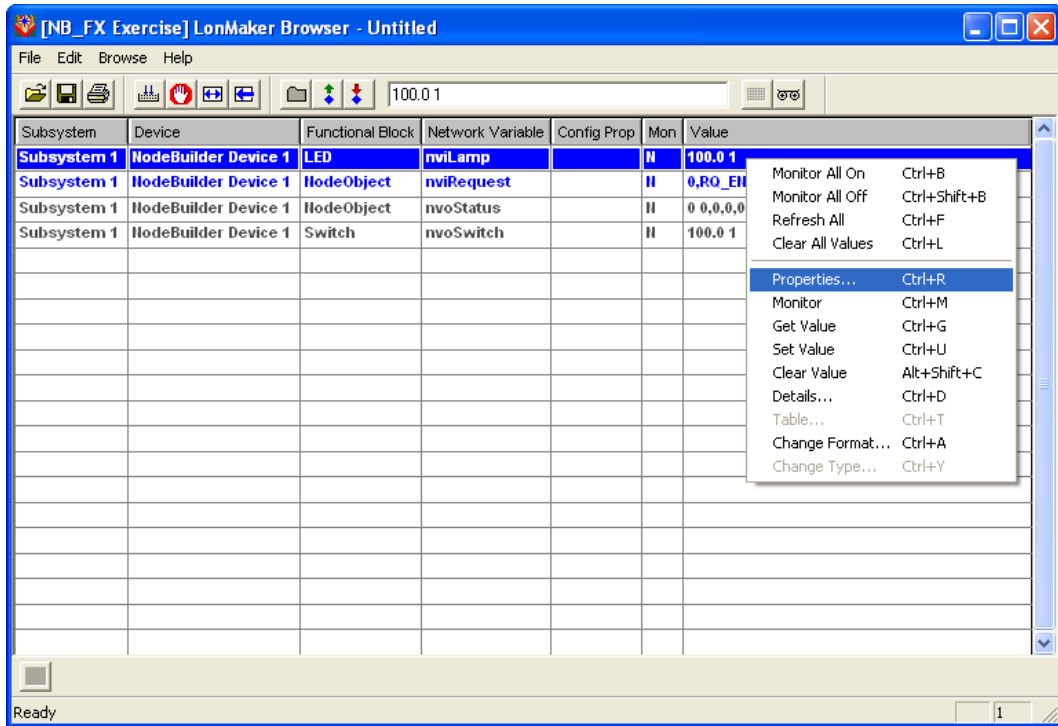
For more information on using the Data Point shape in the IzoT Commissioning tool, see Chapter 6 of the *IzoT Commissioning Tool User's Guide*.

Note: If these steps do not generate the expected result, open the IzoT NodeBuilder tool and check your code. You can also use the NodeBuilder Debugger to help troubleshoot problems (for more information, see Chapter 10, *Debugging a Neuron C Application*).

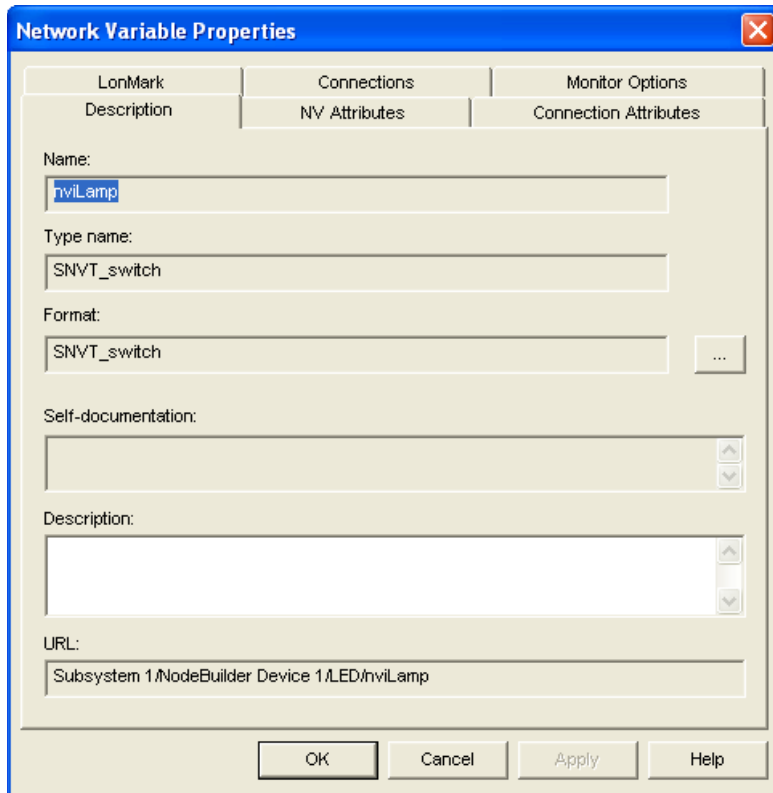
Using the LonMaker Browser


To test your device's interface with the LonMaker Browser, follow these steps:

1. Open the LonMaker drawing containing your device. See the *IzoT Commissioning Tool User's Guide* for more information on opening LonMaker drawings.
2. Right-click the device in your LonMaker drawing, then click **Browse** on the shortcut menu.



- The **Network Variable Properties** or **Configuration Property Properties** dialog opens.



- Verify that the network variable or configuration property has the correct type and size.
- Click the  **Monitor All** button on the toolbar to start polling all network variable and configuration property values.

8. Change network variable and configuration property values and confirm that the device hardware works as designed. For example, toggle a hardware input and observe the value of the corresponding output network variable change. You can then change the value of an input network variable and observe the hardware output change based on the value you entered.

For more information on using the LonMaker Browser, see Chapter 6 of the *IzoT Commissioning Tool User's Guide*.

Note: If these steps do not generate the expected result, open the IzoT NodeBuilder tool and check your code. You can also use the NodeBuilder Debugger to help troubleshoot problems (for more information, see Chapter 10, *Debugging a Neuron C Application*).

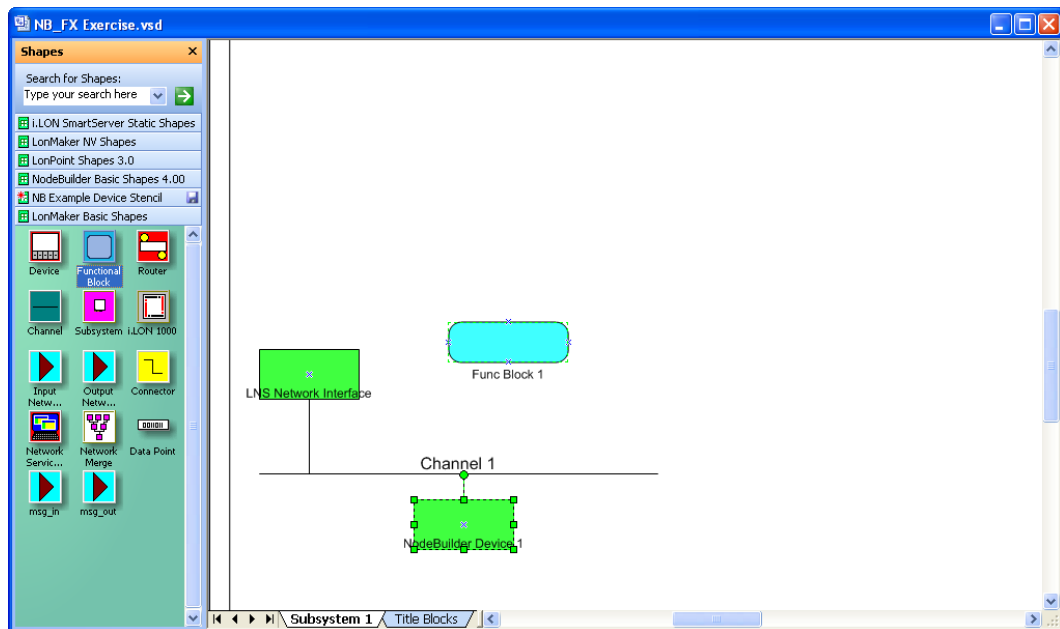
Connecting NodeBuilder Devices

Once you determine that your device is functioning as desired, you can test it as part of a network. You can use the IzoT Commissioning tool to connect your development devices to other devices and verify their operation within a network. This entails creating functional blocks, connecting the network variables within the functional blocks, and verifying that the network variable values are updated appropriately when you use the I/O devices on your device hardware.

Note: You can connect an output network variable of a device to one or more compatible input network variables on the same device. These connections are referred to as *turnaround connections*.

To connect your NodeBuilder device, follow these steps:

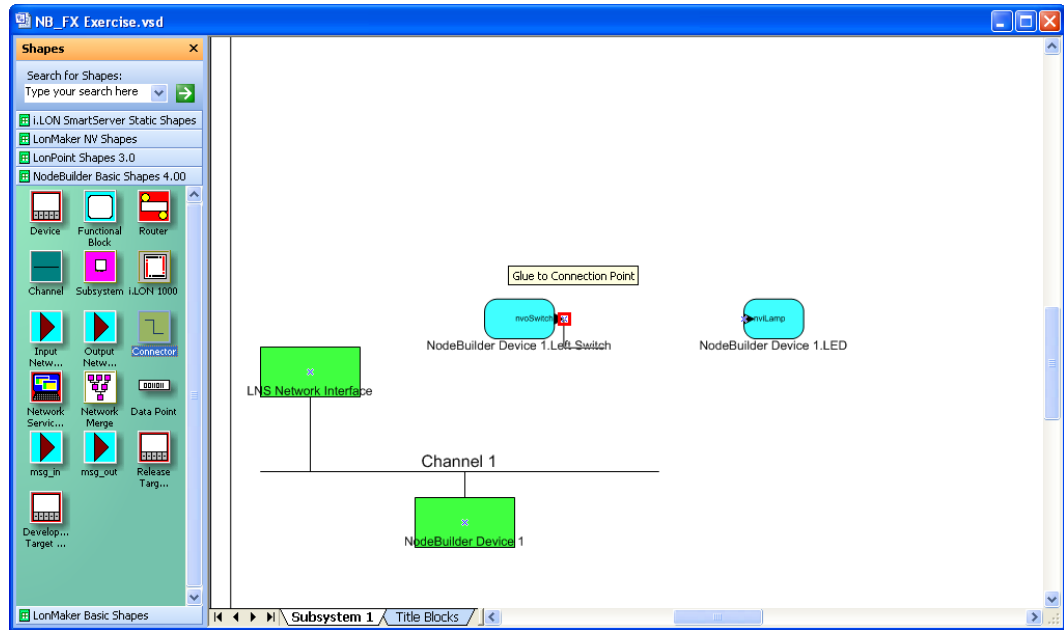
1. Open the LonMaker drawing that contains the NodeBuilder device. The device must be built and it must be associated with the appropriate LNS device template.
2. Drag a **Functional Block** shape from the **NodeBuilder Basic Shapes 4.00** stencil or the **LonMaker Basic Shapes** stencil on the left of the LonMaker window to the drawing.



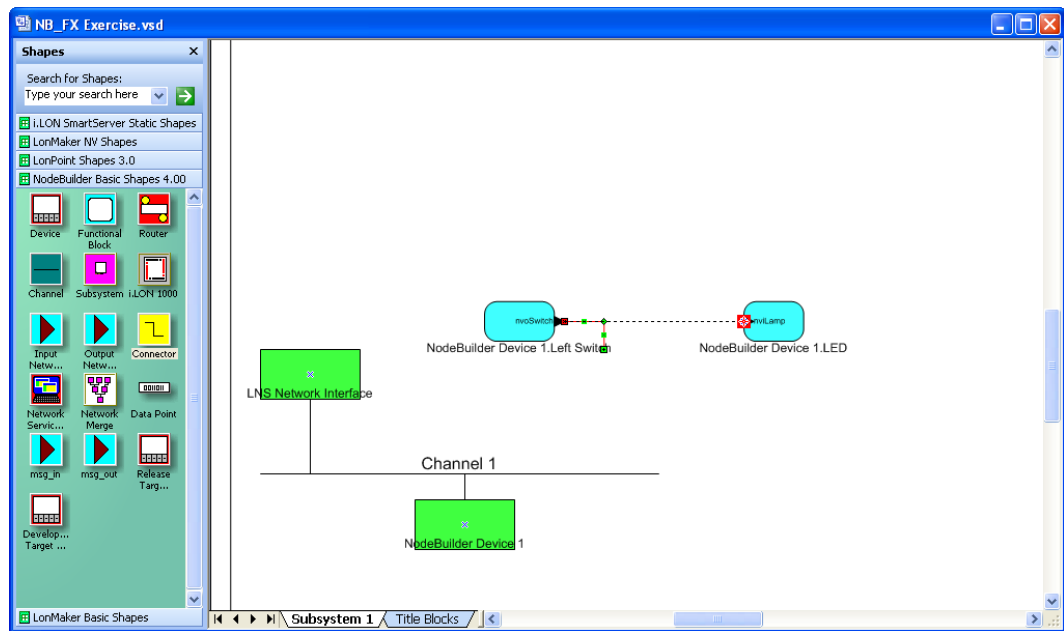
3. The Functional Block wizard opens. You will use this wizard to associate the new functional block shape with your NodeBuilder device and the desired functional block.
4. In the Functional Block wizard, do the following:
 - a. In the **Name** property under **Device**, select your NodeBuilder device.
 - b. In the **Name** property under **Functional Block**, select a functional block from those defined in the device interface.


