ECHELON

LonTalk® Stack Developer's Guide

078-0483-01A

Echelon Corporation
www.echelon.com

Table of Contents

Welcome ix
Audience ix
Related Documentation ix

1 Introduction to LonTalk Stack 1
Overview 2
A LONWORKS Device with a Single Processor Chip 3
A LONWORKS Device with Two Processor Chips 4
ShortStack Developer's Kit 4
LonTalk Stack Developer's Kit 6
Comparing Neuron-Hosted, ShortStack, and LonTalk Stack Devices 7
Requirements and Restrictions for LonTalk Stack 9
Development Tools for LonTalk Stack 10
LonTalk Stack Architecture 10
Overview of the LonTalk Stack Development Process 12

2 Getting Started with the LonTalk Stack Developer's Kit 19
LonTalk Stack Overview 20
Installing the LonTalk Stack Developer's Kit 20
Hardware Requirements 20
Software Requirements 20
Installing the LonTalk Stack Developer's Kit 21
LonTalk Stack Files 21
LonTalk Interface Developer 21
Example LonTalk Stack Applications 22

3 Loading the Echelon Smart Transceiver or Neuron Chip 25
Loading Overview 26
Integrating a Custom Network Interface 28
Defining Incoming Layer 2 Packet Buffers 29
Functions 29

4 Designing the Serial I/O Hardware Interface 31
Overview of the Hardware Interface 32
Reliability 32
Serial Communication Lines 32
The RESET~ Pin 33
Selecting the Link-Layer Bit Rate 34
Host Latency Considerations 36
SCI Interface 36
Performing an Initial Echelon Smart Transceiver Health Check 37

5 Creating a LonTalk Stack Serial MIP Driver 39
Overview of the Link Layer Protocol 40
Code Packet Layout 40
Type Code Values 42
Acknowledgment Rules 44
Sequence Number Cycling and Duplicate Detection 45
Supported MIP Command Set 45
Layer 2 / Layer 5 Modes 46
Product Query Network Management 47

Serial MIP Driver Example 47
Serial MIP Driver API 47
Structures 47
Functions 48

6 Creating a Model File 51

Model File Overview 52
Defining the Device Interface 53
Defining the Interface for a LonTalk Stack Application 53
Choosing the Data Type 54
Defining a Functional Block 55
Declaring a Functional Block 56
Defining a Network Variable 56
Defining a Changeable-Type Network Variable 58
Defining a Configuration Property 59
Declaring a Configuration Property 59
Responding to Configuration Property Value Changes 62
Defining a Configuration Property Array 62
Sharing a Configuration Property 64
Inheriting a Configuration Property Type 66
Declaring a Message Tag 67
Defining a Resource File 68
Implementation-Specific Scope Rules 70
Writing Acceptable Neuron C Code 70
Anonymous Top-Level Types 71
Legacy Neuron C Constructs 71
Using Authentication for Network Variables 71
Specifying the Authentication Key 72
How Authentication Works 73
Managing Memory 74
Address Table 74
Alias Table 75
Domain Table 75
Network Variable Configuration Table 76
Example Model files 76
Simple Network Variable Declarations 76
Network Variables Using Standard Types 76
Functional Blocks without Configuration Properties 77
Functional Blocks with Configuration Network Variables 78
Functional Blocks with Configuration Properties Implemented in a Configuration File 79

7 Using the LonTalk Interface Developer Utility 81

Running the LonTalk Interface Developer 82
Specifying the Project File 82
Specifying the Echelon Smart Transceiver or Neuron Chip Configuration 83
Configuring the LonTalk Stack 84
Configuring the Buffers 85
Configuring the Application 86
Configuring Support for Non-Volatile Data 87
Specifying the Device Program ID 88
Specifying the Model File 89
Specifying Neuron C Compiler Preferences 90

Specifying Code Generator Preferences 91
Compiling and Generating the Files 92
Using the LonTalk Interface Developer Files 93
Copied Files 94
LonNvTypes.h and LonCpTypes.h 94
FtxlDev.h 95
FtxlDev.c 95
project.xif and project.xfb 95
Using Types 95
Bit Field Members 97
Enumerations 98
Floating Point Variables 98
Network Variable and Configuration Property Declarations 100
Constant Configuration Properties 102
The Network Variable Table 103
Network Variable Attributes 103
The Message Tag Table 104

8 Developing a LonTalk Stack Device Application 105

Overview of a LonTalk Stack Device Application 106
Using the LonTalk API 106
Callbacks and Events 108
Integrating the Application with an Operating System 108
Providing Persistent Storage for Non-Volatile Data 109
Restoring Non-Volatile Data 110
Writing Non-Volatile Data 111
Tasks Performed by a LonTalk Stack Application 112
Initializing the LonTalk Stack Device 113
Periodically Calling the Event Pump 113
Sending a Network Variable Update 115
Receiving a Network Variable Update from the Network 117
Handling a Network Variable Poll Request from the Network 120
Handling Changes to Changeable-Type Network Variables 120
Validating a Type Change 121
Processing a Type Change 122
Processing a Size Change 123
Rejecting a Type Change 124
Handling Dynamic Network Variables 124
Communicating with Other Devices Using Application Messages 125
Sending an Application Message to the Network 126
Receiving an Application Message from the Network 126
Handling Management Commands 126
Handling Local Network Management Tasks 127
Handling Reset Events 127
Querying the Error Log 127
Working with ECS Devices 127
Using Direct Memory Files 128
The DMF Memory Window 129
File Directory 130
Shutting Down the LonTalk Stack device 131

9 Developing an IP-852 Router Application 133

Developing an IP-852 Router Application 134

LtLogicalChannel 134
LtIp852Router 134

10 Porting a LonTalk Stack Application 137

Porting Overview 138
OSAL 138
LonLink Driver 138
Service LED 139
Socket Interfaces 139
LonTalkStack Source Files 139
Application-Specific Files for LonTalk Stack Devices 141
Application-Specific Code for IP-852 Interfaces 141
Selecting the Device Type 141
File System Requirements 142

Appendix A LonTalk Interface Developer Command Line Usage 143

Overview 144
Command Usage 144
Command Switches 145
Specifying Buffers 147

Appendix B Model File Compiler Directives 151

Using Model File Compiler Directives 152
Acceptable Model File Compiler Directives 152

Appendix C Neuron C Syntax for the Model File 157

Functional Block Syntax 158
Keywords 158
Examples 160
Functional Block Properties Syntax 161
Keywords 161
Examples 162
Network Variable Syntax 164
Keywords 164
The Network Variable Modifier 164
The Network Variable Storage Class 166
The Network Variable Type 166
The Network Variable Connection Information 167
The Network Variable Initializer 170
The Network Variable Property List 170
Configuration Property Syntax 171
Keywords 171
The Configuration Property Type 172
The Configuration Property Modifiers 172
The Configuration Property Initializer 174
Declaring a Configuration Network Variable 175
Defining a Device Property List 175
Message Tag Syntax 177
Keywords 177

Appendix D LonTalk API 179

Introduction 180
The LonTalk API, Event Handler Functions, and Callback Handler Functions 180

LonTalk API Functions 180
Commonly Used LonTalk API Functions 181
Other LonTalk API Functions 181
Application Messaging API Functions 182
Non-Volatile Data API Functions 182
Extended API Functions 183
Event Handler Functions 184
Commonly Used Event Handler Functions 184
Dynamic Network Variable Event Handler Functions 185
Application Messaging Event Handler Functions 186
Non-Volatile Data Event Handler Functions 186
LonTalk Stack Callback Handler Functions 187
Commonly Used Callback Handler Functions 187
Direct Memory Files Callback Handler Functions 188
Non-Volatile Data Callback Handler Functions 188
The Operating System Abstraction Layer 189
Managing Critical Sections 190
Managing Binary Semaphores 190
Managing Operating System Events 190
Managing System Timing 191
Managing Operating System Tasks 191
Debugging Operating System Functions 191

Appendix E Determining Memory Usage for LonTalk Stack Applications 193

Overview 194
Memory Use for Code 194
Memory Use for Transactions 194
Memory Use for Buffers 195
Memory for LONWORKS Resources 195
Memory for Non-Volatile Data 196
Memory Usage Examples for Data 198

Appendix F Downloading a LonTalk Stack Application Over the Network 201

Overview 202
Custom Application Download Protocol 202
Application Download Utility 203
Download Capability within the Application 203

Appendix G Example LonTalk Stack Applications 205

Overview of the Example Applications 206
Building the Example Applications 207
Running the Examples 207
Running the SimpleLtDevice Example 208
Running the SimpleIp852Device Example 208
Running the Ip852Router Example 208
SimpleLtDevice and SimpleIp852Device Example Application Details 208
Main Function 209
Application Task Function 211
Event Handler Function 212
Application-Specific Utility Functions 213
Callback Handler Function 213

Model File 214
Extending the SimpleLtDevice and SimpleIp852 Examples 214
IP-852 Router Example Application Details 215

Appendix H LonTalk Interface Developer Utility Error and Warning Messages 219

Introduction 220
Error Messages 220
Warning Codes 226
Hint Codes 228

Appendix I Glossary 231

Welcome

Echelon's LonTalk® Stack enables you to add a high-performance ISO/IEC 14908-1 control networking interface to any product that contains a microprocessor, microcontroller, or embedded processor. The LonTalk Stack includes a simple host application programming interface (API), a complete ISO/IEC 14908-1 protocol stack implementation, a link-layer driver, a simple hardware interface, and comprehensive tool support.

This document describes how to port the LonTalk Stack to your processor and how to develop an application for a LONWORKS device using the LonTalk Stack. It describes the architecture of a LonTalk Stack device, and how to develop the device's software. Software development of a LonTalk Stack device includes creating a model file, running the LonTalk Interface Developer utility, and using the LonTalk Stack API functions to program your LonTalk Stack application for the host processor.

Audience

This document assumes that the reader has a good understanding of the LONWORKS platform and programming for embedded processors.

Related Documentation

In addition to this manual, the LonTalk Stack documentation suite includes the following manuals:

- *Neuron C Programmer's Guide*. This manual describes the key concepts of programming using the Neuron® C programming language and describes how to develop a LONWORKS application.
- *Neuron C Reference Guide*. This manual provides reference information for writing programs that use the Neuron C language.
- *Neuron Tools Errors Guide*. This manual describes error codes issued by the Neuron C compiler.

The LonTalk Stack also includes the reference documentation for the LonTalk API, which is delivered as a set of HTML files.

After you install the LonTalk Stack software, you can view these documents from the Windows **Start** menu: select **Programs → Echelon LonTalk Stack Developer's Kit**, and then select the document that you want to view.

The following manuals are available from the Echelon Web site (www.echelon.com/docs) and provide additional information that can help you develop LONWORKS applications:

- *Introduction to the LONWORKS Platform*. This manual provides an introduction to the ISO/IEC 14908 1 Control Network Protocol, and provides a high-level introduction to LONWORKS networks and the tools and components that are used for developing, installing, operating, and maintaining them.

- *LONMARK® Application Layer Interoperability Guidelines.* This manual describes design guidelines for developing applications for open interoperable LONWORKS devices, and is available from the LONMARK Web site, www.lonmark.org.

- *FT 3120 / FT 3150 Echelon Smart Transceiver Data Book.* This manual provides detailed technical specifications on the electrical interfaces, mechanical interfaces, and operating environment characteristics for the FT 3120 and FT 3150® Echelon Smart Transceivers.

- *PL 3120®/PL 3150®/PL 3170™ Power Line Smart Transceiver Data Book*. Provides detailed technical specifications on the electrical interfaces, mechanical interfaces, and operating environment characteristics for the PL 3120®, PL 3150® and PL 3170™ Power Line Smart Transceivers. This data book also provides guidelines for migrating applications to the PL Smart Transceiver using the NodeBuilder® FX Development Tool, the Mini FX Evaluation Kit, or the ShortStack® Developer's Kit.

- *Series 5000 Chip Data Book*. Provides detailed specifications on the electrical interfaces, mechanical interfaces, and operating environment characteristics for the FT 5000 Smart Transceiver and Neuron 5000 Processor.

- *OpenLNS Commissioning Tool User's User's Guide*. This manual describes how to use the OpenLNS Commissioning Tool to design, commission, monitor and control, maintain, and manage a network.

All of the LonTalk Stack documentation, and related product documentation, is available in Adobe® PDF format. To view the PDF files, you must have a current version of the Adobe Reader®, which you can download from Adobe at: http://get.adobe.com/reader/.

1

Introduction to LonTalk Stack

This chapter introduces LonTalk Stack for embedded processors. It describes the architecture of a LonTalk Stack device, including a comparison with other LONWORKS device development solutions. It also describes attributes of a LonTalk Stack device, and the requirements and restrictions of the LonTalk Stack.

Overview

Automation solutions for buildings, homes, utility, transportation, and industrial applications include sensors, actuators, and control systems. A *LONWORKS network* is a peer-to-peer network that uses an international-standard control network protocol for monitoring sensors, controlling actuators, communicating with devices, and managing network operation. In short, a LONWORKS network provides communications and complete access to control network data from any device in the network.

The communications protocol used for LONWORKS networks is the ISO/IEC 14908-1 Control Network Protocol. This protocol is an international standard seven-layer protocol that has been optimized for control applications and is based on the Open Systems Interconnection (OSI) Basic Reference Model (the OSI Model, ISO standard 7498-1). The OSI Model describes computer network communications through seven abstraction layers. The implementation of these seven layers in a LONWORKS device provides standardized interconnectivity for devices within a LONWORKS network. The following table summarizes the CNP layers.

OSI Layer		Purpose	Services Provided
7	Application	Application compatibility	Network configuration, self-installation, network diagnostics, file transfer, application configuration, application specification, alarms, data logging, scheduling
6	Presentation	Data interpretation	Network variables, application messages, foreign frame transmission
5	Session	Control	Request/response, authentication
4	Transport	End-to-end communication reliability	Acknowledged and unacknowledged message delivery, common ordering, duplicate detection
3	Network	Destination addressing	Unicast and multicast addressing, routers
2	Data Link	Media access and framing	Framing, data encoding, CRC error checking, predictive carrier sense multiple access (CSMA), collision avoidance, priority, collision detection
1	Physical	Electrical interconnect	Media-specific interfaces and modulation schemes

Echelon's implementation of the ISO/IEC 14908-1 Control Network Protocol is called the *LonTalk protocol*. Echelon has implementations of the LonTalk protocol in several product offerings, including the Neuron firmware (which is included in a ShortStack® Micro Server), OpenLNS Server, SmartServers, i.LON

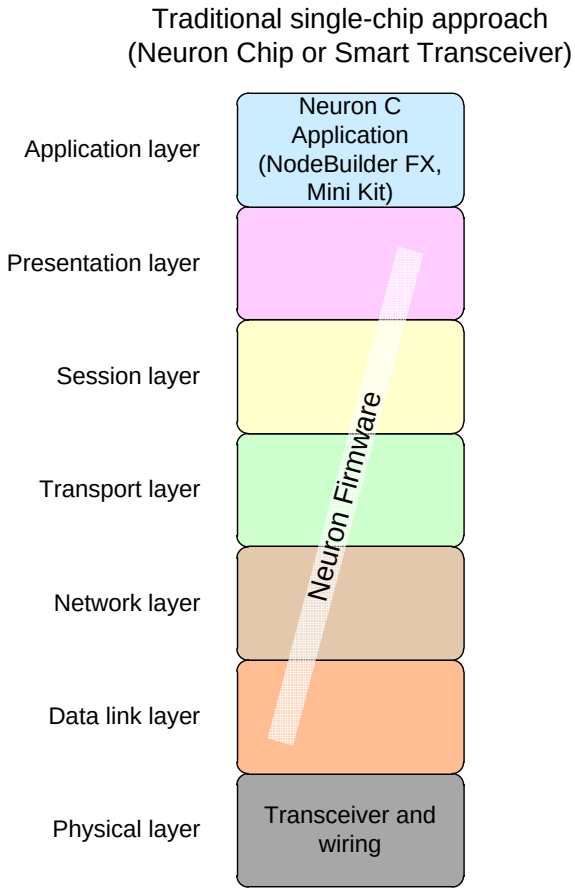
600 IP-852 Routers, and the LonTalk Stack. This document refers to the ISO/IEC 14908-1 Control Network Protocol as the LonTalk protocol, although other interoperable implementations exist.

A LONWORKS Device with a Single Processor Chip

A basic LONWORKS device consists of four primary components:

1. An application processor that implements the application layer, or both the application and presentation layers, of the LonTalk protocol.
2. A protocol engine that implements layers 2 through 5 (or 2 through 7) of the LonTalk protocol.
3. A network transceiver that provides the physical interface for the LONWORKS network communications media, and implements the physical layer of the LonTalk protocol.
4. Circuitry to implement the device I/O.

These components can be combined in a physical device. For example, an Echelon Smart Transceiver product can be used as a single-chip solution that combines all four components in a single chip. When used in this way, the Echelon Smart Transceiver runs the device's application, implements the LonTalk protocol, and interfaces with the physical communications media through a transformer. The following figure shows the seven-layer LonTalk protocol on a single Neuron Chip or Echelon Smart Transceiver.



A LONWORKS device that uses a single processor chip is called a *Neuron-hosted* device, which means that the Neuron-based processor (the Echelon Smart Transceiver) runs both the application and the LonTalk protocol.

For a Neuron-hosted device that uses a Neuron Chip or Echelon Smart Transceiver, the physical layer (layer 1) is handled by the Neuron Chip or Echelon Smart Transceiver. The middle layers (layers 2 through 6) are handled by the Neuron firmware. The application layer (layer 7) is handled by your Neuron C application program. You can create the application program using the Neuron C programming language in either the NodeBuilder® FX Development Tool or the Mini FX.

A LONWORKS Device with Two Processor Chips

Some LONWORKS devices run applications that require more memory or processing capabilities than a single Neuron Chip or Echelon Smart Transceiver can provide. Other LONWORKS devices are implemented by adding a transceiver to an existing processor and application. For these applications, the device uses two processor chips working together:

- An Echelon Smart Transceiver or Neuron Chip.
- A microprocessor, microcontroller, or embedded processor. This is typically called the *host processor*.

A LONWORKS device that uses two processor chips is called a *host-based* device, which means that the device includes an Echelon Smart Transceiver or Neuron Chip plus a host processor.

Compared to the single-chip device, the Echelon Smart Transceiver or Neuron Chip implements only a subset of the LonTalk protocol layers. The host processor implements the remaining layers and runs the device's application program. The Echelon Smart Transceiver or Neuron Chip and the host processor communicate with each other through a link-layer interface.

For a single-chip, Neuron-hosted, device you write the application program in Neuron C. For a host-based device, you write the application program in ANSI C, C++, or other high-level language, using a common application framework and application programming interface (API). This API is called the *LonTalk API*. In addition, for a host-based device, you select a suitable host processor and use the host processor's application development environment, rather than the NodeBuilder FX Development Tool or the Mini FX application, to develop the application.

