ECHELON

Mini EVK User's Guide
Revision 3

078-0302-01D

Echelon Corporation
www.echelon.com

Welcome

This document describes how to use the Mini EVK Evaluation Kit. You can use the Mini EVK to develop a prototype or production control system that requires networking, particularly in the rapidly growing, price-sensitive mass markets of smart light switches, thermostats, and other simple devices and sensors. You can also use the Mini EVK to evaluate the development of applications for such control networks using the LONWORKS® platform.

Related Documentation

The *Introduction to the LONWORKS System* document provides an introduction to the ANSI/CEA-709.1 (EN14908) Control Network Protocol. The *Neuron C Programmer's Guide* document outlines and discusses the key concepts of developing a LONWORKS application, and explains key concepts of programming using the Neuron C Version 2.1 programming language. The *Mini EVK Hardware Guide* describes how to assemble and use the hardware included with the Mini EVK. The *ISI™ Programmer's Guide* describes the ISI protocol, which is used by the Mini EVK and provides for easy development of devices that do not require installation tools.

To view these documents, click the Windows **Start** menu, point to **All Programs**, point to **Echelon Mini EVK**, and then click **Introduction to LonWorks**, **Neuron C Programmer's Guide**, **Mini EVK Hardware Guide,** or **ISI Programmer's Guide**.

System Requirements

To install and use the Mini EVK, your computer must meet the following minimum requirements:

- Microsoft® Windows® XP or Windows 2000
- Pentium® III 800MHz processor
- 128MB RAM minimum (256MB RAM recommended)
- 440MB of available hard-disk space
- 800x600 screen resolution
- CD-ROM drive

Table of Contents

1

Introduction

This chapter introduces the Mini EVK, and describes how to install the Mini EVK software. It also provides a roadmap to follow when reading this document and learning how to use the Mini EVK.

Introducing the Mini Evaluation Kit

Echelon's Mini EVK Evaluation Kit is a tool for evaluating the development of control network applications with the ANSI/CEA-709.1 (EN14908) Control Network Protocol. You can use the Mini EVK to develop a prototype or production control system that requires networking, particularly in the rapidly growing, price-sensitive mass markets of smart light switches, thermostats, and other simple devices and sensors. You can also use the Mini EVK to evaluate the development of applications for such control networks using the LONWORKS platform.

The Mini EVK is available in free topology twisted pair (FT) and power line (PL) versions, both of which leverage Echelon's unique smart transceiver technology. A USB Network Interface is included with the Mini EVK to connect the computer running the Mini EVK software to target hardware devices on twisted pair or power line channels.

Some of the key features of the Mini EVK are listed below:

- Neuron® C compiler for fast development of control applications.
- Simple to use right out of the box.
- Packaged with two working Mini EVB Evaluation Boards with MiniGizmo I/O Boards and preprogrammed code examples.
- Libraries for interoperable self-installation (ISI™). ISI provides for easy development of devices that do not require installation tools, and is also fully compatible with LONWORKS standard installation tools such as the LonMaker® Integration Tool.

Mini EVK vs. NodeBuilder Tool Comparison

The Mini EVK may be the only development platform you require. However, the NodeBuilder® Development Tool is also available for larger applications and faster development. You can start with the Mini EVK and later transition to the NodeBuilder tool to accelerate your development. You can incorporate the source files, hardware templates and Neuron C libraries used in your Mini EVK projects into a NodeBuilder project. For more information on this, see *Using the Mini Application With the NodeBuilder Tool* on page 77.

Table 1.1 compares the Mini EVK and the NodeBuilder tool. For more information on the NodeBuilder tool, see the NodeBuilder Web page at www.echelon.com/nodebuilder.

Table 1.1 Mini EVK / NodeBuilder Tool Comparison

Feature	Mini EVK	NodeBuilder Tool
Neuron C Compiler	✓	✓
Network Variables per Device	32 maximum	62 maximum

Feature	Mini EVK	NodeBuilder Tool
Application Code and Constant Data per Device*	32Kbyte maximum	64Kbyte maximum
Code Wizard	No	✓
Plug-in Wizard	No	✓
Debugger	No	✓
Project Manager	No	✓
Integrated Development Environment	No	✓
Network Installation Tool	Application loader only	Complete network installation and test tool
Target Hardware	Evaluation boards with 64KByte flash memory	Development platform with 64KByte flash and 32KByte RAM; compatible with any standard or custom hardware platform
I/O Boards	Simple I/O boards with LEDs, pushbuttons, and a temperature sensor.	I/O board with LCD display, prototyping area, versatile analog and digital I/O.

*The application code and constant data per device figures represent the maximum application sizes that the Mini EVK and NodeBuilder tools can compile.

Mini EVK Contents

The Mini EVK includes the following hardware:

- A PL 3120® and a PL 3150® EVB Evaluation Board if you are using the PL-20 version, or an FT 3120 and an FT 3150 EVB Evaluation Board if you are using the TP/FT-10 version.
- Two MiniGizmo I/O Boards that can be attached to each EVB.
- A U10 or U20 USB Network Interface you can use to attach the computer running the Mini EVK software to a TP/FT-10 or PL-20 channel for communicating with your target hardware devices.
- A cable for wiring your devices together (TP/FT-10 version only).

The Mini EVK software includes the following applications and examples:

- Mini Application, which you can use to manage Neuron C code, build Neuron C applications, and download those applications into the evaluation boards. For more information on the Mini Application, see Chapter 2, *Using the Mini Application*. For more information on the Neuron C and C# example applications, see Chapter 3, *Using the Mini EVK Example Applications*.
- Several example Neuron C applications you can use when getting started with the LONWORKS platform.
- The NodeBuilder Resource Editor, which provides a simple interface for viewing existing LONMARK® resources and defining your own resources. For more information on the NodeBuilder Resource Editor, see the *NodeBuilder Resource Editor User's Guide*.
- The ISI Developer's Kit, which provides for easy development of devices that do not require installation tools. Consult the *ISI Programmer's Guide* for more information on ISI.
- OpenLDV™ 2.1 library, which is an API used by the Mini EVK software to send and receive ANSI/CEA-709.1 messages through Echelon's family of LONWORKS network interface products. The C# example uses the OpenLDV API, as described in Appendix B of this document.

  Consult the *OpenLDV Programmer's Guide* for more information on OpenLDV. You can download the *OpenLDV Programmer's Guide* and the OpenLDV Developer's Kit from www.echelon.com/openldv.

Installing the Mini EVK Software

Follow the steps below to install the Mini EVK software. Before doing so, make sure your computer meets the requirements listed in the *System Requirements* section on page i. Once you have installed the software, you can begin using the Mini Application, as described in Chapter 2, *Using the Mini Application*.

1. Insert the Echelon Mini EVK CD into a CD-ROM drive. If the installation does not automatically start after a few seconds, start the program manually. You can start the installation by clicking the Windows **Start** button, clicking **Run**, browsing to the setup application, and then clicking **Open**. The main Mini Evaluation Kit installation window opens.
2. Click **Install Products** to continue. The Install Products window opens.
3. Click **Mini EVK Software** to continue. The Mini EVK Software installation includes all the software items listed in the *Mini EVK Contents* section earlier in this chapter. The Welcome window opens.
4. Click **Next** to continue. The License Agreement window opens.
5. Read the license agreement, and click **I Accept the Terms in the License Agreement** if you agree to the license agreement. The Customer Information window opens.
6. Fill in your user name, organization and serial number, and click **Next** to continue. The Ready to Install window opens.

7. Click **Install** to begin the installation. When the installation has completed, a window appears to notify you. Click **Finish** to exit the installation wizard.
8. If you do not have an Adobe Acrobat reader, you can install it by selecting **Install Adobe Acrobat Reader** in the Install Products windows described in step 2.
9. If you are using a PCC-10, PCLTA-10, PCLTA-20, or PCLTA-21 interface instead of the U10 or U20 USB interface included with the Mini EVK, you can install a driver for these interfaces by selecting the **Install the PCC-10/PCLTA-10/20/21 Drivers** software from the Install Products window described in step 2.

Document Roadmap

The remainder of this document describes how to develop Neuron C applications for LONWORKS devices with the Mini EVK, and how to download those applications into the evaluation boards and test them. It also describes how to use the Neuron C and C# example applications included with the Mini EVK software. This content is divided into the following sections:

- Chapter 2, *Using the Mini Application*. This chapter describes how to use the Mini Application to create or modify a Neuron C application, build an application image, and then download the application image into a device. This chapter also describes how to use the Mini Application to reset, wink or test a device.
- Chapter 3, *Using the Mini EVK Example Applications*. The Mini EVK includes several Neuron C example applications you can download into the evaluation boards, as well as the C# Monitoring & Control Example Application, which is a C# application you can use to monitor your evaluation boards from your computer. This chapter describes these applications.
- Chapter 4, *Developing a Neuron C Application*. This chapter introduces the Neuron C Version 2.1 programming language. It describes the basic aspects of the language and provides an overview of how you can use the LONWORKS platform and the Neuron C programming language to construct interoperable devices and systems.
- Chapter 5, *Debugging a Neuron C Application*. This chapter describes how to use the boards and accessories included with the Mini EVK, or additional tools such as the LonMaker Integration Tool or NodeBuilder Development Tool, to debug a Neuron C application.
- Appendix A, *Troubleshooting*. This appendix describes how to resolve problems that may occur when you use the Mini EVK with the evaluation boards.
- Appendix B, *Monitoring & Control Application Overview*. This appendix describes the Monitoring & Control Example Application.

2

Using the Mini Application

This chapter describes how to use the Mini Application to build a Neuron C application image, and how to download an application image into a device. This chapter also describes how to use the Mini Application to reset, wink or test a device.

Starting the Mini Application

You can use the Mini Application to build an application image for a Neuron C application, download the application image into a device, and then test the basic functionality of the application. The remainder of this chapter describes how to perform these tasks.

To start the Mini Application, click the Windows **Start** menu, point to **All Programs**, point to the **Echelon Mini EVK** program folder, and then click **Mini EVK Application**. The Application tab opens.

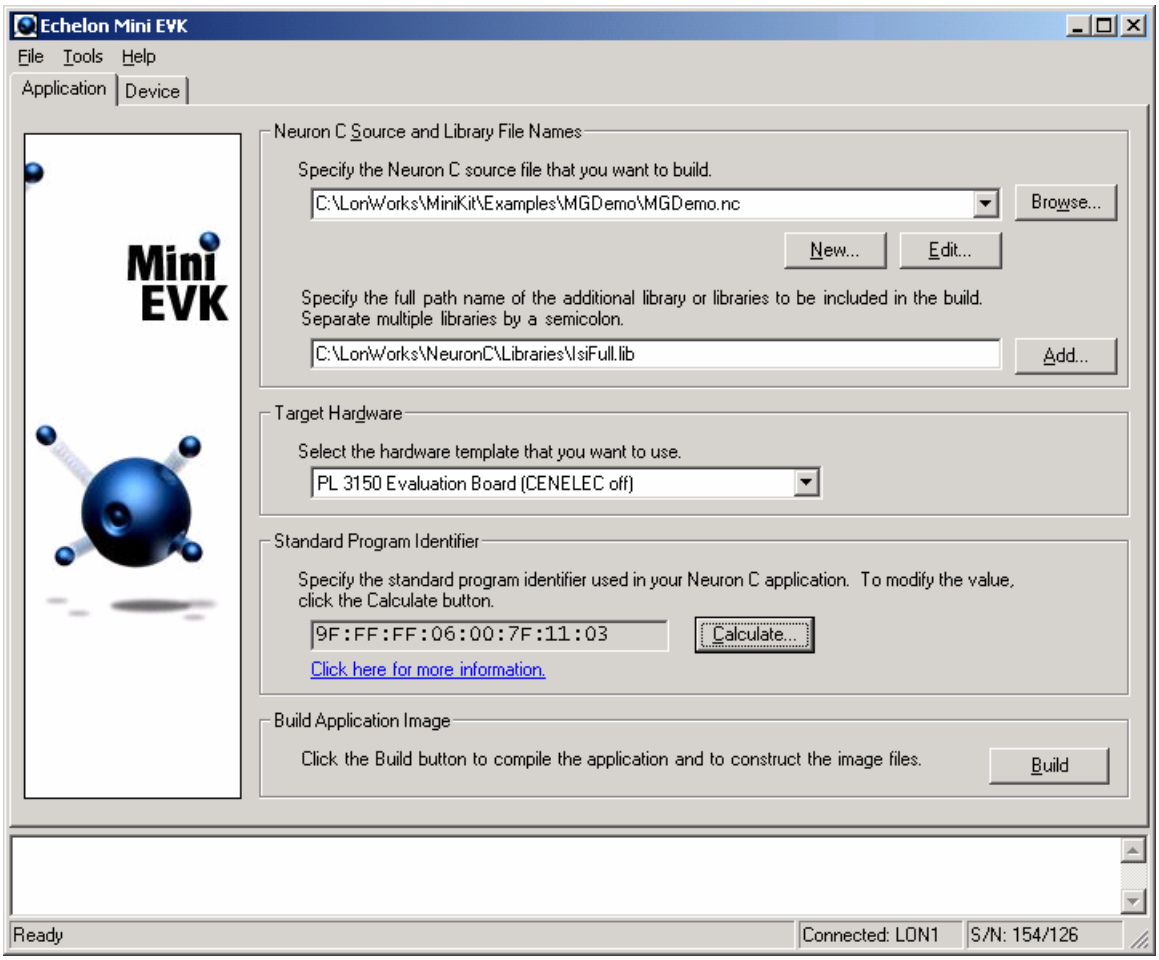

Figure 2.1 Application Tab

Building a Neuron C Application Image

Neuron C is the programming language that you can use to create applications for an evaluation board, as well as for other LONWORKS hardware based on a Neuron Chip or Echelon Smart Transceiver. The Neuron C programming language is introduced in Chapter 4, *Developing a Neuron C Application*, and is described in more detail in the *Neuron C Programmer's Guide* and *Neuron C Reference Guide* documents.

You can create a new Neuron C application, modify an existing Neuron C application or example, or create a Neuron application image for one of the Mini

EVK example Neuron C applications with the Mini Application. To create a Neuron application image, follow these steps:

1. Start the Mini Application and click the **Application** tab, as described in the previous section. You will use the Application tab to select a Neuron C file, optionally select any libraries, select a hardware template, define the program ID, and build the Neuron application image.

2. Enter the Neuron C source file in the first **Neuron C Source and Library File Names** box. Click the arrow to select a recently compiled application. Click **New** to create a new Neuron C application. This creates a new empty source file, and opens the file using your computer's default text editor. Click **Browse** to browse your files for an existing application.

   The Mini EVK includes several example Neuron C applications that you can use. To select one of the example applications, click **Browse** and then navigate to the **Examples** folder of the **MiniKit** folder. Open any of the folders in the **Examples** folder, and then select the **.nc** file to use that example. For descriptions of these examples, including limitations on target hardware, see Chapter 3, *Using the Mini EVK Example Applications*.

3. Click **Edit** to modify the selected application. This opens the application's source file with your computer's default text editor. By default, this is Windows Notepad. You may want to use a different text editor. To do so, open the **Folder Options** in the Windows Control Panel, click the **File Types** tab, select the **NC** extension, and then click **Change** to change the program you want to use to open **.NC** files. The Mini EVK will then use the new editor to edit the Neuron C file.

   For an introduction to Neuron C programming, see Chapter 4, *Developing a Neuron C Application*.

4. Click **Add** next to the second **Neuron C Source and Library File Names** box to add a Neuron C library. The standard libraries other than the ISI libraries are automatically included. These libraries are described in Appendix B of the *Neuron C Programmer's Guide*.

   This step opens the Add Library/Libraries window, which you can use to select a library. This defaults to the LONWORKS **Neuron C\Libraries** directory, which contains the standard Neuron C libraries, as well as the ISI libraries described in the *ISI Programmer's Guide*. Alternatively, you can type the name and path of the library you want to use in the box. You can enter multiple libraries by clicking the **Add** button multiple times, or by typing them in the box and separating them with semicolons (";").

   The Mini Application automatically links the application with all required standard libraries. However, some Neuron C applications have specific library requirements. For example, the example applications contained with the Mini EVK all require the ISI libraries. Seven different ISI libraries are supplied, varying in features provided and application memory required.

To build an application image for the example Neuron C applications, you must select the correct library as shown in Table 2.1. For more information on the ISI libraries, see the *ISI Programmer's Guide*.

Table 2.1 ISI Libraries

Example Application	ISI Library
MGDemo	`IsiFull.lib`
MGSwitch, MGLight	`IsiCompactManual.lib`
MGKeyboard	`IsiCompactAuto.lib`

5. Enter the hardware template for the device in **Target Hardware** box. A hardware template defines the memory layout, transceiver type, and Neuron processor type for the hardware platform to be supported by the application image. Click the arrow to select from a list of available hardware templates. The list includes hardware templates for the evaluation boards.

   To build an application image for one of the evaluation boards included with the Mini EVK, select one of the hardware templates listed in Table 2.2. This choice depends on which Mini EVK model you are using, and whether you are building an application image for a 3120 EVB or a 3150 EVB.

Table 2.2 Hardware Templates

Mini EVK Model	3120 EVB Hardware Template	3150 EVB Hardware Template
Mini EVK PL-20C	PL 3120 EVB, CENELEC	PL 3150 EVB, CENELEC
Mini EVK PL-20N	PL 3120 EVB, Non-CENELEC	PL 3150 EVB, Non-CENELEC
Mini EVK TP/FT-10	FT 3120 Evaluation Board	FT 3150 Evaluation Board

   The CENELEC access protocol is a European-standard protocol for controlling access to a power line used for communication. It is required for PL-20 devices in most of Europe, but is not required or typically used outside of Europe. See Chapter 8 of the *PL 3120 / PL 3150 Smart Transceiver Data Book* for more information on the CENELEC protocol. You can download the *PL 3120 / PL 3150 Smart Transceiver Data Book* from Echelon's Web site at www.echelon.com.

6. Next, enter the application's program ID. The program ID uniquely identifies an application, and must be different for every type of device on

a network. The program ID includes fields that define the manufacturer, device class, device subclass, transceiver type, and model number for a device type.

If you are compiling one of the example applications included with the Mini EVK, you can use any program ID, as long as you use a different program ID for each different application image. To match the application images provided with the Mini EVK, enter one of the program IDs from Tables 2.3 and 2.4 for the example applications.

Table 2.3 Example Application Program IDs – FT Evaluation Boards

Application	FT 3120 Evaluation Board	FT 3150 Evaluation Board
MGKeyboard	9F:FF:FF:05:25:04:04:04	9F:FF:FF:05:25:04:04:03
MGDemo	N/A	9F:FF:FF:05:01:04:04:04
MGLight	9F:FF:FF:1E:28:04:04:04	9F:FF:FF:1E:28:04:04:03
MGSwitch	9F:FF:FF:20:00:04:04:04	9F:FF:FF:20:00:04:04:03

Table 2.4 Example Application Program IDs – PL Evaluation Boards

Application	PL 3120 Board (CENELEC Disabled)	PL 3150 Board (CENELEC Disabled)	PL 3120 Board (CENELEC Enabled)	PL 3150 Board (CENELEC Enabled)
MGKeyboard	9F:FF:FF:05:25:05:11:02	9F:FF:FF:05:25:05:11:03	9F:FF:FF:05:25:05:10:02	9F:FF:FF:05:25:05:10:03
MGDemo	N/A	9F:FF:FF:05:01:05:11:04	N/A	9F:FF:FF:05:01:05:10:04
MGLight	9F:FF:FF:1E:28:05:11:02	9F:FF:FF:1E:28:05:11:03	9F:FF:FF:1E:28:05:10:02	9F:FF:FF:1E:28:05:10:03
MGSwitch	9F:FF:FF:20:00:05:11:02	9F:FF:FF:20:00:05:11:03	9F:FF:FF:20:00:05:10:02	9F:FF:FF:20:00:05:10:03

7. Click **Calculate** to set the program ID that the application will use with the LonMark Standard Program ID Calculator shown in Figure 2.2. You can set the program ID by manually entering it in the Program ID box at the bottom of the dialog, or you can set the fields on the dialog to appropriate values for your application, and calculate a program ID based on those values. For information on how you should set these fields, see *Selecting a Program ID* on page 16.

   When you have configured the fields on the dialog (or entered the program ID you want to use), click **OK** to return to the Application tab.