- c. In the **New FB Name**: property under **Functional Block**, enter the name for the functional block. The functional block name may be up to 85 alphanumeric characters and include embedded spaces; the name may not include the period, backslash, colon, forward slash, or double quote characters.
- d. Select the **Create All Network Variable Shapes** check box.

5. Click **Finish**. The New Functional Block wizard closes and the LonMaker drawing appears. A new functional block shape appears on the drawing.
6. Repeat steps 4–5 for each functional block in your NodeBuilder device. If the device contains any implementation-specific or device network variables or configuration properties (network variables and 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 NodeBuilder Code Wizard can be created by the IzoT Commissioning tool.
7. Connect the output network variable on one functional block to an input network variable on another functional block. To do this follow these steps:
 - a. Drag the **Connector** shape from the **NodeBuilder Basic Shapes 4.00** stencil or the **LonMaker Basic Shapes** stencil to the drawing. Position the left end of the shape over the tip of the output network variable on the 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.

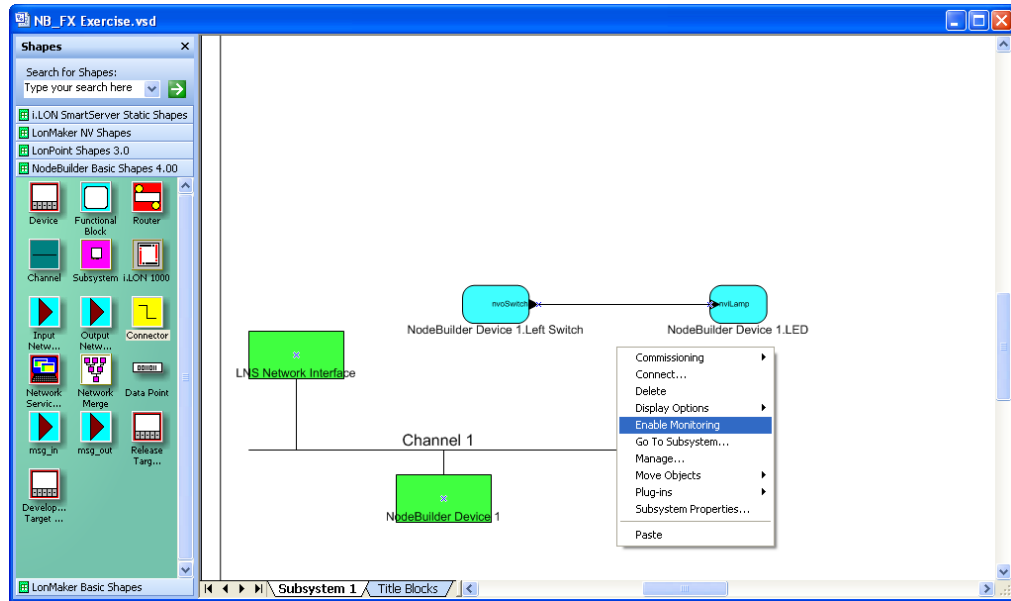


- b. Drag the other end of the **Connector** shape to the input network variable of the other functional block until it snaps into place and a square box appears around the end of the **Connector** shape. There is a brief pause as the IzoT Commissioning tool updates the device over the network.

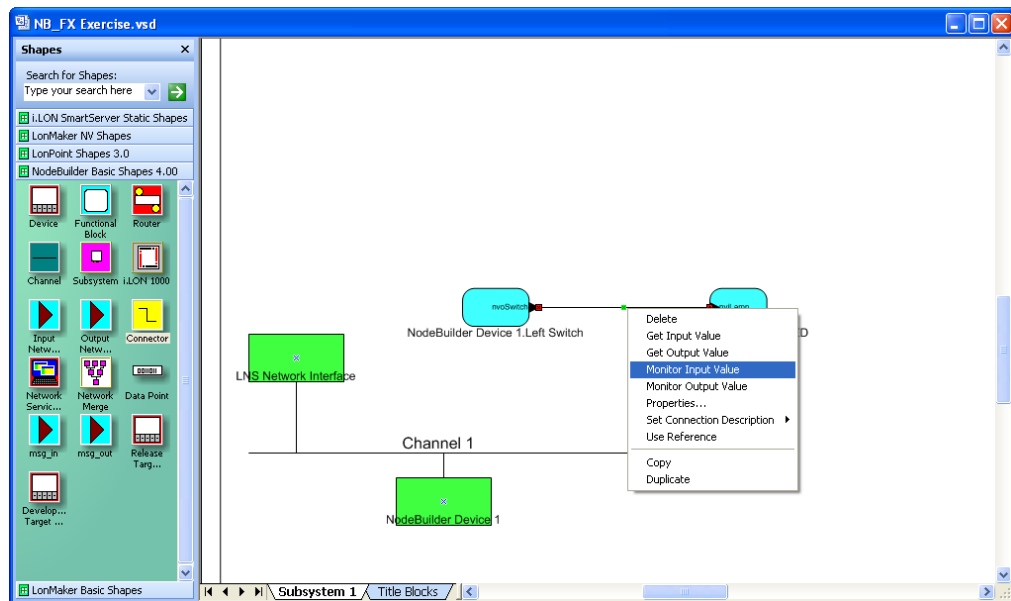


Note: You can also create connections using the **Connector** tool () on the Visio **Standard** toolbar or the **Network Variable Connection** dialog box. See Chapter 4 of the *IzoT Commissioning Tool User's Guide* for more information on creating connection using these methods.

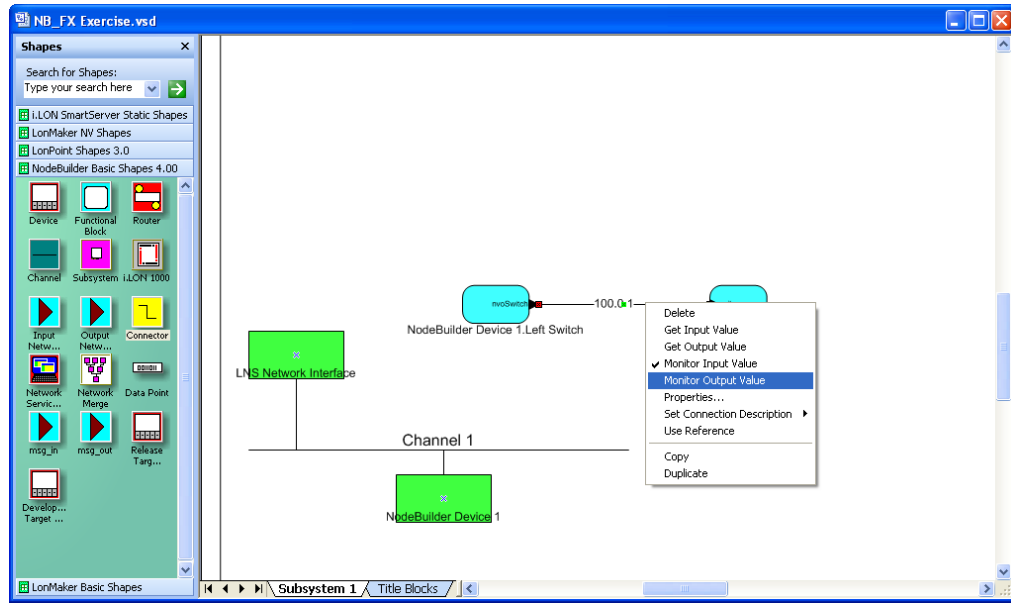
8. Monitor the values of the connected network variables. To do this, follow these steps:
 - a. Right-click an empty space in the LonMaker drawing and then select **Enable Monitoring** on the shortcut menu.



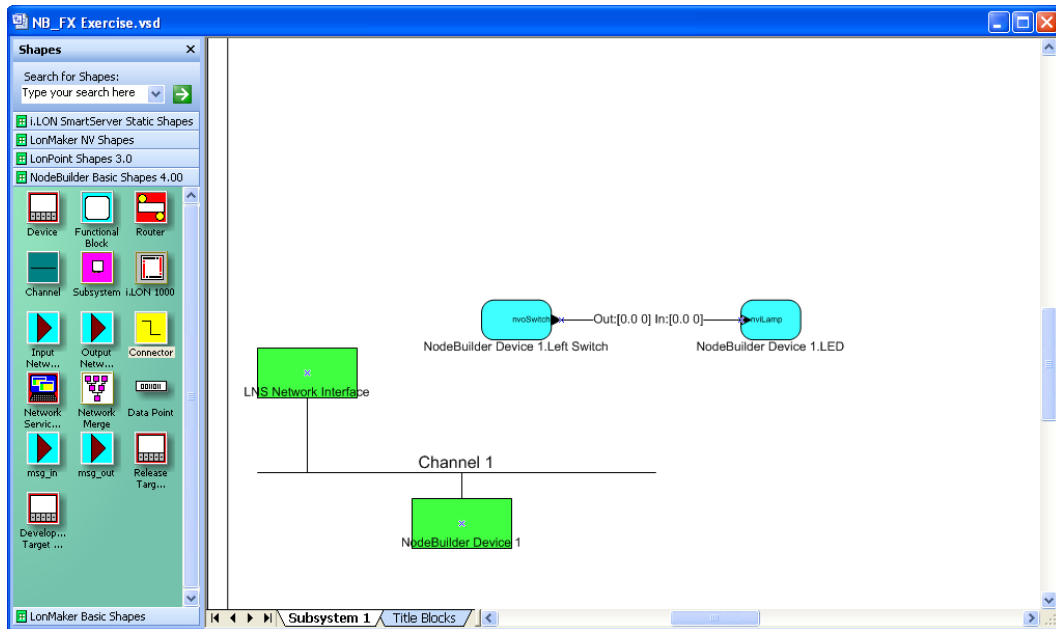
- b. Right-click the new **Connector** shape it and select **Monitor Input Value** to display the current value of the input network variable in the connection.



- c. Right-click the new **Connector** shape it and select **Monitor Output Value** to display the current value of the output network variable in the connection.



9. Toggle a hardware input to test the connection between the network variables change. Observe the hardware output and the current values of the network variables on the **Connector** shape change as you toggle the hardware input.



Debugging a Neuron C Application

This chapter describes how to use the NodeBuilder debugger to troubleshoot your Neuron C application.

Introduction to Debugging

You can use the NodeBuilder debugger within the NodeBuilder Project Manager to control and observe the behavior of your device application over a LONWORKS channel in order to debug it. The debugger lets you set breakpoints, monitor network 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 a single device or debug multiple devices simultaneously.

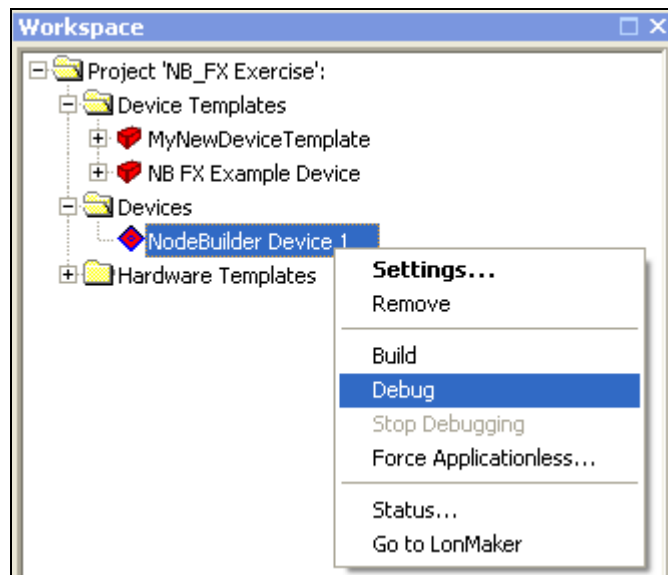
In addition to using the NodeBuilder debugger, you may also connect your device hardware to your computer using a RS-232 or USB interface, and output debugging and tracing information from your application. You can then use a terminal emulation program on your computer, such as Windows HyperTerminal, to view the output and perform runtime debugging.