Echelon provides the following solutions for creating host-based LONWORKS devices:

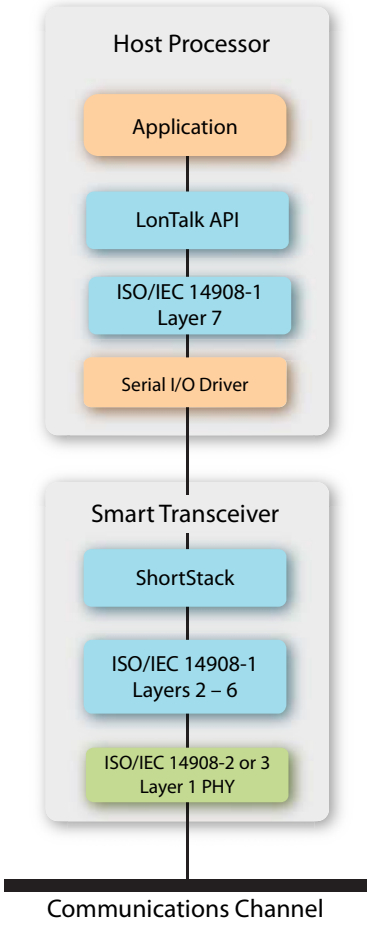
- ShortStack Developer's Kit
- LonTalk Stack Developer's Kit

ShortStack Developer's Kit

The ShortStack Developer's Kit is a set of development tools, APIs, and firmware for developing host-based LONWORKS devices that use the LonTalk Compact API and a ShortStack Micro Server.

A ShortStack Micro Server is an Echelon Smart Transceiver or Neuron Chip with ShortStack firmware that implements layers 2 to 5 (and part of layer 6) of the LonTalk protocol. The host processor implements the application layer (layer 7) and part of the presentation layer (layer 6). The Echelon Smart Transceiver or Neuron Chip provides the physical interface for the LONWORKS communications channel. The ShortStack firmware allows you to use almost any host processor for your device's application and I/O. The following figure displays the ShortStack solution for a host-based LONWORKS device.

A simple serial communications interface provides communications between the ShortStack Micro Server and the host processor. Because a ShortStack Micro Server can work with any host processor, you must provide the serial driver implementation, although Echelon does provide the serial driver API and an example driver for some host processors. An example driver is available for an Atmel® ARM7 microprocessor.



For ShortStack device development, you use the C or C++ programming language. Alternatively, you can develop ShortStack devices using any programming language supported by the host processor if you port the LonTalk Compact API and the application framework generated by the LonTalk Interface Developer utility to that language.

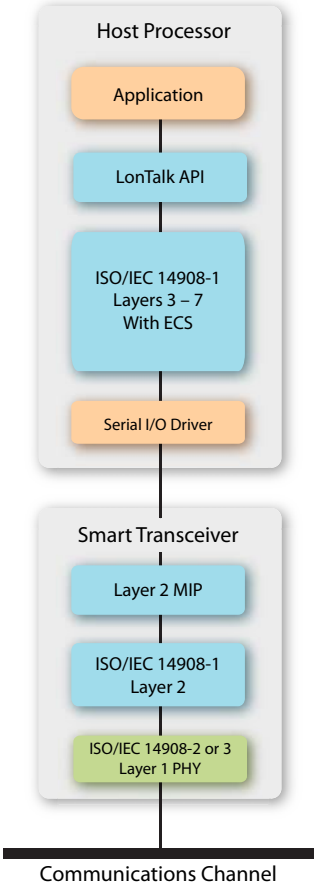
You use the Echelon LonTalk Interface Developer (LID) utility to create the application framework. Your application uses the Echelon LonTalk Compact

API, which is an ANSI C API, to manage communications with the ShortStack Micro Server and devices on the LONWORKS network.

LonTalk Stack Developer's Kit

The LonTalk Stack Developer's Kit is a set of development tools, APIs, and firmware for developing host-based LONWORKS devices that use the Echelon Smart Transceiver or Neuron Chip, a Layer 2 Microprocessor Interface Program (MIP), and the LonTalk API. You can also use the LonTalk Stack to create controllers that are attached to IP-852 channels, and IP-852 routers that route packets between IP-852 and native LonTalk channels.

The Echelon Smart Transceiver or Neuron Chip includes Neuron firmware that implements the data link layer (layer 2) of the LonTalk protocol. The LonTalk Stack provides a Layer 2 MIP that transforms the Echelon Smart Transceiver or Neuron Chip into a network interface that can transmit and receive any packet from the host processor. The LonTalk Stack includes source code that implements layers 3 to 6 and part of layer 7 of the LonTalk protocol that you port to your host processor. The LonTalk Stack also includes a LonTalk API implementation that you port to your host processor that you integrate with your application. This solution enables you to develop high-performance controllers with up to 4,096 network variables and 32,767 address table entries.



To develop the application for your host processor, you use a C or C++ compiler that supports the embedded processor. You will use the Echelon LonTalk Interface Developer utility to create the application framework, and then you can

develop your application using the Echelon LonTalk API to manage communications between the LonTalk Host stack, Echelon Smart Transceiver or Neuron Chip, and other LONWORKS devices on the network.

The LonTalk Stack includes an example communications interface driver for the serial link layer that manages communications between the LonTalk Host stack within the host processor and the Echelon Smart Transceiver or Neuron Chip with Layer 2 MIP. You need to include the physical implementation of the serial link layer in your LonTalk Stack device design, and you need to create the software implementation of the serial interface driver.

Comparing Neuron-Hosted, ShortStack, and LonTalk Stack Devices

The following table compares some of the key characteristics of the Neuron-hosted and host-based solutions for LONWORKS devices.

Characteristic	Neuron-Hosted Solution	ShortStack FX	LonTalk Stack
Maximum number of network variables	254	254 [1]	4096
Maximum number of aliases	254	127 [2]	8192
Maximum number of addresses	15	15	32767
Maximum number of dynamic network variables	0	0	4096
Maximum number of receive transaction records	16	16	32767
Maximum number of transmit transaction records	2	2	32767
Support for the LonTalk Extended Command Set	No	No	Yes [3]

<table>
<tr><td>File access methods supported</td><td>FTP [4] , DMF</td><td>FTP [4], DMF</td><td colspan="2">FTP [4], DMF [5]</td></tr>
<tr><td>Link-layer type</td><td>N/A</td><td>4- or 5-line SCI
or
6- or 7-line SPI</td><td colspan="2">2-line SCI</td></tr>
<tr><td>Typical host API runtime footprint</td><td>N/A</td><td>5-6 KB code with 1 KB RAM (includes serial driver, but does not include optional API or ISI API)</td><td colspan="2"><table><tr><td>Native LonTalk protocol stack.
Includes LonTalk API, Linux Serial MIP driver, and the SimpleDevice example application.</td><td>850 KB</td></tr><tr><td>IP-852 LonTalk protocol stack.
Includes LonTalk API, Linux Serial MIP driver, and the SimpleIp852Device example application.</td><td>955 KB</td></tr><tr><td>Native LonTalk to IP-852 Router.
Includes Linux Serial MIP driver, and the Ip852Router example application.</td><td>965 KB</td></tr></table></td></tr>
<tr><td>Host processor type</td><td>N/A</td><td>Any 8-, 16-, 32-, or 64-bit microprocessor or microcontroller</td><td colspan="2">Any 32- or 64-bit microprocessor or microcontroller</td></tr>
<tr><td>Application development language</td><td>Neuron C</td><td>Any (typically ANSI C)</td><td colspan="2">ANSI C or C++ for the embedded processor</td></tr>
</table>

Notes:

1. ShortStack Micro Servers running on FT 3150 or PL 3150 Echelon Smart Transceivers support up to 254 network variables. ShortStack Micro Servers running on FT 3120 Echelon Smart Transceivers support up to 240 network variables, and ShortStack Micro Servers running on PL 3120 Echelon Smart Transceivers support up to 62 network variables. A custom Micro Server can support up to 254 network variables, depending on available resources.
2. ShortStack Micro Servers running on FT 3150 or PL 3150 Echelon Smart Transceivers support up to 127 aliases. ShortStack Micro Servers running on FT 3120 Echelon Smart Transceivers support up to 120 aliases. ShortStack Micro Servers running on PL 3120 Echelon Smart Transceivers support up to 62 aliases. A custom Micro Server can support up to 127 aliases, depending on available resources.
3. See the *ISO/IEC 14908-1 Control Network Protocol Specification* for more information about the extended command set (ECS) network management commands. This document is available from ISO: www.iso.org/iso/home/store/catalogue_tc/catalogue_detail.htm?csnumber=60203
4. An implementation of the LONWORKS file transfer protocol (FTP) is not provided with the product.
5. For more information about the direct memory files (DMF) feature, see *Using Direct Memory Files*.

The LonTalk Stack solution provides support for any host processor with the highest performance and highest network capacity, and it can be used on native LONWORKS and IP-852 channels. The ShortStack solution provides support for any host processor, and supports TP/FT-10 and PL-20 channels. The ShortStack solution supports fewer network variables and aliases that the LonTalk Stack solution.

Requirements and Restrictions for LonTalk Stack

The LonTalk Stack requires that the application on the host processor use either an embedded operating system or software that implements a minimum set of operating system services.

The LonTalk Stack require about 850 KB of program memory on the host processor, not including the application program or the operating system. In addition, you must provide sufficient additional non-volatile memory for device configuration data and any non-volatile data that you include in your application.

You can implement configuration properties as configuration network variables or in configuration files. To access configuration files, you can implement the LONWORKS file transfer protocol (FTP) or use the direct memory files (DMF) feature. See *Using Direct Memory Files* for more information about when to use FTP or the DMF feature.

Development Tools for LonTalk Stack

To develop an application for a device that uses the LonTalk Stack, you need a development system for the host processor. In addition, you need the LonTalk Stack Developer's Kit, which includes:

- LonTalk API
- LonTalk Host Stack
- LonTalk Interface Developer utility for defining the interface for your LonTalk Stack device and generating the application framework
- Example LonTalk Stack applications

If you are not using an FT 5000 Smart Transceiver with serial interface to your host, you will also need a NodeBuilder FX Development Tool or Mini FX Evaluation Kit to develop the MIP image for your network interface.

You also need a network management tool to install and test your LonTalk Stack device. You can use the OpenLNS Commissioning Tool, or any other tool that can install and monitor LONWORKS devices. See the *OpenLNS Commissioning Tool User's Guide* for more information on the OpenLNS Commissioning Tool.

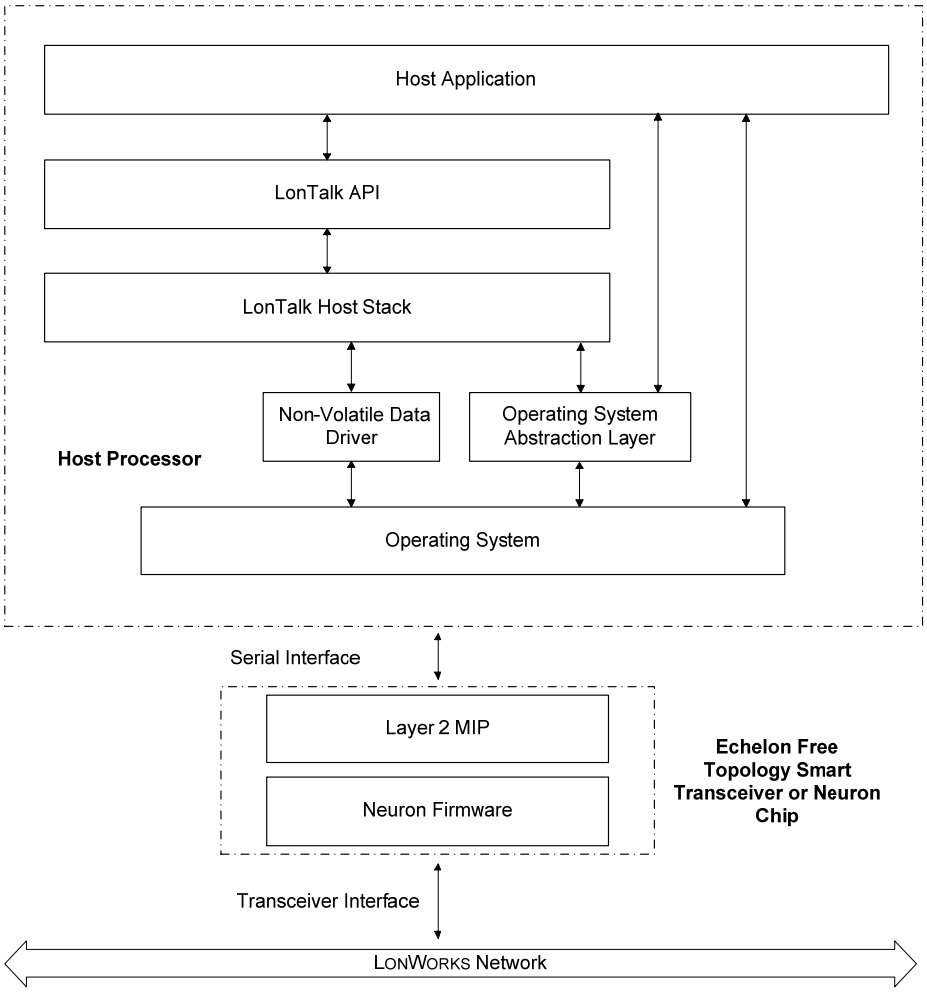
You can use NodeBuilder Code Wizard that is included with the NodeBuilder FX tool, version 3 or later, to help develop your Neuron C model file. The model file is used to define the device's interoperable interface.

LonTalk Stack Architecture

A LonTalk Stack device consists of the following components:

- Echelon Smart Transceiver or Neuron Chip with a Layer 2 MIP.
- A microprocessor, microcontroller, or embedded processor running the following software:
    - Host application that uses the LonTalk API.
    - LonTalk API
    - LonTalk host stack.
    - Non-volatile data (NVD) driver.
    - Operating system abstraction layer (OSAL).
    - Embedded operating system.
    - Serial I/O driver.

The following figure shows the basic architecture of a LonTalk Stack device.



The LonTalk Stack includes source code for the LonTalk API and the LonTalk host stack. The kit also includes source code for additional operating system and hardware APIs that you compile and link with your application. The LonTalk API defines the functions that your application calls to communicate with other devices on a LONWORKS network. The API code provides ANSI C interfaces for the host application.

The LonTalk API consists of the following types of functions:

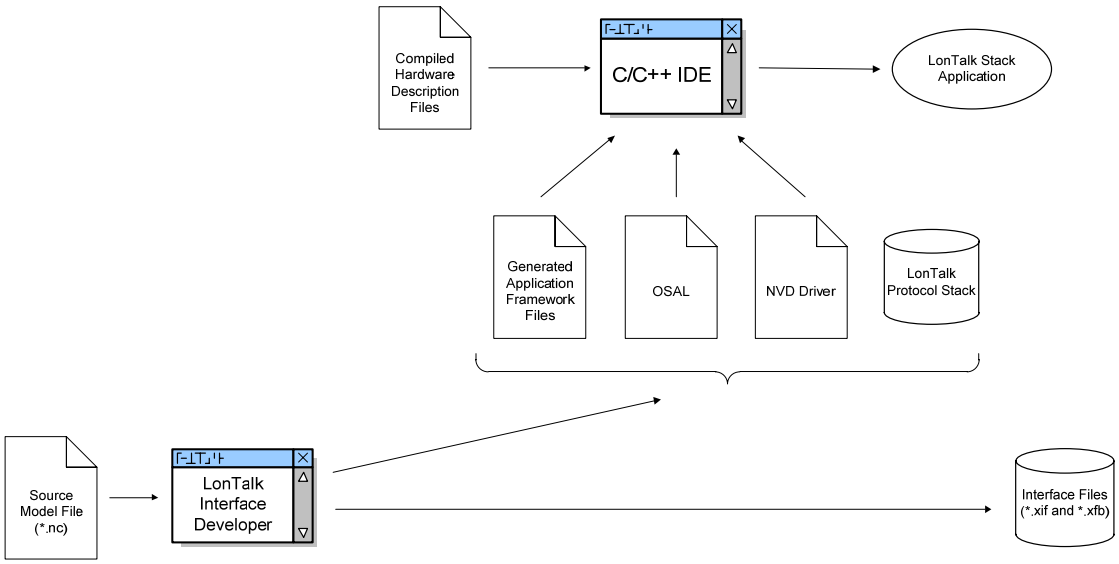
- Functions to initialize the host device after each reset.
- A function that the application must call periodically. This function processes messages pending in any of the data queues.
- Various functions to initiate typical operations, such as the propagation of network variable updates.
- Event handler functions to notify the application of events, such as the arrival of network variable data or an error in the propagation of an application message.
- Functions to interface with the operating system.

Overview of the LonTalk Stack Development Process

The development process for a LonTalk Stack application includes the following steps:

1. Load the Neuron firmware and the Layer 2 MIP on the Echelon Smart Transceiver or Neuron Chip.
2. Create the serial I/O hardware interface between your host processor and the Echelon Smart Transceiver or Neuron Chip.
3. Develop a LonTalk Stack serial driver for your host processor that manages the handshaking and data transfers between the host processor and the Echelon Smart Transceiver or Neuron Chip.
4. Create a model file that defines the interoperable interface of your LonTalk Stack device, including its network inputs and outputs.
5. Use the LonTalk Interface Developer utility to generate application framework files and interface files from the model file.
6. Use a C/C++ development tool to create the LonTalk Stack application, with input from:
   - The application framework files generated by the LonTalk Interface Developer utility
   - The operating system abstraction layer (OSAL) files, which you might need to modify
   - The non-volatile data (NVD) driver files, which you might need to modify
   - The LonTalk host stack
   - The LonTalk API

A LonTalk Stack device is comprised of both hardware and software components; therefore, different people can be involved in the various steps, and these steps can occur in parallel or sequentially. The figure does not imply a required order of steps.



This manual describes the software development process for creating a LonTalk Stack device, which includes the general tasks listed in the following table.