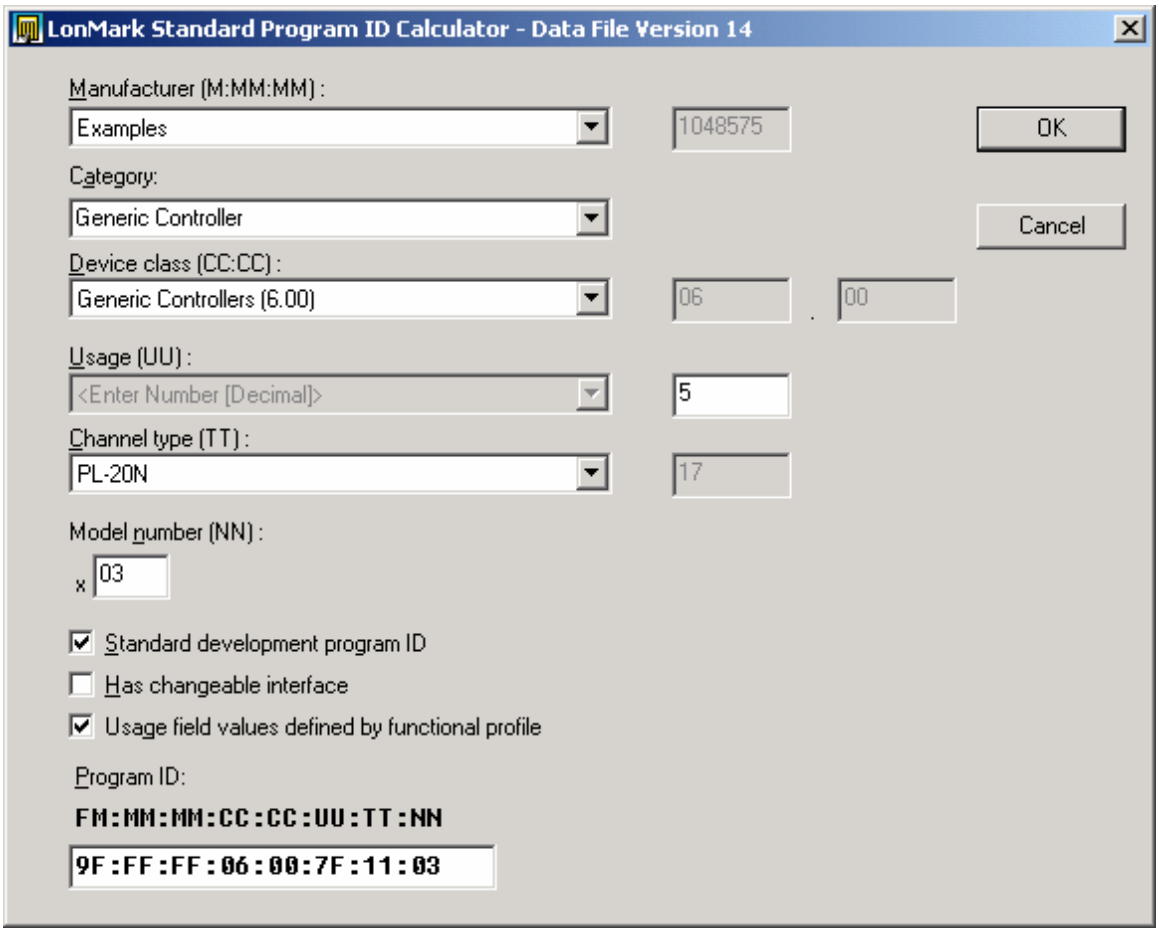

Figure 2.2 Standard Program ID Calculator Dialog

8. Click **Build** to compile the application and create the application image. The status box at the bottom of the Application tab will inform you when the application has successfully compiled, and will also inform you of any build errors.

   Consult Appendix A, *Troubleshooting*, if you are unable to compile your application. The *NodeBuilder Errors* Guide in the **Echelon Mini EVK** program folder describes the compiler, linker, and exporter errors listed in the Status box.

Loading a Neuron C Application Image

You can load a Neuron application image over a LONWORKS network into a Mini EVB, or into any LONWORKS device based on a Neuron Chip or Echelon Smart Transceiver. You can create a Neuron application image as described in the previous section, or you can load an existing Neuron application image.

The 3120 EVB comes with the MGSwitch example application pre-loaded, and the 3150 EVB comes with the MGDemo example application pre-loaded. If you have not already loaded a new application, you can use the MGSwitch and MGDemo applications without loading them. To load any of the other example applications, a custom application, or to reload MGSwitch or MGDemo into an evaluation board, follow these steps:

1. Click the **Device** tab. You will use the Device tab to connect to a device and load the Neuron application image into the device.

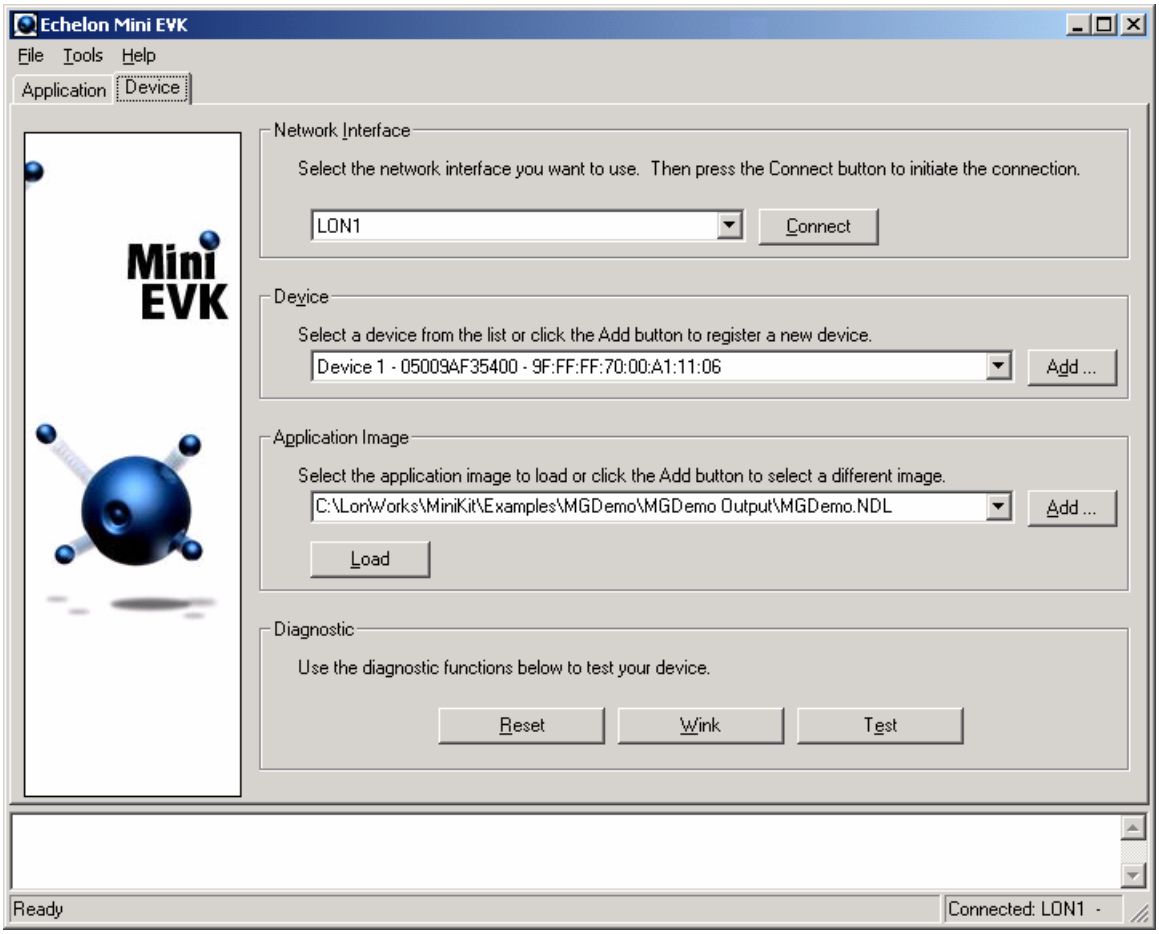

Figure 2.3 Device Tab

2. Select a network interface from the **Network Interface** box, and then click **Connect** to connect the Mini Application to the network interface. The Status bar at the bottom of the window indicates whether or not the Mini Application is connected to a network interface. The network interface connects your computer to a LONWORKS network, and enables the Mini Application to communicate with any LONWORKS devices on the network.

   You can use the U10 or U20 USB Network Interface included with the Mini EVK, or you can use another network interface such as a PCC-10, PCLTA-20, PCLTA-21, *i*.LON™ 10 Ethernet Adaptor, or *i*.LON 100 Internet Server. If you are using the U10 or U20 USB Network Interface included with the Mini EVK and do not have any other network interfaces installed on your computer, select **LON1**.

   The selected network interface typically requires a driver to communicate with the Mini Application. A driver for the U10 or U20 USB Network Interface is automatically installed when you install the Mini EVK software. To use the U10 or U20 USB Network Interface, plug the interface into a USB port and attach it to the channel as described in the

Mini EVK Hardware Guide. For more information on installing and configuring the U10 or U20 USB Network Interface, and on using it to attach your computer to a network channel, see the *LONWORKS USB Network Interface User's Guide*.

WARNING: Only one application can use a network interface at a time, so if you connect the Mini Application to a network interface, you cannot use that network interface with other applications. You must exit the Mini Application to make a connected network interface available to other applications. Multiple LNS applications can share a network interface, but they cannot share a network interface with applications that are not based on the LNS network operating system such as the Mini Application.

WARNING: If you want to use a PCC-10, PCLTA-20, or PCLTA-21 network interface, you must configure it as a layer 5 interface. See the *Mini EVK Hardware Guide* for more information on this.

3. Select a target device in the **Device** box. Click the arrow to view any devices that you have recently added or that have been automatically discovered through the ISI protocol. You can select one of these devices, or you can click the **Add** button to add a new device. If you add a new device, the Add Device dialog shown in Figure 2.4 opens.

Figure 2.4 Add Device Dialog

Enter the Neuron ID of the target device in the **Neuron ID** box and then click **OK**. The Neuron ID is a unique 48-bit (12-hex digit) identifier contained in every LONWORKS device. The Mini Application uses the Neuron ID to communicate with your selected device. For more information on Neuron IDs, see the *Introduction to the LONWORKS System* document in the **Echelon Mini EVK** program folder.

If you do not know the target device's Neuron ID, you can acquire it by pressing the device's Service button. The Service button is typically a push button included on most LONWORKS devices that causes the device to broadcast its Neuron ID on the LONWORKS network.

On a Mini EVB, the service pin button is the black button labeled "**SERVICE**." When the Mini Application receives the service pin message, it displays the Neuron ID in the **Neuron ID** box on the Add Device dialog. Click **OK** to add the device.

You can also add a device using the Service button without opening the Add Device dialog first by pressing the device's Service button. This opens the Service Pin Message window, which you can use to add the device.

Once you have added a device, you will return to the Device tab. Select the device in the Device box, and proceed to step 4.

When you add a device, it will remain in the Device list until you close the Mini Application, or connect to a new network interface. You will need to add the device again when you restart the Mini Application, or when you connect to a different network interface.

4. Select a Neuron application image in the **Application Image** box. Click the arrow to view any application images that you have recently built or added. You can select one of these application images, or you can click the **Add** button to browse your folders for a new application image. You can select a Neuron application image that you built using the Application tab, or you can select an existing Neuron application image. To select an existing Neuron application image, select a file with an **.ndl** extension. The Mini Application builds multiple types of Neuron application image files to provide support for a variety of tools, but you must provide an **.ndl** file when loading a device with the Mini Application.

Click **Load** to load the selected Neuron application image into the selected device. The Status box at the bottom of the Device tab informs you when the application image has been successfully loaded into the device, and also informs you of any load errors.

Consult Appendix A, *Troubleshooting*, if you are unable to load your application image.

NOTE: After you load an application image into a PL Evaluation Board, **LED1** will begin flashing, indicating that the PL Evaluation Board has entered CENELEC configuration mode. If **LED8** is on, then CENELEC is currently enabled. If **LED8** is off, CENELEC is currently disabled.

The initial setting depends on the hardware template you selected when you loaded the application image into the PL Evaluation Board. For example, if you selected the **PL 3120 EVB, CENELEC** template, CENELEC will be enabled by default, and **LED8** will be on. If you selected **PL 3120 EVB, Non-CENELEC**, CENELEC will be disabled by default, and **LED8** will be off.

You can press the **SW8** button to enable or disable CENELEC. When you have made a selection, press the **SW1** button to confirm your selection and exit CENELEC configuration mode. You will not be able to load another application into the PL Evaluation Board, or perform any other network operations, until you have made a selection and exited CENELEC configuration mode.

The PL Evaluation Board will enter CENELEC configuration mode every time you load an application into it (as well as the first time you power up the Evaluation Board). If you want to disable this behavior for any of the Mini example applications, you can do so by commenting out the following line in the Neuron C source file:

```
#define SUPPORT_CCL
```

For more information on CENELEC configuration mode, see the **CENELEC Config Readme.htm** document, which is installed with the Mini EVK software into the **Bin** folder of your **LONWORKS** directory.

Selecting a Program ID

The program ID is a 64-bit (16-hex-digit) identifier that uniquely identifies the application contained within a device. A program ID is typically presented as eight pairs of hexadecimal encoded digits, separated by colons. When formatted as a standard program ID, the 16 hex digits are organized as 6 fields that identify the manufacturer, classification, usage, channel type, and model number of the device. Every standard program ID uses the following format:

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

The LonMark Standard Program ID Calculator dialog shown in Figure 2.2 helps you to select the appropriate values for each part of the program ID. The calculator displays available values based on a program ID definition file included with the Mini EVK. You can update the program ID definition file at any time by downloading the latest standard program ID data from www.lonmark.org/spid. Copy the file into the **Types** folder of your LONWORKS directory (**C:\LonWorks\Types** by default).

Table 2.5 lists and describes the program ID fields.

Table 2.5 Program ID Fields

Program ID Segment	Field	Description
F	**N/A**	A 4-bit format identifier. Set to 8 for LONMARK certified interoperable devices, or to 9 for devices that use the standard program ID format and use LONMARK compatible self-documentation strings to document any functional blocks and configuration properties. Values less than 8 are used by legacy devices and network interfaces—the Mini EVK does not support these program IDs. Format identifiers 10 – 15 (0xA – 0xF) are reserved. Applications that you develop with the Mini EVK should typically use format 9, unless you get them certified by LONMARK International, in which case you can use format 8.

Program ID Segment	Field	Description
M:MM:MM	**Manufacturer**	A 20-bit identifier for the device manufacturer. Click the arrow to select from a list of all the LONWORKS device manufacturers who are members of LONMARK International. If your company is a member of LONMARK International but is not included in the list, download the latest program ID data from www.lonmark.org/spid. If your company is not a member of LONMARK International, get a temporary manufacturer ID from www.lonmark.org/mid. If your company is a LONMARK member, but not listed in the updated program ID list, or if you have a temporary manufacturer ID, select **<Enter Number [Decimal]>** in the **Manufacturer** list, then enter your manufacturer ID in the field to the right of the **Manufacturer** box. Enter the value in decimal, the calculator converts it to hex for the program ID. You do not have to join LONMARK International to get a temporary manufacturer ID, the information required to get one if very minimal, and there is no fee to get one. However, if your company is not a member of LONMARK International, now is a good time to join. For more information, see www.lonmark.org. For example applications, internally used prototypes or applications used in training, select **Examples** as the manufacturer ID (**F:FF:FF**).
CC	**Category**	The general purpose or industry of the device. The **Category** selected determines the device classes that will be available in **Device Class**. Select **ALL** to have **Device Class** show all existing device classes. Select **Profiles By Name** to have **Device Class** show an alphabetical list of all device classes with a standard functional profile. Select **Profiles By Number** to have **Device Class** show a numerical list (sorted by device class number) of all device classes with a standard functional profile.

Program ID Segment	Field	Description
CC	**Device Class**	A 16-bit identifier for the primary function of the device. The primary function of the device is determined by the primary functional profile implemented by your device. Your application will implement at least one functional profile, and may implement multiple functional profiles. If you implement multiple functional profiles, determine which is the primary based on the most typical usage of your device. Enter one of the following depending on your primary functional profile: • If you are using a standard functional profile other than functional profiles 0 through 6 and the functional profile is included in the standard resource file set, select the functional profile name from the list. The device class will be set to the functional profile number for the selected functional profile. • If you are using a standard functional profile other than functional profiles 0 through 6 that has not yet been included in the standard resource file set, select **<Enter Number [Decimal]>** from the list and then enter the functional profile key in the two boxes to the right of Device Class. Enter the last two decimal digits in the second box, and the remaining decimal digits in the first box. • If your primary functional profile is based on standard functional profiles 1 through 5 (you cannot use functional profiles 0 or 6 as the primary functional profile) or a user functional profile, select the proper value from the list of device classes maintained by LONMARK International. To enter a device class value that has not yet been added to the standard list, select **<Enter Number [Decimal]>** and enter a decimal value from 0 to 255 in each of the fields to the right of the Device Class box (the calculator converts the values to hex for the program ID). Non-interoperable applications should still use a standard program ID to describe the device's capability as closely as possible. When in doubt, choose a generic description from the list, such as "generic I/O," although a more specific description should be chosen if possible.

Program ID Segment	Field	Description
UU	**Usage**	An 8-bit identifier for the intended usage 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** box. If you are using a standard usage value, set the **Defined By Functional Profile** check box, click the arrow to select from a list of standard usage values maintained by LONMARK International. You can update the list by downloading the latest program ID data from www.lonmark.org/spid. If the primary functional profile implemented by your device specifies custom usage values, clear the **Defined By Functional Profile** check box, select **<Enter Number[Decimal]>** in the **Usage** list, and then enter a decimal value from 0 – 63 in the box next to the **Usage** box (the calculator translates the value to hex for the program ID). Non-interoperable applications should still use a standard program ID to describe the device's usage as closely as possible.
TT	**Channel Type**	An 8-bit identifier for the channel type supported by the device's LONWORKS transceiver. If you are using an FT EVB or if you are developing a device with an FT Smart Transceiver or FTT-10A transceiver, select **TP/FT-10**. If you are using a PL EVB or if you are developing a device with a PL Smart Transceiver or PLT-22 transceiver, select **PL-20C** or **PL-20N** (select **PL-20C** if you purchased a PL-20C Evaluation Kit, or **PL-20N** if you purchased a PL-20N Evaluation Kit). Applications linking with the ISI libraries must select the program ID so that it reports the channel type correctly. Non-interoperable applications should still use a standard program ID and advertise the channel type field correctly.
NN	**Model Number**	An 8-bit identifier that you assign to specify the product model for your device. Assign a unique model number for the specified manufacturer, device class, usage, and channel type. You can use the same hardware for multiple model numbers depending on the program that is loaded into the hardware. The model number within the program ID does not have to conform to your published product model number.
N/A	**Standard Development Program ID**	This field identifies this device as a development or prototype device. Select this check box if the device has not been certified by LONMARK International. Selecting this check box chooses a format 9 standard program ID.

Program ID Segment	Field	Description
N/A	**Has Changeable Interface**	Select this check box to indicate that the device has a changeable device interface, or if the device has any network variables with changeable types. See the *Neuron C Programmer's Guide* for more information on changeable type network variables.
N/A	**Usage Field Values Defined By Functional Profile**	Select this check box if the primary functional profile implemented by this device defines usage values. Otherwise, clear the check box to specify standard usage values. When selected, the **Usage** field will be set to **<Enter Number>**. Enter the custom usage value in the box to the right of the **Usage** field. .
N/A	**Program ID**	This box is automatically updated when changes are made to the other fields on the dialog. You can also manually enter a program ID here.

Resetting, Winking, and Testing Devices

You can also reset, wink, or test a device with the Mini Application. To do so, open the Mini Application and connect to the device you want to use, as described previously in this chapter. Once you have connected to a device, you have the following options:

- Click **Wink** to send the ANSI/CEA-709.1 Wink command to the device. Devices are not required to respond to this command, but it is recommended. Devices should respond to the Wink command in an application-specific, benign way.

  For example, a device could flash a LED or trigger an audible signal. When you wink an FT or PL EVB, the EVB LEDs will flash on and off for 1.5 seconds.

  Wink commands are often used when installing or diagnosing multiple devices in the field. In such situations, a tool to confirm the identity of a given device is often needed. The Wink command can be used for this purpose.

- Click **Test** to check the current status of the device. When the test completes, the Status box will display information and statistics regarding the test. This includes the current state of the device, as well as statistics such as the number of packets received by the device, the number of packets addressed to the device, and the number of missed or lost messages.

- Click **Reset** to reset the device. You can reset a device to test its reset behavior, or when the device application appears to become unresponsive.

3

Using the Mini EVK Example Applications

This chapter describes how to use the Neuron C and C# example applications included with the Mini EVK.

Mini EVK Example Applications

The Mini EVK includes the following example applications:

- *MGDemo*
- *MGSwitch*
- *MGLight*
- *MGKeyboard*
- *Monitoring & Control*

The MGDemo, MGSwitch, MGLight, and MGKeyboard applications are Neuron C applications that you can load into your evaluation boards and use to create simple LONWORKS networks. For more information on these applications, see the *Neuron C Example Applications* section below.

These examples use the Interoperable Self-installation (ISI) library. ISI is an application-layer protocol that allows installation of devices and connection management without using a separate network management tool. For more information on ISI, see the *ISI Protocol Specification* and *ISI Programmer's Guide* documents.

The Monitoring & Control Example Application is a C# application that you can use to monitor and control evaluation boards running the MGDemo application. You do not need to load the Monitoring & Control Example Application into the evaluation boards, as with the Neuron C example applications, but you do need a 3150 EVB with the MGDemo example loaded to use all the features of the Monitoring & Control Example Application. For more information on the Monitoring & Control Example Application, see *Monitoring & Control Example Application* on page 29.

Neuron C Example Applications

You can use the Neuron C example applications to demonstrate how to use Neuron C to interface with I/O hardware, how to use network variables in Neuron C to communicate on a LONWORKS network, and how to use the ISI library to install devices on a LONWORKS network.