Many of Echelon's evaluation boards include a RS-232 or USB interface to support application-level debugging. These evaluation boards consist of the 6000 FT EVB, 5000 FT EVB, 3150 FT EVB, 3150 PL EVB, 3120 FT EVB, and 3120 PL EVB. For more information on connecting the FT 6000 EVB to a computer for application-level debugging, see the *FT 6000 EVB Hardware Guide*. For more information for connecting the FT 5000 EVB, see the *FT 5000 EVB Hardware Guide*.

Starting the NodeBuilder Debugger

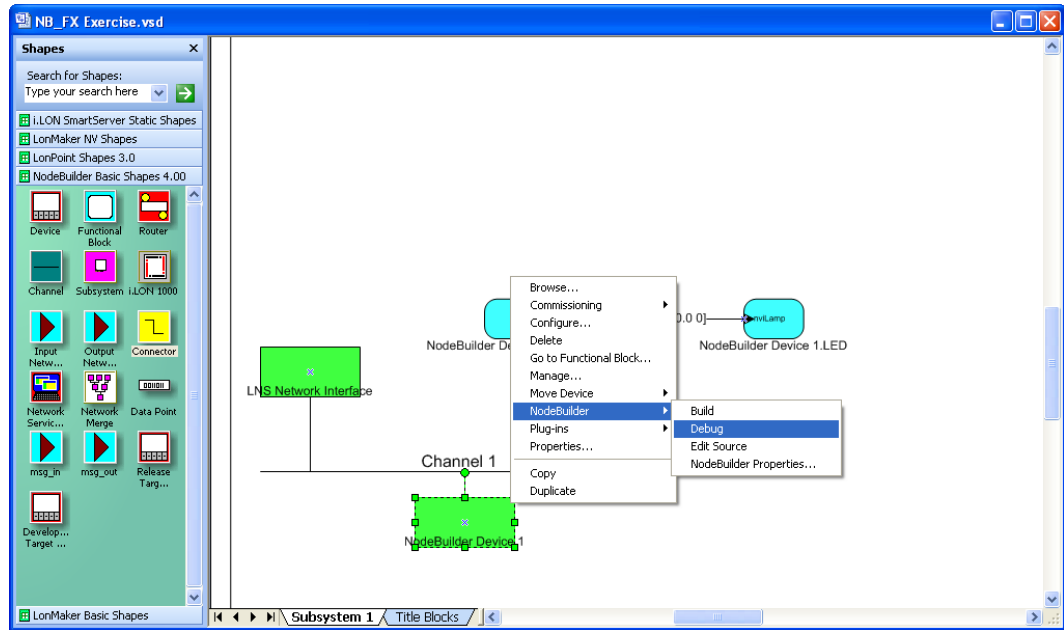
You can start the NodeBuilder debugger from the NodeBuilder Project Manager or from the IzoT Commissioning tool. To start the NodeBuilder debugger, follow these steps:

1. Start the NodeBuilder debugger from the NodeBuilder Project Manager or from the IzoT Commissioning tool.
 - To start the NodeBuilder debugger from the NodeBuilder Project Manager, right-click the device to be debugged under the **Devices** folder in the Project pane and then click **Debug** on the shortcut menu.

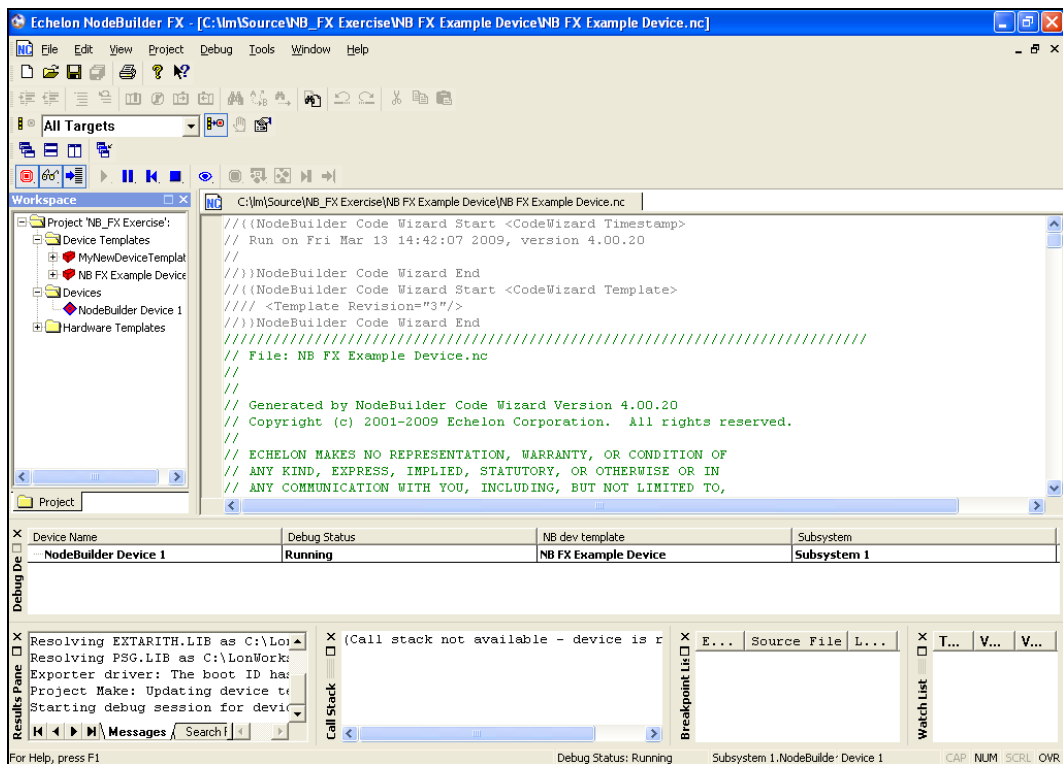


To debug multiple devices at the same time, click one device under the **Devices** folder in the Project pane, hold down CTRL and click the other devices to be debugged, right-click one of the selected devices, and then click **Debug** on the shortcut menu.

- To start the NodeBuilder debugger from the IzoT Commissioning tool, open the LonMaker drawing containing the device, right-click the device to be debugged, point to **NodeBuilder**, and then click **Debug** on the shortcut menu.



2. The NodeBuilder debugger opens.



3. The **Debug** menu appears on the NodeBuilder menu bar and four new panes open in the NodeBuilder project manager: the Debug Device Manager pane, the Breakpoint List pane, the Call Stack pane, and the Watch List pane. The following table describes each of these panes:

<i>Debug Device Manager</i>	Displays which devices are currently being debugged, and lets you pause and resume the application on each device. If at least one debug session is in progress, the status bar will indicate the device currently being debugged and its current state (Running, Halted, Reset, and so on). For more information, see <i>Using the Debug Device Manager</i> later in this chapter.
<i>Breakpoint List</i>	Displays all the breakpoints that have been set. For more information, see <i>Setting and Using Breakpoints</i> later in this chapter.
<i>Call Stack</i>	Displays a list of active function calls when the debugger is halted in application source code. You can this information to trace program execution logic. For more information, see <i>Using the Call Stack</i> later in this chapter.
<i>Watch List</i>	Displays all monitored network variables and their values. For more information, see <i>Using the Watch List Pane</i> later in this chapter.

Except for the Debug Device Manager pane, these panes are docked into the NodeBuilder Project Manager. The Debug Device Manager pane appears as a floating window by default, but you can dock it into the NodeBuilder Project Manager by right-clicking it and selecting the **Allow Docking** option on the shortcut menu. You can enable a pane to be moved and resized by right-clicking the pane and clearing the **Allow docking** option.

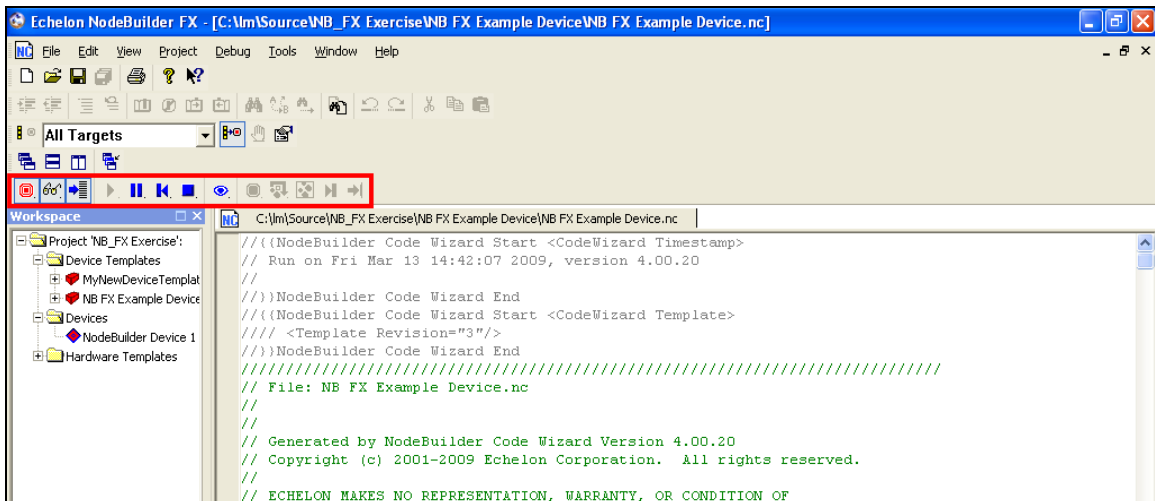
Notes: To stop debugging a single device, right-click the device and select **Stop Debugging** on the shortcut menu. Alternatively, you can click **Debug**, point to **Stop Debugging**, and select **Current Device** from the **Stop Debugging** menu while the appropriate device is displayed in the status bar of the **Debug Device Manager**. To stop debugging all devices, click **Debug**, point to **Stop Debugging**, and select **All Devices** from the **Stop Debugging** menu.

You can also stop debug devices from the Debug Device Manager pane. To stop debugging for one device, right-click the device in the Debug Device Manager pane and select **Stop** on the shortcut menu. To stop debugging for all devices, right-click one device and select **Stop All** on the shortcut menu.













If at least one debug session is in progress, the Results pane contains a **Debug Log** tab, which lists device errors. You can use this tab to dump trace information while debugging.

Using the Debugger Toolbar

When you start the NodeBuilder debugger, the Debugger toolbar opens. By default, the NodeBuilder debugger appears directly above the Project pane and below the Window toolbar in the NodeBuilder Project Manager, but you can move it anywhere.




The following table describes each of the buttons in the Debugger toolbar.

	View Breakpoint List	Toggles the breakpoint list pane. See <i>Setting and Using Breakpoints</i> later in this chapter for more information.
	View Watch List	Toggles the watch list pane. See <i>Using the Watch List Pane</i> later in this chapter for more information.
	View Call Stack	Toggles the call stack pane. See <i>Using the Call Stack</i> later in this chapter for more information.
	Resume	Resumes execution of a halted application. See <i>Stopping an Application</i> later in this chapter for more information.
	Halt	Halts the application running on the current device. See <i>Stopping an Application</i> later in this chapter for more information.
	Reset	Resets the current device.
	Stop	Stops debugging the current device.
	Watch Variable	Opens the Add to Watch List dialog. See <i>Using the Watch List Pane</i> later in this chapter for more information.
	Toggle Breakpoint	Toggles whether the current line of code has a breakpoint. See <i>Setting and Using Breakpoints</i> later in this chapter for more information.
	Step Over	Executes the current line of the application. If the current line contains a function, the function will execute in its entirety. See <i>Stepping Through Applications</i> later in this chapter for more information.
	Step Into	Executes the current line of the application. If the current line contains a function, the application will halt at the first line of the function. See <i>Stepping Through Applications</i> later in this chapter for more information.
	Run to Cursor	Sets an implicit breakpoint at the line that the cursor is on. The application resumes if it is currently halted and continues to execute if it is already running. The application will halt when it reaches this implicit breakpoint. In addition, the breakpoint will be cleared once it is encountered.
	Current Instruction Source Code	When the application is halted, jumps to the line of code on which the application has halted.

Stopping an Application



You can stop an application while it is running in debug mode in three ways: halting the application, running to the cursor, and setting breakpoints.


Once you stop an application, you can step through the application one command at a time (see *Stepping Through Applications* for more information), observe the values of variables in the watch list (see *Using the Watch List Pane* for more information), and observe the condition of the call stack (see *Using the Call Stack* for more information).