Task	Additional Considerations	Reference
Install the LonTalk Developer's Kit and become familiar with it		Chapter 2, *Getting Started with the LonTalk Stack Developer's Kit*
Load an application image file with the Neuron firmware and Layer 2 MIP onto an Echelon Smart Transceiver or Neuron Chip.		Chapter 3, *Loading the Echelon Smart Transceiver or Neuron Chip*
Create the hardware interface between your host processor and the Echelon Smart Transceiver or Neuron Chip.		Chapter 4, *Designing the Serial I/O Hardware Interface*

Task	Additional Considerations	Reference
Develop a LonTalk Stack serial driver for your host processor that manages the handshaking and data transfers between the host processor and the Echelon Smart Transceiver or Neuron Chip.		Chapter 5, *Creating a LonTalk Stack Serial Driver*
Select a microprocessor, microcontroller, or embedded processor.	The LonTalk Stack application runs on any microprocessor, microcontroller, or embedded processor. You must meet the LonTalk Stack hardware and software requirements to ensure that the LonTalk Stack device has sufficient RAM and non-volatile memory.	
Integrate the LonTalk Stack application with your device hardware	You integrate the Echelon Smart Transceiver or Neuron Chip with the device hardware. You can reuse many parts of a hardware design for different applications to create different LonTalk Stack devices.	
Test and verify your hardware design	You must ensure that the host processor and the Echelon Smart Transceiver or Neuron Chip can communicate using the serial interface.	
Select and define the functional profiles and resource types for your device using tools such as the NodeBuilder Resource Editor and the SNVT and SCPT Master List	You must select profiles and types for use in the device's interoperable interface for each application that you plan to implement. This selection can include the definition of user-defined types for network variables, configuration properties or functional profiles. A large set of standard definitions is also available and is sufficient for many applications.	Chapter 6, *Creating a Model File*

Task	Additional Considerations	Reference
Structure the layout and interoperable interface of your LonTalk Stack device by creating a model file	You must define the interoperable interface for your device in a model file, using the Neuron C (Version 2.1) language, for every application that you implement. You can write this code by hand, derive it from an existing Neuron C application, or use the NodeBuilder Code Wizard included with the NodeBuilder Development Tool to create the required code using a graphical user interface.	Chapter 6, *Creating a Model File* Appendix C, *Neuron C Syntax for the Model File*
Use the LonTalk Interface Developer utility to generate device interface data, device interface files, and a skeleton application framework	You must execute this utility, a simple click-through wizard, whenever the model file changes or other preferences change. The utility generates the interface files (including the XIF file) and source code that you can compile and link with your application. This source code includes data that is required for initialization and for complete implementations of some aspects of your device.	Chapter 7, *Using the LonTalk Interface Developer Utility*
Complete the LonTalk API event handler functions and callback handler functions to process application-specific LONWORKS events	You must complete the event handler functions and callback handler functions for every application that you implement, because they provide input from network events to your application, and because they are part of your networked device's control algorithm.	Chapter 8, *Developing a LonTalk Stack Device Application* Appendix D, *LonTalk API*
Modify the Operating System Abstraction Layer (OSAL) files for your application's operating system		*Integrating the Application with an Operating System* in Chapter 8, *Developing a LonTalk Stack Device Application*
Modify the non-volatile data (NVD) driver files	Depending on the type of non-volatile memory that your device uses, you can use one of the non-volatile data drivers provided with the LonTalk Stack, make minor modifications to one of these drivers, or implement your own driver.	*Providing Persistent Storage for Non-Volatile Data* in Chapter 8, *Developing a LonTalk Stack Device Application*

Task	Additional Considerations	Reference
Modify your application to interface with a LONWORKS network by using the LonTalk API function calls	You must make these function calls for every application that you implement. These calls include, for example, calls to the **LonPropagateNv()** function that propagates an updated network variable value to the network. Together with the completion of the event and callback handler functions, this task forms the core of your networked device's control algorithm.	Chapter 8, *Developing a LonTalk Stack Device Application* Appendix D, *LonTalk API*
Test, install, and integrate your LonTalk Stack device using a LONWORKS network tool such as the OpenLNS Commissioning Tool		*OpenLNS Commissioning Tool User's Guide*

2

Getting Started with the LonTalk Stack Developer's Kit

This chapter describes the LonTalk Stack and how to install it.

LonTalk Stack Overview

The LonTalk Stack Developer's Kit contains the source code, firmware, and documentation required to add a high-performance ISO/IEC 14908-1 control networking interface to any smart device. The LonTalk Stack Developer's Kit includes the following components:

- C and C++ source code for the LonTalk host stack and LonTalk API
- Neuron image for a Layer 2 MIP for devices that use an FT 5000 for the network interface
- Library with the Layer 2 MIP for devices that do not use an FT 5000 for the network interface
- A set of example programs that demonstrate how to use the LonTalk API to communicate with a LONWORKS network
- The LonTalk Interface Developer utility, which defines parameters for your host application program and generates required device interface data for your device
- Documentation, including this guide and HTML documentation for the LonTalk API

Installing the LonTalk Stack Developer's Kit

The following sections describe the hardware and software requirements, and how to install the LonTalk Stack.

Note: The LonTalk Stack Developer's Kit is not compatible with the FTXL Developer's Kit. You must uninstall the FTXL Developer's Kit before installing the LonTalk Stack Developer's Kit on your computer.

Hardware Requirements

For the LonTalk Stack Developer's Kit, your computer system must meet the following minimum requirements:

- 1 gigahertz (GHz) or faster 32-bit (x86) or 64-bit (x64) processor
- 1 gigabyte (GB) RAM (32-bit) or 2 GB RAM (64-bit)
- 5 GB available hard disk space

In addition, you must have the following hardware for LONWORKS connectivity:

- LONWORKS compatible network interface, such as a U10 USB Network Interface, SmartServer, or i.LON 600 IP-852 Router.

Software Requirements

For the LonTalk Stack, your computer system must meet one of the following minimum requirements:

- Microsoft Windows 7 (32-bit or 64-bit).
- Microsoft Windows Vista.

- Microsoft® Windows® XP, plus Service Pack 3 or later.

Installing the LonTalk Stack Developer's Kit

To install the LonTalk Stack Developer's Kit, perform the following steps:

1. Download and install the **LonTalkStack200.exe** file from the Echelon Web site.
2. Follow the installation dialogs to install the LonTalk Stack Developer's Kit onto your computer.

In addition to the LonTalk Stack, the installation program also installs:

- LONMARK® Resource Files
- NodeBuilder Resource Editor

LonTalk Stack Files

The LonTalk host stack and LonTalk API are provided as portable ANSI C and C++ files. These files are contained in the [*LonTalkStack*]**Source** directory (the LonTalk Stack Developer's Kit typically installs the *LonTalkStack* directory in the **C:\LonWorks** directory). The LonTalk Interface Developer utility automatically copies these files from the **LonTalkStack\Templates** folder into your project folder, but does not overwrite existing files with the same names.

The following table lists the files included in the LonTalk host stack and LonTalk API. Many of the files are also used by the FTXL Developer's Kit and therefore have an FTXL prefix in their name.

File Name	Description
FtxlApi.h	Function definitions for the LonTalk API
FtxlHandlers.c	Function definitions for the event handler functions and callback handler functions
FtxlNvdFlashDirect.c **FtxlNvdFlashFs.c** **FtxlNvdUserDefined.c**	Functions for managing non-volatile data
FtxlTypes.h	C type definitions that are used by the LonTalk API
LonPlatform.h	Definitions for adjusting your compiler and development environment to the requirements of the LonTalk API

LonTalk Interface Developer

The LonTalk Interface Developer utility generates the device interface data and device interface files required to implement the device interface for your LonTalk Stack device. It also creates a skeleton application framework that you can

modify and link with your application. This framework contains most of the code that is needed for initialization and other required processing.

The executable for the LonTalk Interface Developer utility is named **LID.exe**, and is installed in the LonTalk Interface Developer directory (usually, **C:\LonWorks\InterfaceDeveloper**).

The LonTalk Interface Developer utility also includes a command-line interface that allows make-file and script-driven use of the utility. For more information about the command-line interface, see Appendix A, *LonTalk Interface Developer Command Line Usage*.

For more information about the LonTalk Interface Developer utility, see Chapter 7, *Using the LonTalk Interface Developer Utility*.

Example LonTalk Stack Applications

The LonTalk Stack Developer's Kit includes three example applications that are stored in the **LonWorks\LonTalkStack\Examples** directory. You can build these example applications with Microsoft Visual Studio 2008, and then run them on Windows. To run the examples, you must install OpenLDV 4.0, which you can download for free from the Echelon Web site at www.echelon.com/support/downloads. The following table describes these three example applications:

Function	Description
SimpleLtDevice	Simulates a voltage amplifier device. This device receives an input voltage value, multiplies the value by 2, and outputs the new value. This simulated device connects to a native LonTalk channel via OpenLDV 4.0 (or later), using a standard LONWORKS network interface. This example requires a Layer 2 network interface such as the Echelon U10 USB Network Interface.
SimpleIp852Device	Identical to the **SimpleLtDevice** example, but it connects to an IP-852 channel rather than a native LONWORKS channel. This example requires the Echelon IP-852 Configuration Server (you can download from this app for free from the Echelon Web site at www.echelon.com/support/downloads).
Ip852Router	A router that connects an IP-852 channel to a native LONWORKS channel. This example uses OpenLDV 4.0 (or later) and a standard Layer 2 LONWORKS network interface to communicate with the native LONWORKS channel (for example, U10 USB network interface or PCC-10, PCLTA-20, or PCLTA-21 network interface card).

See Appendix G, *Example LonTalk Stack Applications*, for more information about these examples.

3

Loading the Echelon Smart Transceiver or Neuron Chip

This chapter describes how to load an application image with the Neuron firmware and Layer 2 MIP onto an Echelon Smart Transceiver or Neuron Chip.

Loading Overview

To create a LonTalk Stack device, you first need to load an Echelon Smart Transceiver or Neuron Chip with an application image file. The application image contains Neuron firmware that implements the data link layer of the LonTalk protocol (layer 2), and a Layer 2 MIP that enables the Echelon Smart Transceiver or Neuron Chip to transmit and receive any packet to and from the host processor.

You can load the Echelon Smart Transceiver or Neuron Chip using one of the following three options:

1. Load an Echelon-provided pre-compiled application image file onto an Echelon FT 5000 Smart Transceiver or PL 3120 Smart Transceiver.

   - The FT 5000 Smart Transceiver must be running Neuron Firmware Version 19, using a 20 MHz clock speed, and be attached to a TP/FT-10 channel.
   - The PL 3120 Chip must be running Neuron Firmware Version 14 and be attached to a PL-20 channel.

   The application images files are stored in the **LonWorks\LonTalkStack\Source\Target\Neuron\SMIP** directory. This folder includes .**NME** and .**NDL** files for the FT 5000 Smart Transceiver (**SMIP FT5000.NME** or **SMIP FT5000.NDL**) and the PL 3120 Smart Transceiver (**SMIP PL3100.NME** or **SMIP PL3100.NDL**)

   You can program the .**NME** file directly on the serial EEPROM of the FT 5000 Smart Transceiver. You can load the **.NDL** file on the FT 5000 Smart Transceiver or PL 3120 Smart Transceiver using OpenLNS Commissioning Tool or the NodeLoad utility.

2. Create your own application image with the NodeBuilder FX Development Tool or the Mini FX Evaluation Kit and load it onto a FT 5000 Echelon Smart Transceiver, Series 5000 chip, or Neuron 3120E4 Chip with the appropriate programming tool. For more information on the NodeBuilder tool and the Mini kit, go the Echelon Web site at www.echelon.com/products/tools/development.

   In this scenario, you create your own Neuron C application that specifies the baud rate and the network buffering for the Echelon Smart Transceiver or Neuron Chip. Specify a baud rate of **115,200** to make your Echelon Smart Transceiver compatible with the provided Serial MIP driver example. If you use a different baud rate, update the baud rate in the Serial MIP driver example to make it compatible with your Echelon Smart Transceiver.

   You then generate an application image that includes your Neuron C application and the Layer 2 MIP library (**smip_ft5000.lib** or **smip_pl3100.lib**), and load the application image onto the Echelon Smart Transceiver or Neuron Chip. The LonTalk Stack Developer's Kit includes a Neuron C application example that can be used to build the SMIP (**SMIP PDT.NC**).

The following table lists the Neuron processor and memory combinations, and it lists the application image files and tools that you use to program each onto an Echelon Smart Transceiver or Neuron Chip

Echelon Smart Transceiver	Memory Type	Image File Extension	Programming Tool	Example Programming Tools
Neuron 3120E4 Chip	On-chip EEPROM	APB, NDL, or NEI	Network management tool	NodeLoad utility OpenLNS CT
		NFI	PROM programmer	A universal programmer, such as one from BPM Microsystems or HiLo Systems
FT 5000 Echelon Smart Transceiver	Off-chip EEPROM (minimum 4K) or flash.	APB or NDL	Network management tool	NodeLoad utility OpenLNS CT
		NME or NMF	EEPROM or flash programmer	A universal programmer, such as one from BPM Microsystems or HiLo Systems In-circuit programmer, such as Total Phase™ Aardvark™ I2C/SPI Host Adapter

Notes:

- Information about the NodeLoad utility and OpenLNS CT is available from www.echelon.com.
- Information about BPM Microsystems programmer models is available from www.bpmicro.com. The Forced Programming option in the menu is provided only to refresh the internal memory contents and should not be used to program new devices. In this mode, the programmer simply reads out the contents of the memory and rewrites them.
- Information about HiLo Systems manual programmer models is available from www.hilosystems.com.tw.
- Information about TotalPhase programmers is available from www.totalphase.com.

Notes:

- If you load an NDL file with the NodeLoad Utility, the last step of the process may generate errors when the final network management status checks are performed.
- To prevent link errors, you need to copy an updated symbol file to the appropriate Neuron firmware folder on your development computer, and then specify them as custom system images in the Hardware Template Editor. This file has additional symbols for low-level serial interrupt modifications, and access to the network buffer queues.

  The following table lists where the updated symbol files are stored for the FT 5000 Smart Transceiver and the PL 3120 Smart Transceiver, and to where they need to be copied on your development computer.

Echelon Smart Transceiver	Updated Symbol File	Destination Folder on Development Computer
PL 3120 Smart Transceiver	**Source/Target/Neuron/ Ver14/ b3120E4Xl2smip.sym**	**C:/LonWorks/Images/Ver19**
FT 5000 Smart Transceiver	**Source/Target/Neuron/ Ver19/bft5000l2smip.sym**	**C:/LonWorks/Images/Ver14**

- Before you load an application image onto the Echelon Smart Transceiver or Neuron Chip, you must reset the node and hold the service pin low for 5 seconds to put the node into the application-less state.

  Alternatively, your host application can send the **niMODE_L5** local network interface command to the Layer 2 MIP to switch it to Layer 5 mode. The Layer 2 MIP can then process most network management commands so that a network loader can load the application image.

3. The same as option 2, but you also develop code that implements the network interface with your host processor. See the next section, Integrating a Custom Network Interface, for more information.

Integrating a Custom Network Interface

You can create your own network interface and integrate it with your host processor. The following sections describe the APIs included in the **l2mlib.h** file that you can use to create a custom network interface.

Before creating your network interface, you need to copy two additional updated symbol files to the Version 14 Neuron firmware folder on your development computer, and then specify them as custom system images in the Hardware Template Editor. The following table lists where the updated symbol files are stored, and to where they need to be copied on your development computer.

Updated Symbol File	Destination Folder on Development Computer
Source/Target/Neuron/L2MLIB/ Ver14/sys3150l2mlib.sym	**C:/LonWorks/Images/Ver14**
Source/Target/Neuron/L2MLIB/ Ver14/sys3150l2mlib.nx	**C:/LonWorks/Images/Ver14**

Defining Incoming Layer 2 Packet Buffers

You can define incoming Layer 2 packet buffers using the following syntax:

```
[length] [miCOMM|miINCOMINGL2] [backlog/altpath] NPDU [CRC_HI] [CRC_LO]
```

The length field includes the 2 bytes before the NPDU, the NPDU itself, and the two CRC bytes.

Functions

The following table describes the functions included in the **l2mlib.h** file that you can use to create a custom network interface.

Function	Syntax	Description
l2ml_gol2()	`extern system far void l2ml_gol2(void);`	Switches the network processing to L2 mode. If it is already in L2 mode, this method does nothing. You can return to the scheduler when in L2 mode as the network processor believes there is no incoming traffic.
l2ml_gol5()	`extern system far void l2ml_gol5(void);`	Switches the network processing to L5 mode. If it is already in L5 mode, this method does nothing.
l2ml_getl2packet()	`extern system far unsigned *l2ml_getl2packet(void);`	Returns any L2 packet buffers in the receive queue. This method returns NULL if there are no L2 packet buffers available.

Function	Syntax	Description
l2ml_freel2packet()	`extern system far void l2ml_freel2packet(unsigned *mp);`	Frees any packet buffers returned by **l2ml_getl2packet()** method. This method does not check for NULL pointers. You can only free packet buffers after processing.
l2ml_allocl2buffer()	`extern system far unsigned *l2ml_allocl2buffer(void);` The **length** field indicates the size of the packet buffer (including overhead). The **miCOMM** field is [miCOMM | miTQ] **0x12**. The format of this buffer is [length] [miCOMM | miTQ] [backlog/altpath] **NPDU**.	Allocates a L2 packet buffer for transmitting. This method returns NULL if there are no L2 packet buffers available.
l2ml_sendl2packet()	`extern system far void l2ml_sendl2packet(unsigned *mp);`	Queues the packet buffer allocated by **l2ml_allocl2buffer()** method for transmission. This method does not check for NULL pointers.
l2ml_l2buffsize()	`extern system far unsigned l2ml_l2txbuffsize(void);`	Returns the largest possible size that the NPDU can be for transmission to a like-sized device (same network input buffer size). This is a better measurement than the size passed in from **l2ml_allocl2buffer()** method because it accounts for internal overhead and the receiver's capabilities

4

Designing the Serial I/O Hardware Interface

This chapter describes what you need to design the serial I/O hardware interface between your host processor and the Echelon Smart Transceiver or Neuron Chip for devices that use the Serial MIP.

Overview of the Hardware Interface

This chapter describes the hardware interface, including the requirement for pull-up resistors, selecting a minimum communications interface bit rate, considerations for host latency, specifying the SCI interface, and how to perform an initial health check of the Echelon Smart Transceiver.

Reliability

The LonTalk Stack link layer protocol assumes a reliable serial link and does not include error detection or error recovery. Instead, error detection and recovery are implemented by the LonTalk protocol, and this protocol detects and recovers from errors.

To minimize possible link-layer errors, be sure to design the hardware interface for reliable and robust operations. For example, use a star-ground configuration for your device layout on the device's printed circuit board (PCB), limit entry points for electrostatic discharge (ESD) current, provide ground guarding for switching power supply control loops, provide good decoupling for V_{DD} inputs, and maintain separation between digital circuitry and cabling for the network and power. See the *FT 3120 / FT 3150 Echelon Smart Transceiver Data Book*, the *PL 3120 / PL 3150 / PL 3170 Power Line Echelon Smart Transceiver Data Book*, or the *Series 5000 Chip Data Book* for more information about PCB design considerations for an Echelon Smart Transceiver.

The example applications contain example implementations of the link layer driver, including examples and recommendations for time-out guards within the various states of that driver.

Serial Communication Lines

For the SCI serial interfaces, you must add 10 kΩ pull-up resistors to the two communication lines between the host processor and the Echelon Smart Transceiver or Neuron Chip. These pull-up resistors prevent invalid transactions on start-up and reset of the host processor or the Echelon Smart Transceiver or Neuron Chip. Without a pull-up resistor, certain I/O pins can revert to a floating state during start-up, which can cause unpredictable results.

High-speed communication lines should also include proper back termination. Place a series resistor with a value equal to the characteristic impedance (Z_0) of the PCB trace minus the output impedance of the driving gate (the resistor value should be approximately 50 Ω) at the driving pin. In addition, the trace should run on the top layer of the PCB, over the inner ground plane, and should not have any vias to the other side of the PCB. Low-impedance routing and correct line termination is increasingly important with higher link layer bit rates, so carefully check the signal quality for both the Echelon Smart Transceiver or Neuron Chip and the host when you design and test new LonTalk Stack device hardware, or when you change the link-layer parameters for existing LonTalk Stack device hardware.

The RESET~ Pin

The Echelon Smart Transceiver and Neuron Chip have no special requirements for the **RESET~** (or **RST~**) pin. See the *FT 3120 / FT 3150 Echelon Smart Transceiver Data Book*, the *PL 3120 / PL 3150 / PL 3170 Power Line Echelon Smart Transceiver Data Book*, or the *Series 5000 Chip Data Book* for information about the requirements for this pin.

However, because a LonTalk Stack device uses two processor chips, the Echelon Smart Transceiver or Neuron Chip and the host processor, you have an additional consideration for the **RESET~** pin: Whether to connect the host processor's reset pin to the Echelon Smart Transceiver or Neuron Chip **RESET~** pin.

For most LonTalk Stack devices, you should not connect the two reset pins to each other. It is usually better for the Echelon Smart Transceiver or Neuron Chip and the host application to be able to reset independently. For example, when the Echelon Smart Transceiver or Neuron Chip encounters an error that causes a reset, it logs the reset cause (see *Querying the Error Log*); if the host processor resets the Echelon Smart Transceiver or Neuron Chip directly, possibly before the Echelon Smart Transceiver or Neuron Chip can detect and log the error, your application cannot query the Echelon Smart Transceiver or Neuron Chip error log after the reset to identify the problem that caused the reset. The Echelon Smart Transceiver or Neuron Chip also resets as part of the normal process of integrating the device within a network; there is normally no need for the host application to reset at the same time.