You can load the Neuron C example applications into up to 32 evaluation boards on a LONWORKS network (the ISI protocol supports up to 200 devices, but the version used in the evaluation boards supports up to 32), connect the evaluation boards via the ISI protocol, and then use the applications to exchange data through input and output network variable updates between the evaluation boards. These steps are summarized below. Exact details are provided in the sections following this summary:

1. The 3150 EVB is pre-loaded with the MGDemo application, and the 3120 EVB is pre-loaded with the MGSwitch application. You can use these applications, or you can use the MGLight or MGKeyboard applications that are also included with the Mini EVK. If you loaded a different application into either of you EVBs, or if you want to change the demo application, start the Mini Application, and then load one of the example applications (MGDemo, MGSwitch, MGLight, or MGKeyboard) into each EVB, as described in Chapter 2, *Using the Mini Application*.

2. If you are using an FT Mini EVK, skip to the next step. If you are using a PL Mini EVK and this is the first time you use a PL EVB or if you have just reloaded an example application, **LED1** will flash, indicating that the EVB is in CENELEC configuration mode. The CENELEC access protocol is a European-standard protocol for controlling access to a power line used for communication. It is required for PL-20 devices in most of Europe, but is not required or typically used outside of Europe. If **LED1** is flashing, press the **SW8** button to enable or disable CENELEC. **LED8** indicates the CENELEC access protocol status—if it is on the CENELEC access protocol is enabled, if it is off the CENELEC access protocol is disabled. When you have made a selection, press the **SW1** button to confirm your selection and exit CENELEC configuration mode. Be sure to set all devices on a power line channel to the same mode. You will not be able to load another application into the PL EVB, or perform any other network operations, until you have made a selection and exited CENELEC configuration mode. See Chapter 8 of the *PL 3120 / PL 3150 Smart Transceiver Data Book* for more information on the CENELEC protocol.

3. Push one of the push buttons labeled **SW5 – SW8** on a MiniGizmo attached to an evaluation board with the MGDemo application loaded, or push the button labeled **SW8** on a MiniGizmo attached to an evaluation board with the MGSwitch or MGLight applications loaded. This starts a new connection. The buttons used to start the connection are referred to as the *Connect buttons*, and the LEDs next to the Connect buttons are referred to as the *Connect lights*.

   The Connect light next to the Connect button that you pressed will start blinking. The Connect lights on devices that can join the connection will also start blinking. The MGKeyboard example application does not have a Connect button since it automatically connects without waiting for the installer to push a button. You can skip to step 5 if you are only connecting an MGKeyboard device.

4. Push one of the Connect buttons next to a blinking Connect light on any of the other devices to join the connection. The device's Connect light, as well as the Connect light on the evaluation board used to initiate the connection in step 2, will both illuminate without flashing, indicating they are ready to join the connection.

   Repeat this step until you have added all the devices you want to the connection.

5. Push the Connect button that you used to initiate the connection in step 2 again. This completes the connection.

6. You can now use the example applications. You can also use the Monitoring & Control Example Application to monitor a 3150 EVB running the MGDemo application.

MGSwitch and MGLight Example Applications

You can use the MGSwitch and MGLight example applications to demonstrate how simple switch and lamp devices work on a LONWORKS network, where the switch devices are used to activate or de-activate the lamp devices, or to set the lighting level for the lamp devices. When you use these applications, the evaluation boards running the MGSwitch application represent the switch devices. The evaluation boards running the MGLight application represent the lamp devices, and will respond to output from the MGSwitch devices.

To use the MGSwitch and MGLight applications, follow these steps:

1. Start the Mini Application and load MGSwitch into one or more of the evaluation boards, as described in Chapter 2, *Using the Mini Application*. Then, load MGLight into one or more evaluation boards. You can optionally load MGDemo into a 3150 EVB and use it with MGSwitch or MGLight.

2. Press the button labeled **SW8** on a MiniGizmo attached to any of the evaluation boards used in step 1 to start a new connection. It does not matter which evaluation board you use. For these example applications, the button labeled **SW8** is the Connect button, and the LED next to the **SW8** button is the Connect light. The Connect light will start blinking to indicate that the connection has been initiated. The Connect lights on the other devices that can join the connection will also start blinking.

   NOTE: You can press and hold the Connect button for 8 seconds to cancel the connection.

3. Choose a device you want to add to the connection, and push the device's Connect button to add the device to the connection. The device's Connect light, as well as the Connect light on the device you used to initiate the connection in step 2, will both illuminate without flashing, indicating they are ready to join the connection.

   Repeat this step for each device you want to add to the connection.

4. Push the first Connect button on the device used in step 2 to complete the connection

5. You can now use buttons **SW1** – **SW7** on the evaluation boards running the MGSwitch application to activate or de-activate the LEDs on the evaluation boards running the MGLight application, just as the switch would be used to activate or de-activate the lamp devices it is connected to.

   For example, if you press **SW3** on the MGSwitch MiniGizmo, then **LED1**, **LED2,** and **LED3** will activate on the MiniGizmos attached to the evaluation boards running the MGLight application. Then, you can press **SW3** again on the MGSwitch MiniGizmo to de-activate the LEDs, or press any other of the other buttons on the MGSwitch device to change to a different lighting level.

6. If you included an evaluation board running the MGDemo application in the connection, it will also respond to the MGSwitch application. The I/O

LED on the MiniGizmo attached to the evaluation board running the MGDemo application will activate or de-activate each time the MGSwitch MiniGizmo sends an update.

In addition, the MGDemo application implements **LED1** – **LED4** as simple lights that cannot be dimmed. These LEDs will be illuminated whenever the lighting level on the MGSwitch evaluation board is more than zero.

You can remove an evaluation board running the MGSwitch or MGLight application from a connection by pressing and holding the device's Service button until the Reset light blinks (approximately 10 seconds). To remove an evaluation board running the MGDemo application from a connection, press and hold the Connect button for the connection for approximately 10 seconds.

MGDemo Example Application

You can use the MGDemo example application to demonstrate how to use Neuron C to interface with I/O hardware, how to use network variables in Neuron C to communicate on a LONWORKS network, and how to use the ISI library to install devices on a LONWORKS network. You can use the MGDemo application to interoperate with evaluation boards running the MGSwitch, MGLight, and MGKeyboard applications, as well as with other 3150 EVBs running the MGDemo application. In addition, you can use the Monitoring & Control Example Application to monitor a 3150 EVB running the MGDemo application.

The MGDemo example application implements three types of I/O:

- A temperature sensor reads the local temperature, and stores this data in a pair of output network variables.
- A piezo buzzer can be controlled with a pair of input network variables.
- Four switch/light pairs implement a switch that is hard-wired to a local light, where each pair may be connected to remote switches, remote lights, or remote switch/light pairs, on other devices.

  The four pairs consist of one switch and one light each. Each switch is implemented with a SFPTclosedLoopSensor functional block, and each light is implemented with a SFPTclosedLoopActuator functional block. When you use the MGDemo example application in a self-installed environment, each pair is coupled and can only be connected as one atomic unit. That is, each **LED1** – **LED4** light emulates a light bulb that is physically connected to the corresponding **SW1** – **SW4** switch, while remote switches or remote lights (or remote switch/light pairs) can be added to extend the functionality to a lighting system. When you use the MGDemo example application in a managed environment, you can independently connect each of the switch and light functional blocks, demonstrating the additional flexibility provided in managed networks. For example, you can use the MGDemo example application with the LonMaker tool and connect each of the eight individual functional blocks independently from each other.

To use the MGDemo application, follow these steps:

1. Start the Mini Application, and load the MGDemo application into one or more of the 3150 EVBs as described in Chapter 2, *Using the Mini Application* (this application will not run in a 3120 EVB). The evaluation

board used in this step is referred to as the *MGDemo board* in this section.

2. To demonstrate monitoring and controlling a device from your computer, run the Monitoring & Control Example Application as described in the *Monitoring & Control Example Application* on page 29. You do not need to connect the evaluation board to any other evaluation boards (as described in steps 3 – 5 of this procedure) to use the Monitoring & Control Example Application.

3. Push any of the buttons labeled **SW5, SW6, SW7** or **SW8** on the MGDemo board to start a new connection. These buttons are the Connect buttons for MGDemo, and the LED next to the buttons are the Connect lights. The Connect light next to the Connect button that you press will start blinking. The Connect lights on the other devices that can join the connection also start blinking.

   NOTE: You can press and hold the Connect button for 8 seconds to cancel the connection.

4. Push the blinking Connect buttons on any of the other devices to add those devices to the connection. The device's Connect light, as well as the Connect Light on the MGDemo board, will both illuminate without flashing, indicating they are ready to join the connection.

   Repeat this step for any other devices that you want to add to the connection.

   NOTE: You can press and hold the Connect button on a device for 8 seconds to remove the device from the connection.

5. Push the Connect button on the MGDemo board used in step 3 to complete the connection

6. The Connect buttons that you use to create a connection in steps 3 and 4 determine which button and LED will be used for I/O for the connection, as listed in Table 3.1.

Table 3.1 MGDemo Application

Connect Button	I/O Push Button	I/O LED
SW5	SW1	LED1
SW6	SW2	LED2
SW7	SW3	LED3
SW8	SW4	LED4

7. You can create ISI connections to other evaluation boards running the MGDemo, MGSwitch, or MGLight applications. When you press the I/O button on one of the evaluation boards running the MGDemo application,

the I/O LEDs on all other connected evaluation boards running the MGDemo application will activate. When you press the I/O button again, the I/O LEDs will de-activate. For more information on the MGSwitch and MGLight applications, see *MGSwitch and MGLight Example Application* on page 25.

You can also use the MGDemo application to interoperate with evaluation boards running the MGKeyboard application, as described in the next section, *MGKeyboard Example Application*.

NOTE: The MGDemo application uses **LED1** and **LED2** to signal network and connection addressing conflicts. When a network address conflict is detected and resolved, **LED1** will start flashing. When flashing, **LED1** will not reflect any other input received from the network. Press the **SW1** button to cancel the notification and return **LED1** to its normal state. When a connection conflict is detected and resolved, **LED2** will start flashing, and **LED2** will not reflect any other input received from the network. Press the **SW2** button to cancel the notification and return **LED2** to its normal state.

MGKeyboard Example Application

You can use the MGKeyboard application with the MGDemo application to demonstrate the use of automatic network variable connections with the ISI protocol. The MGKeyboard application implements a simple musical keyboard using the 8 push buttons on the MiniGizmo. To use the MGKeyboard application, follow these steps:

1. Start the Mini Application and load MGKeyboard into a 3120 EVB as described in Chapter 2, *Using the Mini Application*.

2. Load the MGDemo application into a 3150 EVB as described in the previous section.

3. Wait for the ISI T_{auto} protocol timer to expire. By default, this timer takes 12 minutes 30 seconds to expire. When the timer expires, the MGKeyboard application will start an automatic connection process. The MGDemo application will automatically join this connection. No manual intervention is needed in this case.

   The T_{auto} protocol timer is provided to avoid race conditions when a network segment, or the entire site, is powered up. The timer is only relevant the first time the device powers up with a new application, or when it is connected to a new network. Once the connection has been created, it will be immediately operational following a power-cycle or reset. For more information on the T_{auto} protocol timer, see the *ISI Programmer's Guide* and the *ISI Protocol Specification.*

4. Once MGKeyboard has connected with MGDemo, press any of the buttons labeled **SW1 – SW8** on the MiniGizmo connected to the evaluation board running the MGKeyboard application. This sends a network variable update that activates the piezo buzzer on the evaluation board running the MGDemo application. Each of the buttons causes the piezo buzzer to use a different frequency.

Monitoring & Control Example Application

You can use the Monitoring & Control Example Application to monitor a 3150 EVB running the MGDemo example application. You can also use this application to monitor ISI messages from any devices on the same network. The Monitoring & Control Example Application is a Windows application written in Microsoft Visual C#. It requires Microsoft .NET Framework 1.1 to run. The Microsoft .NET Framework is installed on your computer when you install the Mini EVK software if you do not already have it. The Monitoring & Control Example Application uses the OpenLDV API to monitor and control network variable and ISI messages. The OpenLDV API is also installed on your computer when you install the Mini EVK software if you do not already have it. The Monitoring & Control example is installed in the LONWORKS **MiniKit\Examples\Monitor & Control** folder. This example works with any evaluation boards on your network that have the MGDemo application loaded.

This section describes how to use the Monitoring & Control Example Application to monitor a 3150 EVB and to monitor ISI messages. See Appendix B, *Monitoring & Control Application Overview*, for information on how this example is implemented.

To use the Monitoring & Control Application, follow these steps:

1. Load the MGDemo application into a 3150 EVB as described earlier in this chapter.

2. Open the **Echelon Mini EVK** program folder and then select **Monitoring & Control Example Application.** The window shown in Figure 3.1 opens.

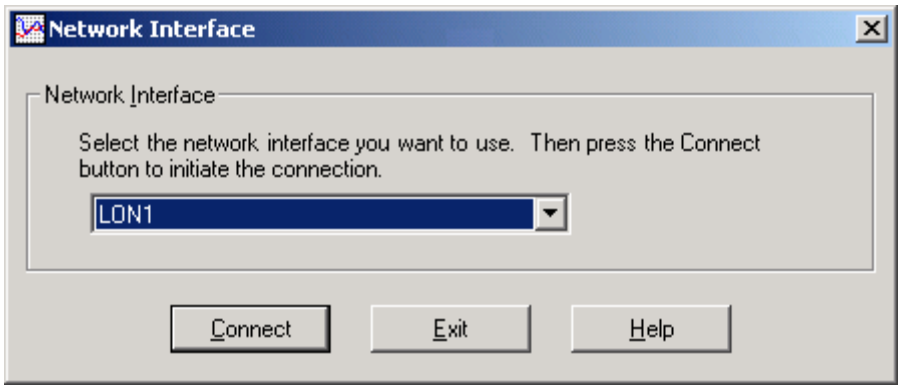

Figure 3.1 Network Interface Window

3. Select a network interface, and then click **Connect**. The window shown in Figure 3.2 opens.

   You can use the U10 or U20 USB Network Interface included with the Mini EVK, or you can use another network interface such as a PCC-10, PCLTA-20, PCLTA-21, *i*.LON 10 Ethernet Adaptor, or *i*.LON 100 Internet Server. If you are using the U10 or U20 USB Network Interface included with the Mini EVK and do not have any other network interfaces installed on your computer, select **LON1**.

WARNING: If you want to use a PCC-10, PCLTA-20, or PCLTA-21 network interface, you must configure it as a layer 5 interface. See the *Mini EVK Hardware Guide* for more information on this.

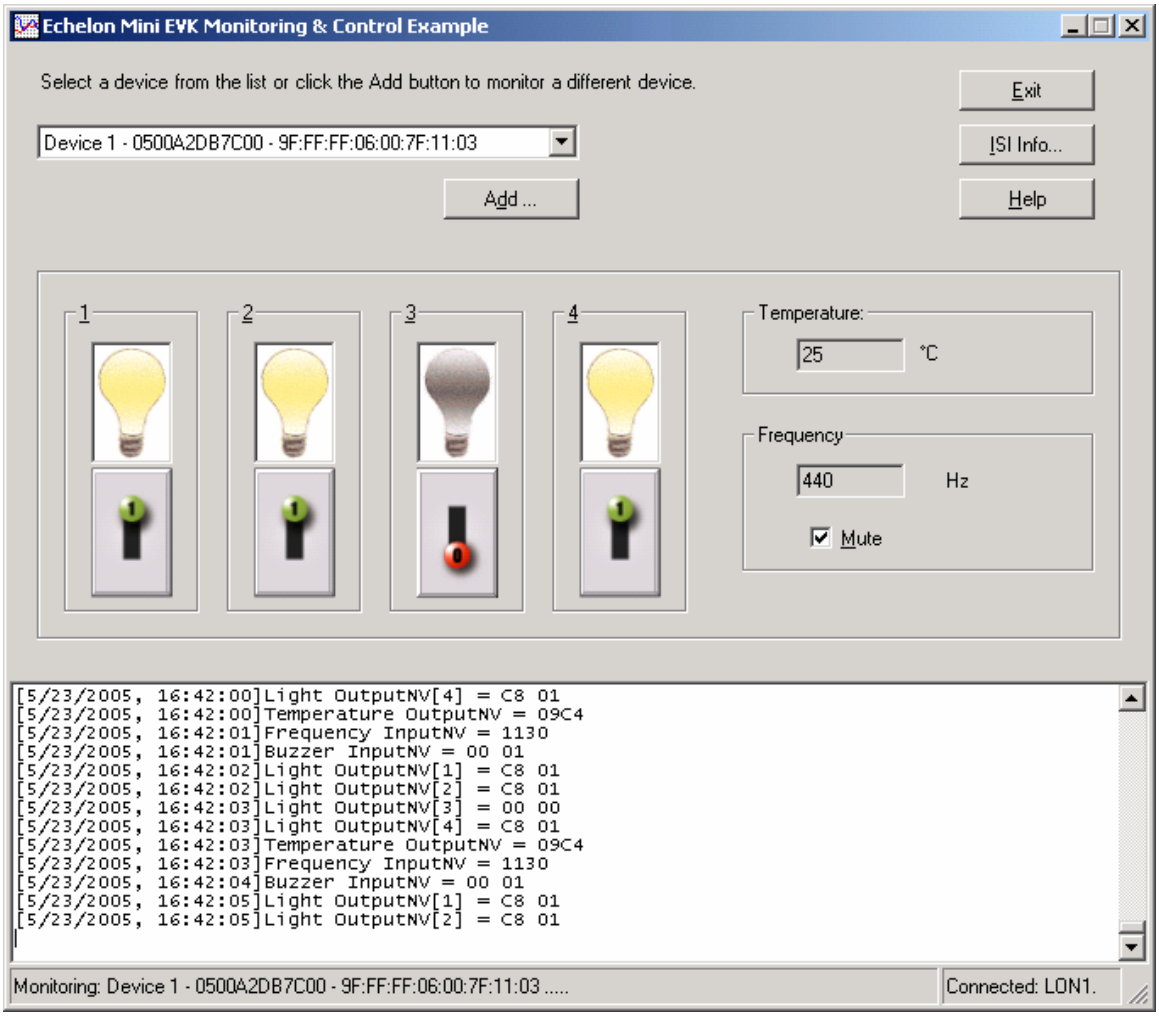

Figure 3.2 Monitoring & Control Application Main Window

4. Select a target device running the MGDemo application in the **Device** box. Click the arrow to view any devices that you have recently added or that have been automatically discovered through the ISI protocol. You can select one of these devices, or you can click the **Add** button to add a new device. If you add a new device, the window shown in Figure 3.3 opens.

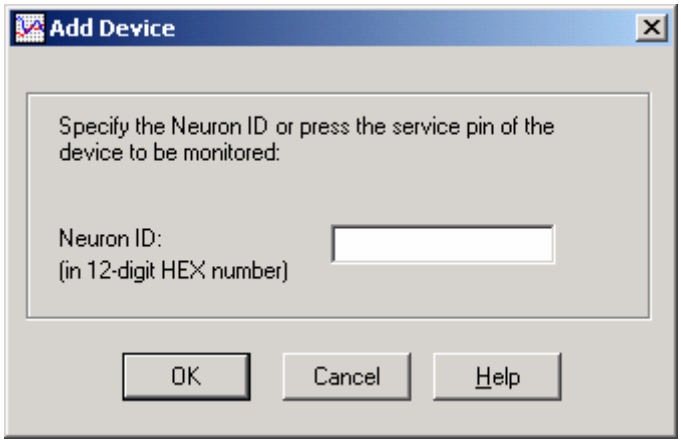

Figure 3.3 Add Device Dialog

If you clicked the **Add** button, enter the Neuron ID of the target device in the **Neuron ID** box, and then click **OK** to return to the window shown in Figure 3.2. The Neuron ID is a unique 48-bit (12-hex digit) identifier contained in every LONWORKS device. The Monitoring & Control Example Application uses the Neuron ID to communicate with your selected device. For more information on Neuron IDs, see *Introduction to the LONWORKS System* in the **Echelon Mini EVK** program folder.

If you do not know the target device's Neuron ID, you can acquire it from the device by pressing the Service button on the device. The Service button is typically a push button that causes the device to broadcast its Neuron ID on the LONWORKS network.

To add a device using the Service button, press the device's Service button. On a Mini EVB, the service pin button is the black button labeled "**SERVICE**." When the Monitoring & Control Example Application receives the service pin message, it displays the Neuron ID in the **Neuron ID** box on the Add Device dialog. Click **OK** to add the device and return to the window shown in Figure 3.2.

6. The window shown in Figure 3.2 displays the current temperature reading returned by the evaluation board's temperature sensor, and the frequency used for the evaluation board's piezo buzzer sound outputs.

   You can use the switch buttons below the light bulbs labeled 1, 2, 3, and 4 to activate and de-activate **LED1** – **LED4** on the evaluation board. In Figure 3.1, **LED1**, **LED2**, and **LED4** are activated. You can activate or de-activate each LED by clicking the switch button below the light bulb. For example, click the switch button below Light Bulb 1 to de-activate **LED1**.

ISI Information

You can use the Monitoring & Control Example Application to monitor ISI domain resource usage messages from any ISI devices. To see ISI information, click **ISI Info** from the Monitoring & Control Example Application's main window. The window shown in Figure 3.4 opens.