To resume execution of an application that you have halted, either click the resume button () on the Debugger toolbar, type <F5>, or select **Go** from the **Debug** menu. The application will continue running until it hits another breakpoint (or the same one again). You can also move your cursor and click the run to cursor button to have the application resume execution until it gets to the line containing the cursor.


The following sections describe the three methods for stopping a device application running in debug mode.

Halting an Application

You can stop an application while it is running in debug mode by clicking the halt button () on the Debugger toolbar. Alternatively, you can click **Debug**, point to **Halt**, and select **Current Device** or **All Devices**. If the device halts in application code, the editing pane displays 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 a “Call stack not available” message appears in the Call Stack pane.



To resume execution of an application that has halted, click the resume button () on the Debugger toolbar, click **Debug** and then click **Go**, or press F5. The application will continue running until it hits another breakpoint (or the same one again). You can also move your cursor and click the run to cursor button to have the application resume execution until it gets to the line containing the cursor.

Running to the Cursor


You can make an application run to a cursor location. To do this, place the cursor in the line where the application is to be halted, and then either click the run to cursor button () on the Debugger toolbar or click **Debug** and then click **Run to Cursor**. The application will automatically halt when it reaches the cursor. If you move the cursor, you will need to set this option again to re-enable this behavior. Note that **Run to cursor** breakpoints will be cleared after the first time that they are encountered.

Setting and Using Breakpoints


You can use breakpoints to set lines in your source code where the application will stop running so you can check variable values, device hardware status, and so on. This lets you identify the line of code 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 click the Toggle Breakpoint button () on the Debugger toolbar. Alternatively, you can either right-click the line of code and select **Toggle Breakpoint** on the shortcut menu; click **Debug**, point to **Breakpoints**, and then click **Toggle Current Line**; or press F9. When you set a breakpoint, the breakpoint 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 you cannot set breakpoints on include comments, **when()** clauses, pre-processor directives, and variable declarations.


When the application reaches a line with a breakpoint, the application halts and an arrow icon appears on top of the breakpoint icon () to the left of the line of code.


Notes:


- For 5000 or 6000 Series chips, you cannot set breakpoints in *interrupt*-tasks or set breakpoints in functions that are called from *interrupt*-tasks. If you set a breakpoint in an *interrupt*-task or in a function called from an *interrupt*-task and interrupts are enabled [with the **interrupt_control()** function], the debug target will report a system error, reset, and then go into the soft-offline state. If you re-enable interrupts in the reset clause before the device can go offline, the NodeBuilder debugger might lose communication with the device and therefore need to set the device applicationless
- Do not edit source files when running an application in debug mode because the source code will no longer reflect the active image in the debugger, and breakpoints may lose synchronization. If you believe breakpoints have lost synchronization, you can stop the debugging session, recompile and load the device application, and then restart the debugging session.
- 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 for it to reach your breakpoint.

Stepping Through Applications

You can step through the code in your application one line at a time after you halt the application. You can *step into* or *step over* a line of code. 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. For more information on stopping a device application running in debug mode, see the previous section, *Stopping an Application*.

When you halt an application, an arrow () appears in the left margin at the line of code where the application was stopped. When you step to the next command, the arrow moves to indicate the current line of source code where the application has been stopped.

To step over the current line of the application, you can either click the step over button () on the Debugger toolbar; click **Debug** and then click **Step Over**; or press <F10>.

To step into the current line of the application, you can either click the step into button () on the Debugger toolbar; click **Debug** and then click **Step Into**; or press <F11>.

Debugging Interrupts for 5000 or 6000 Series chips

If you are debugging a target device that uses a 5000 or 6000 Series chip, you cannot set breakpoints in *interrupt*-tasks or set breakpoints in functions that are called from *interrupt*-tasks. If you set a breakpoint in an *interrupt*-tasks or in a function called from an *interrupt*-task and interrupts are enabled [with the **interrupt_control()** function], the debug target will report a system error, reset, and then go into the soft-offline state. If you re-enable interrupts in the reset clause before the device can go offline, the NodeBuilder debugger might lose communication with the device and therefore need to set the device applicationless.

Using Statement Expansion

The 3100 Series chips use a 2-byte breakpoint instruction for debugging. To support breakpoints in all suitable locations, the compiler must expand some statements to a 2-byte machine instruction (by inserting a benign no-operation performed [NOP] instruction).

The 5000 and 6000 Series chips support single-byte breakpoint instructions for debugging, which enables the debug image for a 5000 or 6000 Series chip to be smaller than that of a 3100 Series chip. To support single-byte breakpoint instructions, no padding is necessary, and the compiler does not need to expand statements.

By default, the statement expansion feature is enabled to support the debugging of 3100 Series devices. If you are debugging a 5000 or 6000 Series device, you can disable the statement expansion feature to reduce the size of the debug image. To do this, right click the target, click **Settings** on the shortcut menu, then select the **Compiler** tab in the **NodeBuilder Device Template Target Properties** dialog. In the **Debug Kernel Options** box, clear the **Expand Statements** check box, and then click **OK**.

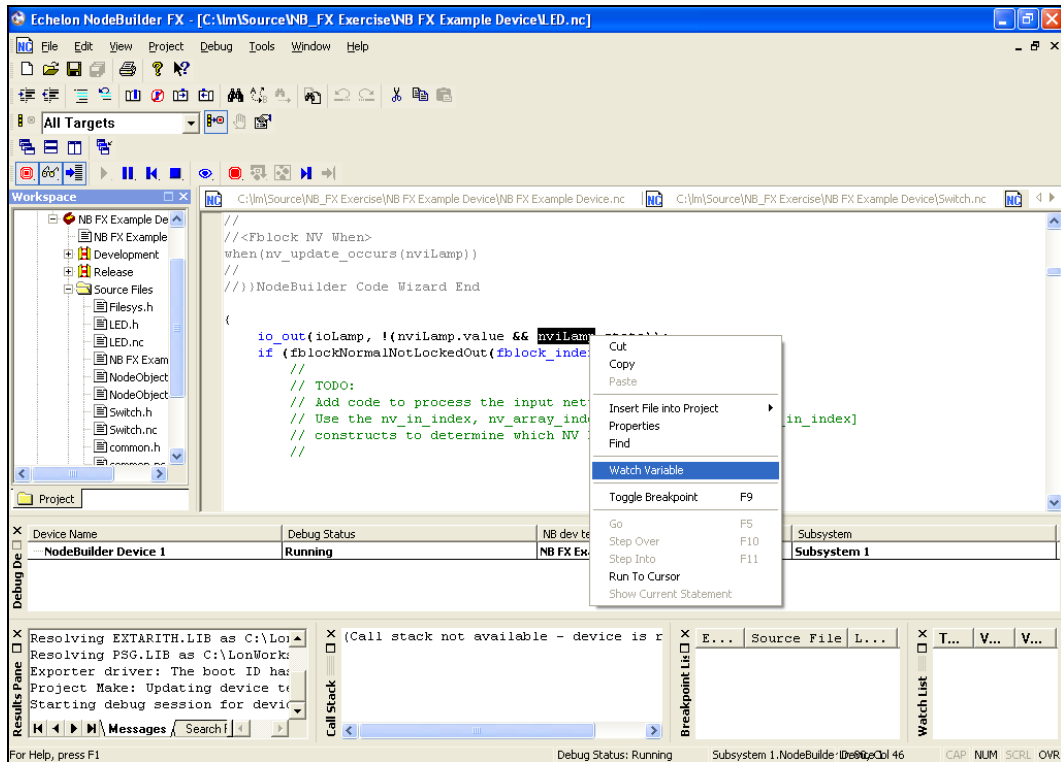
Using the Watch List Pane

You can add variables, network variables, and configuration properties in your device application to the Watch List and then monitor their current values in the Watch List pane.

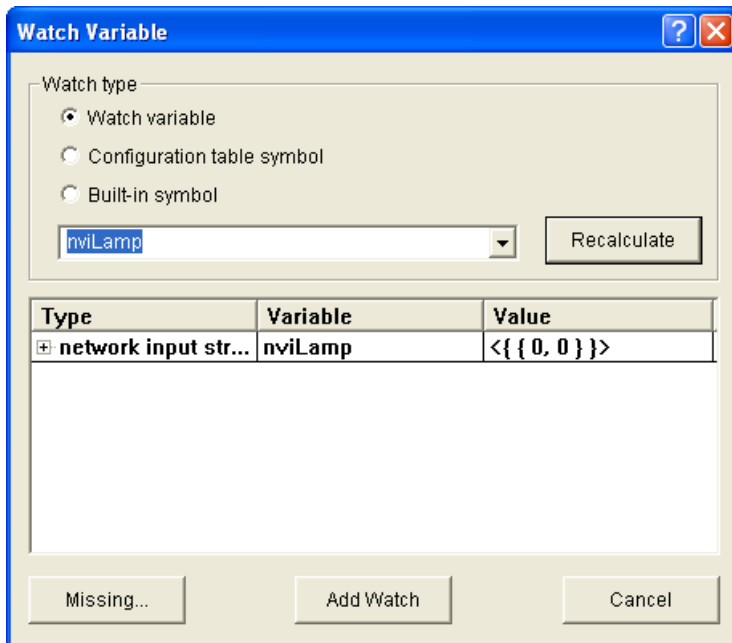
You can monitor local variables when the application is halted in a context where the variables are available. You can monitor global variables and network variables while the application is running. You can also modify the values of global variables and input network variables while the application is running. You can only modify output network variables when the application is halted in the debugger. You cannot monitor the **msg_in**, **msg_out**, **resp_in**, and **resp_out** built-in variables from the debugger.

To add a variable, network variable, or configuration property to the watch list and monitor its value in the Watch List pane, follow these steps:

1. Right-click a variable name or statement in the source code and then click **Watch Variable** on the shortcut menu.



2. The **Watch Variable** dialog opens.



3. If you right-clicked a variable name, the selected variable appears in the **Watch Type** box. You can proceed to step 5.

4. If you right-clicked a statement, the drop-down list in the **Watch Type** box is empty and you need to select one of the following types of variables to watch :

- **Watch variable.** Enter a network variable using its global network variable name or using its functional block member name (for example, using the scope operator “::”). Similarly, you can enter a configuration network variable using its global network variable name or using the corresponding configuration property syntax. See the *Neuron C Programmer’s Guide* and *Neuron C Reference Guide* for more information on referencing configuration network variables (CPNVs). To watch a configuration property that is implemented within a configuration file (file CP), 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:

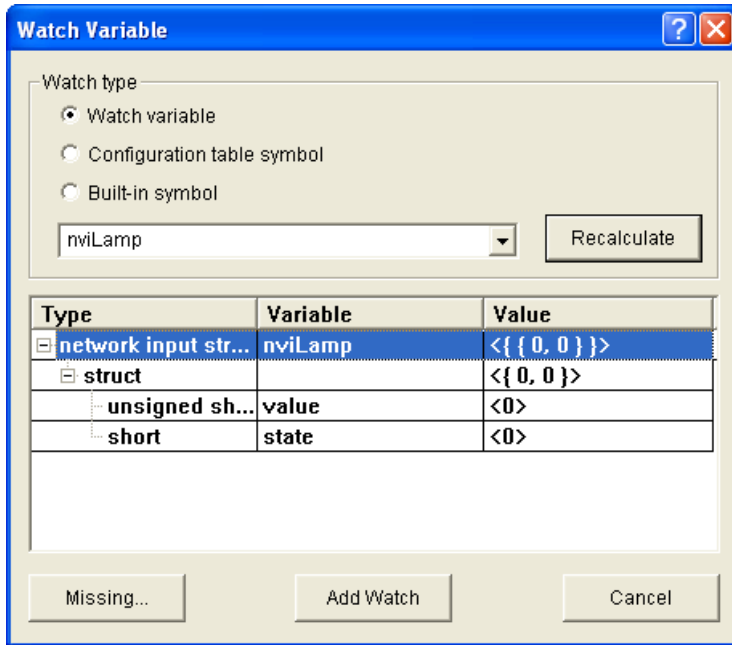
- You cannot watch an entire configuration network variable array. You must specify a single element to be watched using the *<CPindex>* field.
- You can only watch an entire **cp_family** array. In this case, do not specify a *<CPindex>*; the entire array will be displayed in a tree structure in the 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.** Select a configuration table value to be watched from the list of all available configuration table symbols.
- **Built-in Symbol.** Select a built-in symbol value to be watched from the list of all available system symbols. You can 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 the device application.

You can 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 dialog pops up with the message **Symbol Not Found**.

5. Click **Add Watch** to add the selected variable to the Watch List pane. If the variable is a structure or union, you can expand the variable and then the data type under the **Type** column to display all the fields of the structure. For each variable or field in a structure, the watch list displays the type, variable name, and value.