In addition, the host processor should not reset the Echelon Smart Transceiver or Neuron Chip while it is starting up (that is, before it sends the **LonResetNotification** uplink reset message to the host processor).

For devices that require the host application to be able to control all operating parameters of the Echelon Smart Transceiver or Neuron Chip, including reset, you can connect one of the host processor's general-purpose I/O (GPIO) output pins to the Echelon Smart Transceiver or Neuron Chip **RESET~** pin, and drive the GPIO pin to cause an Echelon Smart Transceiver or Neuron Chip reset from within your application or within your serial driver. Alternatively, you can connect one of the host processor's GPIO input pins to the Echelon Smart Transceiver or Neuron Chip **RESET~** pin so that the host application can be informed of Echelon Smart Transceiver or Neuron Chip resets.

A host processor's GPIO output pin should not actively drive the Echelon Smart Transceiver's **RESET~** pin high, but instead should drive the pin low. You can use one of the following methods to ensure that the GPIO pin cannot drive the **RESET~** pin high:

- Ensure that the GPIO pin is configured as an open-drain (open-collector) output
- Ensure that the GPIO pin is configured as a tri-state output
- Place a Schottky diode between the GPIO pin and the **RESET~** pin, with the cathode end of the diode connected to the GPIO pin

Configuring the GPIO pin as either open drain or tri-state ensures that the GPIO pin is in a high-impedance state until it is driven low. Using a Schottky diode is preferable to using a regular diode because a Schottky diode has a low forward voltage drop (typically, 0.15 to 0.45 V), whereas a regular diode has a much

higher voltage drop (typically, 0.7 V), that is, the Schottky diode ensures that the voltage drop is low enough to ensure a logic-low signal.

Host-driven reset of the Echelon Smart Transceiver or Neuron Chip should only be an emergency means to recover from some serious error. In addition, the host application or serial driver should always log the reason or cause for the reset, along with timestamp information. An unrecoverable error that requires a reset of the Echelon Smart Transceiver or Neuron Chip is generally evidence of a malfunction in the host driver, the Echelon Smart Transceiver or Neuron Chip, or the physical link layer, and should be investigated.

Selecting the Link-Layer Bit Rate

The serial link bit rate for the pre-built Layer 2 MIP images is fixed at 115, 200 bps. If you build a custom Layer 2 MIP image, you can specify a lower bit rate if required for your hardware. The minimum bit rate for the serial link between the Echelon Smart Transceiver or Neuron Chip and the host processor is most directly determined by the expected number of packets per second, the type of packets, and the size of the packets. Another factor that can significantly influence the required bit rate is support for explicit addressing, an optional feature that the LonTalk Stack application can enable and disable.

Recommendations: The following recommendations apply to general-use LONWORKS devices:

- Echelon Smart Transceiver or Neuron Chip external clock frequency
  - 10 MHz or higher for TP/FT-10 devices (for Series 5000 devices, specify a minimum 5 MHz system clock rate)
  - 5 MHz or higher for power-line devices
- Bit rate
  - 38 400 bps or higher for TP/FT-10 devices
  - 9600 bps or higher for power-line devices

To generate a more precise estimate for the minimum bit rate for the serial interface, use the following formula:

$$MinBitRate = \left(5 + P_{type} + EA + P_{size}\right) * BPT_{Interface} * PPS_{\exp}$$

where:

- The constant 5 represents general communications overhead
- P_{type} is the packet-type overhead, and has one of the following values:
  - 3 for network-variable messages
  - 1 for application messages
- EA is the explicit-addressing overhead, and has one of the following values:
  - 0 for no explicit-addressing support
  - 11 for explicit-addressing support enabled
- P_{size} is the packet size of the payload, and has one of the following values:

    - sizeof(network_variable)
    - sizeof(message_length)
- $BPT_{Interface}$ represents data transfer overhead for the serial interface, and has one of the following values:
    - 1 bit per transfer for SPI
    - 10 bits per transfer for SCI
- PPS_{exp} is the expected packet-per-second throughput value

Example: For an average network variable size of 3 bytes, no explicit messaging support, and a TP/FT-10 channel that delivers up to 180 packets per second, the minimum bit rate for an SCI interface is 19 200 bps. To allow for larger NVs, channel noise, and other systemic latency, you should consider setting the device bit rate at the next greater value above the minimum calculated from the formula. Thus, for this example, a bit rate of 38 400 or 76 800 bps is recommended.

To calculate the expected packet-per-second throughput value for a channel, you can use the Echelon Perf utility, available from *www.echelon.com/downloads*.

However, the bit rate is not the only factor that determines the link-layer transit time. Some portion of the link-layer transit time is spent negotiating handshake lines between the host and the Echelon Smart Transceiver. For faster bit rates, the handshaking overhead can increase, thus your application might require a faster clock speed for the Echelon Smart Transceiver to handle the extra processing.

Example: For a Series 3100 Echelon Smart Transceiver running at 10 MHz and an ARM7 host running at 20 MHz, the link-layer transit for a 4-byte network variable fetch, the handshaking overhead can be as much as 22% of the total link-layer transit time at 19 200 bps, and as much as 40% at 38 400 bps.

Even though a Series 3100 Echelon Smart Transceiver running at 5 MHz can be sufficient for the demands of a power-line channel, a typical Echelon Smart Transceiver operates at 10 MHz even when used exclusively with a power line channel. The maximum clock rate for an Echelon Smart Transceiver based on a PL 3120, PL 3150, or PL 3170 Echelon Smart Transceiver is 10 MHz.

For a performance test application that attempts to maximize the number of propagated packets, the application is likely to show approximately 3% increased throughput when operating with a 40 MHz Series 3100 Echelon Smart Transceiver compared to a 10 MHz Series 3100 Echelon Smart Transceiver (for FT 5000 Echelon Smart Transceivers, the comparison is between the 20 MHz system clock setting and the 5 MHz system clock setting). However, for a production application, which only occasionally transmits to the network and has unused output buffers available on the Echelon Smart Transceiver, a faster Echelon Smart Transceiver reduces the time required for the handshake overhead (by up to a factor of 4 for Series 3100 devices – or up to a factor of 16 for Series 5000 devices, compared to Series 3100 devices) so that a downlink packet can be delivered to the Echelon Smart Transceiver more quickly, which can improve overall application latency. Thus, depending on the needs of your application, you can use a slower or faster Echelon Smart Transceiver.

Host Latency Considerations

The processing time required by the host processor for an Echelon Smart Transceiver or Neuron Chip can have a significant impact on link-layer transit time for network communications and on the total duration of network transactions. This impact is the host latency for the LonTalk Stack application.

To maintain consistent network throughput, a host processor must complete each transaction as quickly as possible. Operations that take a long time to complete, such as flash memory writes, should be deferred whenever possible. For example, an ARM7 host processor running at 20 MHz can respond to a network-variable fetch request in less than 60 µs, but typically requires 10-12 ms to erase and write a sector in flash memory.

The following formula shows the overall impact of host latency on total transaction time:

$$t_{trans} = \left(2 * \left(t_{channel} + t_{MicroServer} + t_{linklayer}\right)\right) + t_{host}$$

where:

- t_{trans} is the total transaction time
- $t_{channel}$ is the channel propagation time
- $t_{MicroServer}$ is the Echelon Smart Transceiver or Neuron Chip latency (approximately 1 ms for a Series 3100 Echelon Smart Transceiver running at 10 MHz; approximately 65 µs for a FT 5000 Echelon Smart Transceiver running with an 80 MHz system clock)
- $t_{linklayer}$ is the link-layer transit time
- t_{host} is the host latency

The channel propagation time and the Echelon Smart Transceiver latency are fairly constant for each transaction. However, link-layer transit time and host latency can be variable, depending on the design of the host application.

You must ensure that the total transaction time for any transaction is much less than the LONWORKS network transmit timer. For example, the typical transmit timer for a TP/FT-10 channel is 64 ms, and the transmit timer for a PL-20 channel is 384 ms.

Typical host processors are fast enough to minimize link-layer transit time and host latency, and to ensure that the total transaction time is sufficiently low. Nonetheless, your application might benefit from using an asynchronous design of the host serial driver and from deferring time-consuming operations such as flash memory writes.

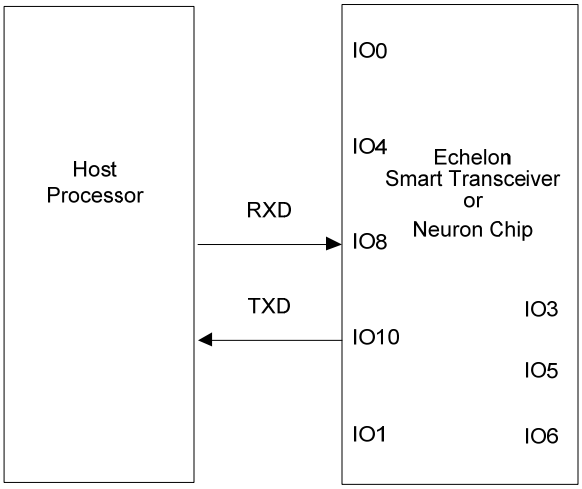
SCI Interface

The LonTalk Stack Serial Communications Interface (SCI) is an asynchronous serial interface between the Echelon Smart Transceiver or Neuron Chip and the host processor. The communications format is:

- 1 start bit

- 8 data bits (least-significant bit first)
- 1 stop bit

The SCI link-layer interface uses two serial data lines: RXD (receive data) and TXD (transmit data). The following diagram summarizes the two serial data lines and their I/O pin assignments. The signal directions are from the point of view of the Echelon Smart Transceiver. An *uplink* transaction describes data exchange from the Echelon Smart Transceiver to the host processor, and uses the TXD line. A *downlink* transaction refers to data exchange from host processor to the Echelon Smart Transceiver, and uses the RXD line.

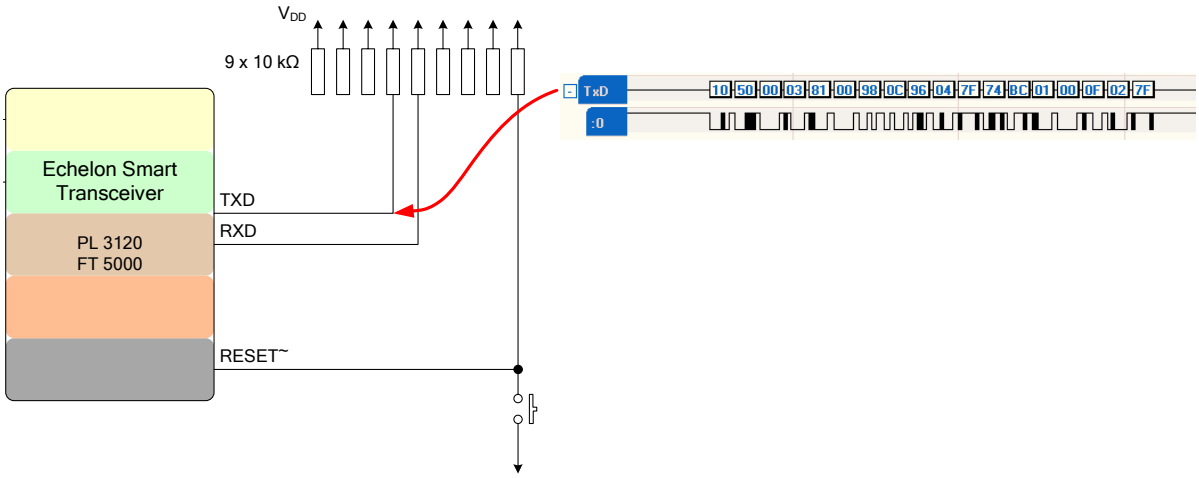


Performing an Initial Echelon Smart Transceiver Health Check

After you load the Layer 2 MIP image into an Echelon Smart Transceiver or Neuron Chip, the Echelon Smart Transceiver or Neuron Chip enters quiet mode (also known as flush mode). While the Echelon Smart Transceiver or Neuron Chip is in quiet mode, all network communication is paused.

The Echelon Smart Transceiver or Neuron Chip enters quiet mode to ensure that only complete implementations of the LonTalk protocol stack attach to a LONWORKS network. In a functioning LonTalk Stack device, the application initializes the Echelon Smart Transceiver or Neuron Chip. After that initialization is complete, the Echelon Smart Transceiver or Neuron Chip leaves quiet mode and enables regular network communication.

To check that the Echelon Smart Transceiver or Neuron Chip is functioning correctly before the host processor has initialized it, you can use an oscilloscope or a logic analyzer to observe the activity on the TXD (IO10) pin that reflects the uplink **LonNiReset** message transfer that follows an Echelon Smart Transceiver or Neuron Chip reset, as shown in the following figure.



Your hardware design should include a switch that connects the **RESET~** pin to ground; you press this switch to reset the Echelon Smart Transceiver or Neuron Chip.

When you press the reset switch for a LonTalk Stack device, the Neuron firmware performs reset processing, as described in the data books for the Echelon Smart Transceiver and Neuron Chips. Then, the Echelon Smart Transceiver or Neuron Chip performs reset processing that is generally independent of the host processor.

5

Creating a LonTalk Stack Serial MIP Driver

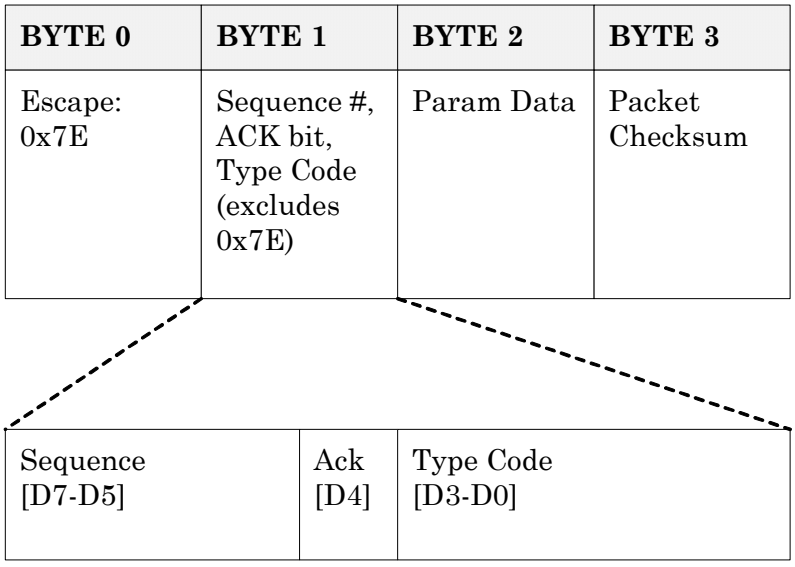
This chapter describes the link-layer protocol (LLP) and how to develop a LonTalk Stack Serial MIP driver for your host processor. This driver manages the handshaking and data transfers between the host and the Echelon Smart Transceiver or Neuron Chip. The driver also manages the buffers in the host for communication with the Echelon Smart Transceiver or Neuron Chip.

Overview of the Link Layer Protocol

The LonTalk Serial MIP driver communicates with the host processor over the built-in SCI serial interface hardware of the FT 5000 Echelon Smart Transceiver, Series 5000 chip, and the PL 3120 Echelon Smart Transceiver. The Serial MIP driver uses the Serial MIP link-layer protocol (LLP), which is a two-signal serial protocol with no extra requirements for handshake signals. The Serial MIP LLP features quick recovery from serial communication errors such as dropped bytes or corrupted serial frames.

Code Packet Layout

The basic component of the Serial MIP LLP is the code packet, which starts with an escape code. If an escape code appears in a normal data stream it is followed by another escape code and interpreted as a single data byte value rather than the start of the code packet. The following figure illustrates the code packet layout.



The second byte contains a 3-bit sequence number in the sequence bits (D5-D7), a single ACK bit (D4), and a 4-bit Type Code bits (D0-D3). You cannot form the escape code by combining a Type Code value with the sequence number (0x0E is not allowed). As a result, you can create 15 different codes.

The sequence number is cycled through between values '1' and '7', with '0' being an initialization value. When a code packet is received that has the same sequence number as the previous code packet received that packet (and any following data) is rejected (this transfer will be acknowledged if necessary). The exception is if the sequence number is zero. The zero sequence number may be used for any idempotent code packet or message.

The third and fourth bytes may contain the escape code value, but they will not interpreted as escape codes. This may cause minor re-synchronization issues if this packet is broken; however, it ensures a constant code packet size.

The packet sum is an 8-bit value that, when added to the first three bytes, results in an 8-bit zero result.

The code packet hast the following features:

- Asynchronous method of presenting itself at any time (by using an escape sequence).
- Checksum verification.
- Data transfer initiation.
- Duplication detection using a sequence number. Duplicates may be sent when an expected response is lost or broken and does not occur within a time-out period.

Data outside of the code packet is restricted to either a *message* or a *multi-byte local network interface command or response* and is always enclosed with a length byte in the front and a checksum at the end. The length byte does not account for the checksum at the end—the inclusion of the checksum is implied. The checksum covers the data that preceded it; therefore, it does include the length byte.

Note: All data outside a code packet must include an escape prefix before any escape data values. For example, if a 0x7E value appears in the sequence it must be followed by another 0x7E (the first 0x7E value is the escape prefix). Checksum bytes and length bytes must be preceded with escape prefixes.

The following table summarizes the layout of the data message used in Layer 5 mode and for the **niNETMGMT** local network interface command (Layer 2 mode is required for the LonTalk Stack). When sending local network interface commands (NI Commands other than the **niCOMM** or **niNETMGMT**) that have additional data, the commands are contained in a data message and they are preceded by a **CpMsg** code packet.

BYTE 0	**BYTE 1**	**BYTE 2..N**	**BYTE N+1**
Length (bytes to follow except checksum)	NI Command	SICB starting w/ message header	Message Checksum of bytes 0-N

The following table summarizes the layout of the data message used in Layer 2 mode.

BYTE 0	**BYTE 1**	**BYTE 2..N**	**BYTE N+1**
Length (bytes to follow except checksum)	NI Command	Priority/AltPath/DeltaBacklog byte, NPDU, CRC (2 Bytes)	Message Checksum of bytes 0-N

Type Code Values

The following table lists the values for the Type Code byte. Uplink means that data is transferred from the Echelon Smart Transceiver or Neuron Chip to the host processor. Downlink mans that data is transferred from the host processor to the Echelon Smart Transceiver or Neuron Chip.