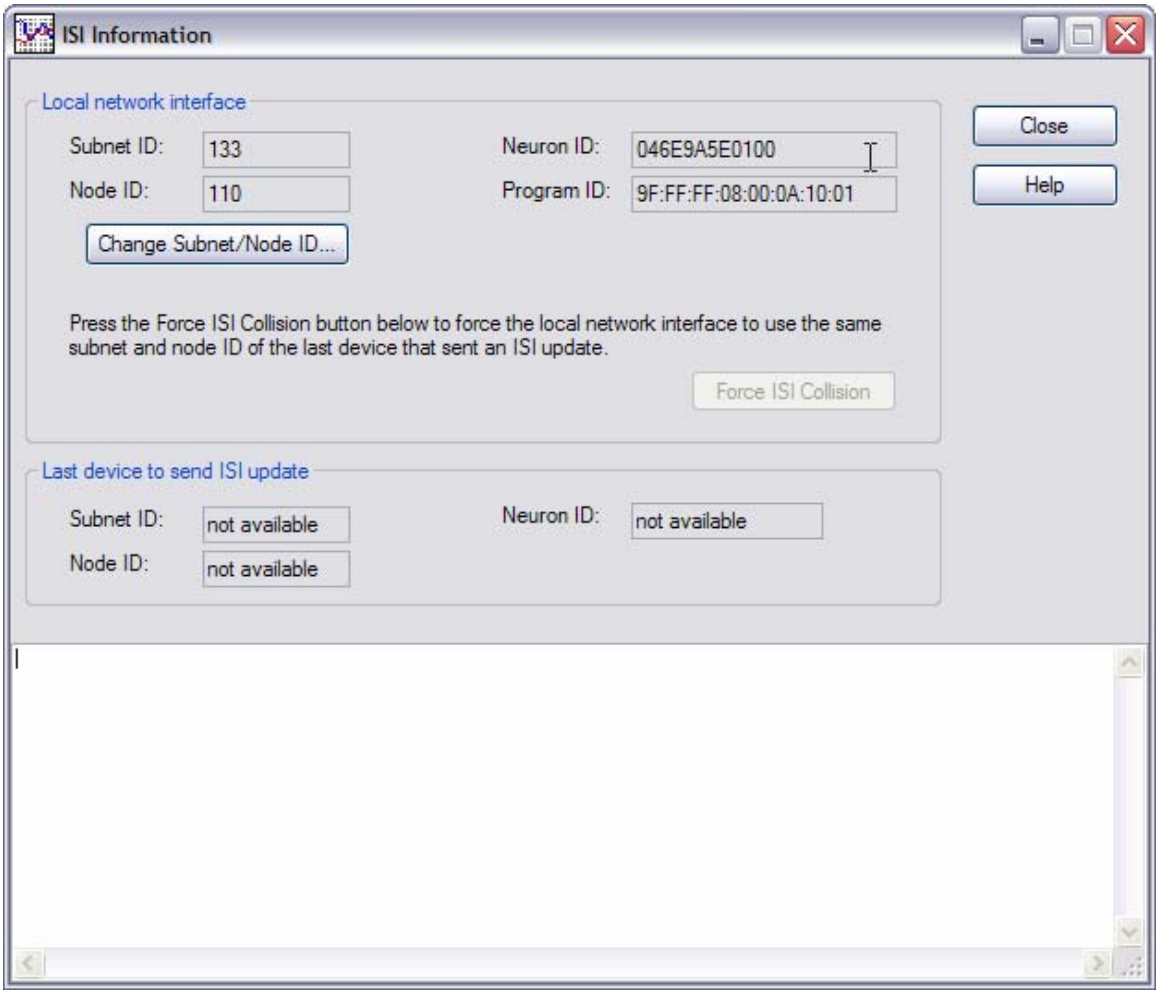

Figure 3.4 ISI Information Window

The subnet ID, node ID, and Neuron ID of the network interface used by the Monitoring & Control Application are displayed in the **Local Network Interface** box. The subnet ID, node ID, and Neuron ID of the last device to send an update over the channel are displayed in the **Last Device to Send Update** box.

You can force an ISI collision to see how the ISI protocol recovers from ISI collisions. To force the local network interface to change its subnet and node ID, click **Force ISI Collision**. The ISI protocol enables the ISI library to detect and resolve network address conflicts, thereby maintaining a self-installed network consisting of devices with unique addresses without the use of a central database. During the lifetime of an ISI-compliant self-installed network, it is possible (though unlikely) that multiple devices may temporarily use the same subnet/node address. The **Force ISI Collision** button demonstrates the effect of this by forcing the Mini computer's network interface to use the same subnet/node address as the last device to send it an update. The remote device (i.e. the last device to send an update) will recognize the situation, take defensive action, and allocate itself a new subnet/node address. You can use ISI Information window or Echelon's LonScanner™ Protocol Analyzer to observe the resulting message exchange. You can also observe the action by monitoring the

IsiUpdateUserInterface() and **IsiDiagnostics()** calls on the remote device. Consult the *ISI Programmer's Guide* for more information on these calls.

You can also manually change the Mini computer's network interface subnet and node IDs by following these steps:

1. Click **Change Subnet/Node ID**. This opens the dialog shown in Figure 3.5.

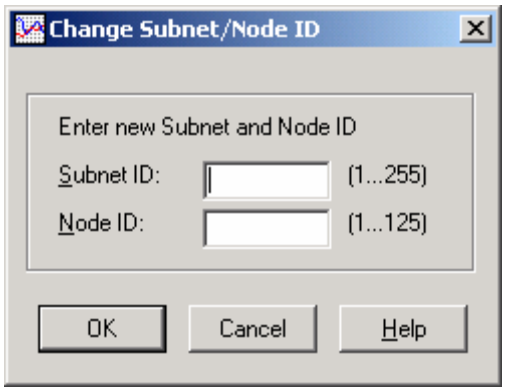

Figure 3.5 Change Subnet/Node ID Dialog

2. Enter the subnet ID in the **Subnet ID** box, and the node ID in the **Node ID** box. If the device you are monitoring is on a power line channel (such as a PL Evaluation Board), the subnet ID must be in the range of 128 – 192. If the device you are monitoring is on a TP/FT-10 channel (such as an FT Evaluation Board), the subnet ID must be in the range of 64 – 125. In most cases, each subnet supports up to 127 devices. However, node IDs 126 and 127 are reserved for managed tool use, so the node ID you enter must be in the range of 1 – 125.

3. Click **OK** to save and apply the change.

4

Developing a Neuron C Application

This chapter introduces the Neuron C Version 2.1 programming language, and provides an overview on how to use the LONWORKS platform and the Neuron C programming language to construct interoperable devices and systems. The chapter also introduces key concepts of Neuron C such as event-driven scheduling, network variables, configuration properties, and functional blocks.

What Is Neuron C?

Neuron C Version 2.1 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, and event-handling extensions to ANSI C, which make it a powerful tool for the development of LONWORKS applications. Following are a few of the extensions to the ANSI Standard C language:

- A new network communication model, based on *functional blocks* and *network variables*, simplifies and promotes data sharing between like or disparate devices.
- A new network configuration model, based on functional blocks and *configuration properties*, facilitates interoperable network configuration tools.
- A new 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* supports the powerful I/O capabilities of Neuron Chips and Smart Transceivers. Powerful *event-driven programming* extensions, based on new **when** statements, provide easy handling of network, I/O, and timer events.

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 system firmware support to provide the look and feel of a C program, 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 LonMaker Integration 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, each of which provides 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 profiles.

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 configuration properties, the standard types are called standard configuration property types (SCPTs). For network variables, the standard types are called standard network variable types (SNVTs). For functional blocks, the standard types are called standard functional profiles. If you cannot find standard types or profiles that meet your requirements, Neuron C also provides full support for user network variable types (UNVTs), user configuration property types (UCPTs), and user functional profiles.

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 common with ANSI C's **main()** function. Instead, a Neuron C application uses *when* statements to specify application code to be executed in response to various system events, much in the way of an interrupt handler or .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.

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 *timer*. Timers can be declared and manipulated like variables. When a timer expires, the system 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 provides many built-in *I/O objects*. These I/O objects are standardized I/O device drivers for the Neuron Chip or Smart Transceiver I/O hardware. Each I/O object fits into the event-driven programming model. A function-call interface is provided to interact with each I/O object. I/O models may be used with the Neuron Chip or Smart Transceiver's **IO***n* pins, and include simple models such as bit I/O and complex timer/counter models, models for serial I/O, and many other models for commonly used I/O tasks.

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;
```

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 the *Timers* chapter 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 discussed in great detail in the *Functions* chapter 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 or a when 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.

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 to zero (variables declared with the **eeprom** or **config** class, as well as configuration properties declared with the **config_prop** or **cp_family** keywords, are only initialized when the application image is first loaded).

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. 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. The Neuron C language does not permit the use of **const** with **auto**.
- **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 **eeprom** 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.

- **eeprom**
- **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 **eeprom** and **config** variables are also initialized at load time, except when the **uninit** class modifier is included in these variable definitions.

Automatic variables cannot be declared **const** because Neuron C does not implement initializers in declarations of automatic variables.

Global RAM variables are initialized at reset (that is, 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 to zero costs no extra code space, as it is a firmware feature.

Initialization of I/O objects, input network variables (except for **eeprom**, **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, nor are their values preserved when the program execution leaves the local scope.

Neuron C Declarations

Both ANSI C and Neuron C support declarations of the following:

Declaration	*Example*
•Simple data items	**int a, b, c;**
•Data types	**typedef unsigned long ULONG;**
•Enumerations	**enum hue {RED, GREEN, BLUE};**
•Pointers	**char *p;**
•Functions	**int f(int a, int b);**
•Arrays	**int a[4];**
•Structures and unions	**struct s {** **int field1;** **unsigned field2 : 3;** **unsigned field3 : 4;** **};**

In addition, Neuron C Version 2.1 supports declarations of the following:

Declaration	*For Example:*
•I/O objects	**IO_0 output oneshot relay_trigger;**
•Timers	**mtimer led_on_timer;**
•Network variables	**network input SNVT_temp nviTemperature;**

• Configuration Properties	**SCPTdefOutput cp_family cpDefaultOut;**
• Functional Blocks	**fblock SFPTnodeObject { … } myNode;**
• Message tags	**msg_tag command;**

Getting Started with Neuron C

This section introduces Neuron C with a series of examples. Neuron C concepts are introduced one at a time, and each concept is integrated into the following examples:

- A digital sensor that senses a push button.
- An analog sensor that reads a temperature value.
- A digital actuator that controls an LED.
- A serial actuator that controls a device via a serial connection.

The first four examples in the *Input/Output* section below are example functions used in the remaining examples, which are complete Neuron C applications that you can copy to Windows Notepad or another editor, save to a file with a **.nc** extension, and then compile and download the file to one of the evaluation boards included with the Mini EVK.

Note that the source file must not use the **.c** file extension, as is common to ANSI-C programmers. Instead, the **.nc** file extension is recommended for Neuron C source code. When compiling code packaged in a file with a **.c** file extension, the Neuron C Compiler classifies the source as "Pure C" and disables most of the Neuron C extensions. The resulting image file cannot be loaded into a Neuron Chip or Smart Transceiver. The Pure C feature may be used with the NodeBuilder Development Tool, but not the Mini EVK, to create user-defined function libraries.

The digital sensor, analog sensor, and digital actuator examples all use the I/O hardware on the MiniGizmo I/O board, so the MiniGizmo must be attached to your evaluation board to run these examples. The serial actuator example uses the serial interface on the evaluation boards. To use this interface, insert the JP201 I010 jumper on the evaluation board to enable the serial output port, connect the board to your computer using a male DB-9 to female DB-9 serial extension cable (pins wired straight through), and then run Windows HyperTerminal on your computer to monitor the serial output. Configure HyperTerminal for direct connection to your serial port (typically COM1 or COM2), 4800 bps, 8 data bits, no parity, one stop bit, and no flow control. Compatible serial extension cables include the Radio Shack #26-117 6-Ft. Serial RS-232C Cable, Outpost.com (Fry's Electronics) #2007427 Serial Mouse Extension Cable, cdw.com #086872 Belkin Serial Extension DB9M toDB9F 6' Cable, or the Belkin F2N209-06-T Serial Extension DB9M to DB9F 6' Cable.

Input/Output

A Neuron Chip or Smart Transceiver may be connected to one or more physical I/O devices via up to 12 versatile I/O pins. Examples of simple I/O

devices include temperature and position sensors, valves, switches, and LED displays. Neuron Chips and Smart Transceivers can also be connected to other microprocessors. The Neuron firmware implements numerous *I/O objects* that manage the interface to these devices for a Neuron C application. I/O objects are discussed in detail in Chapter 2, *Focusing on a Single Device*, of the *Neuron C Programmer's Guide* and in the *Neuron C Reference Guide*.

To set up I/O, declare the I/O objects that monitor and control the Neuron Chip or Smart Transceiver I/O pins, named **IO_0** – **IO_10** or **IO_11** (depending on the Neuron Chip or Smart Transceiver model). To perform I/O, you use the following built-in I/O functions: **io_in()**, **io_out()**, **io_set_direction(), io_select()**, **io_change_init()**, and **io_set_clock()**. The **io_out_request()** function is used to perform I/O with the **parallel** I/O object.

For more information on Neuron C I/O, see the *Neuron C Reference Guide* and the Echelon engineering bulletins listed in Table 4.1. These engineering bulletins are available at www.echelon.com/mini.

Table 4.1 Neuron C I/O Engineering Bulletins

Document Title	Contents	Part Number
Analog-To-Digital Conversion With the Neuron Chip	Describes some of the more popular analog to digital (A/D) conversion schemes available for use with a Smart Transceiver or Neuron Chip. Provides schematics, parts lists and code examples.	005-0019-01
Driving a Seven Segment Display with the Neuron Chip	Describes how a Smart Transceiver or Neuron Chip can be used to drive a seven-segment display controller chip, the Motorola MC14489, using the Neurowire device. Includes Neuron C software drivers to display decimal numbers from binary data.	005-0014-01
EIA-232C Serial Interfacing with the Neuron Chip	Describes a simple level conversion circuit to allow a Smart Transceiver or Neuron Chip to communicate with RS-232C devices. Also includes Neuron C software to drive an RS-232C CRT terminal.	005-0008-01
Neuron Chip Quadrature Input Function Interface	Describes the use of the quadrature device in a Smart Transceiver or Neuron Chip to interface to external devices such as shaft encoders.	005-0003-01
Parallel I/O Interface to the Neuron Chip	Describes hardware and software to interface a Smart Transceiver or Neuron Chip to a microprocessor using the parallel I/O port.	005-0021-01

Document Title	Contents	Part Number
Scanning a Keypad with the Neuron Chip	Describes how a Smart Transceiver or Neuron Chip can be used to scan a simple 16-key switch matrix to provide a numeric or special-function keyboard without the use of a keyboard encoder.	005-0004-01
Using the Hardware Serial Peripheral Interface (SPI) and Neurowire I/O Object Models to Interface with Peripherals and Microcontrollers	Describes communications between Smart Transceivers or Neuron Chips and other microcontrollers for designs that intend to make use of the SPI interface for simpler applications and also for understanding how the SPI interfaces are implemented in the Smart Transceivers and Neuron Chips. Neuron C code examples of an SPI interface are explained in this engineering bulletin, and the source code is available for download.	005-0165

Digital Sensor Example

The following example function reads the push buttons on a MiniGizmo and returns TRUE if any are pressed. The push buttons are connected to a 74HC165 8-bit parallel-in/serial-out shift register. Data is shifted on the Smart Transceiver **IO4** (clock) and **IO5** (data) pins, with a latch strobe on **IO6** (active low to capture). The push button readings are debounced under application control.

```
// Configure the I/O pins
IO_4 input bitshift numbits(8) clockedge(-) ioButtons;
IO_6 output bit ioButtonLd = 1;

// Read the MiniGizmo buttons
boolean GetButton(void) {
    // Latch the button inputs
    io_out(ioButtonLd, 0);
    io_out(ioButtonLd, 1);

    // Shift in and return TRUE if any buttons are pressed
    return !((unsigned) io_in(ioButtons) == 0xFF);
}
```

Analog Sensor Example

The following example function reads the temperature sensor on a MiniGizmo. The **enable_io_pullups** compiler directive is required when using the MiniGizmo temperature sensor. The temperature sensor is based on a 1-Wire ("Touch I/O") Dallas DS18S20 digital thermometer. The Touch I/O interface is connected to the Smart Transceiver **IO7** pin.

```
#pragma enable_io_pullups

// Configure the I/O pins
IO_7 touch ioThermometer;
IO_10 output serial baud (4800) ioSerialOut;
```

```
#define DS18S20_SKIP_ROM    0xCCu
#define DS18S20_CONVERT     0x44u
#define DS18S20_READ        0xBEu

// Get a temperature reading from the Touch temperature sensor
SNVT_temp_p GetTemperature(void) {
    union {
        SNVT_temp_p snvtTempP;
        unsigned    Bytes[2];
    } CurrentTemperature;
    CurrentTemperature.snvtTempP = 32767l;

    if (touch_reset(ioThermometer)) {
        (void) touch_byte(ioThermometer, DS18S20_SKIP_ROM);
        (void) touch_byte(ioThermometer, DS18S20_READ);

        CurrentTemperature.Bytes[1]
            = touch_byte(ioThermometer, 0xFFu);
        CurrentTemperature.Bytes[0]
            = touch_byte(ioThermometer, 0xFFu);

        if (touch_reset(ioThermometer)) {
            // Scale the raw reading
            CurrentTemperature.snvtTempP *= 50l;

            // start the next conversion cycle:
            (void) touch_byte(ioThermometer, DS18S20_SKIP_ROM);
            (void) touch_byte(ioThermometer, DS18S20_CONVERT);
        } else {
            CurrentTemperature.snvtTempP = 32767l;
        }
    }
    return CurrentTemperature.snvtTempP;
}
```

Digital Actuator Example

The following example function sets the state of the eight LEDs on a MiniGizmo. The LEDs are connected to a 74HC595 8-bit serial-in/parallel-out shift register. Data is shifted on **IO2** (clock) and **IO3** (data), a rising edge on **IO1** strobes data into the latch. LEDs are driven active-low

```
// Configure the I/O pins
IO_1 output bit ioLEDLd = 1;
IO_2 output bitshift numbits(8) ioLEDs;

// Set the MiniGizmo LEDs to all on or all off
void SetLED(const boolean state) {
    // Shift out the LED value
    io_out(ioLEDs, state ? 0 : 0xFF);

    // Latch the new value
    io_out(ioLEDLd, 0);
    io_out(ioLEDLd, 1);
}
```

Serial Actuator Example

The following example function sends a string to the serial port on an EVB. The JP201 I010 jumper on the evaluation board must be inserted to enable the serial

output port. This connects the IO10 output to an EIA-232 line driver that is in turn connected to pin 2 as a serial data output on the RS-232 connector.

```
// Configure the I/O pins
IO_10 output serial baud (4800) ioSerialOut;

// Send a string to the serial port
const char errorString[] = "String too long.";
void PrintConsole(const char *message) {
    if (strlen(message) <= 100) {
        io_out(ioSerialOut, message, (unsigned) strlen(message));
    } else {
        io_out(ioSerialOut, errorString,
               (unsigned) strlen(errorString));
    }
}
```

Hello World Example

The following example sends "hello, world" to the serial port on a MiniGizmo when the device is reset.

```
#include <string.h>

#pragma num_alias_table_entries 2
#pragma run_unconfigured

// Configure the I/O pins
IO_10 output serial baud (4800) ioSerialOut;

// Send a string to the serial port
const char errorString[] = "String too long.";
void PrintConsole(const char *message) {
    if (strlen(message) <= 100) {
        io_out(ioSerialOut, message, (unsigned) strlen(message));
    } else {
        io_out(ioSerialOut, errorString,
               (unsigned) strlen(errorString));
    }
}

when (reset) {
    PrintConsole("hello, world\n\r");
}
```

Timers

You can create millisecond and second timers. The millisecond timers provide a timer duration of 1 – 64,000 milliseconds (or .001 – 64 seconds). The second timers provide a timer duration of 1 – 65,535 seconds. For more accurate timing of durations of 64 seconds or less, use the millisecond timer. These are separate from the two hardware timer/counters used for I/O in the Neuron core.

To set up a timer, you declare a millisecond or second timer. You can declare up to 16 timers. To start a timer, you set its value. To execute code when a timer expires, you create a when task that monitors the timer.

Digital Sensor and Serial Actuator Example

The following example application reads the MiniGizmo push buttons every 50 milliseconds and sends an “On” or “Off” value to the serial port if it has changed. A value of On means any of the buttons is pressed, and a value of Off means all the buttons are off. The timer code is highlighted in bold.