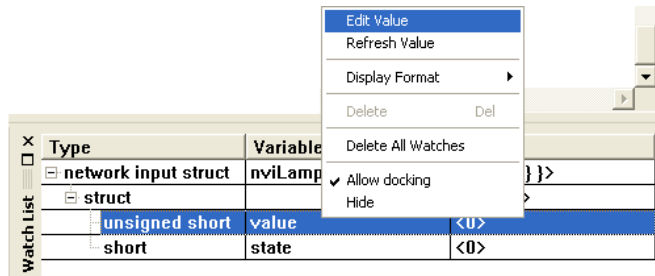
Scalar network variables contain a single field that contains their value. If the variable does not exist, a **Symbol Not Found** dialog opens.

6. Optionally, you can edit the value of a variable or a field in a structure in the Watch List pane.

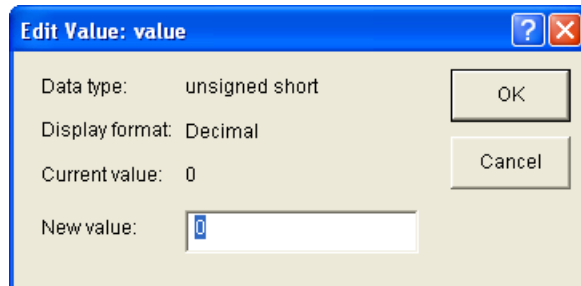
- a. To edit the value of a scalar variable, double-click anywhere in the row containing the variable or right-click the variable and then click **Edit Value** on the shortcut menu.

To edit the value of an enumeration, expand the variable, double-click anywhere in the row containing the field, or right-click the field and then click **Edit Value** on the shortcut menu.

To edit the value of a structure, expand the variable and expand the type, double-click anywhere in the row containing the field, or right-click the field and then click **Edit Value** on the shortcut menu.



- b. The **Edit Value** dialog opens.



- c. Enter the new value for the variable and then click **OK**. If you are editing the value of an enumerated type, select an enumeration from the list or click **Enter in Decimal** or **Enter in Hex** and then enter the desired index of the enumeration.
- d. Click **OK** to save the value.

Notes:

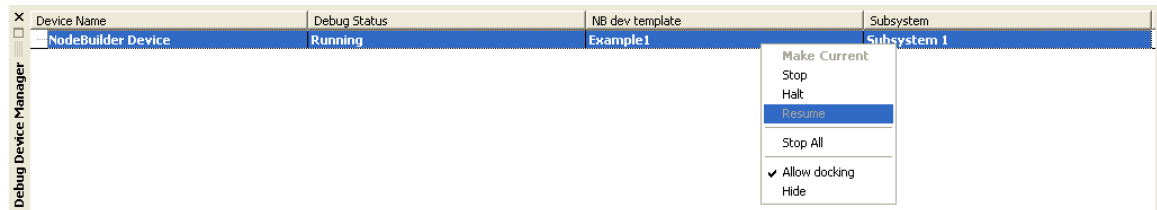
- To remove a variable from the watch list, right-click the variable in the Watch List pane and click **Delete** on the shortcut menu. To remove all variables from the Watch List pane, right-click anywhere in the Watch List pane and click **Delete all Watches** on the shortcut menu.
- You can display the values in the Watch List pane in either decimal or hexadecimal format. You can set the default format by clicking **Project**, clicking **Settings**, clicking the **Debugger** tab in the **NodeBuilder Project Properties** dialog, and then selecting the desired default format in the **Default Display Radix** option. You can override the default setting for individual entries in the Watch List pane by right-clicking in the Watch List pane, pointing to **Format**, and then selecting the desired format on the shortcut menu. Individual entries within each of the variables can also be displayed using string, signed 32-bit, and floating point format where applicable.

Using the Call Stack Pane

The Call Stack pane displays the functions that have been called when the application is halted. If your device application is halted within a function, this lets you determine if that function was called from within another function, and if so, which one. If the device application is within multiple functions, the most recently called one will be on the top of the call stack list. You can double-click any entry on the call stack list to be taken to the line of the function call.

Using the Debug Device Manager Pane

The Debug Device Manager pane displays the status (running, halted, or reset) of all devices that are currently being debugged. The **Reset** status is only displayed if the device is reset while halted. You can right-click a device in the Debug Device Manager pane and select one of the following options on the shortcut menu:



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 and removes the device from the Debug Device Manager pane. To restart debugging for this device, right-click the device under the **Devices** folder in the Project pane and click **Debug** on the shortcut menu.

Halt

Halts the application in the selected device. For more information about stopping and starting device applications, see *Stopping an Application* earlier in this chapter.

Resume

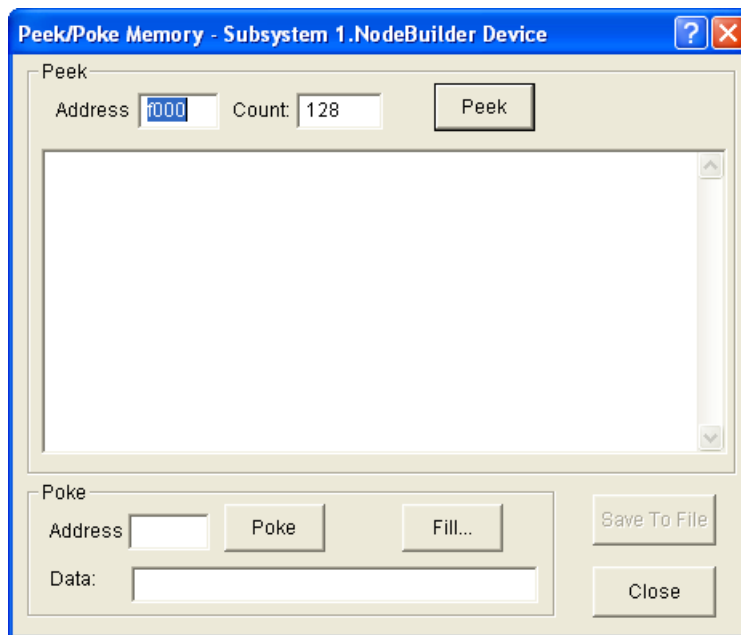
Resumes running a halted application in the current device.

<i>Stop All</i>	Stops debugging all devices, removes all the devices from the Debug Device Manager pane, and closes the NodeBuilder debugger. To restart debugging for a device, right-click the device under the Devices folder in the Project pane and click Debug on the shortcut menu.
<i>Allow Docking</i>	Docks the Debug Device Manager pane into the NodeBuilder Project Manager. The Debug Device Manager pane appears as a floating window that you can move and resize by default.
<i>Hide</i>	Select this option to hide the debug manager window. To view the debug manager window again, click View , select Debug Windows , and then select Debug Device Manager .

Peeking and Poking Memory

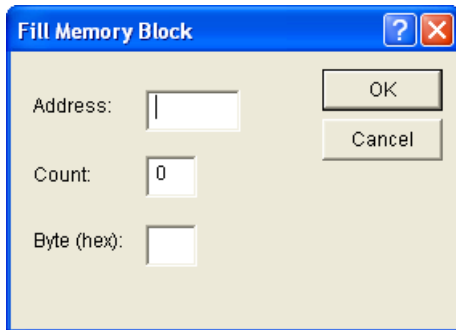
You can use the NodeBuilder debugger to view (peek) and modify (poke) the memory contents of the device being debugged. You must be careful when modifying memory contents because you can render a device inoperable by writing to an inappropriate memory location. To view and modify memory, follow these steps:

1. Click **Debug** and then click **Peek/Poke Memory**. The **Peek/Poke Memory** dialog opens:



2. To inspect memory, enter the **Address** and **Count** properties in the **Peek** box at the top of the dialog, and then click **Peek**. The **Peek** box displays the number of bytes in the **Count** property starting at the address in the **Address** property. The data is displayed in both hexadecimal and ASCII format. You can save the results of the peek by clicking **Save to File**.
3. To modify memory, enter the **Address** property and enter the **Data** property (in hexadecimal format) in the **Poke** box at the bottom of the dialog, and then click **Poke**. The data in the **Data** property is written to the device starting at the address in the **Address** property. To write multiple bytes of data, separate each byte with spaces, commas, tabs, newlines, hyphens, or colons.

You can fill multiple bytes of memory with the same value. To do this, click **Fill**. The **Fill Memory Block** dialog opens.



In the **Address** field, enter the address to start writing in. In the **Count** field, enter the number of bytes to write. In the **Byte** field, enter a two digit hexadecimal value. Click **OK** to write the value in **Byte** a number of times equal to **Count** starting at the address in **Address**. You are returned to the **Peek/Poke Memory** dialog.

4. Click **Close** to return to the NodeBuilder debugger.

Executing Code in Development Targets Only

You can designate code for execution in development targets only. This lets you build simultaneously to development and release targets and include test code that executes on the development targets only. To have one or more lines of code execute on development targets only, put the statement **#ifdef** **_DEBUG** before the code, and the statement **#endif** after the statement. The following code sample demonstrates how to do this:

```
#ifdef _DEBUG
    //Test code.  Executes on development targets only
    <test code>
#endif
```

You can not define network variables or configuration properties or make any changes to the external interface inside the **#ifdef** clause. This is because both release and development targets have the same program ID.

The **_DEBUG** macro is predefined for development targets, but not for the release targets. To edit the predefined macros for a development target, right-click **Development** folder in the Project pane, and then click **Settings** on the shortcut menu. The **NodeBuilder Device Template Target Properties** dialog opens with the **Compiler** tab selected. Enter a symbol in the **Defines:** property, which can be tested using the **ifdef** directive.

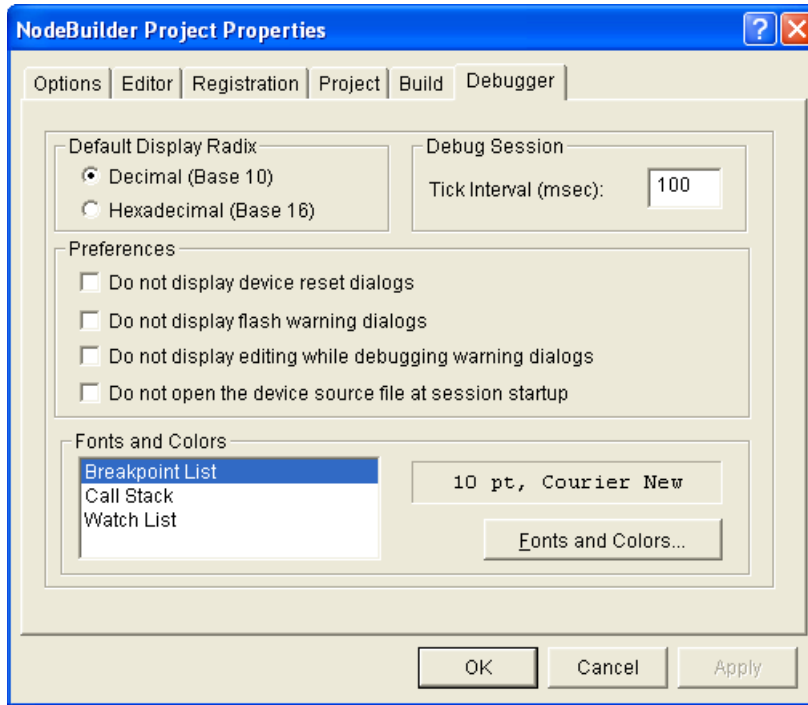
Using the Debug Error Log Tab

When you start a debugging session, the **Debug Error Log** tab is added to the results pane. This tab provides rudimentary tracing capabilities and debugging timing-related problems in the debugger when a debug session is in progress. You can use the Neuron C **error_log()** function to output specific error codes to the debug tab in response to specific events. See the *Neuron C Programmer's Guide* and *Neuron C Reference Guide* for more information about the **error_log()** function.

Setting Debugger Options

You can set options for the NodeBuilder debugger following these steps:

1. Click **Project**, click **Settings**, and then click the **Debugger** tab in the **NodeBuilder Project Properties** dialog. Alternatively, you can right-click the **Project** folder at the top of the Project pane and then click **Settings** on the shortcut menu.



2. You can set the following options:

Default Display Radix Specifies the default format in which data is displayed in the Watch List pane. You can choose to monitor data in the Watch List pane in **Decimal** or **Hexadecimal** format.

Tick Interval Specifies how frequently (in milliseconds) the debugger processes incoming debug messages from the device. The default interval is **100** ms.

Preferences

Do not Display Device Reset Dialogs Suppresses warnings 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. This check box is cleared by default.

Do not Display Flash Warning Dialogs Suppresses warnings when you set a breakpoint in application code that resides in flash memory. This check box is cleared by default.