Value	Type	Description	Uplink / Downlink
0	**CpNull**	No data is being sent. Use this value for acknowledge-only packets, or for pinging.	U/D
1	**CpFail**	The previous transfer was broken or in error. This is typically due to checksum errors or fragmented transfers. The correct response to this command is to resend the previous transfer.	U/D
2	**CpMsg**	One message follows. A message is any multi-byte network interface transfer and requires an application output buffer or packet buffer in the downlink case. The number of messages is stored in the **Param Data** field. This value is limited to one in both the downlink and uplink case. The message follows the code packet checksum byte, and consists of a length byte, the NI command, the message itself, and a checksum.	U/D
3	**CpMsgReq**	Optional. Sent by the host for requesting both the attention of the Serial MIP and requesting an uplink **CpMsgAck**. The **Param Data** field contains either a **0** or a **1** informing the Serial MIP that the following message is either a non-priority message (0) or a priority message (1). The MIP Serial driver does not use priority messaging; therefore, this value is ignored. This allows the Serial MIP to respond to the **CpMsgReq** based on actual buffer availability. Alternatively, you can send the entire **CpMsg** plus message to the MIP and a **CpMsgReject** may be sent uplink if there are no available buffers.	D

Value	Type	Description	Uplink / Downlink
4	**CpMsgAck**	The Serial MIP is entering the ready-receive state. This is the Serial MIP's response to the **CpMsgReq**.	U
5	**CpMsgReject**	An attempt to transfer a message downlink is rejected because of a lack of buffer space. This code packet will be a response to a downlink **CpMsgReq** code packet or a response to a **CpMsg** without the **CpMsgReq** being sent. This code indicates that the offered downlink traffic is more than the Serial MIP can handle (it has no more APP or NET output buffers). Upon receiving this code, the device driver should repeat the message send process until it succeeds.	U
6	**CpNiCmdShort**	Sends a single byte local network interface command. The command is stored in the **Param Data** field.	U/D
7	**CpResync**	This command is sent by the host to start a new session with the Serial MIP. When received it will reset the transmit and receive sequence numbers to zero so that any subsequent sequence numbers will be accepted rather than rejected. This command is also a good way to establish that communications are functioning with the Serial MIP. The Serial MIP always responds to this command with a **CpNull** packet, so the host can determine that the serial link is not simply echoing data.	D

Notes:

- Use symmetrical timeouts on the host processor. There is an inter-byte timeout on the client side of 40ms when receiving either a code packet or a message body. There is a timeout of 250ms when waiting for the start of a downlink message **CpMsg** following a **CpMsgReq / CpMsgAck** sequence.

- All length fields do not count for escape prefixes. Instead, they reflect the length of the real data stream above the link-layer protocol. All length fields do not account for the checksum.
- Broken code packets (code packets with missing or corrupt bytes) are not responded to at all, and they rely on time-out mechanisms for re-transmission.
- Broken message streams are responded to as soon as the error condition is observed. The response is the **CpFail** code packet.

Acknowledgment Rules

The **Ack** bit is used to send a success response to a previous transfer, specifically transfers that send data that is above the LLP (non-LLP data). These are transfers of messages and transfers of local network interface commands.

To reduce traffic, not all data transfers require acknowledgement. Following are the acknowledgement rules:

- If a transfer requires an acknowledgment and there is no other data that needs to be sent, a **CpNull** packet is sent with the **Pass/Ack** bit set. Otherwise, if there is outgoing traffic to be sent the **Ack** bit will be included in the following transfer. This is true for both uplink and downlink data messages.
- Code packets that do not require an acknowledgment via the **Ack** bit are packets that result in an immediate response anyway. The response implies acknowledgment. The following table lists these types of code packets:

Code Packet	Description
CpMsgReq	The Serial MIP will respond with either the **CpMsgAck**, or a criss-cross could occur and an unrelated code packet will be sent by the Serial MIP.
CpMsgAck	The host does not need to acknowledge this message. Instead it provides an implied acknowledgement by sending the **CpMsg** code packet followed by a message packet

- Data that is sent and requires acknowledgement will persist in the source until the acknowledgment is received. While the persistent data is waiting for acknowledgement, the **Acknowledge Wait** timeout, which is 300ms for the Serial MIP and the driver, causes the persistent data to be re-sent.

Sequence Number Cycling and Duplicate Detection

The sequence number is used to reject duplicate non-LLP data. Duplicate LLP data does not effect the LLP; therefore, the sequence number is only cycled for each transfer of non-LLP data. For example, two consecutive **CpMsgReq** packets have no effect—the second **CpMsgReq** packet reinstates the ready-receive state and the **CpMsgAck** is re-sent.

Supported MIP Command Set

The following table lists the MIP commands you can use in your LonTalk Stack Serial MIP driver.

Uplink local network interface commands that must also convey additional data (for example, the **niL2MODE** response command) will always result in a data message that is at least 3 bytes in length. This is because the host driver expects at least 4 bytes of data (3 bytes plus checksum) to appear following a code packet.

Uplink means that data is transferred from the Echelon Smart Transceiver to the host processor. Downlink mans that data is transferred from the host processor to the Echelon Smart Transceiver

Value	Name	Description	Uplink / Downlink
0x1x	**niCOMM**	Passes completion events to the host processor.	
0x2x	**niNETMGMT**	Performs local network management or network diagnostic commands on the network interface.	
0x31	**niERROR|niCRC**	The Layer 2 mode MIP will convey this command uplink whenever it senses an error – in this implementation it is limited to receive CRC errors (0x31).	U
0x50	**niRESET**	When sent downlink this will reset the MIP. Following any reset the MIP will send this uplink.	U/D
0x60	**niFLUSH_CANCEL**	Exits the flush state.	D
0x70	**niONLINE**	Sets the MIP state to soft-online.	D
0x80	**niOFFLINE**	Sets the MIP state to soft-offline. No messaging can be handled in this state.	D

Value	Name	Description	Uplink / Downlink
0x90	**niFLUSH**	Sets the MIP to the "flush" state.	D
0xA0	**niFLUSH_IGNORE**	Sets the MIP to the "flush ignore comm" state.	D
0xCx	**niIO_SET**	Directly controls the MIP's four I/O pins, IO0 – IO3, for general purpose I/O control from the Neuron. The L.S. 4 bits are used to control these pins.	D
0xD0	**niMODE_L5**	Sets the MIP to Layer 5 mode. If already in this mode the MIP will reply with this command. Otherwise the MIP will reset and resume in the new mode. This change is persistent across resets. Layer 5 mode is not compatible with the LonTalk host stack.	U/D
0xD1	**niMODE_L2**	Sets the MIP to Layer 2 mode. If already in this mode the MIP will reply with this command. Otherwise the MIP will reset and resume in the new mode. This change is persistent across resets.	U/D
0xE0	**niSSTATUS**	Provides status information. When sent downlink, the MIP responds with the **niSSTATUS** command followed by the following 4 bytes of data: [TXID Value], [MIP Version], [Layer 2:1 or Layer 5:0 Mode], [Serial checksum error count]	U/D
0xE6	**niSERVICE**	In Layer 5 mode, sends a Service Pin message.	D

Layer 2 / Layer 5 Modes

The default mode for the Serial MIP is L2 mode. The L2/L5 mode is maintained in non-volatile memory.

Product Query Network Management

The Serial MIP supports the **Product Query** network management command from the host only. The code for this command is the Network Management Escape code [0x7D] followed by the sub-command and command values of [0x01], [0x01]. The response includes **PRODUCT** and **MODEL** values based on whether the MIP is currently in L2 or L5 mode. The App Version is 3.1 (31); the TXID is defined when the Serial MIP image is built, and is **4** for a TP/FT-10 channel.

Serial MIP Driver Example

The LonTalk Stack Developer's Kit includes a Linux Serial MIP driver example in the **LonTalkStack/Source/Target/Drivers/Linux/SMIP** folder that demonstrates how to create a LonTalk Stack Serial MIP driver. You can use this example for your LonTalk Stack device, or you can customize it to meet your specifications.

Serial MIP Driver API

The following sections describe the structures and functions in the Serial MIP driver API.

Structures

Structure	Description

`#define MAXLONMSG 114`
`typedef struct LDV_Message {`
`  BYTE NiCmd;`
`  BYTE Length;`
`  BYTE ExpAppMessage[MAXLONMSG];`
`} LDV_Message;`
	Standard structure for handling messages. This structure is used for passing network interface commands, SICBs, and L2 packet buffers.
	The **NiCmd** byte is the network interface command.
	The **Length** byte is the size of **ExpAppMessage**.

`typedef struct LLPStats {`
`  DWORD AckTMOs;`
`  DWORD RxTmos;`
`  DWORD CsumErrors;`
`  DWORD CpFails;`
`  DWORD ULDuplicates;`
`  DWORD UlDiscarded;}`
`LLPStats;`
	Structure for handling LLP statistics. The following describes each statistic:
	AckTMOs. Number of Acknowledged timeouts.
	RxTmos. Number of receive timeouts.
	CsumErrors. Number of uplink checksum errors.
	CpFails. Number of uplink **CpFail** messages received (implies downlink cs error).
	ULDuplicates Number of duplicates

Structure	Description
	sensed. **UlDiscarded**. Number of tossed uplinks.

Functions

Function	Syntax	Description

SciMipOpen	`LdvRetVal SciMipOpen(`
	`  WORD iComPort,`
	`  DWORD baudrate,`
	`  HANDLE hNotifier);`
		Opens the serial interface driver.
		iComPort. The index to the serial port.
		baudrate. The serial port baud rate,
		hNotifier. A handle to an event that will be set by the driver whenever received messages are available. The driver never closes this handle.
		If the driver is already open, then the **SciMipClose()** function is called first.
		This function returns **LDV_OK** or **LDV_DEVICE_ERR** (if there was a failure with the serial port or thread creation).

SciMipClose	`LdvRetVal SciMipClose(void);`	Closes the serial interface driver by closing the serial port and deleting any threads created during **SciMipOpen()**. Returns **LDV_OK** or **LDV_NOT_OPEN** (if the driver was not open).

Function	Syntax	Description
SciMipRead	`LdvRetVal SciMipRead ( LDV_Message *pMsg, int size);`	De-queues one uplink message or local network interface command if there is one available. **size**. Indicates the size of the structure of **pMsg**. This function returns one of the following vlaues: **LDV_OK**. Successful. **LDV_NO_MSG_AVAIL**. No messages are available. **LDV_NOT_OPEN**. The driver is not open **LDV_INVALID_BUF_LEN** The value of **size** is too small for the message.
SciMipWrite	`LdvRetVal SciMipWrite( LDV_Message *pMsg);`	Queues a message or local network interface command to the driver. Returns one of the following values: **LDV_NOT_OPEN**. The driver is not open. **LDV_INVALID_BUF_LEN** The length embedded in the structure at **pMsg** is too large for the driver buffers. **LDV_NO_BUFF_AVAIL**. The driver buffers are full. **LDV_OK** if successful. Messages that are queued have not necessarily been delivered to the MIP yet. The queue depth is currently set to 4.

Function	Syntax	Description
SciMipStatistics	`LdvRetVal SciMipStatistics( LLPStats *pLlps, int size, BOOL bIfClear);`	Returns a structure containing device driver statistics. Set the **size** argument to *sizeof(LLPStats)*. To clear the internal statistics, set **bIfClear** to TRUE.
SciMipSetMode	`LdvRetVal SciMipSetMode( LLPMode mode);`	This Serial MIP driver supports a mode where downlink messages are transmitted without the normal **CpMsgReq** request. This mode works under controlled downlink traffic conditions. To enable this mode, set the **mode** argument to **S10LLP_MODE_ NOREQUESTS**.
SciMipSetKeys	`LdvRetVal SciMipSetKeys( SCMKeys *pKeys);`	Sets the two keys used for MIP/Host authentication. If the driver is currently open, the driver will initiate the authentication process when this API is called. The actual keys used by the MIP are not embedded in the driver; they must be supplied by the client software.

6

Creating a Model File

You use a model file to define your device's interoperable interface, including its network inputs and outputs. The LonTalk Interface Developer utility converts the information in the model file into device interface data and a device interface file for your application. This chapter describes how to create a model file using the Neuron C programming language.

Syntax for the Neuron C statements in the model file is described in Appendix C, *Neuron C Syntax for the Model File*.

Model File Overview

The interoperable application interface of a LONWORKS device consists of its functional blocks, network variables, configuration properties, and their relationships. The *network variables* are the device's means of sending and receiving data using interoperable data types. The *configuration properties* are the device's means of providing externally exposed configuration data, again using interoperable data types. The configuration data items can be read (and typically also written) by a network tool. The device interface is organized into *functional blocks*, each of which groups together a collection of network variables and configuration properties that are used to perform one task. These network variables and configuration properties are called the *functional block members*.

The model file describes the functional blocks, network variables, configuration properties, and their relationships, that make up the interoperable interface for a LonTalk Stack device, using the Neuron C programming language. Neuron C is based on ANSI C, and is designed for creating a device's interoperable interface and implementing its algorithms to run on Neuron Chips and Echelon Smart Transceivers. However, you do not need to be proficient in Neuron C to create a model file for a LonTalk Stack application because the model file does not include executable code. All tools required to process model files are included with the LonTalk Stack; you do not need to license another Neuron C development tool to work with a LonTalk Stack model file. The model file uses Neuron C Version 2.1 declaration syntax.

The LonTalk Interface Developer utility included with the LonTalk Stack Developer's Kit uses the model file to generate device interface data and device interface files. You can use any of the following methods to create a model file:

- Manually create a model file
  A model file is a text file that you can create with any text or programming editor, including Windows Notepad. Model files have the **.nc** file extension. This chapter describes the types of Neuron C statements you can include in a model file. Appendix C describes the syntax for the Neuron C statements.
- Reuse existing Neuron C code
  You can reuse an existing Neuron C application that was originally written for a Neuron Chip or an Echelon Smart Transceiver as a model file. The LonTalk Interface Developer utility uses only the device interface declarations from a Neuron C application program, and ignores all other code. You might have to delete some code from an existing Neuron C application program, or exclude this code using conditional compilation, as described later in this chapter.
- Automatically generate a model file
  You can use the NodeBuilder Code Wizard, included with the NodeBuilder FX Development Tool, to automatically generate a model file. Using the NodeBuilder Code Wizard, you can define your device interface by dragging functional profiles and type definitions from a graphical view of your resource catalog to a graphical view of your device interface, and refine them using a convenient graphical user interface. When you complete the device interface definition, click the **Generate Code and Exit** button to automatically generate your model file. Use the main file produced by the NodeBuilder Code Wizard as your model

file. NodeBuilder software is not included with the LonTalk Stack, and must be licensed separately. See the *NodeBuilder FX User's Guide* for details about using the NodeBuilder Code Wizard.

See Appendix C, *Neuron C Syntax for the Model File*, for the detailed Neuron C syntax for each type of statement that can be included in the model file.

Defining the Device Interface

You use a model file to define the device interface for your device. The device interface for a LONWORKS device consists of its:

- Functional blocks
- Network variables
- Configuration properties

A *functional block* is a collection of network variables and configuration properties, which are used together to perform one task. These network variables and configuration properties are called the *functional block members*.

Functional blocks are defined by *functional profiles*. A functional profile is used to describe common units of functional behavior. Each functional profile defines mandatory and optional network variables and configuration properties. Each functional block implements an instance of a functional profile. A functional block must implement all of the mandatory network variables and configuration properties defined by the functional profile, and can also implement any of the optional network variables and configuration properties defined by the functional profile. In addition, a functional block can implement network variables and configuration properties that are not defined by the functional profile – these are called *implementation-specific* network variables and configuration properties.

The primary inputs and outputs to a functional block are provided by network variables. A *network variable* is a data item that a device application expects to get from other devices on a network (an *input network variable*) or expects to make available to other devices on a network (an *output network variable*). Network variables are used for operational data such as temperatures, pressures, switch states, or actuator positions.

A *configuration property* is a data item that specifies the configurations for a device (its network variables and functional blocks). Configuration properties are used for configuration data such as set points, alarm thresholds, or calibration factors. Configuration properties can be set by a network management tool (such as OpenLNS Commissioning Tool or a customized plug-in created for the device), and allow a network integrator to customize a device's behavior.

These interface components, and the resource files used to define them, are described in the following sections.

Defining the Interface for a LonTalk Stack Application

Within the model file, you define a simple input network variable with the following syntax:

network input *type name***;**

Example: The following declaration defines an input network variable of type "SNVT_type" with the name "nviAmpere".

```
network input SNVT_amp nviAmpere;
```

You define a simple output network variable using the same syntax, but with the **output** modifier:

network output *type name***;**

Example: The following declaration defines an output network variable of type "SNVT_type" with the name "nvoAmpere".

```
network output SNVT_amp nvoAmpere;
```

By convention, input network variable names have an *nvi* prefix and output network variables have an *nvo* prefix.

See *Network Variable Syntax* for the full network variable declaration syntax.

The LonTalk Interface Developer utility reads the network variable declarations in the model file to generate device-specific code. For the example of the *nviAmpere* and *nvoAmpere* pair of network variables above, the utility generates a standard ANSI C type definition for the *SNVT_amp* network variable type and implements two global C-language variables:

```
typedef ncsLong  SNVT_amp;
…
volatile SNVT_amp nviAmpere;
SNVT_amp nvoAmpere;
```

The **ncsLong** data type defines the host equivalent of a Neuron C signed long variable. This type is defined in the **LonPlatform.h** file.

Your LonTalk Stack application can simply read the *nviAmpere* global C variable to retrieve the most recently received value from that input network variable. Likewise, your application can write the result of a calculation to the *nvoAmpere* global C variable, and call the appropriate LonTalk API function to propagate the network variable to the LONWORKS network.

Choosing the Data Type

Many functional profiles define the exact type of each member network variable. The *SNVT_amp* type used in the previous section is such a type. Using a different network variable type within a functional profile that requires this network variable type renders the implementation of the profile not valid.

Other profiles are generic profiles that allow various network variable types to implement a member. The *SFPTopenLoopSensor* functional block (described in *Defining a Functional Block*) is an example for such a generic functional profile. This profile defines the *nvoValue* member to be of type *SNVT_xxx*, which means "any standard network variable type."

Implementing a generic profile allows you to choose the standard network variable type from a range of allowed types when you create the model file.

For added flexibility, if the specific functional profile allows it, your application can implement changeable-type network variables. A *changeable-type network variable* is network variable that is initially declared with a distinct default type (for example, *SNVT_volt*), but can be changed during device installation to a different type (for example, *SNVT_volt_mil*).

Using changeable-type network variables allows you to design a generic device (such as a generic proportional-integral-derivative (PID) controller) that supports a wide range of numeric network variable types for set-point, control, and process-value network variables.

See *Defining a Changeable-Type Network Variable* or more information about implementing changeable-type network variables for LonTalk Stack applications.

You can also define your own nonstandard data types. The NodeBuilder Resource Editor utility, which is included with the LonTalk Stack Developer's Kit, allows you to define your own, nonstandard data types for network variables or configuration properties, and allows definition of your own, nonstandard functional profiles. These nonstandard types are called user-defined types and user-defined profiles.

Defining a Functional Block

The first step for defining a device interface is to select the functional profile, or profiles, that you want your device to implement. You can use the NodeBuilder Resource Editor included with the LonTalk Stack Developer's Kit to look through the standard functional profiles, as described in *Defining a Resource File*. You can find detailed documentation for each of the standard functional profiles at *types.lonmark.org*.

For example, if your device is a simple sensor or actuator, you can use one of the following standard profiles:

- Open-loop sensor (**SFPTopenLoopSensor**)
- Closed-loop sensor (**SFPTclosedLoopSensor**)
- Open-loop actuator (**SFPTopenLoopActuator**)
- Closed-loop actuator (**SFPTclosedLoopActuator**).

If your device is more complex, look through the other functional profiles to see if any suitable standard profiles have been defined. If you cannot find an existing profile that meets your needs, you can define a user functional profile, as described in *Defining a Resource File*.