```
#include <string.h>

#pragma num_alias_table_entries 2
#pragma run_unconfigured

// Configure the I/O pins
IO_4 input bitshift numbits(8) clockedge(-) ioButtons;
IO_6 output bit ioButtonLd = 1;
IO_10 output serial baud (4800) ioSerialOut;

// Read the MiniGizmo buttons
boolean GetButton(void) {
    // Latch the button inputs
    io_out(ioButtonLd, 0);
    io_out(ioButtonLd, 1);

    // Shift in and return TRUE if any buttons are pressed
    return !((unsigned) io_in(ioButtons) == 0xFF);
}

// Send a string to the serial port
const char errorString[] = "String too long.";
void PrintConsole(const char *message) {
    if (strlen(message) <= 100) {
        io_out(ioSerialOut, message, (unsigned) strlen(message));
    } else {
        io_out(ioSerialOut, errorString,
               (unsigned) strlen(errorString));
    }
}

// Repeat every 50 milliseconds
mtimer repeating scanTimer = 50;

// Read the buttons when the timer expires
when(timer_expires(scanTimer)) {
    boolean button;
    static boolean lastButton;

    button = GetButton();
    if (button != lastButton) {
        lastButton = button;
        PrintConsole(button ? "On\n\r" : "Off\n\r");
    }
}
```

Analog Sensor and Serial Actuator Example

The following example application reads the temperature sensor once a second on a MiniGizmo and sends its value to the serial port if it has changed. The timer code is highlighted in bold.

```
#include <s32.h>
#include <string.h>
```

```
#pragma enable_io_pullups
#pragma num_alias_table_entries 2
#pragma run_unconfigured

// Configure the I/O pins
IO_7 touch ioThermometer;
IO_10 output serial baud (4800) ioSerialOut;

#define DS18S20_SKIP_ROM    0xCCu
#define DS18S20_CONVERT     0x44u
#define DS18S20_READ        0xBEu

// Get a temperature reading from the Touch temperature sensor
SNVT_temp_p GetTemperature(void) {
    union {
        SNVT_temp_p snvtTempP;
        unsigned    Bytes[2];
    } CurrentTemperature;
    CurrentTemperature.snvtTempP = 32767l;

    if (touch_reset(ioThermometer)) {
        (void) touch_byte(ioThermometer, DS18S20_SKIP_ROM);
        (void) touch_byte(ioThermometer, DS18S20_READ);

        CurrentTemperature.Bytes[1]
            = touch_byte(ioThermometer, 0xFFu);
        CurrentTemperature.Bytes[0]
            = touch_byte(ioThermometer, 0xFFu);

        if (touch_reset(ioThermometer)) {
            // Scale the raw reading
            CurrentTemperature.snvtTempP *= 50l;

            // start the next conversion cycle:
            (void) touch_byte(ioThermometer, DS18S20_SKIP_ROM);
            (void) touch_byte(ioThermometer, DS18S20_CONVERT);
        } else {
            CurrentTemperature.snvtTempP = 32767l;
        }
    }
    return CurrentTemperature.snvtTempP;
}

// Send a string to the serial port
const char errorString[] = "String too long.";

void PrintConsole(const char *message) {
    if (strlen(message) <= 100) {
        io_out(ioSerialOut, message, (unsigned) strlen(message));
    } else {
        io_out(ioSerialOut, errorString, (unsigned)
strlen(errorString));
    }
}

// Repeat every second
mtimer repeating readTimer = 1000;

// Read the temperature when the timer expires
when(timer_expires(readTimer)) {
    s32_type s32_temperature;
    SNVT_temp_p temperature;
```

```
    static SNVT_temp_p lastTemperature;
    char temperatureString[15];

    temperature = GetTemperature();
    if (temperature != lastTemperature) {
        lastTemperature = temperature;

        // Convert temperature to a string and print
        s32_from_slong(temperature / 100, &s32_temperature);
        s32_to_ascii(&s32_temperature, temperatureString);
        PrintConsole(temperatureString);
        // Print new line
        PrintConsole("\n\r");
    }
}
```

Digital Actuator Example

The following example application toggles the state of the LEDs on a MiniGizmo once a second. The timer code is highlighted in bold.

```
#pragma num_alias_table_entries 2
#pragma run_unconfigured

// Configure the I/O pins
IO_1 output bit ioLEDLd = 1;
IO_2 output bitshift numbits(8) ioLEDs;

// Set the MiniGizmo LEDs to all on or all off
void SetLED(const boolean state) {
    // Shift out the LED value
    io_out(ioLEDs, state ? 0 : 0xFF);

    // Latch the new value
    io_out(ioLEDLd, 0);
    io_out(ioLEDLd, 1);
}

// Repeat every second
mtimer repeating updateTimer = 1000;

// Toggle the LEDs when the timer expires
when(timer_expires(updateTimer)) {
    static boolean lastState;
    SetLED(lastState = !lastState);
}
```

Network Variables

The ANSI/EIA/CEA-709.1 (EN14908-1) protocol employs a data-oriented application layer that supports the sharing of data between devices, rather than simply the sending of commands between devices. With this approach, application data such as temperatures, pressures, states, and text strings can be sent to multiple devices—each of which may have a different application for each type of data. This results in smaller, simpler, and more maintainable applications than traditional command-based systems.

Applications exchange data 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 units, scaling, and encoding of the data. The LONWORKS platform defines standard types for network variables called *standard network variable types* (SNVTs). There are SNVT definitions for essentially every physical quantity, and other more abstract definitions tailored for certain industries and common applications. Device manufacturers may also create custom network variable types called *user network variable types* (UNVTs). You can view existing network variable type definitions and define new types using the NodeBuilder Resource Editor included with the Mini EVK, and you can also view the on the Web at types.lonmark.org. For more information on the NodeBuilder Resource Editor, see the *NodeBuilder Resource Editor User's Guide* included with the Mini EVK.

Network variables of identical type and length but opposite directions can be *connected* to allow the devices to share information. For example, an application on a lighting device could have an input network variable that was of the switch type, while an application on a dimmer-switch device could have an output network variable of the same type. A network tool or self-installation code on each device could be used to connect these two devices, allowing the dimmer switch to control the lighting device, as shown in the following figure. To send an update, the dimmer-switch application writes to its copy of the network variable. The Neuron firmware automatically propagates the update to the lighting application, which gets the new value from its local network variable. The application program does not require any explicit instructions for addressing, sending, receiving, managing, retrying, authenticating, or acknowledging network variable updates.

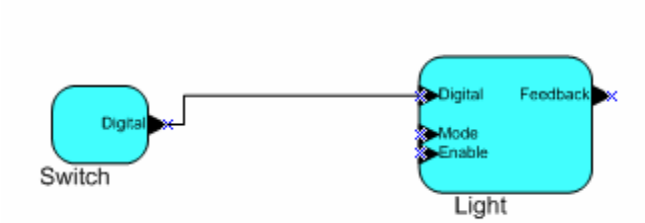

Figure 4.1 Network Variable Unicast Connection

The direction indicated by the triangle in the above figure indicates the direction of the network variable. A single network variable may be connected to multiple network variables of the same type but opposite direction. A single *network variable* output connected to multiple inputs is called a *fan-out connection* or a *multicast connection*. A single *network variable* input that receives inputs from multiple *network variable* outputs is called a *fan-in connection*. Figure 4.2 shows the same dimmer switch being used to control three lights using a fan-out connection:

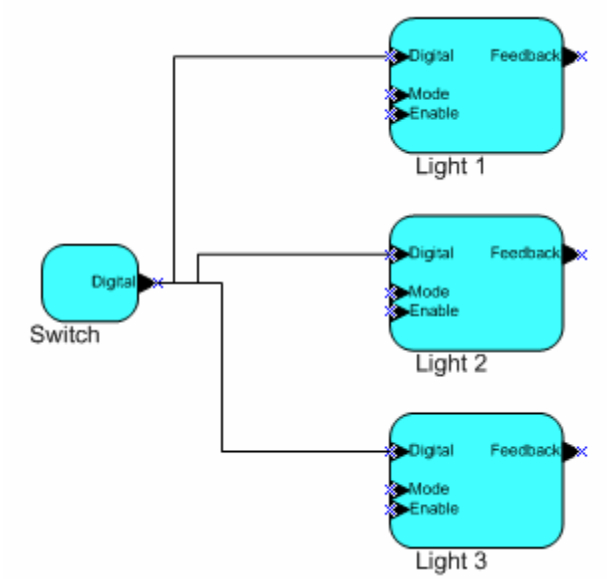

Figure 4.2 Network Variable Multicast Connection

Network variables greatly simplify the process of developing and installing distributed systems because devices can be defined, developed, and produced individually, then connected and reconnected easily into many new LONWORKS applications. Network variables are discussed in detail in Chapter 3, *How Devices Communicate Using Network Variables*, or the *Neuron C Programmer's Guide* and also in the *Neuron C Reference Guide*.

To use an output network variable, you declare and write to it much like other C variables. To use an input network variable, you declare and read from it, also like a C variable. In the case of an input network variable you can also respond to update events that occur when the network variable is updated over the network.

Digital Sensor Example

The following example application reads the MiniGizmo push buttons every 50 milliseconds and sends its value to a network variable if it has changed, replacing the console output in the previous digital sensor example. The output network variable is a **SNVT_switch** structure with value and state fields. A value of 200 and state of 1 means any of the buttons is pressed, and a value of zero and a state of zero means all the buttons are off. The network variable code is highlighted in bold. A network tool such as the LonMaker tool is required to test this application.

```
#pragma num_alias_table_entries 2
#pragma run_unconfigured

// Define the device interface
network output SNVT_switch nvoSwitch;

// Configure the I/O pins
IO_4 input bitshift numbits(8) clockedge(-) ioButtons;
IO_6 output bit ioButtonLd = 1;

// Read the MiniGizmo buttons
boolean GetButton(void) {
```

```
    // Latch the button inputs
    io_out(ioButtonLd, 0);
    io_out(ioButtonLd, 1);

    // Shift in and return TRUE if any buttons are pressed
    return !((unsigned) io_in(ioButtons) == 0xFF);
}

// Repeat every 50 milliseconds
mtimer repeating scanTimer = 50;

// Read the buttons when the timer expires
when(timer_expires(scanTimer)) {
    boolean button;
    static boolean lastButton;

    button = GetButton();
    if (button != lastButton) {
        lastButton = button;
        nvoSwitch.value = button ? 200U : 0;
        nvoSwitch.state = button ? 1 : 0;
    }
}
```

Analog Sensor Example

The following example application reads the MiniGizmo temperature sensor once a second and sends its value to a network variable if it has changed, replacing the console output of the previous analog sensor example. The output network variable is a **SNVT_temp_p** value, which is a fixed-point scalar value representing hundredths of degrees Celsius. The network variable code is highlighted in bold. A network tool such as the LonMaker Integration Tool is required to test this application.

```
#pragma enable_io_pullups
#pragma num_alias_table_entries 2
#pragma run_unconfigured

// Define the device interface
network output SNVT_temp_p nvoTemperature;

// Configure the I/O pins
IO_7 touch ioThermometer;

#define DS18S20_SKIP_ROM    0xCCu
#define DS18S20_CONVERT     0x44u
#define DS18S20_READ        0xBEu

// Get a temperature reading from the Touch temperature sensor
SNVT_temp_p GetTemperature(void) {
    union {
        SNVT_temp_p snvtTempP;
        unsigned    Bytes[2];
    } CurrentTemperature;
    CurrentTemperature.snvtTempP = 32767l;

    if (touch_reset(ioThermometer)) {
        (void) touch_byte(ioThermometer, DS18S20_SKIP_ROM);
        (void) touch_byte(ioThermometer, DS18S20_READ);

        CurrentTemperature.Bytes[1]
```

```
            = touch_byte(ioThermometer, 0xFFu);
        CurrentTemperature.Bytes[0]
            = touch_byte(ioThermometer, 0xFFu);

        if (touch_reset(ioThermometer)) {
            // Scale the raw reading
            CurrentTemperature.snvtTempP *= 50l;

            // start the next conversion cycle:
            (void) touch_byte(ioThermometer, DS18S20_SKIP_ROM);
            (void) touch_byte(ioThermometer, DS18S20_CONVERT);
        } else {
            CurrentTemperature.snvtTempP = 32767l;
        }
    }
    return CurrentTemperature.snvtTempP;
}

// Repeat every second
mtimer repeating readTimer = 1000;

// Read the temperature when the timer expires
when(timer_expires(readTimer)) {
    SNVT_temp_p temperature;
    static SNVT_temp_p lastTemperature;

    temperature = GetTemperature();
    if (temperature != lastTemperature) {
        lastTemperature = temperature;

        // Send network variable update
        nvoTemperature = temperature;
    }
}
```

Digital Actuator Example

The following example application controls the state of the MiniGizmo LEDs based on a network variable input, replacing the timer control in the previous digital actuator example. The network variable code is highlighted in bold. A network tool such as the LonMaker Integration Tool is required to test this application.

```
#pragma num_alias_table_entries 2
#pragma run_unconfigured

// Define the device interface
network input SNVT_switch nviLight;

// Configure the I/O pins
IO_1 output bit ioLEDLd = 1;
IO_2 output bitshift numbits(8) ioLEDs;

// Set the MiniGizmo LEDs to all on or all off
void SetLED(const boolean state) {
    // Shift out the LED value
    io_out(ioLEDs, state ? 0 : 0xFF);

    // Latch the new value
    io_out(ioLEDLd, 0);
    io_out(ioLEDLd, 1);
}
```

```
// Update LEDs when an nviLight update is received
when(nv_update_occurs(nviLight)) {
    SetLED(nviLight.state && nviLight.value);
}
```

Serial Actuator Example

The following example application sends a string received from a network variable to the serial port on a MiniGizmo. The output network variable is a **SNVT_str_asc** structure containing a single field with a zero-terminated ASCII string of up to 30 characters. The network variable code is highlighted in bold. A network tool such as the LonMaker Integration Tool is required to test this application.

```
#include <string.h>

#pragma num_alias_table_entries 2
#pragma run_unconfigured

// Define the device interface
network input SNVT_str_asc nviString;

// Configure the I/O pins
IO_10 output serial baud (4800) ioSerialOut;

// Send a string to the serial port
const char errorString[] = "String too long.";
void PrintConsole(const char *message) {
    if (strlen(message) <= 100) {
        io_out(ioSerialOut, message, (unsigned) strlen(message));
    } else {
        io_out(ioSerialOut, errorString,
               (unsigned) strlen(errorString));
    }
}

// Print the network variable value when an nviString update
// is received
when(nv_update_occurs(nviString)) {
    PrintConsole(nviString.ascii);
    // Print new line
    PrintConsole("\n\r");
}
```

Configuration Properties

A *configuration property (CP)* is a data item that, like a network variable, is part of the device interface for a device. Configuration properties characterize the behavior of a device in the system. Network tools manage this attribute and keep a copy of its value in a database to support maintenance operations. If a device fails and needs to be replaced, the configuration property data stored in the database is downloaded into the replacement device to restore the behavior of the replaced device in the system.

Configuration properties facilitate interoperable installation and configuration tools by providing a well-defined and standardized interface for configuration data. Each configuration property type is defined in a resource

file that specifies the data encoding, scaling, units, default value, range, and behavior for configuration properties based on the type. A rich variety of *standard configuration property types (SCPTs)* are defined. SCPTs provide standard type definitions for commonly used configuration properties such as dead-bands, hysteresis thresholds, and message heartbeat rates. You can also create your own *user configuration property types (UCPTs)* that are defined in resource files that you create with the NodeBuilder Resource Editor.

Digital Sensor Example

The following example application adds a configuration property that documents the location of the digital sensor example application from the previous section. This configuration property is used by network tools to document the location of a device, and is not used by the application. The **ignore_notused** compiler directive is used to prevent a symbol not used warning when compiling this application. The configuration property code is highlighted in bold. A network tool such as the LonMaker Integration Tool is required to test this application.

```
#pragma num_alias_table_entries 2
#pragma run_unconfigured

// Define the device interface
network input SCPTlocation cp cpLocation = {"Unknown"};
#pragma ignore_notused cpLocation
network output SNVT_switch nvoSwitch;

// Configure the I/O pins
IO_4 input bitshift numbits(8) clockedge(-) ioButtons;
IO_6 output bit ioButtonLd = 1;

// Read the MiniGizmo buttons
boolean GetButton(void) {
    // Latch the button inputs
    io_out(ioButtonLd, 0);
    io_out(ioButtonLd, 1);

    // Shift in and return TRUE if any buttons are pressed
    return !((unsigned) io_in(ioButtons) == 0xFF);
}

// Repeat every 50 milliseconds
mtimer repeating scanTimer = 50;

// Read the buttons when the timer expires
when(timer_expires(scanTimer)) {
    boolean button;
    static boolean lastButton;

    button = GetButton();
    if (button != lastButton) {
        lastButton = button;
        nvoSwitch.value = button ? 200U : 0;
        nvoSwitch.state = button ? 1 : 0;
    }
}
```

Functional Blocks and Functional Profiles

A device application is divided into one or more *functional blocks*. A functional block is a portion of a device's application that performs a task by receiving configuration and operational data inputs, processing the data, and sending operational data outputs. A functional block may receive inputs from the network, hardware attached to the device, or from other functional blocks on a device. A functional block may send outputs to the network, to hardware attached to the device, or to other functional blocks on the device.

The device application implements a functional block for each function on the device to which other devices should communicate, or that requires configuration for particular application behavior. Each functional block is defined by a *functional profile*. A functional profile is a template for functional block, and a functional block is an implementation of a functional profile.

The network inputs and outputs of a functional block, if any, are provided by network variables and configuration properties as described in the previous sections. The network variables provide the operational data inputs and outputs for the functional block. The configuration properties configure the behavior of the functional block.

For example, a light switch could implement a functional block based on the **SFPTopenLoopSensor** profile, combining a **SNVT_switch** typed network variable that represents the current switch position with a configuration property that contains the default state for the switch into one logical unit. This logical unit—the functional block—can be disabled, enabled, tested, and managed by a network integrator.

Each functional profile defines mandatory and optional network variables and mandatory and optional configuration properties. A functional block must implement all the mandatory network variables and configuration properties defined by the functional profile, and may implement any of the optional network variables and configuration properties defined by the functional profile. In the example above, the mandatory member network variable **nvoValue** is implemented with the **nvoSwitch** network variable.

Functional profiles are defined in *resource files*. You can use standard functional profiles or you can define your own functional profiles in your own resource files using the NodeBuilder Resource Editor. A functional profile defined in a resource file is also called a *functional profile template* (FPT).

You can automatically embed data within your device that identifies its device interface to network tools that are used to install the device. This data is called *self-identification* (SI) data and *self-documentation* (SD) data. The Neuron C compiler generates this data based on the functional blocks, network variables, and configuration properties that you declare, as well as the resource files that you provide. You can add your own documentation to the SD data to further document your device and its interface.

You can include network variable names in the SD data using the **#pragma enable_sd_nv_names** directive. You can also express further details, requirements and recommendations, such as the use of authenticated data transfer, for each network variable. See Chapter 3 of the *Neuron C Programmer's Guide* for more information on this.

An application image for a device created by the Neuron C compiler will contain self-identification information unless the **#pragma disable_snvt_si** directive is used. See the *Compiler Directives* chapter of the *Neuron C Reference Guide* for more information.

Including self-identification and self-documentation data with a device makes it easier to install, as it allows easy, plug-and-play style, integration in multi-vendor networks. While self-identification and self-documentation simplify installation, these methods do not expose any of the algorithms used within the application.

LONMARK International provides a procedure for developers to certify devices as being interoperable. Certification confirms that the device is interoperable in a networked system. LONMARK interoperable devices conform to all ANSI/EIA/CEA-709.1 (EN14908-1) protocol layer 1 – 6 requirements as specified by the *LONMARK Layer 1 – 6 Interoperability Guidelines*, and conform to all aspects of application-layer design, as discussed in the *LONMARK Application Layer Interoperability Guidelines*.

Contact LONMARK International at www.lonmark.org for more details about becoming a member and certifying your devices.

Digital Sensor Example

The following example application adds a Switch functional block to the previous digital sensor example. The network variable output is changed to be a member of the functional block. The functional block code is highlighted in bold. The **enable_sd_nv_names** compiler directive is not required to implement functional blocks, but simplifies network integration by putting network variable names in the application image for a device. A network tool such as the LonMaker Integration Tool is required to test this application.

```
#pragma enable_sd_nv_names
#pragma num_alias_table_entries 2
#pragma run_unconfigured

// Define the device interface
network input SCPTlocation cp cpLocation = {"Unknown"};
#pragma ignore_notused cpLocation
network output SNVT_switch nvoSwitch;

fblock SFPTswitch {
       nvoSwitch implements nvoSwitch;
} fbSwitch external_name("Switch");

// Configure the I/O pins
IO_4 input bitshift numbits(8) clockedge(-) ioButtons;
IO_6 output bit ioButtonLd = 1;

// Read the MiniGizmo buttons
boolean GetButton(void) {
    // Latch the button inputs
    io_out(ioButtonLd, 0);
    io_out(ioButtonLd, 1);

    // Shift in and return TRUE if any buttons are pressed
    return !((unsigned) io_in(ioButtons) == 0xFF);
}
```

```
// Repeat every 50 milliseconds
mtimer repeating scanTimer = 50;

// Read the buttons when the timer expires
when(timer_expires(scanTimer)) {
    boolean button;
    static boolean lastButton;

    button = GetButton();
    if (button != lastButton) {
        lastButton = button;
        nvoSwitch.value = button ? 200U : 0;
        nvoSwitch.state = button ? 1 : 0;
    }
}
```

Analog Sensor Example

The following example application adds an Open Loop Sensor functional block to the previous analog sensor example. The network variable output is changed to be a member of the functional block. The functional block code is highlighted in bold. A network tool such as the LonMaker Integration Tool is required to test this application.