Do not Display Editing while Debugging Warning Dialogs Suppresses warnings when you edit code while in a debugging session. Editing code in a debugging session can cause unpredictable debugger behavior and is not recommended. This check box is cleared by default.

Do not Open the Device Source File at Session Startup Prevents the source file (<template name>.nc) from 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.

Fonts and Colors

Specifies the font, font size, and color used for text in the Breakpoint List, Call Stack, and Watch List panes. To change the font and color used in a pane, click the pane and then click **Fonts and Colors**.

3. Click **OK** to save the settings.

Appendix A

Using the Command Line Project Make Facility

This appendix describes how to use the command line project make facility with the project make command.

Using the NodeBuilder Command Line Project Make Facility

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 invoke the NodeBuilder Command Line Project Make Utility and build a project, open a Windows command prompt and enter 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.

You can use the following command line switches:

- | | |
|---|--|
| -? <cmd>
(or --help <cmd>) | Displays usage help for the <cmd> command. Providing no command at all also displays the list of the available commands and a brief usage hint. |
| -@ <file> | Uses <file> as input to the project make. This file can contain command line switches to be used by the project make facility. You can set the Generate build script option in the Build tab of the NodeBuilder Project Properties dialog to have the IzoT 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 <device template name>.cmd. If multiple targets are built, a separate command file will be generated for each. |
| --always (or -a) | Causes NodeBuilder 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. |
| -b <nbd> | Indicates that a build operation will be made on the selected NodeBuilder device template (.Nbd extension) for the target specified by the -t command switch. The device template will be compiled, linked, and exported. You can only specify a single device template per make command. |
| -c <nbd> | Specifies a NodeBuilder device template file (.Nbd extension) to be compiled. You can only specify a single device template per make command. |

--defloc <dir>	<p>Specifies a directory to search for the default command file. The default command file for the project make facility must be named lonpmk32.def. If a default directory that does not contain this file is specified, the command will fail silently. If no default directory is specified, the current directory will be searched for lonpmk32.def.</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 --file 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 file called lonpmk.32.log:</p> <pre>--mkscript c:\temp\lonpmk32.log</pre>
--mkscript <file>	Generates a file that contains all the command switches and arguments that are used in this invocation of the project make command. This file can be used (for example) as a log of the build or to recreate the build on another computer.
--n	Reconfirms build status after build completion.
--nadep <nadep>	Specifies the location of the assembler dependency file. By default, this file is located in the IM subdirectory of the target folder (for example, Development or Release).
--ncdep <ncdep>	Specifies the location of the compiler dependency file. By default, this file is located in the IM subdirectory of the target folder (for example, Development or Release).
--nldep <nldep>	Specifies the location of the linker dependency file. By default, this file is located in the IM subdirectory of the target folder (for example, Development or Release).
--nodefaults	Disables processing of default command files (see the description of the --defloc command switch for more information).
--nxdep <nxdep>	Specifies the location of the exporter dependency file. By default, this file is located in the IM subdirectory of the target folder (for example, Development or Release).
-p <project file>	Specifies the NodeBuilder project that contains the NodeBuilder device template to be built. NodeBuilder project files have the .NbPrj extension.
-q <nbd>	Indicates that a query operation will be performed on the specified NodeBuilder device template for the target specified by the -t command switch. This command will indicate whether the target needs to be built.
--silent	Suppresses banner message display.
-t <Development Release>	Specifies on which target the build, clean, or query operation will be invoked.
-v	Causes the project make facility to be run in verbose mode.

-x <nbdt> Indicates that a clean operation will be performed on the specified NodeBuilder device template for the target specified by the **-t** command switch. A clean operation removes all files and folders produced by a build.

The following example demonstrates a minimal command line invocation of the Project Make Facility:

```
PMK -p=Test.nbprj -b=MyDevice.nbdt -t=Development
```

This command performs a conditional build on the development target that is contained within the device template **MyDevice**, which is part of the project **Test**.

For more information about the NodeBuilder and Neuron C command line tools, see Appendix A of the *Neuron C Programmer's Guide*.

Appendix B

Using Source Control With a NodeBuilder Project

This appendix describes how to manage a NodeBuilder project using a source control application.

Using Source Control with a NodeBuilder Project

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. 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:

<i><LonWorks></i>	The LONWORKS folder, which is typically C:\LonWorks.
<i><NbDtFolder></i>	The folder that contains the NodeBuilder device template file. NodeBuilder device template files use the .NbDt file extension. By default <i><NbDtFolder></i> is a subfolder of the NodeBuilder project folder.
<i><mnfr></i>	Your manufacturer name (for example, ACME Corporation).
<i><lang></i>	A valid device resource file language identifier such as ENU, GER, FRA, and so on.
<i><project></i>	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:

NodeBuilder Project Files (.NbPrj and .NbOpt)	<p>The file <i><project>.NbPrj</i> is the NodeBuilder project file. It holds pointers to all the NodeBuilder device templates and any user-defined hardware templates required for a build. This file would be checked in for convenience.</p> <p>The file <i><project>.NbOpt</i> 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. This file would not normally be checked in. The options in this file are a matter of personal preference, and do not effect device file set.</p> <p>Although NodeBuilder project folders and all their subfolders can be moved and re-opened from the new location with the Open Project dialog, moving a project folder can cause compilation errors due to absolute file references in use, or due to device resource files being moved. Try to use relative references rather than absolute file name paths whenever possible.</p> <p>To improve project-to-project compatibility, do not use the Include Search Path option in the Project tab of the NodeBuilder Project Properties dialog.</p> <p>The default location for project files is: C:\lm\source\<project></p>
NodeBuilder Device Template Files (.NbDt)	<p>These files hold most of the data required to build a device file set and NodeBuilder device template.</p> <p>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 <i><NbDtFolder></i>.</p>

Neuron C Source Files
(.nc, .c, and .h)

The main source file, *<Device Template>.nc*, is stored in the **C:\Lm\Source\<Project>\<Device Template>** folder. This file and any files included with the **#include** directive must be checked in.

Standard header files are stored in the **C:\LonWorks\NeuronC\Include** folder. These files should never be edited because future installs will overwrite modified files and changes would be lost. Check these files 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

Includes user-defined libraries, build script files, and other user-defined files

NodeBuilder Hardware Template Files

Describe the hardware that will be used to host the application. This data includes Neuron Chip model, clock rate, memory map, and so on.

(.NbHwt)

Standard hardware templates are stored in the **C:\LonWorks\NodeBuilder\Templates\Hardware\Standard** folder.

These files should never be edited because future updates to the IzoT 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 the **C:\LM\Source\Templates\Hardware\User** folder.

Resource Files

(.TYP, .FMT, .FPT, and
.<lang>)

These resource files comprise resource file sets, which hold definitions of functional profiles, network variable types, and configuration property types. Resource file sets are generated with the NodeBuilder Resource Editor. For more information on creating and editing resource file sets, see the *NodeBuilder Resource Editor User's Guide*.

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 **C:\LonWorks\Types** folder and enter the following command:

```
mkcat -a<ResourceFolderPath>
```

Note: Do not check-in the device resource file catalog (**LDRF.cat** by default) because it might contain references to device resource files that are unique to each computer.

Appendix C

Glossary

This appendix provides definitions for many terms commonly used with NodeBuilder device development.

3100 Series Chip

The term used to collectively refer to all previous-generation Neuron chips, including the 3150 and 3120 Neuron chips; the 3150 and 3120 FT Smart Transceivers; and the 3170, 3150, and 3120 PL Smart Transceivers.

5000 Series Chip

The term used to collectively refer to the Neuron 5000 Processor and FT 5000 Smart Transceiver.

6000 Series Chip

The term used to collectively refer to the Neuron 6000 Processor and FT 6000 Smart Transceiver.

Application Device

A LONWORKS device that runs a LonTalk Application (OSI Layer 7). The application may run on a Neuron Chip, in which case the device is called a “Neuron hosted” device.

Application Image

Device firmware that consists of the object code generated by the Neuron C compiler from the user’s application program and other application-specific parameters, including the following:

- Network variable fixed and self-identification data
- Network variable external interface data (XIF file)
- Program ID string
- Optional self-identification and self-documentation data
- Number of address table entries
- Number of domain table entries
- Number and size of network buffers
- Number and size of application buffers
- Number of receive transaction records
- Input clock speed of target Neuron Chip
- Transceiver type and bit rate

Application Program

The software code in a LONWORKS device that defines how it functions. The application program, also referred to as the application or the application layer, may be in the device when you purchase it, or you may load it into the device from application image files (.APB, .NDL, and .NXE extensions) using the IzoT Commissioning tool. The application program interfaces with the LonTalk firmware to communicate over the network. It may reside completely in the Neuron Chip, or it may reside on an attached host processor (in a host-based device).

Backup

A .zip file containing a saved version of one to all of the following components: a LonMaker drawing, LNS network database, and NodeBuilder project. Backup files are used to protect against accidental file corruption or hardware failure, or to copy a LonMaker network design or NodeBuilder project from one computer to another.

Binding

Process of connecting network variables. Binding creates logical connections (virtual wires) between LONWORKS devices. Connections define the data that devices share with one another. Tables containing binding information are stored in the Neuron Chip’s EEPROM, and may be updated by the IzoT Commissioning tool.

Changeable-Type Network Variable

A network variable that has a type and length that can be changed to that of another network variable type of equal or smaller size. You can use changeable-type network variables to implement generic functional blocks that work with different types of inputs and outputs.

Channel

The physical media between devices upon which the devices communicate. The LonTalk protocol is media independent; therefore, numerous types of media can be used for channels: twisted pair, power line, fiber optics, IP, and RF, and other types.

Clock Multiplier

For 5000 and 6000 Series chips, you can select the frequency at which the Neuron Chip runs to modify the internal system clock speed. You can select multipliers of ½, 1, 2, 4, and 8 to adjust the internal system clock speed from 5 MHz to 80 MHz (based on a crystal running at 10 MHz).

Commissioning

The process in which the IzoT Commissioning tool downloads network and application configuration data into a physical device. For devices whose application programs are not contained in ROM, the IzoT Commissioning tool also downloads the application program into non-volatile RAM in the device. Devices are usually either commissioned and tested one at a time, or commissioned and then brought online and tested incrementally.

Code Wizard Template

Defines the general infrastructure and layout of a Neuron C application generated with the NodeBuilder Code Wizard. Code templates supply many utility functions for managing device and functional block status, which you can use in your application, as needed.

Configuration Properties (CPs)

Configuration properties define the behavior of an application device by determining the manner in which data is manipulated and when data it is transmitted. Configuration properties can be applied at the device, functional block, or network variable level. Configuration properties determine the functions to be performed on the values stored in network variables. For example, a configuration property may specify a minimum change that must occur on a physical input to a device before the corresponding output network variable is updated.

Configured

A device state where the device has both an application image and a network image. This indicates that the device is ready for network operation.

Connector Shape

A single connector used to connect a pair of network variables within the same subsystem.

Control Network Protocol (CNP)

Echelon's implementation of the ISO/IEC 14908-1 standard. The CNP provides a standard method for devices on a LonWorks network to exchange data. The CNP defines the format of the messages being transmitted between devices, and it 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.

Data Point

A network variable, configuration property, or functional block state (enabled or in override) that the IzoT Commissioning tool can monitor and/or control.

Data Point Shape

A shape in the LonMaker Basic Stencil of the IzoT Commissioning tool that you can use to monitor and control the values of network variables and configuration properties, and the states of functional blocks (enabled or in override).

Device

A device that communicates on a LONWORKS network. A device may be an application device, network service device, or a router. Devices are sometimes referred to as nodes in LONWORKS documentation.

Device Interface

The logical interface to a device. A device's interface specifies the number and types of functional blocks; number, types, directions, and connection attributes of network variables; and the number of message tags. The program ID field is used as the key to identify each external interface. Each program ID uniquely defines the static portion of the interface. However, two devices with identical static portions may differ if dynamic network variables are added or removed, or if the types of changeable network variables are changed. Thus it is possible to have devices with the same program ID but different external interfaces.

Device Interface File (XIF)

A file that documents a device's interface with a network. The file can be a text file (.XIF extension), or it can be a binary file (.XFB extension).

Device-Specific Configuration Property

A configuration property that has values that can be modified independent of the network database. Changes made to a device-specific configuration property are not updated in the network database.

Device Template

A device template contains all the attributes of a given device type, such as its functional blocks, network variables, and configuration properties. You can create a device template by importing a device interface (XIF) file supplied by the device manufacturer, or by uploading the device interface definition from the physical device. A device template is identified by its name and its program ID. Both must be unique within a network—you cannot have two device templates with the same name or the same program ID in a single network.