Example: The following example shows a simple functional block declaration.

```
network output SNVT_amp nvoAmpere;

fblock SFPTopenLoopSensor {
   nvoAmpere implements nvoValue;
} fbAmpMeter;
```

This functional block:

- Is named *fbAmpMeter* (network management tools use this name unless you include the **external_name** keyword to define a more human-readable name)
- Implements the standard profile **SFPTopenLoopSensor**
- Includes a single network variable, named *nvoAmpere*, which implements the **nvoValue** network variable member of the standard profile

Declaring a Functional Block

A functional block declaration, by itself, does not cause the LonTalk Interface Developer utility to generate any executable code, although it does create data that implements various aspects of the functional block. Principally, the functional block creates associations among network variables and configuration properties. The LonTalk Interface Developer utility uses these associations to create the self-documentation (SD) and self-identification (SI) data in the device and in its associated device interface file (.xif or .xfb extension).

The functional block information in the device interface file, or the SD and SI data, communicates the presence and names of the functional blocks contained in the device to a network management tool.

Network-variable or configuration members of a functional block also have self-documentation data, which is also automatically generated by the LonTalk Interface Developer utility. This self-documentation data provides details about the particular network variable or configuration property, including whether the network variable or configuration property is a member of a functional block.

Functional blocks can be implemented as single blocks or as arrays of functional blocks. In a functional block array, each member of the array implements the same functional profile, but has different network variables and typically has different configuration properties that implement its network variable and configuration property members.

Example: The following example shows a simple array of 10 functional blocks.

```
network output SNVT_amp nvoAmpere[10];

fblock SFPTopenLoopSensor {
   nvoAmpere[0] implements nvoValue;
} fbAmpMeter[10];
```

This functional block array:

- Contains ten functional blocks, *fbAmpMeter[0]* to *fbAmpMeter[9]*, each implementing the **SFPTopenLoopSensor** profile.
- Distributes the ten nvoAmpere network variables among the ten functional blocks, starting with the first network variable (at network variable array index zero). Each member of the network variable array applies to a different network variable member of the functional block array.

Defining a Network Variable

Every network variable has a type, called a *network variable type*, that defines the units, scaling, and structure of the data contained within the network variable. To connect a network variable to another network variable, both must have the same type. This type matching prevents common installation errors from occurring, such as connecting a pressure output to a temperature input.

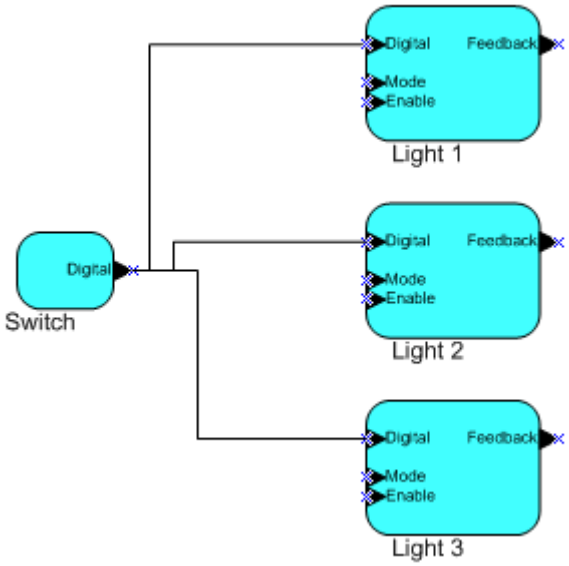
Type translators are also available to convert network variables of one type to another type. Some type translators can perform sophisticated transformations between dissimilar network variable types. Type translators are special functional blocks that require additional resources, for example, a dedicated type-translating device in your network.

You can minimize the need for type translators by using standard network variable types (SNVTs) for commonly used types, and by using changeable-type network variables, where appropriate. You can also define your own user network variable types (UNVTs).

You can use the NodeBuilder Resource Editor to look through the standard network variable types, as described in *Defining a Resource File*, or you can browse the standard profiles online at types.lonmark.org.

You can connect network variables on different devices that are of identical type, but opposite direction, to allow the devices to share information. For example, an application on a lighting device could have an input network variable of the switch type, while an application on a dimmer-switch device could have an output network variable of the same type. You can use a network tool, such as OpenLNS Commissioning Tool, to connect these two devices, allowing the switch to control the lighting device, as shown in the following figure.

A single network variable can be connected to multiple network variables of the same type but opposite direction. The following figure shows the same switch being used to control three lights.



The LonTalk Stack application in a device does not need to know anything about where input network variables come from or where output network variables go. After the LonTalk Stack application updates a value for an output network variable, it uses a simple API function call to have the LonTalk host stack propagate it.

Through a process called *binding* that takes place during network design and installation, the LonTalk Stack is configured to know the logical address of the

other devices (or groups of devices) in the network that expect a specific network variable, and the LonTalk Stack assembles and sends the appropriate packets to these devices. Similarly, when the LonTalk Stack receives an updated value for an input network variable required by its application program, it reads the data from the network and passes the data to the application program.

The binding process creates logical connections between an output network variable in one device and an input network variable in another device or group of devices. You can think of these connections as "virtual wires." For example, the dimmer-switch device in the dimmer-switch-light example above could be replaced with an occupancy sensor, without requiring any changes to the lighting device.

Network variable processing is transparent, and typical networked applications do not need to know whether a local network variable is bound ("connected") to one or more network variables on the same device, to one or more other devices, or not bound at all. For those applications that do require such knowledge, API functions (such as **LonQueryNvConfig()**, **LonQueryAliasConfig()**, **LonNvIsBound()**, and **LonMtIsBound()**) are supplied to query the related information.

Defining a Changeable-Type Network Variable

A *changeable-type network variable* is a network variable that supports installation-time changes to its type and its size.

You can use a changeable-type network variable to implement a generic functional block that works with different types of inputs and outputs. Typically, an integrator uses a network management tool plug-in that you create to change network variable types.

For example, you can create a general-purpose device that can be used with a variety of sensors or actuators, and then create a functional block that allows the integrator to select the network variable type depending on the physical sensor or actuator that is attached to the device during installation.

Restrictions:

- Each changeable-type network variable must be declared with an initial type in the model file. This initial type defines the default type and the maximum size of the network variable.
- A changeable-type network variable must be a member of a functional block.
- Only network variables that are not bound can change their type. To change the type of a bound network variable, you must first unbind (disconnect) the network variable.
- Only a network management tool, such as OpenLNS Commission Tool, can change the type of a changeable-type network variable. The LonTalk Stack device does not initiate type changes.

To create a changeable-type network variable for a LonTalk Stack application, perform the following tasks:

1. Declare the network variable with the **changeable_type** keyword. You must declare an initial type for the network variable, and the size of the initial type must be equal to the largest network variable size that your application supports. The initial type must be one of the interoperable standard or user network variable types.
2. Select **Has Changeable Interface** in the LONMARK Standard Program ID Calculator (included with the LonTalk Interface Developer utility) to set the changeable-interface bit in the program ID when you create the device template.
3. Declare a **SCPTnvType** configuration property that applies to the changeable-type network variable. This configuration property is used by network management tools to notify your application of changes to the network variable type.
4. You can optionally also declare a **SCPTmaxNVLength** configuration property that applies to the changeable-type network variable. This configuration property informs network management tools of the maximum type length supported by the changeable-type network variable. This value is a constant, so declare this configuration property with the **const** modifier.
5. Implement code in your LonTalk Stack application to process changes to the **SCPTnvType** value. This code can accept or reject a type change. Ensure that your application can process all possible types that the changeable-type network variable might use at runtime.
6. Implement code to provide information about the current length of the network variable.

The OpenLNS CT Browser provides integrators with a user interface to change network variable types. However, you can provide a custom interface for integrators to change network variable types on your device. For example, the custom interface could restrict the available types to those types supported by your application, thus preventing configuration errors.

See *Handling Changes to Changeable-Type Network Variables* for information about how your application should handle changes to changeable-type network variables.

Defining a Configuration Property

Like network variables, configuration properties have types, called *configuration property types*, that determine the units, scaling, and structure of the data that they contain. Unlike network variable types, configuration property types also specify the meaning of the data. For example, standard network variable types represent temperature values, whereas configuration property types represent specific types of temperature settings, such as the air temperature weighting used during daytime control, or the weighting of an air temperature sensor when calculating an air temperature alarm.

Declaring a Configuration Property

You declare a configuration property in a model file. Similar to network variable types, there are standard and user-defined configuration property types. You can

use the NodeBuilder Resource Editor to look through the standard configuration property types, as described in *Defining a Resource File*, or you can browse the standard profiles online at types.lonmark.org. You can also define your own configuration property type, if needed.

You can implement a configuration property using either of the following techniques:

- A configuration property network variable
- A configuration file

A *configuration network variable* (also known as a configuration property network variable or configuration NV) uses a network variable to implement the configuration property. In this case, a LONWORKS device can modify the configuration property, just like any other network variable. A configuration NV can also provide your application with detailed notification of updates to the configuration property. However, a configuration NV is limited to a maximum of 31 bytes, and a LonTalk Stack application is limited to a maximum of 4096 network variables, including configuration NV s. Use the **network … config_prop** syntax described in to implement a configuration property as a configuration network variable. By convention, configuration NV names start with an *nci* prefix, and configuration properties in files start with a *cp* prefix.

A *configuration file* implements the configuration properties for a device as one or two blocks of data called value files, rather than as separate externally exposed data items. A value file consists of configuration property records of varying length concatenated together. Each value file must fit as contiguous bytes into the memory space in the device. When there are two value files, one contains writeable configuration properties, and the second contains read-only data. To allow a network management tool to access the data items in the value file, you specify a provided template file, which is an array of text characters that describes the elements in the value files. When you use the Direct Memory Files feature, the total size of the directory, template file, and value files cannot exceed 65 535 bytes (64 KB -1). When you use FTP, individual files cannot exceed 2 147 483 647 bytes (2 GB -1, or 2^{31} -1 bytes).

Other devices cannot connect to or poll a configuration property implemented in a configuration file. To modify a configuration property implemented in a configuration file, a network management tool must modify the configuration file, for which your application must provide an appropriate access method.

You must implement configuration properties within a configuration file if any of the following apply to your application:

- The total number of network variables (including configuration network variables and dynamic network variables) exceeds the total number of available network variables (a maximum of 4096 for a LonTalk Stack device, but potentially fewer than 4096 depending on the resources available).
- The size of a single configuration property exceeds the maximum size of a configuration network variable (31 bytes).
- Your device cannot use a configuration network variable (CPNV). For example, for a device that uses a configuration property array that applies to several network variables or functional blocks with one instance of the configuration property array each, the configuration

property array must be shared among all network variables or functional blocks to which it applies. In this case, the device must implement the configuration properties within a configuration file.

In addition, you might decide whether to implement configuration properties within a configuration file for performance reasons. Using the direct memory files (DMF) feature can be faster than using configuration network variables if you have more than a few configuration properties because multiple configuration properties can be updated during a single write to memory (especially during device commissioning). However, FTP can be faster than DMF if there are many configuration properties to be updated.

Use the **cp_family** syntax described in *The Configuration Property Type* to implement a configuration property as a part of a configuration file.

When implementing configuration property files, the LonTalk Interface Developer utility combines all configuration properties declared using the **cp_family** keyword, and creates the value files and a number of related data structures.

However, you must provide one of two supported mechanisms to access these files:

- An implementation of the LONWORKS file transfer protocol
- Support for the direct memory files feature

The LonTalk Interface Developer utility provides most of the required code to support direct memory files. However, if you use FTP, you must also implement the LONWORKS file transfer protocol within your application program. You would typically implement the LONWORKS file transfer protocol only if the total amount of related data exceeds (or is likely to exceed) the size of the direct memory file window.

See the File Transfer engineering bulletin at www.echelon.com/docs for more information about the LONWORKS file transfer protocol; see *Using Direct Memory Files* for more information about the direct memory files feature.

To indicate which file access method the application should use, you must declare the appropriate network variables in your model file:

- For direct memory files, declare an output network variable of type **SNVT_address**. If your device implements the **SFPTnodeObject** functional profile, you use this network variable to implement the profile's **nvoFileDirectory** member. If your device does not implement the **SFPTnodeObject** functional profile, simply add this network variable to the model file. You do not need to initialize this network variable (any initial value is ignored – the LonTalk Interface Developer utility calculates the correct value).
- For FTP, declare at least two mandatory network variables, an input network variable of type **SNVT_file_req**, and an output network variable of type **SNVT_file_status**. You also need to define a message tag for the transfer of the data. In addition, you need an input network variable of type **SNVT_file_pos** to support random access to the various files. You must also implement the LONWORKS file transfer protocol within your application program.

The LONWORKS file transfer protocol and the direct memory files features are mutually exclusive; your device cannot implement both.

Responding to Configuration Property Value Changes

Events are not automatically generated when a configuration property implemented in a configuration file is updated, but you can declare your configuration property so that a modification to its value causes the related functional block to be disabled and re-enabled, or causes the device to be taken offline and brought back online after the modification, or causes the entire device to reset. These state changes help to synchronize your application with new configuration property values.

Your application could monitor changes to the configuration file, and thus detect changes to a particular configuration property. Such monitoring would be implemented in the FTP server or direct memory files driver.

However, many applications do not need to know that a configuration property value has changed. For example, an application that uses a configuration property to parameterize an algorithm that uses some event as a trigger (such as a network variable update or a change to an input signal) would not typically need to know of the change to the configuration property value, but simply consider the most recent value.

Defining a Configuration Property Array

You can define a configuration property as:

- A single configuration property
- An array of configuration properties
- A configuration property array

A single configuration property either applies to one or more network variables or functional blocks within the model file for the device, or the configuration property applies to the entire device.

When you define an array of configuration properties, each element of the array can apply to one or more network variables or functional blocks within the model file.

When you define a configuration property array, the entire array (but not each element) applies to one or more network variables or functional blocks within the model file. That is, a configuration property array is atomic, and thus applies in its entirety to a particular item.

Assuming that the device has sufficient resources, it is always possible to define arrays of configuration properties. However, configuration property arrays are subject to the functional profile definition. For each member configuration property, the profile describes whether it can, cannot, or must be implemented as a configuration property array. The profile also describes minimum and maximum dimensions for the array. If you do not implement the configuration property array as the profile requires, the profile's implementation becomes incorrect.

Example:

This example defines a four-channel analog-to-digital converter (ADC), with the following properties:

- Four channels (implemented as an array of functional blocks)
- One gain setting per channel (implemented as an array of configuration properties)
- A single offset setting for the ADC (implemented as a shared configuration property)
- A linearization setting for all channels (implemented as a configuration property array)

```
#include <s32.h>
#define CHANNELS    4

network output   SNVT_volt    nvoAnalogValue[CHANNELS];

network input cp SCPTgain     nciGain[CHANNELS];
network input cp SCPToffset   nciOffset;
network input cp SCPTsetpoint nciLinearization[5];

fblock SFPTopenLoopSensor {
  // the actual network variable that implements the
  // mandatory 'nvoValue' member of this profile:
  nvoAnalogValue[0] implements nvoValue;
} fbAdc[CHANNELS] external_name("Analog Input")
fb_properties {
  // one gain factor per channel:
  nciGain[0],
  // one offset, common to all channels:
  static nciOffset,
  // one linearization array for all channels:
  static nciLinearization = {
    {0, 0}, {2, 0}, {4, 0}, {6, 0}, {8, 0}
  };
};
```

This example implements a single output network variable, of type **SNVT_volt**, per channel to represent the most recent ADC reading. This network variable has a fixed type, defined at compile-time, but could be defined as a changeable-type network variable if needed for the application.

There is one gain setting per channel, implemented as an array of configuration network variables, of type **SCPTgain**, where the elements of the array are distributed among the four functional blocks contained in the functional block array. Because the **SCPTgain** configuration property has a default gain factor of 1.0, no explicit initialization is required for this configuration property network variable.

There is a single offset setting, implemented as a configuration network variable, of type **SCPToffset**. This configuration NV applies to all channels, and is shared among the elements of the functional block array. The **SCPToffset** configuration property has a default value of zero.

The **SCPToffset** configuration property is a type-inheriting configuration property. The true data type of a type-inheriting property is the type of the

network variable to which the property applies. For an **SFPTopenLoopSensor** standard functional profile, the **SCPToffset** configuration property applies to the functional block, and thus implicitly applies to the profile's primary member network variable. In this example, the effective data type of this property is **SNVT_volt** (inherited from **nvoAnalogValue**).

The example also includes a five-point linearization factor, implemented as a configuration property array of type **SCPTsetpoint**. The **SCPTsetpoint** configuration property is also a type-inheriting configuration property, and its effective data type is also **SNVT_volt** in this example.

Because the **SCPTsetpoint** linearization factor is a configuration property array, it applies to the entire array of functional blocks, unlike the array of **SCPTgain** configuration property network variables, whose elements are distributed among the elements of the functional block array. In this example, the linearization configuration property array is implemented with configuration property network variables, and must be shared among the elements of the functional block array.

To implement the linearization array of configuration properties such that each of the four functional blocks has its own linearization data array, you must implement this configuration property array in files, and declare the configuration property with the **cp_family** modifier.

The following table shows the relationships between the members of the functional-block array. As the table shows, each channel has a unique gain value, but all channels share the offset value and linearization factor.

Channel	Gain	Offset	Linearization
fbAdc[0]	nciGain[0]	nciOffset	nciLinearization[0..4]
fbAdc[1]	nciGain[1]		
fbAdc[2]	nciGain[2]		
fbAdc[3]	nciGain[3]		

Sharing a Configuration Property

The typical instantiation of a configuration property is unique to a single device, functional block, or network variable. For example, a configuration property family whose name appears in the property list of five separate network variables has five instantiations, and each instance is specific to a single network variable. Similarly, a network variable array of five elements that includes the same configuration property family name in its property list instantiates five members of the configuration property family, and each one applies to one of the network variable array elements.

Rather than creating extra configuration property instances, you can specify that functional blocks or network variables share a configuration property by including the **static** or **global** keywords in the configuration property declaration.

The **global** keyword causes a configuration property member to be shared among all the functional blocks or network variables whose property list contains that configuration property family name. The functional blocks or network variables in the configuration property family can have only one such global member. Thus, if you specify a global member for both the functional blocks and the network variables in a configuration property family, the global member shared by the functional blocks is a *different* member than the global member shared by the network variables.

The **static** keyword causes a configuration property family member to be shared among all elements of the array it is associated with (either network variable array or functional block array). However, the sharing of the static member does not extend to other network variables or functional blocks outside of the array.

Example 1:

```
// CP for throttle (default 1 minute)
SCPTmaxSndT cp_family cpMaxSendT = { 0, 0, 1, 0, 0 };

// NVs with shared throttle:
network output SNVT_lev_percent nvoValue1
   nv_properties {
      global cpMaxSendT
   };
network output SNVT_lev_percent nvoValue2
   nv_properties {
      global cpMaxSendT       // the same as the one above
   };
network output SNVT_lev_percent nvoValueArray[10]
   nv_properties {
      static cpMaxSendT       // shared among the array
                              // elements only
   };
```

In addition to sharing members of a configuration property family, you can use the **static** or **global** keywords for a configuration network variable to specify sharing. However, a shared configuration property network variable cannot appear in two or more property lists without the **global** keyword because there is only one instance of the network variable (configuration property families can have multiple instances).

A configuration property that applies to a device cannot be shared because there is only one device per application.