```
#pragma enable_io_pullups
#pragma enable_sd_nv_names
#pragma num_alias_table_entries 2
#pragma run_unconfigured

// Define the device interface
network output SNVT_temp_p nvoTemperature;

fblock SFPTopenLoopSensor {
       nvoTemperature implements nvoValue;
} fbSwitch external_name("Temperature");

// Configure the I/O pins
IO_7 touch ioThermometer;

#define DS18S20_SKIP_ROM    0xCCu
#define DS18S20_CONVERT     0x44u
#define DS18S20_READ        0xBEu

// Get a temperature reading from the Touch temperature sensor
SNVT_temp_p GetTemperature(void) {
    union {
        SNVT_temp_p snvtTempP;
        unsigned    Bytes[2];
    } CurrentTemperature;
    CurrentTemperature.snvtTempP = 32767l;

    if (touch_reset(ioThermometer)) {
        (void) touch_byte(ioThermometer, DS18S20_SKIP_ROM);
        (void) touch_byte(ioThermometer, DS18S20_READ);

        CurrentTemperature.Bytes[1]
            = touch_byte(ioThermometer, 0xFFu);
        CurrentTemperature.Bytes[0]
            = touch_byte(ioThermometer, 0xFFu);

        if (touch_reset(ioThermometer)) {
            // Scale the raw reading
```

```
            CurrentTemperature.snvtTempP *= 50l;

            // start the next conversion cycle:
            (void) touch_byte(ioThermometer, DS18S20_SKIP_ROM);
            (void) touch_byte(ioThermometer, DS18S20_CONVERT);
        } else {
            CurrentTemperature.snvtTempP = 32767l;
        }
    }
    return CurrentTemperature.snvtTempP;
}

// Repeat every second
mtimer repeating readTimer = 1000;

// Read the temperature when the timer expires
when(timer_expires(readTimer)) {
    SNVT_temp_p temperature;
    static SNVT_temp_p lastTemperature;

    temperature = GetTemperature();
    if (temperature != lastTemperature) {
        lastTemperature = temperature;

        // Send network variable update
        nvoTemperature = temperature;
    }
}
```

Digital Actuator Example

The following example application adds an Open Loop Actuator functional block to the previous digital actuator example. The network variable input is changed to be a member of the functional block. The functional block code is highlighted in bold. A network tool such as the LonMaker Integration Tool is required to test this application.

```
#pragma enable_sd_nv_names
#pragma num_alias_table_entries 2
#pragma run_unconfigured

// Define the device interface
network input SNVT_switch nviLight;

fblock SFPTopenLoopActuator {
       nviLight implements nviValue;
} fbSwitch external_name("Light");

// Configure the I/O pins
IO_1 output bit ioLEDLd = 1;
IO_2 output bitshift numbits(8) ioLEDs;

// Set the MiniGizmo LEDs to all on or all off
void SetLED(const boolean state) {
    // Shift out the LED value
    io_out(ioLEDs, state ? 0 : 0xFF);

    // Latch the new value
    io_out(ioLEDLd, 0);
    io_out(ioLEDLd, 1);
}
```

```
// Update LEDs when an nviLight update is received
when(nv_update_occurs(nviLight)) {
    SetLED(nviLight.state && nviLight.value);
}
```

Serial Actuator Example

The following example application adds an Open Loop Actuator functional block to the previous serial actuator example. The network variable input is changed to be a member of the functional block. The functional block code is highlighted in bold. A network tool such as the LonMaker Integration Tool is required to test this application.

```
#include <string.h>

#pragma enable_sd_nv_names
#pragma num_alias_table_entries 2
#pragma run_unconfigured

// Define the device interface
network input SNVT_str_asc nviString;

fblock SFPTopenLoopActuator {
       nviString implements nviValue;
} fbSwitch external_name("Serial Out");

// Configure the I/O pins
IO_10 output serial baud (4800) ioSerialOut;

// Send a string to the serial port
const char errorString[] = "String too long.";
void PrintConsole(const char *message) {
    if (strlen(message) <= 100) {
        io_out(ioSerialOut, message, (unsigned) strlen(message));
    } else {
        io_out(ioSerialOut, errorString,
               (unsigned) strlen(errorString));
    }
}

// Print the network variable value when an nviString update
// is received
when(nv_update_occurs(nviString)) {
    PrintConsole(nviString.ascii);
    // Print new line
    PrintConsole("\n\r");
}
```

Self-installation

The network inputs and outputs described in the *Input/Output* section must be connected to cause the data sent by an application to an output network variable to be received by another application via a network variable input. These connections may be created by a network tool such as the LonMaker Integration Tool, or by your device application with the Neuron C ISI library. Networks can start out as self-installed networks using ISI and, as size or complexity grows beyond the ISI limits, can be upgraded into a managed

network. Developing an application using the Neuron ISI library is described in the *ISI Programmer's Guide*.

Digital Sensor Example

The following example application adds ISI support to the previous digital sensor example. The network variable output will automatically connect to all digital actuators offering connections in the network. The ISI code is highlighted in bold.

```
#pragma enable_sd_nv_names
#pragma num_alias_table_entries 2

#include <isi.h>

// Define the device interface
network input SCPTlocation cp cpLocation = {"Unknown"};
#pragma ignore_notused cpLocation
network output SNVT_switch nvoSwitch;

fblock SFPTswitch {
       nvoSwitch implements nvoSwitch;
} fbSwitch external_name("Switch");

// Configure the I/O pins
IO_4 input bitshift numbits(8) clockedge(-) ioButtons;
IO_6 output bit ioButtonLd = 1;

// Start the ISI engine
when(reset) {
    IsiStartS(isiFlagNone);
}

// Process ISI Messages
when(msg_arrives) {
    if (IsiApproveMsg()) {
        (void) IsiProcessMsgS();
    }
}

// Read the MiniGizmo buttons
boolean GetButton(void) {
    // Latch the button inputs
    io_out(ioButtonLd, 0);
    io_out(ioButtonLd, 1);

    // Shift in and return TRUE if any buttons are pressed
    return !((unsigned) io_in(ioButtons) == 0xFF);
}

// Repeat every 50 milliseconds
mtimer repeating scanTimer = 50;

// Read the buttons when the timer expires
when(timer_expires(scanTimer)) {
    boolean button;
    static boolean lastButton;

    button = GetButton();
    if (button != lastButton) {
        lastButton = button;
        nvoSwitch.value = button ? 200U : 0;
```

```
        nvoSwitch.state = button ? 1 : 0;
    }
}

// Call IsiTick() every 250 milliseconds
mtimer repeating isiTimer = 250;

when(timer_expires(isiTimer)) {
    IsiTickS();
}

// Override IsiGetAssembly() with a version that connects to
// automatic connections offering a digital actuator
unsigned IsiGetAssembly(const IsiCsmoData * pCsmo, boolean Auto) {
    if (Auto &&                   // Automatic Connection
        pCsmo->Profile   == 4 &&  // SFPTopenLoopActuator
        pCsmo->Direction == isiDirectionInput &&
        pCsmo->Width     == 1) {
            return nvoSwitch::global_index;
    }
    return ISI_NO_INDEX;
}
```

Analog Sensor Example

The following example application adds ISI support to the previous analog sensor example. The network variable output may be manually connected to a space comfort controller. The ISI code is highlighted in bold.

```
#pragma enable_io_pullups
#pragma enable_sd_nv_names
#pragma num_alias_table_entries 2

#include <isi.h>
#include <control.h>

// Define the device interface
network output SNVT_temp_p nvoTemperature;

fblock SFPTopenLoopSensor {
       nvoTemperature implements nvoValue;
} fbSwitch external_name("Temperature");

// Configure the I/O pins
IO_7 touch ioThermometer;

// Variable to keep track of the state of the ISI engine
IsiEvent isiState;

// Start the ISI engine
when(reset) {
    IsiStartS(isiFlagNone);
}

// Process ISI Messages
when(msg_arrives) {
    if (IsiApproveMsg()) {
        (void)IsiProcessMsgS();
    }
}

#define DS18S20_SKIP_ROM    0xCCu
```

```
#define DS18S20_CONVERT     0x44u
#define DS18S20_READ        0xBEu

// Get a temperature reading from the Touch temperature sensor
SNVT_temp_p GetTemperature(void) {
    union {
        SNVT_temp_p snvtTempP;
        unsigned    Bytes[2];
    } CurrentTemperature;
    CurrentTemperature.snvtTempP = 32767l;

    if (touch_reset(ioThermometer)) {
        (void) touch_byte(ioThermometer, DS18S20_SKIP_ROM);
        (void) touch_byte(ioThermometer, DS18S20_READ);

        CurrentTemperature.Bytes[1]
            = touch_byte(ioThermometer, 0xFFu);
        CurrentTemperature.Bytes[0]
            = touch_byte(ioThermometer, 0xFFu);

        if (touch_reset(ioThermometer)) {
            // Scale the raw reading
            CurrentTemperature.snvtTempP *= 50l;

            // start the next conversion cycle:
            (void) touch_byte(ioThermometer, DS18S20_SKIP_ROM);
            (void) touch_byte(ioThermometer, DS18S20_CONVERT);
        } else {
            CurrentTemperature.snvtTempP = 32767l;
        }
    }
    return CurrentTemperature.snvtTempP;
}

// Repeat every second
mtimer repeating readTimer = 1000;

// Read the temperature when the timer expires
when(timer_expires(readTimer)) {
    SNVT_temp_p temperature;
    static SNVT_temp_p lastTemperature;

    temperature = GetTemperature();
    if (temperature != lastTemperature) {
        lastTemperature = temperature;

        // Send network variable update
        nvoTemperature = temperature;
    }
}

// Call IsiTick() every 250 milliseconds
mtimer repeating isiTimer = 250;

when(timer_expires(isiTimer)) {
    IsiTickS();
}

// Poll the service pin and use it as the ISI Connect button.
// Holding it down will cancel any pending enrollment.
#define LONG_SERVICE_PIN 50
mtimer repeating serviceTimer = 50;
unsigned ServicePinActivation;
```

```
    when(timer_expires(serviceTimer)) {
        if (service_pin_state()) {
            ServicePinActivation++;
            if (ServicePinActivation > LONG_SERVICE_PIN) {
                IsiCancelEnrollment();
            }
        } else if (ServicePinActivation) {
            if (isiState == isiNormal) {
                IsiOpenEnrollment(nvoTemperature::global_index);
            } else if (isiState == isiPending ||
                       isiState == isiApprovedHost) {
                IsiCreateEnrollment(nvoTemperature::global_index);
            }
            ServicePinActivation = 0;
        }
    }

    // MyCsmoData defines the connections details for the automatic
    // ISI network variable connection advertised by this device
    static const IsiCsmoData MyCsmoData =
         { ISI_DEFAULT_GROUP, isiDirectionOutput, 1, 2, 105u, 0 };

    // Create the enrollment open message
    void IsiCreateCsmo(unsigned Assembly, IsiCsmoData* pCsmoData) {
        memcpy(pCsmoData, &MyCsmoData, sizeof(IsiCsmoData));
    #pragma ignore_notused  Assembly
    }

    // Relay information provided by the ISI engine
    void IsiUpdateUserInterface(IsiEvent Event, unsigned Parameter) {
        switch (Event) {
            // When these event occur, the ISI engine is in the
            // normal state
            case isiImplemented:
            case isiRun:
            case isiWarm:
            case isiCancelled:
            case isiDeleted:
                isiState = isiNormal;
                break;
            // Otherwise set the state to the event
            default:
                isiState = Event;
                break;
        }
    #pragma ignore_notused Parameter
    }

    // Override IsiGetAssembly() with a version that connects to
    // automatic connections offering a digital actuator
    unsigned IsiGetAssembly(const IsiCsmoData * pCsmo, boolean Auto) {
         if (!Auto &&                  //Manual Connection
             pCsmo->Profile   == 14 && //SFPTspaceComfortController
             pCsmo->Direction == isiDirectionInput &&
             pCsmo->Width     == 1) {
                return nvoTemperature::global_index;
        }
        return ISI_NO_INDEX;
}
```

Digital Actuator Example

The following example application adds ISI support to the previous digital actuator example. The network variable input may be automatically connected to all digital sensors in the network programmed to accept enrollment from a **SFTPopenLoopActuator** functional block. The ISI code is highlighted in bold.

```
#pragma enable_sd_nv_names
#pragma num_alias_table_entries 2

#include <isi.h>

// Define the device interface
network input SNVT_switch nviLight;

fblock SFPTopenLoopActuator {
       nviLight implements nviValue;
} fbSwitch external_name("Light");

// Configure the I/O pins
IO_1 output bit ioLEDLd = 1;
IO_2 output bitshift numbits(8) ioLEDs;

// Start the ISI engine
when(reset) {
    IsiStartS(isiFlagNone);
}

// Process ISI Messages
when(msg_arrives) {
    if (IsiApproveMsg()) {
        (void)IsiProcessMsgS();
    }
}

// Set the MiniGizmo LEDs to all on or all off
void SetLED(const boolean state) {
    // Shift out the LED value
    io_out(ioLEDs, state ? 0 : 0xFF);

    // Latch the new value
    io_out(ioLEDLd, 0);
    io_out(ioLEDLd, 1);
}

// Update LEDs when an nviLight update is received
when(nv_update_occurs(nviLight)) {
    SetLED(nviLight.state && nviLight.value);
}

// Call IsiTick() every 250 milliseconds
mtimer repeating isiTimer = 250;

when(timer_expires(isiTimer)) {
    IsiTickS();
}

// MyCsmoData defines the connection details for the automatic ISI
// network variable connection advertised by this device.
static const IsiCsmoData MyCsmoData =
       { ISI_DEFAULT_GROUP, isiDirectionInput, 1, 4, 95u, 0 };
```

```
// Call IsiInitiateAutoEnrollment() after a specified period has
// passed
void IsiUpdateUserInterface(IsiEvent Event, unsigned Parameter) {
    if (Event == isiWarm &&
        !IsiIsConnected(nviLight::global_index)) {
        IsiInitiateAutoEnrollment(&MyCsmoData,
                                  nviLight::global_index);
    }
#pragma ignore_notused Parameter
}

// Create the CSMR messages to be periodically sent
void IsiCreateCsmo(unsigned Assembly, IsiCsmoData* pCsmoData) {
    memcpy(pCsmoData, &MyCsmoData, sizeof(IsiCsmoData));
#pragma ignore_notused  Assembly
}
```

Serial Actuator Example

The following example application adds ISI support to the previous serial actuator example. The network variable input is changed to a switch input, and may be automatically connected to all digital sensors in the network programmed to accept enrollment from a **SFTPopenLoopActuator** functional block. The ISI code is highlighted in bold.

```
#include <string.h>
#include <isi.h>

#pragma enable_sd_nv_names
#pragma num_alias_table_entries 2

// Define the device interface
network input SNVT_switch nviSwitch;

fblock SFPTopenLoopActuator {
       nviSwitch implements nviValue;
} fbSwitch external_name("Serial Out");

// Configure the I/O pins
IO_10 output serial baud (4800) ioSerialOut;

// Start the ISI engine
when(reset) {
    IsiStartS(isiFlagNone);
}

// Process ISI Messages
when(msg_arrives) {
    if (IsiApproveMsg()) {
        (void)IsiProcessMsgS();
    }
}

// Send a string to the serial port
const char errorString[] = "String too long.";
void PrintConsole(const char *message) {
    if (strlen(message) <= 100) {
        io_out(ioSerialOut, message, (unsigned) strlen(message));
    } else {
        io_out(ioSerialOut, errorString,
               (unsigned) strlen(errorString));
    }
```

```
}

// Print the network variable value when an nviSwitch update
// is received
when(nv_update_occurs(nviSwitch)) {
    PrintConsole(nviSwitch.value ? "On\n\r" : "Off\n\r");
}

// Call IsiTick() every 250 milliseconds
mtimer repeating isiTimer = 250;

when(timer_expires(isiTimer)) {
    IsiTickS();
}

// MyCsmoData defines the connection details for the automatic
// ISI network variable connection advertised by this device
static const IsiCsmoData MyCsmoData =
       { ISI_DEFAULT_GROUP, isiDirectionInput, 1, 4, 95u, 0 };

// Call IsiInitiateAutoEnrollment() after a specified period
// has passed
void IsiUpdateUserInterface(IsiEvent Event, unsigned Parameter) {
    if (Event == isiWarm &&
        !IsiIsConnected(nviSwitch::global_index)) {
        IsiInitiateAutoEnrollment(&MyCsmoData,
                                  nviSwitch::global_index);
    }
#pragma ignore_notused Parameter
}