Download

An installation process in which data, such as the application program, network configuration, and/or application configuration, is transferred over the network into a device.

Free Topology

A connection scheme for the communication bus that removes traditional transmission line restrictions of trunks and drops of specified lengths and at specified distances, and terminations at both ends. Free topology allows wire to be strung from any point to any other, in bus, daisy chained, star, ring, or loop topologies, or combinations thereof. It only requires one termination anywhere in the network. This can reduce the cost of wiring by a factor of two or more.

FT 5000 EVB

A LONWORKS evaluation board that uses Echelon's FT 5000 Smart Transceiver. It features a compact design that includes the following I/O devices that you can use to develop prototype devices and run the FT 5000 EVB examples: 4 x 20 character LCD display, 4-way joystick with center push button, 2 push-button inputs, 2 LED outputs, digital light sensor, and digital temperature sensor.

FT 5000 Smart Transceiver

A chip that integrates a Neuron 5000 processor core and a TP/FT-10 transceiver. See *Neuron 5000 Processor* for more information about the key features of the Neuron 5000 processor.

FT 6000 EVB

A LONWORKS evaluation board that uses Echelon's FT 6000 Smart Transceiver. It features a compact design that includes the following I/O devices that you can use to develop prototype devices and run the FT 6000 EVB examples: 4 x 20 character LCD display, 4-way joystick with center push button, 2 push-button inputs, 2 LED outputs, digital light sensor, and digital temperature sensor.

FT 6000 Smart Transceiver

A chip that integrates a Neuron 6000 processor core and a TP/FT-10 transceiver.

FT/PL 3150 EVB

A LONWORKS evaluation board that uses Echelon's FT or PL 3150 Smart Transceiver. It is connected to a MiniGizmo board that includes eight push buttons, eight LEDs, a temperature sensor, and a piezo buzzer. In a managed network, you can bind compatible network variables in applications running on the FT 3150 EVB and FT 5000 EVBs. In a self-installed network, you can use the ISI protocol to connect the FT 3150 EVB running the *MGSwitch*, *MGLight*, or *MGDemo* applications to an FT 5000 EVB running the *NcSimpleIsiExample* or *NcMultiSensorExample* applications.

FT/PL 3120 EVB

A LONWORKS evaluation board that uses Echelon's FT or PL 3120 Smart Transceiver. It is connected to a MiniGizmo board that includes eight push buttons, eight LEDs, a temperature sensor, and a piezo buzzer. In a managed network, you can bind compatible network variables in applications running on the FT 3120 EVB and FT 5000 and 6000 EVBs. In a self-installed network, you can use the ISI protocol to connect the FT 3120 EVB running the *MGSwitch* or *MGLight* applications to an FT 5000 or FT 6000 EVB running the *NcSimpleIsiExample* or *NcMultiSensorExample* applications.

Functional Block (FB)

A collection of network variables, configuration properties, and associated behavior that defines a desired system functionality. Functional blocks define standard formats and semantics for how information is exchanged between devices on a network.

Functional Block Array

A set of identical functional blocks. A functional block array is useful if your device contains two or more identical switches, lights, dials, controllers, or other I/O components that will each have an identical external interface. In addition, a functional block array saves code space and reduces the number of when-tasks in your code.

Functional Profile

A LONMARK specification that enables equipment specifiers to select the functionality they need for a system. A functional profile is a template for a type of functional block that defines mandatory and optional network variable and configuration property members along with their intended usage. A small number of functional profiles are available for generic devices such as simple sensor and actuators. Many industry-specific functional profiles are available for industry-specific applications. Industry-specific profiles are developed through a review and approval process, including a cross-functional review to ensure the profile will interoperate within an individual subsystem and also provide interoperability with other subsystems in the network.

Gizmo 4 I/O Board

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.

***i*.LON IP-852 Router**

An *i*.LON IP-852 router forwards ISO/IEC 14908-2 packets enveloped in ISO/IEC 14908-4 packets over an IP-852 channel. *i*.LON IP-852 routers include the *i*.LON SmartServer with IP-852 routing, *i*.LON 100 e3 Internet Server with IP-852 routing, and the *i*.LON 600 LONWORKS-IP Server.

I/O Object

An instantiation of an I/O model. An I/O object consists of a specific I/O model, and its pin assignment, modifiers, and name.

IP-852 Channel

Also known as an ANSI/CEA-852 LONWORKS/IP channel, an IP-852 channel carries ISO/IEC 14908-2 packets enveloped in ISO/IEC 14908-4 packets. An IP-852 channel is a LONWORKS channel that uses a shared IP network to connect IP-852 devices and is defined by a group of IP addresses. These IP addresses form virtual wires that connect IP-852 devices so they can communicate with each other. IP-852 devices include the LNS Server computers, LonMaker computers, and *i*.LON IP-852 routers. An IP-852 channel enables a remote full client to connect directly to a LONWORKS network and perform monitoring and control tasks.

IP-852 Network Interface

Formally called VNI, an IP-852 network interface enables IP-852 devices such as LNS Server computers, LonMaker computers, and *i*.LON IP-852 routers to be attached to IP-852 channels. An IP-852 network interface requires that the LONWORKS-IP Configuration Server be configured before trying to communicate with remote devices or remote computers.

Implementation-specific NVs/CPs

Network variables and configuration properties that are not defined in the functional profiles used by their parent functional blocks. Implementation-specific network variables and configuration properties (those implemented as configuration network variables [CPNVs]) appear in Virtual functional blocks instead of their parent functional blocks when you are using the IzoT Commissioning tool or other network tool.

Note: If you use implementation-specific network variables in your device interface, your device will not comply with interoperability guidelines version 3.4 (or better) and therefore cannot be certified by LONMARK. A better alternative for adding members to a functional profile is to create a user-defined functional profile template (UFPT) that inherits from an existing standard functional profile template (SFPT), and then add new mandatory or optional member network variables to the UFPT. This method results in a new functional profile that you can easily reuse in new devices. See the *NodeBuilder Resource Editor User's Guide* for more information on creating UFPTs.

Interoperable Self-installation (ISI) Protocol

The standard protocol for performing self-installation in LONWORKS networks. ISI is an application-layer protocol that lets you install and connect devices without using a separate network management tool.

ISI Mode

An installation scenario in which the ISI protocol is used (instead of the LonMaker tool or other network tool) to install devices and create network variables connections.

IzoT Commissioning Tool

An OpenLNS network tool that uses Visio as its graphical user interface. The IzoT Commissioning tool is used to design, commission, maintain, and document distributed control networks comprised of both LONMARK and other LONWORKS devices.

IzoT NodeBuilder Tool

A hardware and software platform that is used to develop applications for Neuron Chips and Echelon Smart Transceivers. The IzoT NodeBuilder tool provides complete support for creating, debugging, testing, and maintaining LONWORKS devices. You can use the IzoT NodeBuilder tool all to create many types of devices, including VAV controllers, thermostats, washing machines, card-access readers, refrigerators, lighting ballasts, blinds, and pumps. You can use these devices in a variety of systems including building controls, factory automation, and transportation.

IzoT Router

The IzoT router is included with the FT 6000 EVK. It includes the IzoT Server Stack and the FT Terminators. The IzoT router supports FT and Ethernet interfaces and can be used to connect the FT 6000 EVBs to a host that is running the IzoT NodeBuilder software and IzoT Commissioning Tool.

LNS Device Template

A device template automatically generated by the IzoT NodeBuilder tool when you build a device application. The LNS device template defines the external interface to the device, and it is used by the IzoT Commissioning tool and other OpenLNS network tools to configure and bind the device

LNS Network Database

Each LONWORKS network has its own LNS network database (also referred to as the network database), which includes the network and device configuration data for that network. The network database also contains extension records, which are user-defined records for storing application data.

LNS Server

The computer containing the LNS global database acts as the LNS Server. The LNS global database contains the group of LONWORKS networks being managed with the LNS Server.

Local Client

A LonMaker computer that is also running the LNS Server.

Local Device

An FT 6000 EVB board running the *NcMultiSensorExample* application that receives **SNVT_lux** and/or **SNVT_temp_p** output network variable updates from another device (a remote device). The local device displays the temperature and light level values received from the remote device in the Remote Info Mode panel on its LCD. A remote device may be another FT 6000 EVB board running the *NcMultiSensorExample* application.

LonMaker Browser

Part of the IzoT Commissioning Tool, the LonMaker Browser is an LNS plug-in that provides a table view of the network variables and configuration properties of selected devices and/or functional blocks. The LonMaker Browser can be used to monitor and control the network variables and configuration properties in a network.

LonMaker Drawing

A LonMaker drawing contains the graphical representation of a LONWORKS network.

LonMaker Network Design

A LonMaker network design consists of an LNS network database and a LonMaker drawing.

LonMaker Shape

A reusable drawing object related specifically to a LONWORKS device.

LONMARK

A distinctive logo applied to LONWORKS devices that have been certified to the interoperability standards of LONMARK International.

LONWORKS 2.0 Platform

The next generation of LONWORKS products designed to both increase the power and capability of LONWORKS devices, and to decrease the costs of device development and devices.

LONWORKS Network

A network of intelligent devices (such as sensors, actuators, and controllers) that communicate with each other using a common protocol over one or more communications channels.

LONWORKS Technology

The technology that allows for the creation of open, interoperable control networks that communicate with the LonTalk protocol. LONWORKS technology consists of the tools and components required to build intelligent device and to install them in control networks.

LTM-10A Platform

A complete LONWORKS device with downloadable flash memory and RAM that you can use for testing your applications and I/O hardware prototypes. You can connect a Gizmo 4 I/O Board to the LTM-10A Platform.

Mandatory Network Variable/Configuration Property

A network variable/configuration property that must be implemented by the functional block, as specified by the functional profile that the functional block is instantiating.

Monitored Connection

A connector shape or reference connection on which network variable values are displayed and updated.

Network Interface

A LONWORKS device that provides a layer 6 LonTalk interface to an external host computer such as a computer or a handheld maintenance tool

Network Variable (NV)

Network variables allow a device to send and receive data over the network to and from other devices. Network variables are data items (such as temperature, the state of a switch, or actuator position setting) that a particular device application program expects to receive from other devices on the network (an *input network variable*) or expects to make available to other devices on the network (an *output network variable*).

Network Variable/Configuration Property Types

A network variable or configuration property type defines the structure and contents of the object. A network variable type can be either a standard network variable type (SNVT) or a user-defined network variable type (UNVT). A configuration property type can be a standard configuration property type (SCPT) or a user-defined configuration property type (UCPT)

Neuron 5000 Processor

Echelon's next-generation Neuron chip designed for the LONWORKS 2.0 platform. The Neuron 5000 processor is faster, smaller, and cheaper than previous-generation Neuron chips. The Neuron 5000 processor includes a fourth processor for interrupt service routine (ISR) processing.

The Neuron 5000 processor supports an internal system clock speed of 5 MHz to 80 MHz (using a 10 MHz external crystal). The Neuron 5000 processor includes 16KB of on-chip ROM to store the Neuron firmware image and 64 KB on-chip RAM (44 KB is user-accessible). The Neuron 5000 processor requires at least 2KB of off-chip EEPROM to store configuration data, and you can use a larger capacity EEPROM device or an additional flash device (up to 64KB) to store your application code, configuration data, and an upgradable Neuron firmware image. The Neuron 5000 processor

supports the mapping of external non-volatile memory from 0x4000 to 0xDFFF in the Neuron address space (a maximum of 42KB).

Neuron Assembler (NAS)

A Neuron C tool that is used to produce Neuron object files.

Neuron C

A programming language based on ANSI C that you can use to develop applications for Neuron Chips and Smart Transceivers. It includes network communication, I/O, and event-handling extensions to ANSI C, which make it a powerful tool for the development of LONWORKS device applications.

Neuron Chip

A semiconductor component specifically designed for providing intelligence and networking capabilities to low-cost control devices. The Neuron core includes up to four processors that provide both communication and application processing capabilities. Two processors execute the layer 2 through 6 implementation of the ISO/IEC 14908-1 protocol and the third executes layer 7 and the application code. LONWORKS 2.0 Neuron cores include a fourth processor for interrupt service routine (ISR) processing.

Neuron C Compiler (NCC)

A Neuron C tool that is used to produce Neuron assembly source files from Neuron C source code.

Neuron Exporter (NEX)

A Neuron C tool that takes input from the compiler and the linker and produces the following types of files: downloadable application image files (**.APB**, **.NDL**, and **.NXE** extensions), programmable application image files (**.NRI**, **.NFI**, **.NEI**, **.NME**, and **.NMF**, extensions), and device interface files (**.XIF** and **.XFB** extensions).