Example 2:

The following model file defines a three-phase ammeter, implemented with an array of three **SFPTopenLoopSensor** functional blocks. The hardware for this device contains a separate sensor for each phase, but a common analog-to-digital converter for all three phases. Each phase has individual gain factors, but shares one property to specify the sample rate for all three phases.

```
#define  NUM_PHASES     3

SCPTgain cp_family cpGain;
SCPTupdateRate cp_family cpUpdateRate;

network output SNVT_amp nvoAmpere[NUM_PHASES];
```

```
fblock SFPTopenLoopSensor {
   nvoAmpere[0] implements nvoValue;
} fbAmpereMeter[NUM_PHASES]  external_name("AmpereMeter")
   fb_properties {
      cpGain,
      static cpUpdateRate
   };
```

Inheriting a Configuration Property Type

You can define a configuration property type that does not include a complete type definition, but instead references the type definition of the network variable to which it applies. A configuration property type that references another type is called a *type-inheriting configuration property*. When the configuration property family member for a type-inheriting configuration property appears in a property list, the instantiation of the configuration property family member uses the type of the network variable. Likewise, a configuration network variable can be type-inheriting; however, for configuration network variable arrays and arrays of configuration network variables, each element of the array must inherit the same type.

Type-inheriting configuration properties that are listed in an **nv_properties** clause inherit the type from the network variable to which they apply. Type-inheriting configuration properties that are listed in an **fb_property** clause inherit their type from the functional profile's principal network variable member, an attribute that is assigned to exactly one network variable member.

Recommendation: Because the type of a type-inheriting configuration property is not known until instantiation, specify the configuration property initializer option in the property list rather than in the declaration. Likewise, specify the *range-mod* string in the property list because different *range-mod* strings can apply to different instantiations of the property.

Restrictions:

- Type-inheriting configuration network variables that are also shared can only be shared among network variables of identical type.
- A type-inheriting configuration property cannot be used as a device property, because the device has no type from which to inherit.

A typical example of a type-inheriting configuration property is the **SCPTdefOutput** configuration property type. Several functional profiles list the **SCPTdefOutput** configuration property as an optional configuration property, and use it to define the default value for the sensor's principal network variable. The functional profile itself, however, might not define the type for the principal network variable.

The following example implements a **SFPTopenLoopSensor** functional block with an optional **SCPTdefOutput** configuration property. The configuration property inherits the type from the network variable it applies to, **SNVT_amp** in this case.

Example 1:

```
SCPTdefOutput cp_family cpDefaultOutput;

network output SNVT_amp nvoAmpere nv_properties {
```

```
   cpDefaultOutput = 123
};

fblock SFPTopenLoopSensor {
   nvoAmpere implements nvoValue;
} fbAmpereMeter;
```

The initial value (123) must be provided in the instantiation of the configuration property, because the type for **cpDefaultOutput** is not known until it is instantiated.

You can also combine type-inheriting configuration properties with network variables that are of changeable type. The type of such a network variable can be changed dynamically by a network integrator when the device is installed in a network.

Example 2:

```
SCPTdefOutput cp_family cpDefaultOutput;
SCPTnvType cp_family    cpNvType;

network output changeable_type SNVT_amp nvoValue
   nv_properties {
      cpDefaultOutput = 123,
      cpNvType
   };

fblock SFPTopenLoopSensor {
   nvoValue implements nvoValue;
} fbGenericMeter;
```

The **nvoValue** principal network variable, although it is of changeable type, must still implement a default type (**SNVT_amp** in the example). The **SCPTdefOutput** type-inheriting configuration property inherits the type information from this initial type. Therefore, the initializer for **cpDefaultOutput** must be specific to this instantiation. Furthermore, the initializer must be valid for this initial type.

If the network integrator decides to change this type at runtime, for example, to **SNVT_volt**, then it is in the responsibility of the network management tool to apply the formatting rules that apply to the new type when reading or writing this configuration property. However, your application has the responsibility to propagate the new type to this network variable's type-inheriting configuration properties (if any).

Declaring a Message Tag

You can declare a message tag in a model file. A *message tag* is a connection point for application messages. Application messages are used for the LONWORKS file transfer protocol, and are also used to implement standard and proprietary interfaces to LONWORKS devices as described in Chapter 8, *Developing a LonTalk Stack Device Application*.

Message tag declarations do not generate code, but result in a simple enumeration, whose members are used to identify individual tags. There are two basic forms of message tags: bindable and nonbindable.

Example:

```
msg_tag myBindableMT;
msg_tag bind_info(nonbind) myNotBindableMT;
```

Similar to network variables, you can connect bindable message tags together, thus allowing applications to communicate with each other through the message tags (rather than having to know specific device addressing details). Each bindable message tag requires one address-table space for its exclusive use.

Sending application messages through bindable message tags is also known as sending application messages with implicit addressing.

Nonbindable message tags enable (and require) the use of explicit addresses, which the sending application must provide. However, these addresses do not require address-table space.

Defining a Resource File

Functional profiles, network variable types, and configuration property types are defined in *resource files*. LONWORKS resource files use a standard format that is recognized by all interoperable network management tools, such as the OpenLNS Commissioning Tool. This standard format enables device manufacturers to create definitions for user functional profiles, user network variable types (UNVTs), and user configuration property types (UCPTs) that can be used during installation by a network integrator using any interoperable network management tool.

A set of standard functional profiles, standard network variable types (SNVTs), and standard configuration property types (SCPTs) is defined by a standard resource file set distributed by LONMARK International (www.lonmark.org). A functional profile defined in a resource file is also called a *functional profile template*.

Resource files are grouped into *resource file sets*, where each set applies to a specified range of program IDs. A complete resource file set consists of a type file (.TYP extension), a functional profile definitions file (.FPT extension), a format file (.FMT extension), and one or more language files (.ENG, .ENU, or other extensions).

Each set defines functional profiles, network variable types, and configuration properties for a particular type of device. The program ID range is determined by a *program ID template* in the file, and a *scope* value for the resource file set. The scope value specifies which fields of the program ID template are used to match the program ID template to the program ID of a device. That is, the range of device types to which a resource file applies is the scope of the resource file.

The program ID template has an identical structure to the program ID of a device, except that the applicable fields might be restricted by the scope. The scope value is a kind of filter that indicates the relevant parts of the program ID. For example, the scope can specify that the resource file applies to an individual device type, or to all device types.

You can specify a resource file for any of the following scopes:

0 – Standard
Applies to all devices.

1 – Device Class
Applies to all devices with the specified device class.

2 – Device Class and Subclass
Applies to all devices with the specified device class and subclass.

3 – Manufacturer
Applies to all devices from the specified manufacturer.

4 – Manufacturer and Device Class
Applies to all devices from the specified manufacturer with the specified device class.

5 – Manufacturer, Device Class, and Device Subclass
Applies to all devices from the specified manufacturer with the specified device class and device subclass.

6 – Manufacturer, Device Class, Device Subclass, and Device Model
Applies to all devices of the specified type from the specified manufacturer.

For scopes 1 through 6, the program ID template included in the resource file set specifies the components. Network management tools match this template against the program ID for a device when searching for an appropriate resource file.

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

Example: Consider a resource file set with a program ID template of 81:23:45:01:02:05:04:00, with manufacturer and device class scope (scope 4). Any device with the manufacturer ID fields of the program ID set to 1:23:45 and the device class ID fields set to 01:02 would be able to use types defined in this resource file set. However, resources on devices of the same class, but from a different manufacturer, could not access this resource file set.

A resource file set can also use information in any resource file set that has a numerically lower scope, as long as the relevant fields of their program ID templates match. For example, a scope 4 resource file set can use resources in a scope 3 resource file set, assuming that the manufacturer ID components of the resource file sets' program ID templates match.

Scopes 0 through 2 are reserved for standard resource definitions published by Echelon and distributed by LONMARK International. Scope 0 applies to all devices, and scopes 1 and 2 are reserved for future use. Because scope 0 applies to all devices, there is a single scope 0 resource file set called the *standard resource file set*.

The LonTalk Stack includes the scope 0 standard resource file set that defines the standard functional profiles (SFPTs), SNVTs, and SCPTs (updates are also available from LONMARK International at www.lonmark.org). The kit also includes the NodeBuilder Resource Editor that you can use to view the standard resource file set, or use to create your own user functional profiles (UFPTs), UNVTs, and UCPTs.

You can define your own functional profiles, types, and formats in scope 3 through 6 resource files.

Most OpenLNS tools, including the OpenLNS Commissioning Tool assume a default scope of 3 for all user resources. OpenLNS automatically sets the scope to

the highest (most specific) applicable scope level. See the *NodeBuilder FX User's Guide* for information about developing a plug-in to set the scope, or see the *OpenLNS Commissioning Tool User's Guide* (or online help) for information about modifying a device shape to set the scope.

Implementation-Specific Scope Rules

When you add implementation-specific network variables or configuration properties to a standard or user functional profile, you must ensure that the scope of the resource definition for the additional item is numerically less than or equal to the scope of the functional profile, and that the member number is set appropriately. For example:

- If you add an implementation-specific network variable or configuration property to a standard functional block (SFPT, scope 0), it must be defined by a standard type (SNVT, or SCPT).
- If you implement a functional block that is based on a manufacturer scope resource file (scope 3), you can add an implementation-specific network variable or configuration property that is defined in the same scope 3 resource file, and you can also add an implementation-specific network variable or configuration property that is defined by a SNVT or SCPT (scope 0).

You can add implementation-specific members to standard functional profiles using inheritance by performing the following tasks:

1. Use the NodeBuilder Resource Editor to create a user functional profile with the same functional profile key as the standard functional profile.
2. Set **Inherit Members from Scope 0** in the functional profile definition. This setting makes all members of the standard functional profile part of your user functional profile.
3. Declare a functional block based on the new user functional profile.

Add implementation-specific members to the functional block.

Writing Acceptable Neuron C Code

When processing the model file, the LonTalk Interface Developer utility distinguishes between three categories of Neuron C statements:

- Acceptable
- Ignored – ignored statements produce a warning
- Unacceptable – unacceptable statements produce an error

Appendix B, *Model File Compiler Directives*, lists the acceptable and ignored compiler directives for model files. All other compiler directives are not accepted by the LonTalk Interface Developer utility and cause an error if included in a model file. A statement can be unacceptable because it controls features that are meaningless in a LonTalk Stack device, or because it refers to attributes that are determined by the LonTalk protocol stack or by other means.

The LonTalk Interface Developer utility ignores all executable code and I/O object declarations. These constructs cause the LonTalk Interface Developer utility to issue a warning message. The LonTalk Interface Developer utility

predefines the **_FTXL** and **_MODEL_FILE** macros, so that you can use **#ifdef** or **#ifndef** compiler directives to control conditional compilation of source code that is used for standard Neuron C compilation and as an LonTalk Stack model file.

All constructs not specifically mentioned as unacceptable or ignored are acceptable.

Anonymous Top-Level Types

Anonymous top-level types are not valid. The following Neuron C construct is not valid:

```
network output struct {int a; int b;} nvoZorro;
```

This statement is not valid because the type of the **nvoZorro** network variable does not have a name. The LonTalk Interface Developer utility issues an error when it detects such a construct.

Using a named type solves the problem, for example:

```
typedef struct {
   int a;
   int b;
} Zorro;
network output Zorro nvoZorro;
```

The use of anonymous sub-types is permitted. For example, the LonTalk Interface Developer utility allows the following type definition:

```
typedef struct {
   int   a;
   int   b;
   struct {
      long  x;
      long  y;
      long  z;
   }  c;
} Zorro;
network output Zorro nvoZorro;
```

Legacy Neuron C Constructs

You must use the Neuron C Version 2.1 syntax described in this manual. You cannot use legacy Neuron C constructs for defining LONMARK-compliant interfaces. That is, you cannot use the **config** modifier for network variables, and you cannot use Neuron C legacy syntax for declaring functional blocks or configuration properties. The legacy syntax used an **sd_string()** modifier containing a string that starts with a '&' or '@' character.

Using Authentication for Network Variables

Authentication is a special acknowledged service between one source device and one or more (up to 63) destination devices. Authentication is used by the destination devices to verify the identity of the source device. This type of service is useful, for example, if a device containing an electronic lock receives a message to open the lock. By using authentication, the electronic lock device can verify

that the "open" message comes from the owner, not from someone attempting to break into the system.

Authentication doubles the number of messages per transaction. An acknowledged message normally requires two messages: an update and an acknowledgment. An authenticated message requires four messages, as illustrated in the next section. These extra messages can affect system response time and capacity.

A device can use authentication with acknowledged updates or network variable polls. However, a device cannot use authentication with unacknowledged or repeated updates.

For a program to use authenticated network variables or send authenticated messages, you must perform the following steps:

1. Declare the network variable as authenticated, or allow the network management tool to specify that the network variable is to be authenticated.
2. Specify the authentication key to be used for this device using a network management tool, and enable authentication. You can use the OpenLNS Commissioning Tool to install a key during network integration, or your application can use the **LonQueryDomainConfig()** and **LonUpdateDomainConfig()** API functions to install a key locally.

Specifying the Authentication Key

All devices that read or write a given authenticated network variable connection must have the same authentication key. This 48-bit authentication key is used in a special way for authentication, as described in the next section, *How Authentication Works*. If a device belongs to more than one domain, you must specify a separate key for each domain.

The key itself is transmitted to the device only during the initial configuration. All subsequent changes to the key do not involve sending it over the network. The network management tool can modify a device's key over the network, in a secure fashion, with a network management message.

Alternatively, your application can use a combination of the **LonQueryDomainConfig()** and **LonUpdateDomainConfig()** API calls to specify the authentication keys during application start-up.

If you set the authentication key during device manufacturing, you must perform the following tasks to ensure that the key is not exposed to the network during device installation:

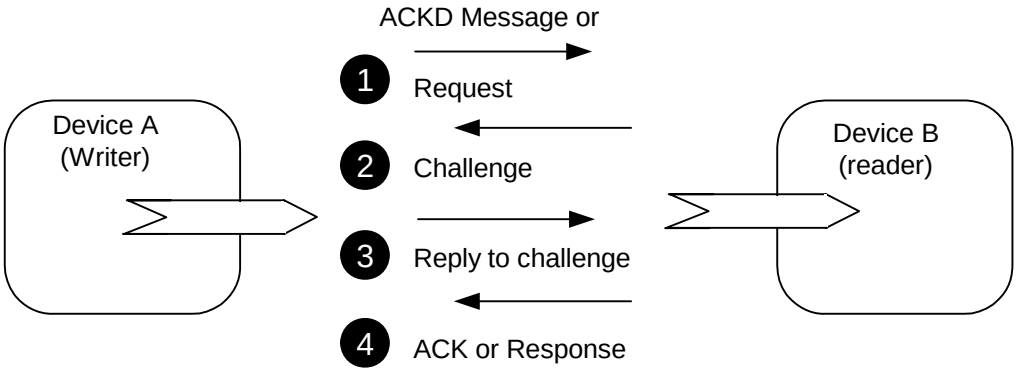
1. Specify that the device should use network-management authentication (set the configuration data in the **LonConfigData** data structure, which is defined in the **FtxlTypes.h** file).
2. Set the device's state to configured. An unconfigured device does not enforce authentication.
3. Set the device's domain to a unique domain value to avoid address conflicts during device installation.

If you do not set the authentication key during device manufacturing, the device installer can specify authentication for the device using the network management

tool, but must specify an authentication key because the device has only a default key.

How Authentication Works

The following figure illustrates the authentication process:



1. Device A uses the acknowledged service to send an update to a network variable that is configured with the authentication attribute on Device B. If Device A does not receive the challenge (described in step 2), it sends a retry of the initial update.
2. Device B generates a 64-bit random number and returns a challenge packet that includes the 64-bit random number to Device A. Device B then uses an encryption algorithm (part of the LonTalk host stack) to compute a transformation on that random number using its 48-bit authentication key and the message data. The transformation is stored in Device B.
3. Device A then also uses the same encryption algorithm to compute a transformation on the random number (returned to it by Device B) using its 48-bit authentication key and the message data. Device A then sends this computed transformation to Device B.
4. Device B compares its computed transformation with the number that it receives from Device A. If the two numbers match, the identity of the sender is verified, and Device B can perform the requested action and send its acknowledgment to Device A. If the two numbers do not match, Device B does not perform the requested action, and an error is logged in the error table.

If the acknowledgment is lost and Device A tries to send the same message again, Device B remembers that the authentication was successfully completed and acknowledges it again.

If Device A attempts to update an output network variable that is connected to multiple readers, each receiver device generates a different 64-bit random number and sends it in a challenge packet to Device A. Device A must then transform each of these numbers and send a reply to each receiver device.

The principal strength of authentication is that it cannot be defeated by simple record and playback of commands that implement the desired functions (for example, unlocking the lock). Authentication does not require that the specific

messages and commands be secret, because they are sent unencrypted over the network, and anyone who is determined can read those messages.

It is good practice to connect a device directly to a network management tool when initially installing its authentication key. This direct connection prevents the key from being sent over the network, where it might be detected by an intruder. After a device has its authentication key, a network management tool can modify the key, over the network, by sending an increment to be added to the existing key.

You can update the device's address without having to update the key, and you can perform authentication even if the devices' domains do not match. Thus, a LonTalk Stack device can set its key during device manufacturing, and you can then use a network management tool to update the key securely over the network.

Managing Memory

The LonTalk Interface Developer Neuron C compiler generates four tables that affect memory usage. The LonTalk host stack and network management tools use these tables to define the network configuration for a device. The LonTalk Interface Developer utility allocates space for the following tables:

- Address table
- Alias table
- Domain table
- Network variable configuration table

See the *ISO/IEC 14908-1 Control Network Protocol Specification* for more information about these tables. This document is available from ISO:

www.iso.org/iso/home/store/catalogue_tc/catalogue_detail.htm?csnumber=60203.

See Appendix E, *Determining Memory Usage for LonTalk Stack Applications*, for information about how to calculate the memory requirements for you LonTalk Stack application.

Address Table

The address table contains the list of network addresses to which the device sends network variable updates or polls, or sends implicitly-addressed application messages. You can configure the address table through network management messages from a network management tool.

By default, the LonTalk Interface Developer utility calculates the size of the address table. The utility calculates the required number of address table entries based on parameters defined in the device's interface, such as the number of static polling input network variables, static non-constant output network variables, bindable message tags, the number of aliases, and the number of dynamic network variables. The utility always allocates at least 15 address table entries. Within the LonTalk Interface Developer utility, you can override the automatic calculation of the table size and specify any number of entries, from 0 to 4096.

The maximum number of address table entries that a device could require is determined by the expected maximum number of different destination entries that the device requires for connections (network variables and bindable message tags).

The size of the address table affects the amount of RAM and non-volatile memory required for the device. When the LonTalk Interface Developer utility calculates the size of the address table, it attempts to balance the need to limit the amount of resources required (small address table) and the need for comprehensive coverage (large address table). Although you generally do not need to, you can override the automatically calculated value with one that reflects the use of the device.

Alias Table

An alias is an abstraction for a network variable that is managed by network management tools and the LonTalk host stack. Network management tools use aliases to create connections that cannot be created solely with the address and network variable tables. Aliases provide network integrators with more flexibility for how devices are installed into networks.

The alias table has no default size, and can contain up to 8192 entries. The LonTalk Interface Developer utility calculates the size of the alias table. The utility calculates the required number of alias table entries based on parameters defined in the device's interface, such as the number of static network variables and the number of supported dynamic network variables. The utility always allocates at least 5 alias table entries, unless the device does not support any network variables. Within the LonTalk Interface Developer utility, you can override the automatic calculation of the table size and specify any number of entries, from 0 to 8192.

The maximum number of aliases that a device could require depends on its involvement in network variable connections and the characteristics of these connections. The size of the alias table also affects the performance of the device, because the alias table must be searched whenever network variable updates arrive. When the LonTalk Interface Developer utility calculates the size of the alias table, it attempts to balance the need for performance (small alias table) and the need for comprehensive coverage (large alias table). Although you generally do not need to, you can override the automatically calculated value with one that reflects the use of the device.