// Create the CSMR messages to be periodically sent
void IsiCreateCsmo(unsigned Assembly, IsiCsmoData* pCsmoData) {
    memcpy(pCsmoData, &MyCsmoData, sizeof(IsiCsmoData));
#pragma ignore_notused  Assembly
}
```

Advanced Neuron C Concepts

This section discusses advanced Neuron C concepts.

Event-Driven vs. Polled Scheduling

Although the Neuron C applications typically use event-driven network variable updates, Neuron C also allows you to construct polled networks variables that are only updated when requested by another device or tool. Chapter 3, *How Devices Communicate Using Network Variables*, of the *Neuron C Programmer's Guide* provides further information on polling.

Low-Level Messaging

In addition to the functional block and network variable communication model, Neuron C also supports application messages. You can use application messages—in place of or in conjunction with network variables—to implement proprietary or standard special-purpose interfaces to your devices. Standard special-purpose interfaces include the LONWORKS file transfer protocol and the ISI protocol. Application messages are described in

Chapter 6, *How Devices Communicate Using Application Messages*, of the *Neuron C Programmer's Guide.*

Feedback Network Variable Connections

A typical network variable connection uses one output network variable that provides sensor data, and one input network variable that receives the sensor reading. The connections may use the acknowledged messaging service, and rely on the ANSI/EIA/CEA-709.1 (EN14908-1) protocol to ensure delivery of the related data. This scenario is known as an *open loop connection scenario*. In this scenario, the sensor has no feedback about the state of the actuators.

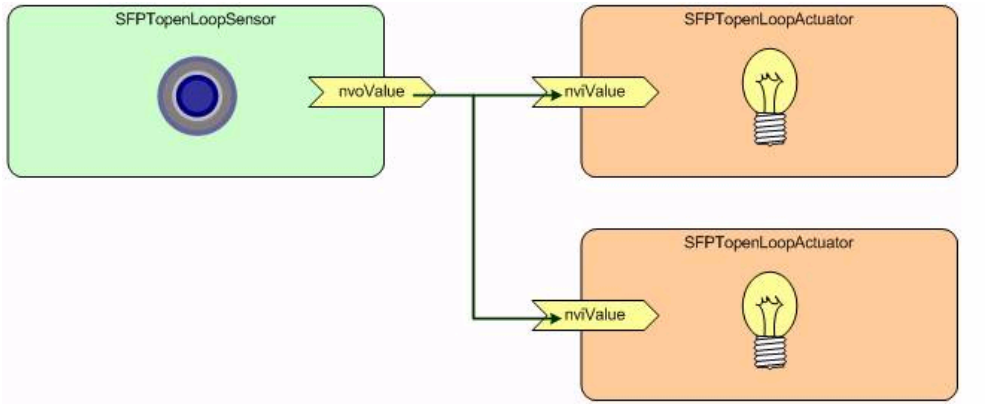

Figure 4.3 Open Loop Connection Scenario

The alternative is known as a closed loop connection scenario, or feedback connection scenario. A feedback connection contains sensors with an additional feedback input network variable, and actuators with an additional feedback output network variable. In this scenario, it is possible for the sensor to get feedback on the state of any connected actuators.

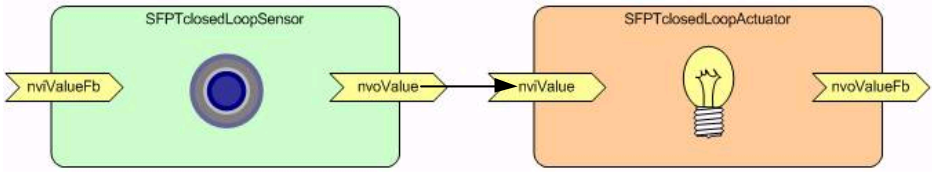

Figure 4.4 Closed Loop Connection Scenario

Feedback connection scenarios are typically implemented to address one of two issues:

- Feedback loops allow for synchronization. For example, multiple light switches implemented as toggling buttons, and a lamp. With the feedback signal, the switches can be synchronized and always know the lamp's status. Switch A can be used to turn the light on at the bottom of the stairs, and switch B can later be used to turn the light back off at the top of the stairs.
- Feedback loops allow for complete transport control. The acknowledged protocol service provides a ready-to-use mechanism for delivery supervision, but, similar to a registered letter, control stops at the point of delivery. The switch will know from the acknowledgment that the light has received the command, but it will not know whether the light has actually turned on.

Feedback loops allow for tied control; the lamp can for example sense the light level and provide the feedback signal based on real measurement (rather than assumption).

Simple feedback loops contain a single source and a single destination. Creating the feedback connection is straight-forward, as shown in Figure 4.5.

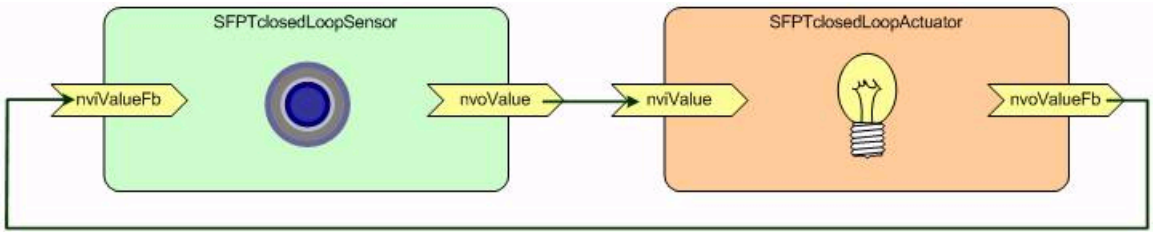

Figure 4.5 Simple Feedback Loops

With multiple sources or destinations, however, two distinct shapes of feedback connections are possible: a daisy-chained feedback connection, and a star-shaped feedback connection. These are shown in Figures 4.6 and 4.7

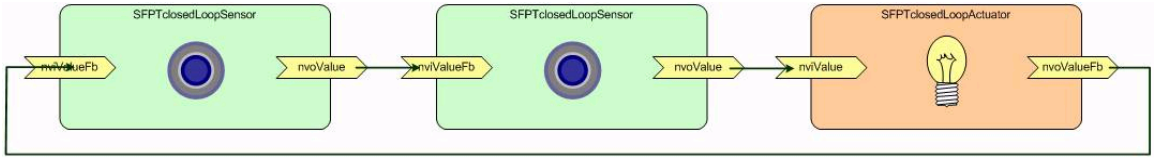

Figure 4.6 Daisy-chained Feedback Connection

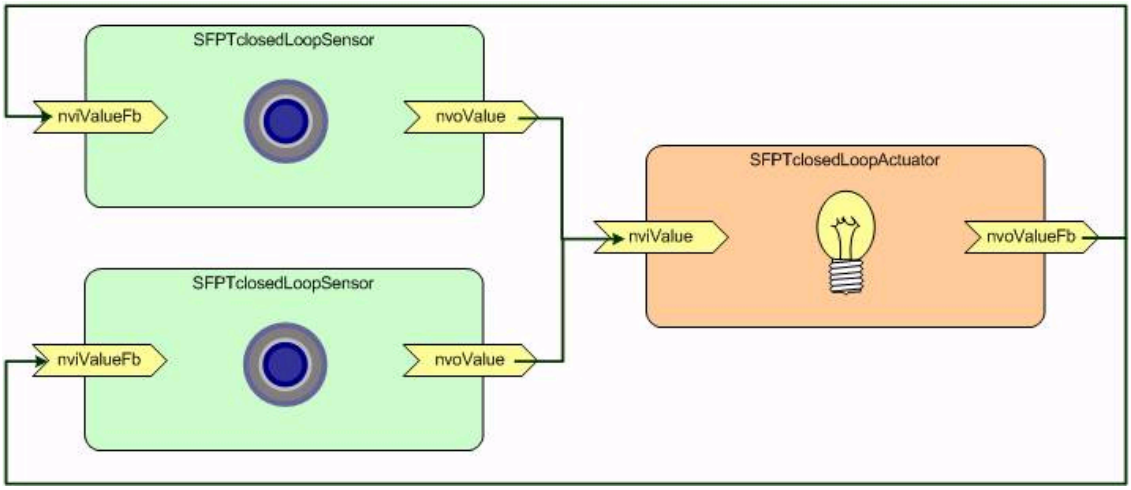

Figure 4.7 Star-shaped Feedback Connection

The daisy-chained feedback loop is often considered inferior, mostly because the loop breaks as a result of a single failure in the entire chain (subject to the application, this might actually be desirable behaviour), and also due to the fact that changes take time to propagate through the chain. For example, a large number of lights will not change the state simultaneously, but one after the other.

However, an outstanding advantage of the daisy-chain feedback loop is the simple and elegant transport control provided. The initiator device (switch B, for example) only needs to await exactly one feedback network variable update to know the loop has been completed.

To achieve the same level of transport control in a star-shaped feedback connection, the initiator needs to know the number of expected feedback network variable updates. The initiator device might be able to determine that number dynamically by inspecting the appropriate system tables, but this information is not always available. If the star-shaped feedback connection is made using the acknowledged service and group addressing, the group's member count is not known to the device by default.

An advantage of the star-shaped feedback connection is that it allows for quick propagation of new values, and oscillation is easier to control.

Oscillation

Like any looping construct, feedback loops can start to oscillate. This is most obvious in a daisy-chain scenario: the loop starts to oscillate if the sensor propagates revised data before the previous loop has been completed. Consider a series of On/Off/On commands issued by a switch in quick succession. If these commands propagate quickly enough, the loop will carry three different sets of data. If the initiator device was designed with a feedback loop for transport control in mind, this might have no negative effects. If, however, the initiator was designed with synchronization in mind yet connected in a daisy-chain, the loop can oscillate.

Oscillation is less likely with a star-shaped connection, but in both cases, manufacturers should document recommended connection scenarios for their devices. When designing closed loop applications, the following is recommended:

- Feedback loops that serve the purpose of synchronization should use a star-shaped feedback connection if possible. When a daisy-chain feedback connection is supported, sensors should only issue new values once the loop has been completed, or once a sufficient timeout has expired, whichever comes first.
- Closed loop sensors should ignore feedback (input) network variable updates if the update contains a network variable value that equals the value of the corresponding output network variable.
- Feedback loops that serve the purpose of tied transport control should be created under control of specialized software. A configuration tool should use knowledge of the exact connection shape to set configuration properties with each network variable, indicating the expected number of feedback updates.
- Due to its superior stability and speed of propagation, star-shaped feedback connections are preferred. Daisy-chained feedback connections are possible where a specialized connection tool (as described in the previous recommendation) is not available, and where oscillation is not possible due to sufficiently slow sensors.

Detecting First Application Start

Sometimes applications need to identify the first reset after a new application image has been loaded. This can be used to initialize data in the uninitialized EEPROM memory area supported by some Neuron Chip and Smart Transceiver models, where regular initializers are not supported. The same feature may also be used to set other defaults, such as enabling the ISI engine. This can be easily accomplished by taking advantage of the fact that Neuron C variables declared

with the **eeprom** storage class don't get initialized during reset, but only with a new application image. The following code fragment illustrates this approach:

```
eeprom boolean FirstStart = TRUE;
when(reset) {
   if (FirstStart) {
         // do whatever needs doing in this special case
         …
         // reset the FirstStart flag:
         FirstStart = FALSE;
   }
   }
```

Reset Processing

When a device is reset while being configured, the application executes the `when(reset)` task. Code in this task typically takes care of initialization of special application data as well of attached I/O circuitry. Some of these initialization tasks can take a long time, and may make the device seem unresponsive. The reset task should not take longer than five seconds, and cannot take longer than 18 seconds. If more than a few seconds of processing is required for the reset task, any time-consuming power-up or reset-processing must be moved into the first when-task that gets executed after the `when(reset)` task has completed. A simple way to accomplish this is by setting an application-defined flag, as shown in the following code fragment:

```
// declare the flag:
boolean ResetFlag = TRUE;

when (reset) {
   // perform all immediate initialization here
}

when (ResetFlag) {
   // perform all delayed reset processing here
   ..
   // signal completion by changing the flag:
   ResetFlag = FALSE;
 }
```

5

Debugging a Neuron C Application

This chapter describes how to use the boards and accessories included with the Mini EVK, or additional tools such as the LonMaker Integration Tool or NodeBuilder Development Tool, to debug a Neuron C application.

Debugging a Neuron C Application

You can use the boards and accessories included with the Mini EVK to debug a Neuron C application, or you can use additional tools such as the LonMaker Integration Tool or NodeBuilder Development Tool for more efficient and productive debugging. The following sections describe some of the methods you can use to debug your Neuron C applications. The first section describes how you can debug with the boards and accessories included with the Mini EVK. The next two sections describe how you can improve your debugging with additional tools that are not included with the Mini EVK.

Debugging with I/O

You can debug an application using I/O statements as described in the *Input/Output* section in Chapter 4. For example, if your device has LED outputs, you can use the outputs to signal events from within your application. If your device has serial outputs, you can send outputs to the serial port that you can monitor with Windows HyperTerminal on your computer.

If your device does not have appropriate outputs to support debugging, you can initially develop your application on one of the evaluation boards included with the Mini EVK. If your application does not use the IO10 I/O pin, you can use the serial port on the evaluation board to display debug output on your computer as described in *Getting Started with Neuron C* in Chapter 4. If your application does use the IO10 I/O pin, you can temporarily disable the I/O to this pin to debug your application on the evaluation board using the serial connection.

The following example application reads the MiniGizmo push buttons every 50 milliseconds and sends an "On" or "Off" value to the serial port if it has changed. A value of On means any of the buttons is pressed, and a value of Off means all the buttons are off. The serial output code is highlighted in bold. For more details on this code, see the examples in Chapter 4.

```
#include <string.h>

#pragma num_alias_table_entries 2
#pragma run_unconfigured

// Configure the I/O pins
IO_4 input bitshift numbits(8) clockedge(-) ioButtons;
IO_6 output bit ioButtonLd = 1;
IO_10 output serial baud (4800) ioSerialOut;

// Read the MiniGizmo buttons
boolean GetButton(void) {
    // Latch the button inputs
    io_out(ioButtonLd, 0);
    io_out(ioButtonLd, 1);

    // Shift in and return TRUE if any buttons are pressed
    return !((unsigned) io_in(ioButtons) == 0xFF);
}

// Send a string to the serial port
const char errorString[] = "String too long.";
```

```
void PrintConsole(const char *message) {
    if (strlen(message) <= 100) {
        io_out(ioSerialOut, message, (unsigned) strlen(message));
    } else {
        io_out(ioSerialOut, errorString,
               (unsigned) strlen(errorString));
    }
}

// Repeat every 50 milliseconds
mtimer repeating scanTimer = 50;

// Read the buttons when the timer expires
when(timer_expires(scanTimer)) {
    boolean button;
    static boolean lastButton;