Neuron Firmware

A complete operating system including an implementation of the ISO/IEC 14908-1 protocol used by a Neuron chip. The Neuron firmware is a program that is inserted into programmable read-only memory (programmable ROM) of a Neuron chip.

Neuron ID

A 48-bit number assigned to each Neuron Chip at manufacture time. Each Neuron Chip has a unique Neuron ID, making it like a serial number.

Neuron Librarian (NLIB)

A Neuron C tool that is used to create and manage libraries, or to add and remove individual object files to and from an existing library.

Neuron Linker (NLD)

A Neuron C tool that is used to produce Neuron executable files. It links the application image, user-libraries, system libraries, and the Neuron firmware.

Neuron Object File

A Neuron object file (**.NO** extension) is an intermediate file that contains the data and executable code in binary form, and contains information about exported and imported symbols. Neuron object files are the link between the Neuron Assembler and the Neuron Linker, but other data also contributes to the linking

Node Object

A functional block that monitors the status of all functional blocks in a device and makes the status information available for monitoring by the IzoT Commissioning tool. A LONMARK-compliant device that has more than one functional block must have a node object.

NodeBuilder Device Template

An XML file with a **.NbDt** extension that specifies the information required for the IzoT NodeBuilder tool to build the device application. The NodeBuilder device template includes a list of Neuron C source code files and the hardware template name

NodeBuilder Hardware Template

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 IzoT NodeBuilder tool. You can use these or create your own. Third-party development platform suppliers may include NodeBuilder hardware templates for their platforms

NodeBuilder Project

A NodeBuilder project collects all the information about a set of devices that you are developing.

NodeBuilder Project Manager

The NodeBuilder Project Manager provides an integrated view of an entire NodeBuilder project and provides the tools for defining and building a NodeBuilder device.

Non-const Device-specific Configuration Property

A configuration property that can be changed by the device application, an LNS network tool such as the LonMaker tool, or another tool not based on LNS. For example, a thermostat may include a user interface that allows the user to change the setpoint.

OffNet

A management mode in which network configuration changes are stored in the network database, but not propagated to the devices on the network. To send the changes to the devices, you place the IzoT Commissioning tool OnNet. If the IzoT Commissioning tool is OffNet and attached to the network, you can still perform read operations on the network.

OnNet

A management mode in which network configuration changes are propagated immediately to the devices on the network.

OpenLNS

A network operating system that provides services for interoperable LONWORKS installation, maintenance, monitoring, and control tools such as the IzoT Commissioning tool. Using the services provided by the OpenLNS client/server architecture, tools from multiple vendors can work together to install, maintain, monitor, and control LONWORKS networks. The OpenLNS architecture consists of the following elements:

1. The OpenLNS Client application program, which can be used to develop, monitor and control LONWORKS networks.
2. The OpenLNS Object Server ActiveX Control, which is a language-independent programming interface to access the LONWORKS network.
3. The Network Services Server (NSS), which maintains an image of the network.
4. The Data Server, which provides services for monitoring and control.
5. The Network Services Interface (NSI), which is the physical interface to the network.

Optional Network Variable/Configuration Property

A network variable/configuration property that may be implemented by the functional block, as specified by the functional profile that the functional block is instantiating.

PCC-10

A type II PC (formerly PCMCIA) card network services interface (NSI) that includes an integral FTT-10 transceiver. Other transceiver types can be connected to the PCC-10 via external transceiver “pods”. The PCC-10 is the best NSI to use with laptop, notebook, or embedded PCs.

PCLTA-10/20

A ½ size ISA card network services interface (NSI). Unlike the PCNSI, it includes a twisted pair transceiver onboard, eliminating the need to attach a separate SMX transceiver assembly. The PCLTA-10 also supports the Windows plug-and-play standard. The PCLTA-10/20 is the best NSI to use on a host computer attached to a twisted-pair channel.

PCNSI

A half-length ISA card network services interface (NSI). Requires an SMX transceiver to interface to any LONWORKS communications channel. The PCNSI has two modes of operation – NSI mode and network interface mode. In NSI mode, the host treats the PCNSI card as a smart peripheral device that provides access to an NSS either locally on the PC or remotely via the LONWORKS network. In network interface mode, the host uses the PCNSI card as a standard LONWORKS network interface.

The PCNSI card is supported, but it is not recommended for use with the NodeBuilder FX tool. For better performance, use the USB 10/20 network interface included with the NodeBuilder FX tool, or use a PCLTA-10/20 or PCC-10 adapter.

Peer-To-Peer

A control strategy in which independent intelligent devices share information directly with each other and make their own control decisions without the need or delay of using an intermediate, central, or master controller. Because of the enhanced system reliability introduced by eliminating the master (a single point of failure) and the reduced installation and configuration cost inherent in peer-to-peer designs, LONWORKS technology is intended to implement a peer-to-peer control strategy.

PL-20

The power line LONWORKS channel type.

Program ID

A unique, 16-hex digit ID that uniquely identifies the device application.

Project Make Facility (PMK)

A Neuron C tool that manages the build process (it minimizes the number of build steps required), and handles program ID management tasks and automatic boot ID processing.

Remote Client

A LonMaker computer that communicates with the LNS Server (running on a separate computer) over a LONWORKS channel (an IP-852 or TP/XF-1250 channel) or over an LNS/IP interface. The IzoT NodeBuilder tool cannot be run on a remote client.

Remote Network Interface (RNI)

A network interface that enables you to connect an LNS or OpenLDV-based application to a LONWORKS network via a TCP/IP connection. RNIs include the *i.LON SmartServer*, *i.LON 100 e3 Internet Server*, *i.LON 600 LONWORKS-IP Server*, and *i.LON 10 Ethernet Adapter*.

Resource File

A file included with a LONWORKS device that defines the components of the device interface to be used by the IzoT Commissioning tool or other LNS network tool. Defined components include network variable types, configuration property types, and functional profiles implemented by the device application. Resource files allow for the correct formatting of the data, and they are necessary for LONMARK certification of a device.

SLTA-10

A serial NSI interface with built-in twisted pair transceiver that connects to any host with an EIA-232 (formerly RS232) port. It can also connect to the host remotely using a Hayes-compatible modem. The SLTA-10 is the best NSI to use for remote application or for portable hosts that do not contain a type II PC slot or a USB interface.

The SLTA-10 adapter is supported, but not recommended unless dial-up operation through a modem and a serial connection is required. You should use a PCC-10 or U10/20 USB network interface instead. For accessing remote networks, you can use an RNI such as the *i.LON SmartServer*, *i.LON 100 e3 Internet Server*, *i.LON 600 LONWORKS-IP Server*, and *i.LON 10 Ethernet Adapter*.

Self-Installed Network

A network that has network addresses and connections created without the use of a network management tool. In a self-installed network, each device contains code (the Neuron C ISI library, which implements the ISI protocol) that replaces parts of the network management server's functionality, resulting in a network that no longer requires a special tool or server to establish network communication or to change the configuration of the network.

Service Pin

Each Neuron Chip has a service pin used during installation to acquire the Neuron Chip's Neuron ID. When this pin is grounded, the Neuron Chip sends a broadcast message containing its Neuron ID and program ID, which is called service pin message or packet. The method used to ground the service pin varies from device to device. Examples of mechanical methods include grounding via a push button or using a magnetic reed switch. By attaching one of the device's I/O pins to the service pin, the service pin can also be put under software control as long as the device is configured. For example, the device can ground the pin when the device is moved or when a predefined series of I/O occurs. The service pin can also be used to drive an LED that indicates the Neuron Chip's state. The service LED is solid on when the Neuron Chip is applicationless, blinks slowly when the Neuron Chip has an application and is unconfigured, is off when the Neuron Chip has an application and is configured, and blinks once quickly each time the Neuron Chip is reset.

Smart Transceiver

A chip that integrates a Neuron network processor core and a transceiver.

Standard Configuration Property Type (SCPT)

A standard set of configuration property types defined by LONMARK International to facilitate interoperability. SCPTs are defined for a wide range of configuration properties used in many kinds of functional profiles, such as hysteresis bands, default values, minimum and maximum limits, gain settings, and delay times. SCPTs should be used in a LONWORKS network wherever applicable. In situations where there is not an appropriate SCPT available, manufacturers may define UCPTs for configuring their devices. See the LONMARK Web site for a current list and documentation.

Standard Functional Profile Template (SFPT)

A standard set of functional profiles defined by LONMARK International. See the LONMARK Web site for a current list and documentation. See *Functional Profile* for more information about functional profile templates.

Standard Network Variable Type (SNVT)

A standard set of network variable types defined by LONMARK International to facilitate interoperability by providing a well-defined interface for communication between devices made by different manufacturers. See the Echelon or LONMARK Web site for a current list and documentation.

Stencil

A collection of master shapes that can be reused in Visio.

Target Device

A LONWORKS device application that is built by the IzoT 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.

TP/FT-10

The free topology twisted pair LONWORKS channel type, which has 78Kbps bit rate.

U10/20 USB Network Interface.

A low-cost, high-performance LONWORKS network interface with a built-in TP/FT-10 or PL-20 transceiver that can be used with USB-enabled computers and controllers.

User-defined Configuration Property Type (UCPT)

A non-standard data structure used for configuration of the application program in a LONMARK device. UCPTs should be used only when there is no appropriate standard configuration property type (SCPT) defined. LONMARK-certified devices must have UCPTs documented in resource files according to a standard format, in order to allow the devices to be configured without the need for proprietary configuration tools.

User-defined Functional Profile Template (UFPT)

A non-standard functional profile template defined by a device manufacturer. UFPTs should be used only when there is no appropriate standard functional profile template (SFPT) defined. See *Functional Profile* for more information about functional profile templates.

User-defined Network Variable Type (UNVT)

A non-standard network variable type defined by the manufacturer of a device. UNVTs should be used only when there is no appropriate standard network variable type (SNVT) defined. LONMARK-certified devices must have UNVTs documented in resource files according to a standard format, in order to allow the devices to be interoperable.

Virtual Functional Block

A static functional block that contains the network inputs and outputs for a device that are not part of other functional blocks on the device.

WireShark

WireShark is an open source network protocol analyzer used to capture, analyze, characterize, and display network packets so you can pinpoint network or device faults and identify potential solutions.

Appendix D

NodeBuilder Software License Agreement

When installing the NodeBuilder software, you must agree to the terms of the software license agreement detailed in this appendix.

IzoT™ NodeBuilder® Development Tool

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 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.

IzoT NodeBuilder 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 Documentation 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 purposes 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 IzoT 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 OpenLNS Commissioning Tool, Microsoft Visio, or any other software product shipped with the IzoT NodeBuilder Development Tool product and not contained in the NodeBuilder directories as identified in the Documentation.
- “NodeBuilder Example Applications” means the Neuron C 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” directory and its subdirectories, (ii) as generated by the NodeBuilder Code Wizard, or (iii) otherwise containing wording in the source code clearly identifying such source code as an “Example Application”.

- “IzoT Device” means a product designed for use in a network based upon Echelon’s IzoT Platform.
- “Your Device” means an IzoT Device that you developed by using the Licensed Software.
- “Your IzoT Network Services Application” means Your software product that makes calls to the IzoT Network Services Server (as described in the Documentation) and incorporates the IzoT Network Services Server software.
- “You(r)” means Licensee, i.e. the company, entity or individual who has rightfully acquired the IzoT NodeBuilder Development Tool.

LICENSE

You may:

- use the Licensed Software solely to develop Your Devices and Your IzoT Network Services Applications and prepare your derivative works of the NodeBuilder Example Applications to develop Your Devices and Your IzoT Network Services Applications;
- install and use the Licensed Software for such purposes on one (1) primary computer (the “Primary Computer”);
- 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;
- copy the Documentation, provided that You reproduce, unaltered, all proprietary notices on or in the copy;
- make one (1) copy of the Licensed Software in machine readable form solely for backup purposes, provided that You reproduce, unaltered, all proprietary notices on or in the copy;
- keep the original media on which the Licensed Software was provided by Echelon solely for backup or archival purposes;
- make, use, and sell Your Devices;
- develop, use, sell, and distribute Your IzoT Network Services Applications; and
- 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:

- 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;
- copy the Licensed Software except as permitted above;
- 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);
- 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
- 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.

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.

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 IzoT Network Services Applications, and that Echelon Corporation will not support Your Devices and Your

IzoT Network Services Applications, or (ii) inform the end-user that there will be no support for Your Devices and Your IzoT Network Services Applications.

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).

© 2014 Echelon. LON, the Echelon logo, IzoT, and NodeBuilder are trademarks of Echelon Corporation that may be registered in the United States and other countries. A complete list of Echelon’s trademarks is available at www.echelon.com. All rights reserved.