    button = GetButton();
    if (button != lastButton) {
        lastButton = button;
        PrintConsole(button ? "On\n\r" : "Off\n\r");
    }
}
```

Debugging with the LonMaker Integration Tool

You can use the LonMaker Integration Tool to install LONWORKS devices in a network, and then configure, monitor, and test those devices. The LonMaker tool includes three useful interfaces for testing your Neuron C applications. They are the LonMaker Browser, the LonMaker Device Manager, and connection monitoring.

The LonMaker Browser is a standalone application that monitors all the network outputs from your device and allows you to control all the network inputs to your device. You can open the LonMaker Browser on any device or functional block in the network. It then displays all the network variables and configuration properties for the selected network variables and configuration properties. You can change the value of any of the input network variables or writeable configuration properties. Figure 5.1 shows an example LonMaker Browser display from the MGDemo application.

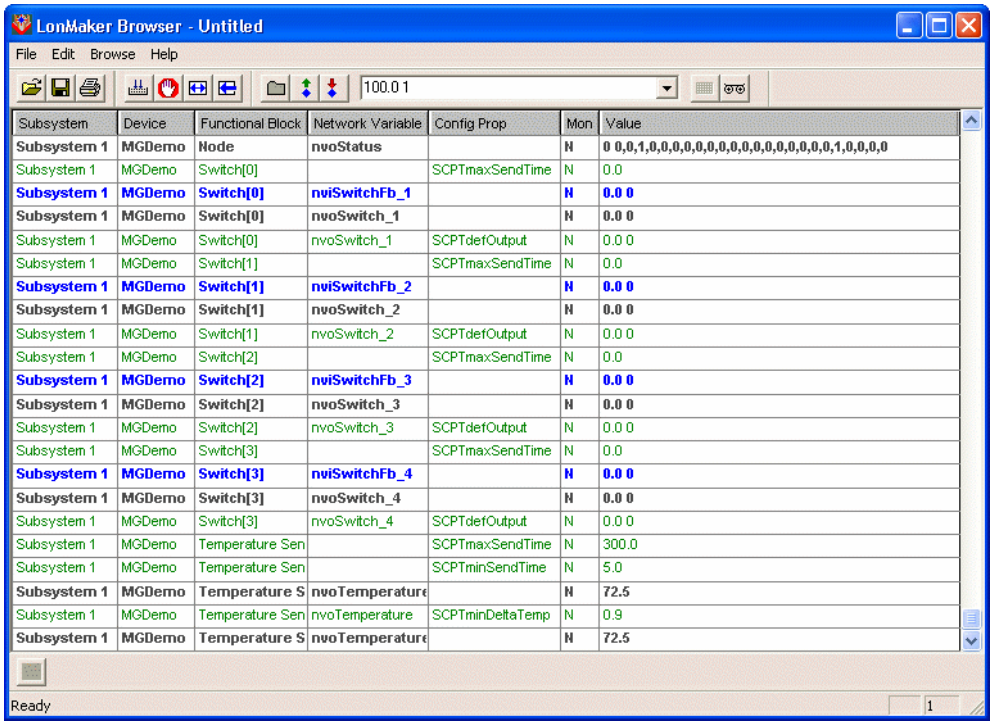

Figure 5.1 LonMaker Browser

The LonMaker Device Manager allows you to control the state of your device and its functional blocks. You can use the device manager to reset your device, put your device online or offline, and test network communication with your device. You can also use the Manage dialog to enable or disable individual functional blocks on your device, and to invoke the self-test function of any of your functional blocks that support self-test. Figure 5.2 shows an example test report from the Manage dialog.

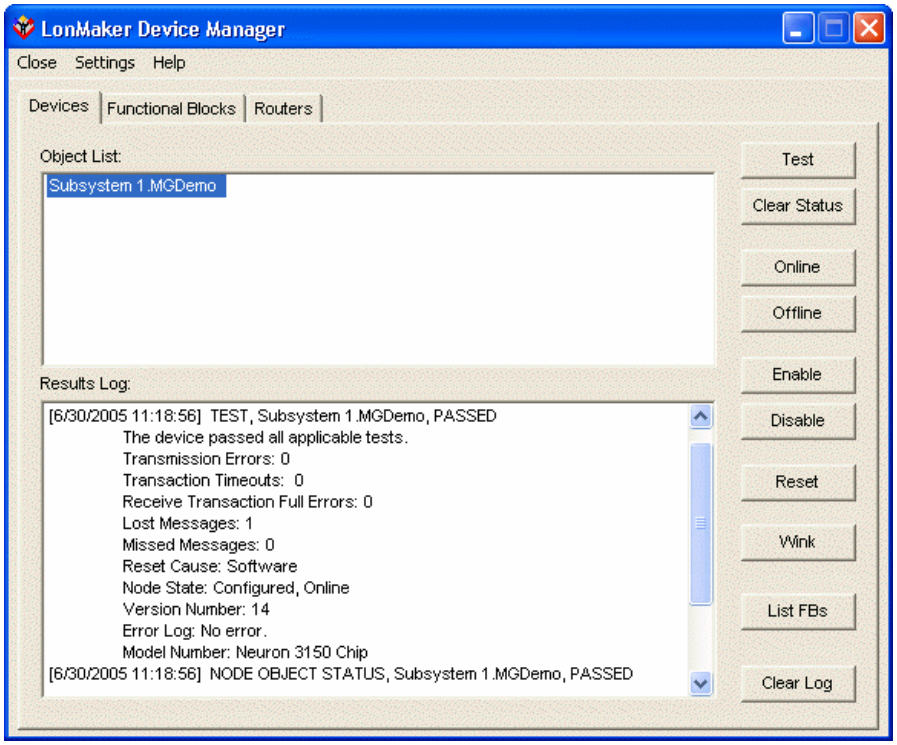

Figure 5.2 LonMaker Device Manager

The LonMaker tool allows you to connect network variables on your devices, and then monitor those connections on the same page that you use to create the connections. Figure 5.3 shows an example LonMaker drawing showing an MGSwitch device connected to an MGDemo device, with connection monitoring enabled to show the current value of the switch and the current output of the temperature sensor.

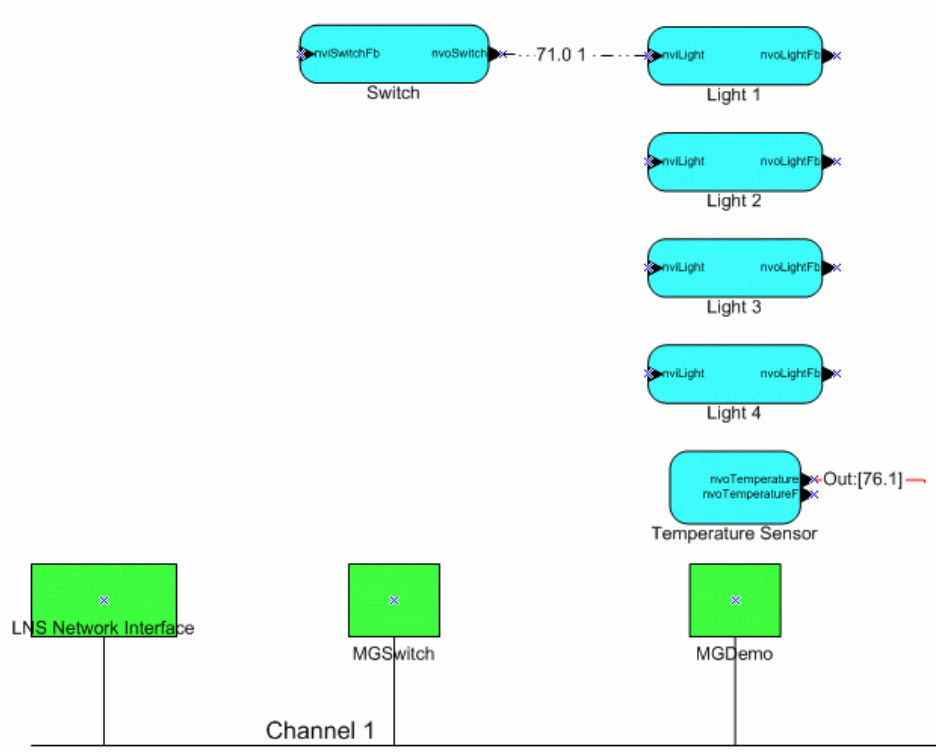

Figure 5.3 LonMaker Drawing with Connection Monitoring

You cannot simultaneously use the same network interface with both the LonMaker tool and the Mini Application. See *Using the Mini Application with the LonMaker Tool* section later in this chapter for information on using the LonMaker tool with the Mini Application.

Debugging with the NodeBuilder Development Tool

You can use the NodeBuilder Development Tool to develop and debug Neuron C applications. The NodeBuilder Development Tool includes a source-level debugger for Neuron C, called the NodeBuilder debugger. The NodeBuilder debugger allows you to control and observe your application's behavior to facilitate debugging. The debugger allows you to set breakpoints, monitor variables, halt the application, step through the application, view the call stack, and peek and poke memory. You can make changes to the code as you debug and debug multiple devices simultaneously.

Following is an example window from the NodeBuilder debugger showing the MGDemo example application stopped at a breakpoint upon completing a reading from the temperature sensor. Once stopped at a breakpoint, you can execute the application one step at a time to verify correct behavior and you can monitor any of the variables in your application.

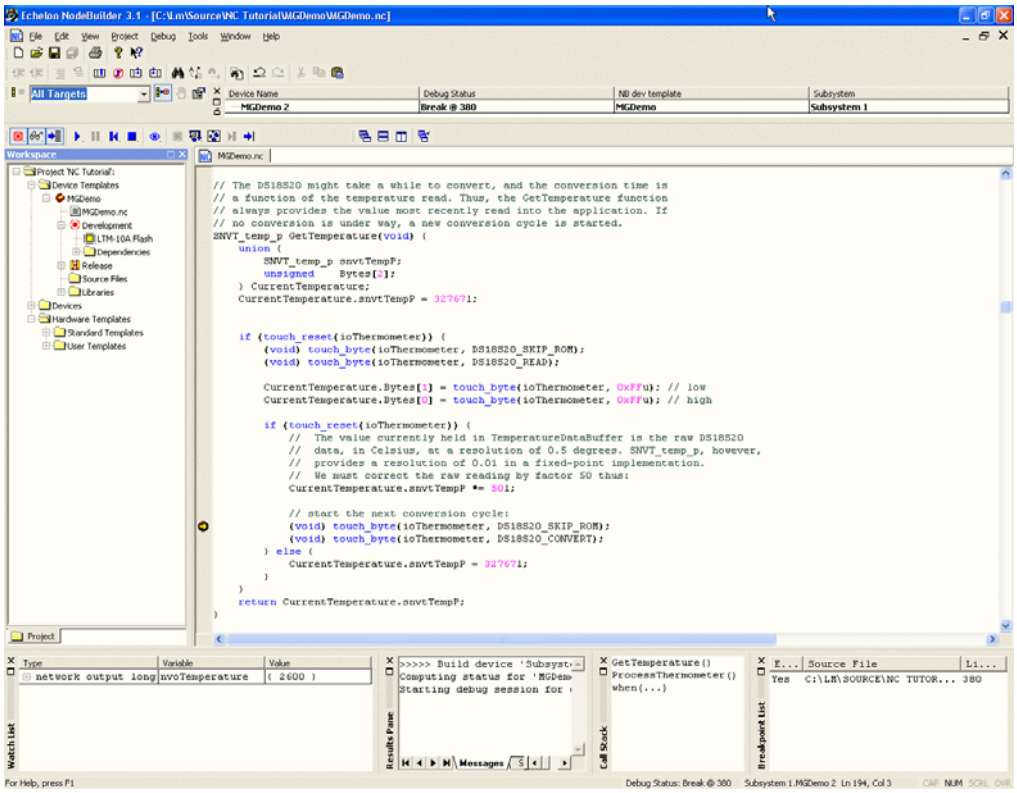

Figure 5.4 NodeBuilder Debugger

You cannot simultaneously use the same network interface with both the LonMaker tool that is included with the NodeBuilder tool and the Mini Application. See *Using the Mini Application with the LonMaker Tool* later in this chapter for information on using the LonMaker tool with the Mini Application, or see *Using the Mini Application With the NodeBuilder Tool* for more information on using the NodeBuilder tool with the Mini Application.

Using the Mini Application with LNS Applications

You can use the Mini Application with LNS applications such as the LonMaker Integration Tool or NodeBuilder Development Tool, but you cannot have the same network interface open in both the Mini Application and an LNS application at the same time. To use the Mini Application with an LNS application, you must do one of the following:

- Do not connect to the network interface in the Mini Application. You can use an LNS network tool such as the LonMaker Integration Tool to load an application that you compile with the Mini Application into a target device, and test that target device.
- Use a separate network interface for the Mini Application and LNS applications. For example, you can install two U10 USB Network Interfaces in your computer, use one of the interfaces with the Mini Application, and use the other with your LNS applications.
- Ensure that the Mini Application and LNS application are not attached to the network interface at the same time. An example of this procedure is described in the next section.

Using the Mini Application with the LonMaker Tool

To switch from using the Mini Application on a network interface to the LonMaker tool with the same network interface, follow these steps:

1. Close the Mini Application. This closes the network interface connection from the Mini Application.

2. Either start the LonMaker tool and attach to the same network interface, or if the LonMaker tool is already running, open the **LonMaker** menu, click **Network Properties**, click the **Network Interface** tab, select the **Network Interface**, set **Network Attached**, and then click **OK**.

To switch from using the LonMaker tool on a network interface to the Mini Application with the same network interface, follow these steps:

1. Either exit the LonMaker tool or open the **LonMaker** menu, click **Network Properties**, click the **Network Interface** tab, clear **Network Attached**, and then click **OK**.

2. Start the Mini Application, click the **Device** tab, select the **Network Interface**, and then click **Connect**.

Using the Mini Application With the NodeBuilder Tool

You can incorporate the source files, hardware templates and Neuron C libraries used in your Mini EVK projects into a NodeBuilder project. This section provides guidelines to follow when doing so.

The Mini EVK build process requires the automatic creation of NodeBuilder device template files. The Mini EVK uses the name of the Neuron C source file as the name of the device template file. For example, compiling the **zorro.nc** source file with the Mini EVK leads to the creation of a hidden **zorro.nbdt** device template file.

If you build an existing source file with the Mini application, and if the folder containing the source file already contains the associated device template file, the build process loses its transparency. This could occur if a source file was previously built with NodeBuilder, and you build the same source file with the Mini application. The Mini EVK will not overwrite the previously defined device template, and it will not delete it at the end of the build process. Instead, it will add a new target device template for the Mini build process, and merges preferences that apply to the entire device template.

You can avoid this by choosing different names for the device template and for the source file when you build the source file with NodeBuilder, as shown in Figure 5.5.

Figure 5.5 New Device Template Wizard

With these settings, you can use the same device template file for both build processes. Possible conflicts resulting from the sharing of the same NodeBuilder device template file can be resolved by viewing and editing the device template preferences in the NodeBuilder tool.

Appendix A

Troubleshooting

This appendix describes how to resolve problems that may occur while you are using the Mini EVK.

1. I cannot compile the MGKeyboard application. When I try to compile the application, the "`Typename 'UCPTfrequency' not found in Device Resource Files; 'SNVT*', 'SCPT*', 'UNVT*', 'UCPT*' are reserved [NCC#389]`" error is reported.

- For a TP/FT-10 channel, set the application's program ID to 9F:FF:FF:05:00:05:04:xx.
- For a PL-20N (power line PL-20 Smart Transceiver with CENELEC protocol disabled) target channel type, set the application's program ID to 9F:FF:FF:05:00:05:11:xx.
- For a PL-20C (power line PL-20 Smart Transceiver with CENELEC protocol enabled) target channel type, set the application's program ID to 9F:FF:FF:05:00:05:10:xx.
- See the header comment in the `MGKeyboard.nc` file for more details.

2. My application will not link to the ISI library. The linker reports "cannot relocate segment..." when I try to add the library to my application.

- Make sure you link with the correct version of the ISI library. See the header comment in your application source file for recommendations.
- The memory requirements of your application and all required libraries together might simply exceed the amount of available memory. See chapter 8, *Memory Management*, in the *Neuron C Reference Guide* of a list of suggestions *What to Try When a Program Doesn't Fit in a Neuron Chip*.

3. My application will not link to the ISI library. The linker reports "symbol not found..." when I try to build my application.

- Make sure you link with the correct version of the ISI library. The Neuron linker does not automatically link the application with the ISI library; you must provide a reference to that library through the *Libraries* section in the Mini Application.
- Make sure the reference to the offending symbol name is spelled correctly in your source code. Remember that symbol names in the C language family are case-sensitive.

4. The Mini Application reports error #10 when I load an application into my device.

The application cannot be loaded into the device, as the device does not contain the amount of memory required to load the application. This error will occur if you attempt to load the MGDemo application into an FT 3120 or PL 3120 Evaluation Board.

5. What can I do to improve application load-times on power line channels?

Use a TP/FT-10 channel for most of the application development and debugging. When doing so, you must make sure to include use of the targeted power line channel in your testing, as timing and latencies will vary from those observed in the faster development channel.

6. What can I do to improve application load-times?

You can reduce the application's size and have faster downloads, but you won't be downloading the same application anymore. Another solution is to avoid downloads altogether. You can program the application's **.NEI** file into the flash parts used with the FT 3150 and PL 3150 Evaluation Boards.

Flash parts supported by both the Mini software and hardware include the AT29C512 and AT29C010 devices.

7. I can connect my MGSwitch or MGLight application only once. What am I doing wrong?

Nothing. MGSwitch and MGLight are based on the minimum-footprint **IsiCompactManual.lib** version of the ISI library. This does not support extending, removing, or replacing connections. Once the device has joined a connection, you must return the device to factory defaults to clear out the connection information, and re-connect as desired. To return the device to factory defaults, press and hold the device's service pin button until the device resets (approximately 10 seconds).

8. I use the MGDemo application with multiple connections. When I try to add a new connection, or when I try to extend an exiting connection, the MGDemo application seems to ignore my request and no connection LED starts flashing on the evaluation board. What's wrong?

You might have run out of connection table space. The ISI library includes a default ISI connection table that defaults to 8 entries. Once all entries are used, you can no longer add a new connection, or extend an existing connection. You must remove an existing connection first, or implement a larger connection table by overriding the **IsiGetConnectionTableSize()**, **IsiGetConnection()** and **IsiSetConnection()** functions.

While the MGDemo example only implements 6 connection assemblies requiring one connection table entry each, an extension to any of these existing connection requires an additional connection table entry. Furthermore, an existing connection can only be replaced if at least one free connection table entry is available to hold data related to the pending enrollment process, even if this connection will later replace another local connection and thus free another connection table record.

9. How can I extend a connection with the MGDemo application?

You can create new connections and replace existing connections using the MGDemo example application. To create a new connection or replace an existing connection, use the **SW5 – SW8** Connect buttons. You can also extend an MGDemo connection using the I/O buttons. To extend an existing connection

using an I/O button, press and hold the **SW1 – SW4** I/O button on the evaluation board running the MGDemo application when accepting or confirming an connection, then press and release the related **SW5 – SW8** Connect button on the evaluation board being added to the connection, and then release the original I/O button again.

Appendix B

Monitoring & Control Application Overview

This appendix describes the structure of the Monitoring & Control C# Example Application, the class interactions, and the different classes it contains.

Monitoring & Control C# Example

The Monitoring & Control Example Application is a C# application that monitors ISI messages and uses the OpenLDV API to monitor and control network variables on devices running the MGDemo example. This chapter describes the structure of the Monitoring & Control C# example.

The Monitoring & Control Example Application is a Windows application written in Microsoft Visual C# .NET 2003. It requires Microsoft .NET Framework 1.1 to run. During the Mini EVK installation, the installer will install the Microsoft .NET Framework 1.1 if your system does not have it.

Monitoring & Control Example Hierarchy

Figure B.1 shows the hierarchy of the classes introduced in this chapter. There are four major segments of the class hierarchy:

- *User Interface and Application Specific Implementation*
- *ISI Support*
- *OpenLDV Adapter*
- *LDV32.DLL*

These are described in more detail in the following sections.

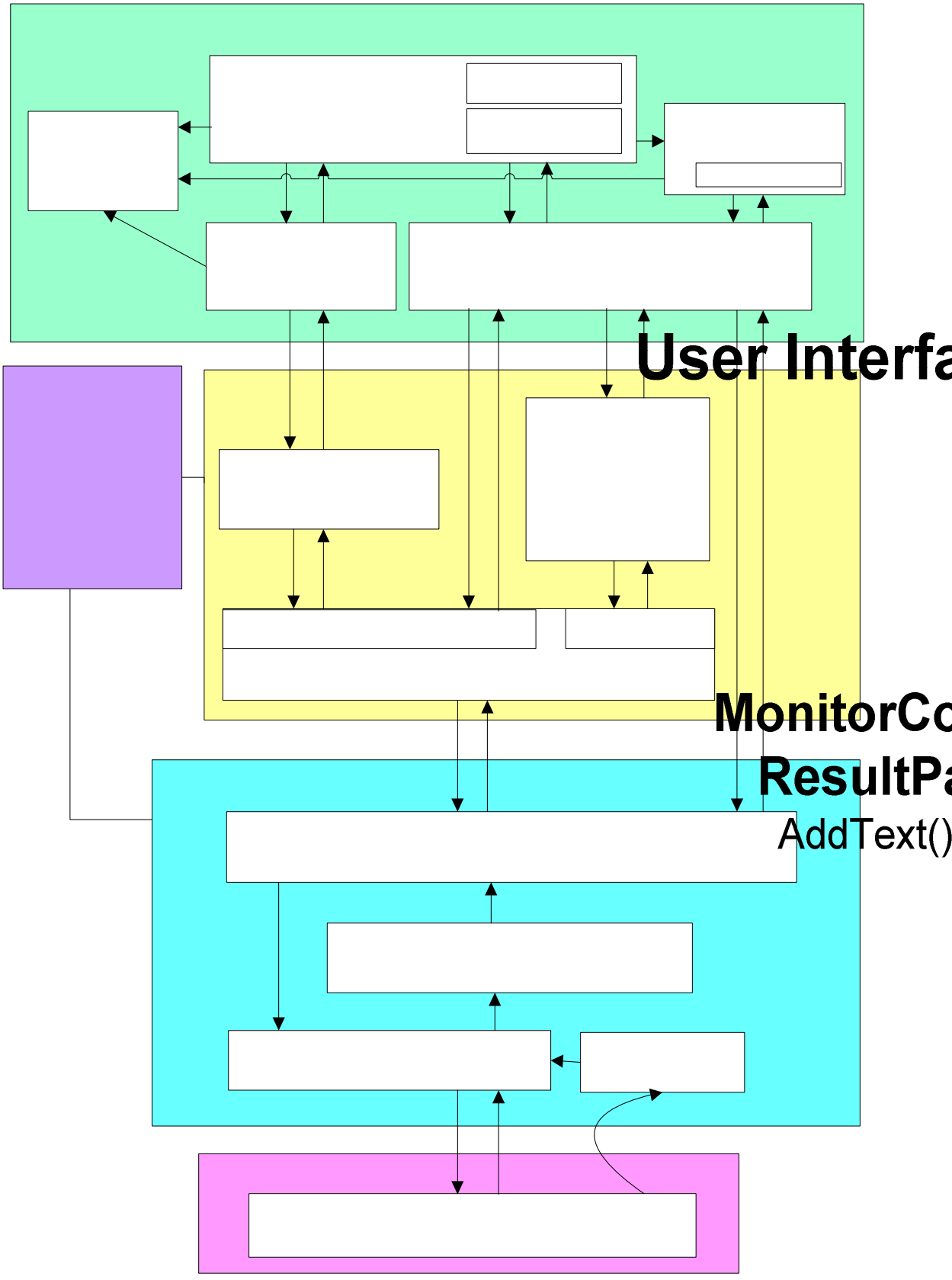

Figure B.1 Monitoring & Control Example Hierarchy

User Interface and Application Specific Implementation

This section describes the classes, files, and dialogs that make up the Monitoring & Control Example Application's user interface.

Main User Interface Window

The Monitoring & Control Example Application provides a main dialog-based user interface. The **MonitorControlMain** class contains the implementation of the main user interface window. This includes the event handlers related to that user interface, such as the various click event handlers related to the buttons.

Network Interface Selection Form

When you start the Monitoring & Control Example Application, the Network Interface dialog opens. You can use this dialog to select a network interface from a list of available network interfaces that are currently registered and attached to the system, can operate as layer 5 network interface, and are not in use by other applications. This information is obtained by using the OpenLDV 2.1 **ldv_get_matching_devices** function to obtain information about the network interface devices that match the specified set of capabilities. The default interface is the last network interface selected, or the first network interface in the list if you are running the software for the first time.

The example code saves the persistent information for the last network interface selected will be saved in the `HKCU\SOFTWARE\MonitorControlExample\Network Interface` Windows Registry entry. After you select a network interface, you can press **Connect** to attach to the network interface or quit the application. If there is an error while attaching to the network interface, an error message will be displayed. You can then try to select a different network interface from the list. The **ConnectNiForm.cs** file contains the implementation of the network interface selection dialog.

Add Device Dialog/Service Pin Handling

You can use the Add Device dialog to register a new device. You can access this dialog by clicking **Add Device** on the main window. You identify the device to be registered by either specifying the device's Neuron ID, or by pressing the device's Service button. When the application receives a service pin message from a device, the Neuron ID of the device that sent the message will be displayed in the Neuron ID box on the Add Device dialog. You can ignore the service pin message, or begin monitoring the device by pressing **OK**. In this case, the main dialog will be reset and all inputs will now be polled from the newly selected device. The **AddDeviceForm** class contains the implementation of the Add Device dialog.

ISI Information Window

The Monitoring & Control Example Application displays ISI messages in the ISI Information window. You can access this window by clicking **ISI Info** on the

main user interface window. The **IsiInfoForm** class contains the implementation of the ISI Information window.

Change Subnet/Node ID Dialog

You can change the Subnet/Node ID of the network interface that is currently used by the Monitoring & Control Example Application with the Change Subnet/Node ID dialog. You can access this dialog by clicking **Change Subnet/Node ID** on the ISI Information window. The **`ChangeSNForm`** class contains the implementation of the Change Subnet/Node ID dialog.

Progress Log Window

The progress log window is created based on the **System.Windows.Forms.UserControl** form. It uses the multiline **TextBox** control to display the text. The **MonitorControlMain** class and the **IsiInfoForm** class use this window to display the progress/log.

Monitor Control Engine

This module contains of the main implementation of the monitoring and control process. The monitoring process is performed on a background thread that uses round-robin network variable polling. It interacts with evaluation boards running the MGDemo application. This module also processes the network variable update that is requested from the **MonitorControlMain** class as a response from some user interactions (pressing the switch button or set/clear the Mute control for the piezo buzzer). The **MonitorControlMain** class subscribes to NVUpdate events to receive the network variable updates. The **MonitorControlEngine** class contains the implementation of the monitoring and control process.

Network Interface Configuration

The Monitoring & Control Example Application supports ISI-compliant management of network addressing, implementing the ISI-S addressing scheme. It uses a fixed, standardized 3-byte domain ID—0x49, 0x53, 0x49 (ASCII "ISI")—on the primary domain, and a zero-length domain in a clone domain configuration on the secondary domain. As an ISI device, this example uses a fixed subnet/node ID of 1/1 in the secondary domain, and for the primary domain it randomly allocates a node ID value from the range 2...125, and also randomly assigns its own subnet ID (with a value between 64 and 127 if using TP/FT-10 transceiver and between 128 and 191 if using PL-20 transceiver).

For more information on domain ID management and Subnet/Node ID allocation for an ISI device, see the *ISI Protocol Specification*. The **MonitorControlNi** class contains the implementation of the network interface configuration.

ISI Support

The Monitoring & Control Example Application includes a typical controller implementation of the ISI protocol. It implements ISI-compliant management of network addressing, the ISI-S addressing scheme, and the ISI periodic DRUM

(domain resource usage message) that helps detect duplicate subnet/node IDs. When the Monitoring & Control Example Application receives the DRUM, it evaluates the subnet/node ID contained in the message. In the event that the collision of subnet/node ID is detected, the example application reallocates a new subnet/node ID and broadcast the new assigned subnet/node ID using the DRUM. The **IsiConfig** class contains the implementation of the ISI configuration.

IsiAppMsg Class

The **IsiAppMsg** class supports ISI messages. This class is derived from the **NiMsg** class, which is the base class for all network interface messages. The example application only uses the DRUM. The class can be modified to support more ISI messages.

Network Management

This section describes the classes and files used for the network management tasks performed by the Monitoring & Control Application.

AppImage Class

The Application Image module reads the application configuration, the SNVT information, and fills in the NV Descriptor table with the SNVT data, the NV direction, and the current connection information for each network variable available on the device. This class also supports NV updates and NV polls.

NetMgmtMsg Class

The **NetMgmtMsg** class contains functions that support the following network operations, both on the local network interface as well as on the remote device (Neuron ID addressing):

- Query Status
- Update Domain
- Leave Domain
- Set Node Mode
- Ready Memory
- Wink

It also supports the following operations on the remote target only:

- Query Net Variable Config
- Query SNVT
- Update NV data
- Fetch NV data

OpenLDV Adapter

This section describes the classes and files of the Monitoring & Control Example Application that invoke the OpenLDV API.

Operator Class

The **Operator** class is derived from the **Dispatcher** class. It completes the dispatching of uplink messages, and processes complex operations messages. It also overrides some of the virtual methods to stop uplink data from propagating into the client application. The Operator also contains transaction services like the **NiSendMsgWait()** class, which handles outgoing messages (addressed to the local network interface or a remote device by subnet/node, Neuron ID, and broadcast) and waits for completion messages.

Dispatcher Class

The **Dispatcher** class is derived from the **Connector** class. It completes the dispatching of uplink messages. All uplink dispatching follows the same scheme: a related virtual method is called, where the default implementation (provided in this class) fires the related **On*** event. The **Dispatcher** class is an example for an application- specific message dispatcher. It can be re-written to adapt to each specific application.

Connector Wrapper Class

The **Connector** class is a Microsoft .NET wrapper that is built on top of the OpenLDV API. The Connecter class provides an interface via the **Open**, **Close**, **Read** and **Write** methods.

The class further provides thread-safe, synchronized access to the downlink message path (**ldv_write)**. The Connector class also implements and controls an uplink reader thread (Listener method), which fetches packets from the interface (**ldv_read**), deciphers the uplink data, and then dispatches it to the relevant event handler.

The **Connector** example class also implements the low-level uplink data notification, so that the application's user interface can subscribe to the event to trace the incoming/outgoing packets.

SessionEventTrap Class

The **SessionEventTrap** class captures xDriver session events. xDriver is a component of OpenLDV that is used to manage connections to remote network interfaces such as the *i*.LON 100 Internet Server. These events require a valid window handle, which is the reason that this class is derived from the **System.Windows.Forms.Form** form and implemented as a hidden form. The form captures the xDriver session events, and relays them back to the Connector class, which then fires the **OnSesionChange** event for client notification.

AppBuffer / ISIAppBuffer

The **AppBuffer.cs** and **ISIAppBuffer** implementation files contain the collection of conversion classes, which together form the application message buffer structure and support conversion between the unmanaged OpenLDV API and the managed Microsoft .NET environment. The explicit conversion classes are required to handle most issues related to **AppBuffer** structures, as Microsoft .NET has no concepts of bit fields, which are used by the OpenLDV interface and API.

LDV32.DLL

The **ldv32.dll** dynamic link library implements the OpenLDV API. This API provides low-level access to any Echelon LONWORKS network interfaces available on your computer. The network interfaces may be directly attached to your computer, as with a U10 or U20 USB network interface, a PCC-10 PC card network interface, or a PCLTA-21 PCI network interface. Or, the network interfaces may be remotely attached via an Ethernet or dial-up connection such as an *i*.LON 10 Ethernet Adaptor, an *i*.LON 100 Internet Server, or an SLTA-10 serial network interface.

When used in conjunction with the *i*.LON 10 Ethernet Adaptor and *i*.LON 100 Internet Server, the OpenLDV driver software provides secure interfaces including RC4 encryption, MD5 authentication, and protection from replay attacks, as well as transparent, fault tolerant session recovery when the IP connection or power to the i.LON interface is interrupted. Support is provided for uplink connections, wherein a remote network initiates a call, either dial-up or broadband, into a service center. Uplink connections are commonly used in large remote access systems in which hundreds or thousands of sites report back to a single service center.

The **ldv32.dll** dynamic link library is an unmanaged DLL. The **Connector** class described in the previous section wraps this DLL with a managed interface.

For more information on developing OpenLDV applications, download and install the OpenLDV Developer's Kit from www.echelon.com/downloads.

www.echelon.com