Domain Table

The number of domain table entries is dependent on the network in which the device is installed; it is not dependent on the application.

The LonTalk Interface Developer utility always allocates 2 domain table entries. From the command-line interface for the LonTalk Interface Developer utility, you can override the number of entries. However, LONMARK International requires all interoperable LONWORKS devices to have two domain table entries. Reducing the size of the domain table to one entry will prevent certification.

Network Variable Configuration Table

This table contains one entry for each network variable that is declared in the model file. Each element of a network variable array counts separately.

The maximum size of the network variable configuration table is 4096 entries. You cannot change the size of this table, except by adding or deleting static network variables or by increasing or decreasing the number of dynamic network variables.

Example Model files

This section describes a few example model files, with increasing levels of complexity.

See *Network Variable and Configuration Property Declarations* for information about mapping types and items declared in the model file to those shown in the LonTalk Interface Developer utility-generated application framework.

Simple Network Variable Declarations

This example declares one input network variable and one output network variable. Both network variables are declared with the **SNVT_count** type. The names of the network variables (**nviCount** and **nvoCount**) are arbitrary. However, it is a common practice to use the "nvi" prefix for input network variables and the "nvo" prefix for output network variables.

```
network input  SNVT_count     nviCount;
network output SNVT_count     nvoCount;
```

The LonTalk Interface Developer utility compiles this model file into an application framework that contains, among other things, two global C variables in the **FtxlDev.c** file:

```
volatile SNVT_count nviCount;
SNVT_count nvoCount;
```

When an update occurs for the input network variable (**nviCount**), the LonTalk host stack stores the updated value in the global variable. The application can use this variable like any other C variable. When the application needs to update the output value, it updates the **nvoCount** variable, so that the LonTalk Host stack can read the updated value and send it to the network.

For more information about how the LonTalk Interface Developer utility-generated framework represents network variables, see *Using Types*.

Important: This example is not interoperable because it does not use functional blocks to define the purpose of these network variables. However, this type of declaration can define a functioning device for an initial test application.

Network Variables Using Standard Types

A more complete example includes the use of more complex standard network variable types and declarations. This example provides the model for a simple electricity meter, where all input data is retrieved from the network through the **nviAmpere**, **nviVolt**, and **nviCosPhi** input network variables. The result is

posted to the **nvoWattage** output network variable. A second **nvoUsage** output network variable is polled and uses non-volatile storage to count the meter's total lifetime.

```
network input                  SNVT_amp          nviAmpere;
network input                  SNVT_volt         nviVolt;
network input                  SNVT_angle        nviCosPhi;
network output                 SNVT_power        nvoWattage;
network output polled eeprom   SNVT_elapsed_tm   nvoUsage;
```

The LonTalk Interface Developer utility generates type definitions in the **LonNvTypes.h** file for all of the above network variables. However, it does not generate type definitions in the **LonCpTypes.h** file because there are no configuration properties.

In addition to the type definitions and other data, the LonTalk Interface Developer utility generates the following global C variables for this model file:

```
volatile SNVT_amp nviAmpere;
volatile SNVT_volt nviVolt;
volatile SNVT_angle nviCosPhi;
SNVT_power nvoWattage;
SNVT_elapsed_tm nvoUsage;
```

The declaration of the **nvoUsage** output network variable uses the network variable modifiers **polled** and **eeprom**. The LonTalk Interface Developer utility stores these attributes in the network-variable table (**nvTable[]**) in the **FtxlDev.c** file. The API uses this table to access the network variables when the application runs. In addition, the application can query the data in this table at runtime.

Important: This example is not interoperable because it does not use functional blocks to define the purpose of these network variables. However, this type of declaration can define a functioning device for an initial test application.

Functional Blocks without Configuration Properties

The following model file describes a similar meter application as in the previous example, but implements it using functional blocks to provide an interoperable interface:

- A node object based on the *SFPTnodeObject* functional profile to manage the entire device
- An array of three meters, each based on the same user-defined *UFPTenergyMeter* profile, implementing three identical meters.

Configuration properties are not used in this example.

```
// Node object
network input           SNVT_obj_request  nviNodeRequest;
network output polled   SNVT_obj_status   nvoNodeStatus;

fblock SFPTnodeObject {
   nviNodeRequest implements nviRequest;
   nvoNodeStatus  implements nvoStatus;
} NodeObject external_name("NodeObject");

// UFPTenergyMeter
```

```
// Implements the meter from the previous example.
network input                 SNVT_amp    nviAmpere[3];
network input                 SNVT_volt   nviVoltage[3];
network input                 SNVT_angle  nviCosPhi[3];
network output                SNVT_power  nvoWattage[3];
network output polled eeprom  SNVT_elapsed_tm  nvoUsage[3];

fblock UFPTenergyMeter {
   nvoWattage[0]  implements nvoWattage;
   nviAmpere[0]   implements nviAmpere;
   nviVoltage[0]  implements nviVoltage;
   nviCosPhi[0]   implements nviCosPhi;
   nvoUsage[0]    implements nvoUsage;
} Meter[3] external_name("Meter");
```

Because functional blocks only provide logical grouping of network variables and configuration properties, and meaning to those groups, but do not themselves contain executable code, the functional blocks appear only in the self-documentation data generated by the LonTalk Interface Developer utility, but not in any generated executable code.

Functional Blocks with Configuration Network Variables

The following example takes the above example and adds a few configuration properties implemented as configuration network variables. A **cp** modifier in the network variable declaration makes the network variable a configuration network variable. The **nv_properties** and **fb_properties** modifiers apply the configuration properties to specific network variables or the functional block.

```
// Configuration properties for the node object
network input cp SCPTlocation nciLocation;

// Network variables for the node object
network input           SNVT_obj_request  nviNodeRequest;
network output polled   SNVT_obj_status   nvoNodeStatus;

fblock SFPTnodeObject {
   nviNodeRequest implements nviRequest;
   nvoNodeStatus  implements nvoStatus;
} NodeObject external_name("NodeObject")
fb_properties {
   nciLocation
};

// Config properties for the Meter
network input cp SCPTminSendTime nciMinSendTime[3];
network input cp SCPTmaxSendTime nciMaxSendTime[3];
network input cp UCPTcoupling    nciCoupling;

// Network variables for the meter
network input  SNVT_amp       nviAmpere[3];
network input  SNVT_volt      nviVoltage[3];
network input  SNVT_angle     nviCosPhi[3];
network output SNVT_power     nvoWattage[3] nv_properties {
   nciMinSendTime[0],
```

```
   nciMaxSendTime[0]
};

network output polled eeprom SNVT_elapsed_tm nvoUsage;

fblock UFPTenergyMeter {
   nvoWattage[0]  implements nvoWattage;
   nviAmpere[0]   implements nviAmpere;
   nviVoltage[0]  implements nviVoltage;
   nviCosPhi[0]   implements nviCosPhi;
   nvoUsage[0]    implements nvoUsage;
} Meter external_name("Meter") fb_properties {
   static nciCoupling
};
```

This example implements two arrays of configuration network variables, *nciMinSendTime* and *nciMaxSendTime*. Each element of these two arrays applies to one element of the *nvoWattage* array, starting with *nciMinSendTime[0]* and *nciMaxSentTime[0]*. Each element of the *nvoWattage* array of network variables in turn implements the *nvoWattage* member of one element of the *Meter* array of functional blocks, again starting with *nvoWattage[0]*.

The user-defined *UCPTcoupling* configuration property *nciCoupling* is shared among all three meters, configuring the meters as three single-phase meters or as one three-phase meter in this example. There is only a single *nciCoupling* configuration property, and it applies to every element of the array of three *UFPTenergyMeter* functional blocks.

The LonTalk Interface Developer utility creates a network variable table for the configuration network variables and the persistent *nvoUsage* network variable.

Functional Blocks with Configuration Properties Implemented in a Configuration File

This example implements a device similar to the one in the previous example, with these differences:

1. All configuration properties are implemented within a configuration file instead of as a configuration network variable
2. A *SNVT_address* type network variable is declared to enable access to these files through the direct memory files feature
3. An *SFPTnodeObject* node object has been added to support the SNVT address network variable

```
// Config properties for the node object
SCPTlocation cp_family cpLocation;

// Network variables for the node object
network input           SNVT_obj_request  nviNodeRequest;
network output polled   SNVT_obj_status   nvoNodeStatus;
const network output polled SNVT_address  nvoFileDirectory;

// Node object
fblock SFPTnodeObject {
   nviNodeRequest       implements nviRequest;
```

```
   nvoNodeStatus        implements nvoStatus;
   nvoFileDirectory     implements nvoFileDirectory;
} NodeObject external_name("NodeObject") fb_properties {
   cpLocation
};

// Config properties for the Meter
SCPTminSendTime cp_family cpMinSendTime;
SCPTmaxSendTime cp_family cpMaxSendTime;
UCPTcoupling    cp_family cpCoupling;

// Network variables for the meter
network input  SNVT_amp       nviAmpere[3];
network input  SNVT_volt      nviVoltage[3];
network input  SNVT_angle     nviCosPhi[3];
network output SNVT_power     nvoWattage[3] nv_properties {
   cpMinSendTime,
   cpMaxSendTime
};

network output polled eeprom SNVT_elapsed_tm nvoUsage[3];

fblock UFPTenergyMeter {
   nvoWattage[0]  implements nvoWattage;
   nviAmpere[0]   implements nviAmpere;
   nviVoltage[0]  implements nviVoltage;
   nviCosPhi[0]   implements nviCosPhi;
   nvoUsage[0]    implements nvoUsage;
} Meter[3] external_name("Meter") fb_properties {
   static cpCoupling
};
```

The addition of the *SNVT_address* typed network variable *nvoFileDirectory* is important for enabling the direct memory files feature for access to the configuration property files. The LonTalk Interface Developer initializes this network variable's value correctly, and creates all required structures and code for direct memory file access; see *Using Direct Memory Files* for more information.

Alternatively, you can use the LONWORKS File Transfer Protocol (FTP) to access the file directory and the files in the directory. In this case, you need to implement the network variables and message tags as needed for the implementation of a LONWORKS FTP server in the model file, and provide application code in your host to implement the protocol. See the File Transfer engineering bulletin at www.echelon.com/docs for more information about the LONWORKS file transfer protocol.

7

Using the LonTalk Interface Developer Utility

You use the model file, described in Chapter 6, and the LonTalk Interface Developer utility to define the network inputs and outputs for your device, and to create your application's skeleton framework source code. You use this skeleton application framework as the basis for your LonTalk Stack application development.

The utility also generates device interface files that are used by a network management tool when designing a network that uses your device.

This chapter describes how to use the LonTalk Interface Developer utility and its options, and describes the files that it generates and how to use them.

Running the LonTalk Interface Developer

You use the LonTalk Interface Developer utility to create the application framework files that are required for your LonTalk Stack application. The LonTalk Interface Developer utility also generates the device interface files (*.xif and *.xfb) that can be used by network management tools to design a network that uses your device.

To create the device interface data and device interface files for your device, perform the following tasks:

1. Create a model file as described in Chapter 6, *Creating a Model File*.
2. Start the LonTalk Interface Developer utility: from the Windows **Start** menu, select **Programs → Echelon LonTalk Stack → LonTalk Interface Developer**.
3. In the LonTalk Interface Developer utility, specify the program ID, the model file for the device, and other preferences for the utility. The utility uses this information to generate a number of files that your application uses. See *Using the LonTalk Interface Developer Files*.
4. Add the **FtxlDev.h** ANSI C header file to your LonTalk Stack application with an include statement:

   ```
   #include "FtxlDev.h"
   ```

The LonTalk Interface Developer utility creates the application framework files and copies other necessary files (such as the LonTalk API files and the LonTalk host stack) to your project directory.

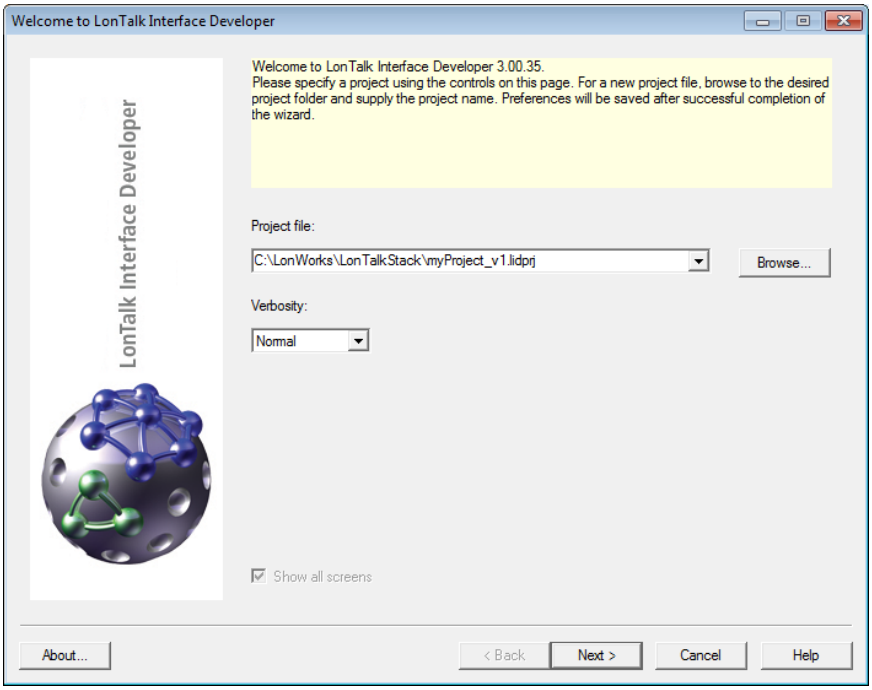
If you modify the LonTalk Interface Developer utility-generated files without first copying them, any changes that you make will be overwritten the next time you run the utility. However, the LonTalk Interface Developer utility does not overwrite or modify the LonTalk API files.

After you have created the LonTalk Interface Developer utility-generated files, modify and add code to your application using the LonTalk API to implement desired LONWORKS functionality into your LonTalk Stack application. See Chapter 8, *Developing a LonTalk Stack Device Application*, for information about how to use the LonTalk API calls to implement LONWORKS tasks.

Note: The LonTalk Interface Developer, source code, and examples include many instances of "FTXL". This is because the LonTalk Interface Developer was initially developed for the FTXL Development Kit and the LonTalk Stack uses the same API as FTXL.

Specifying the Project File

From the Welcome to LonTalk Interface Developer page of the utility, you can enter the name and location of a new or existing LonTalk Stack project file (.**lidprj** extension). The LonTalk Interface Developer utility uses this project file to maintain your preferences for this project. The base name of the project file is also used as the base name for the device interface files that the utility generates.



You can include a project version number in the name of the project to facilitate version control and project management for your LonTalk Interface Developer projects.

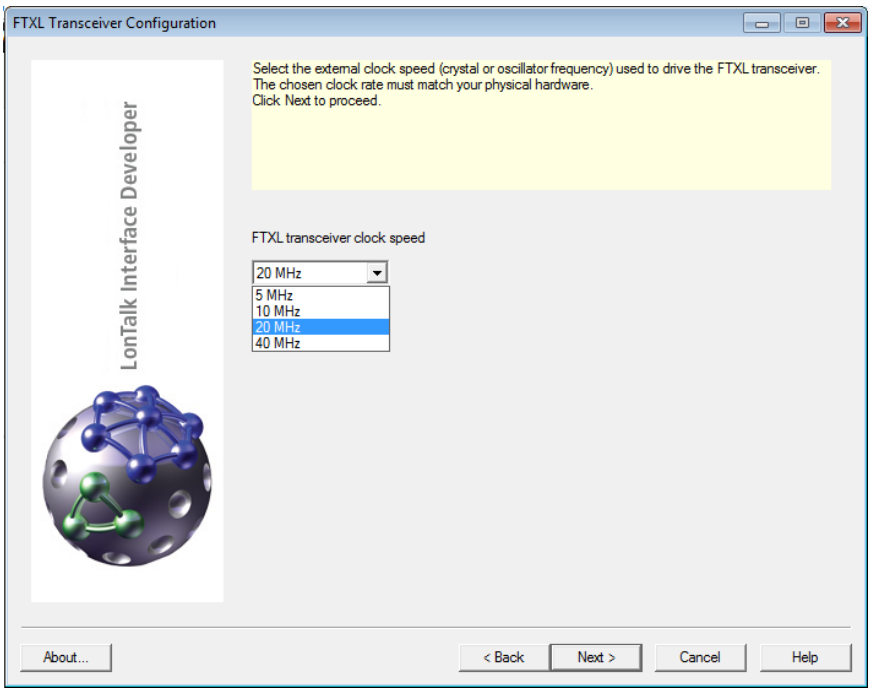
The utility creates all of its output files in the same directory as the project file. Your application's model file does not need to be in this directory; from the utility's Model File Selection page, you can specify the name and location of the model file.

The location of the LonTalk Interface Developer project file can be the same as your application's project folder, but you can also generate and maintain the LonTalk Interface Developer's project in a separate folder, and manually link the latest generated framework with your application by copying or referencing the correct location.

Click **Next** and then click **Yes** to confirm the creation of the new project.

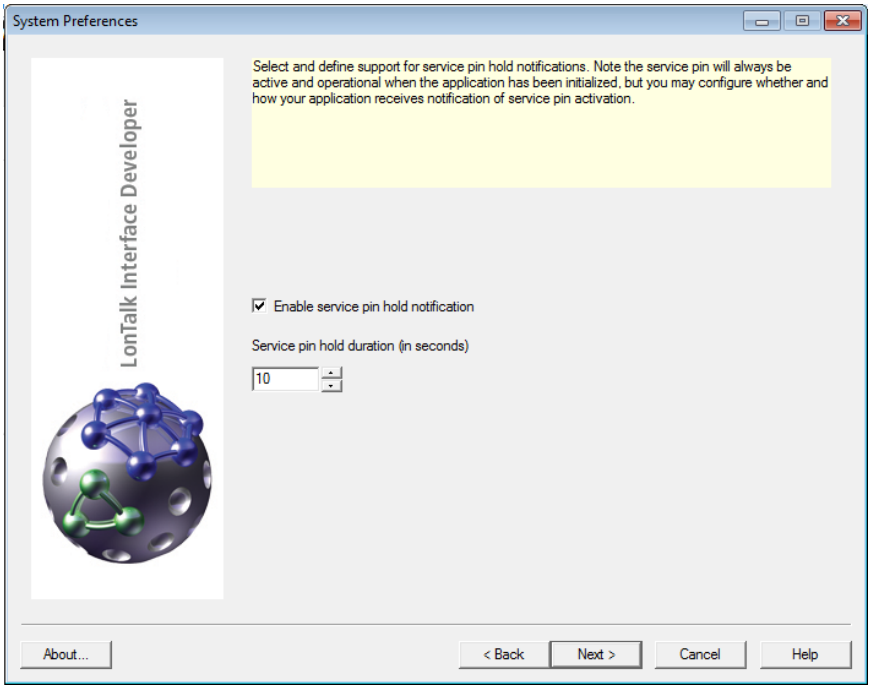
Specifying the Echelon Smart Transceiver or Neuron Chip Configuration

From the Echelon Smart Transceiver or Neuron Chip Configuration page of the utility, you can specify the clock speed for the Echelon Smart Transceiver or Neuron Chip.



Select a clock speed and then click **Next**.

In the **System Preferences** dialog, click **Next** (the LonTalk Stack does not have access to the service pin; therefore, the options in this dialog do not affect your LonTalk Stack device application).

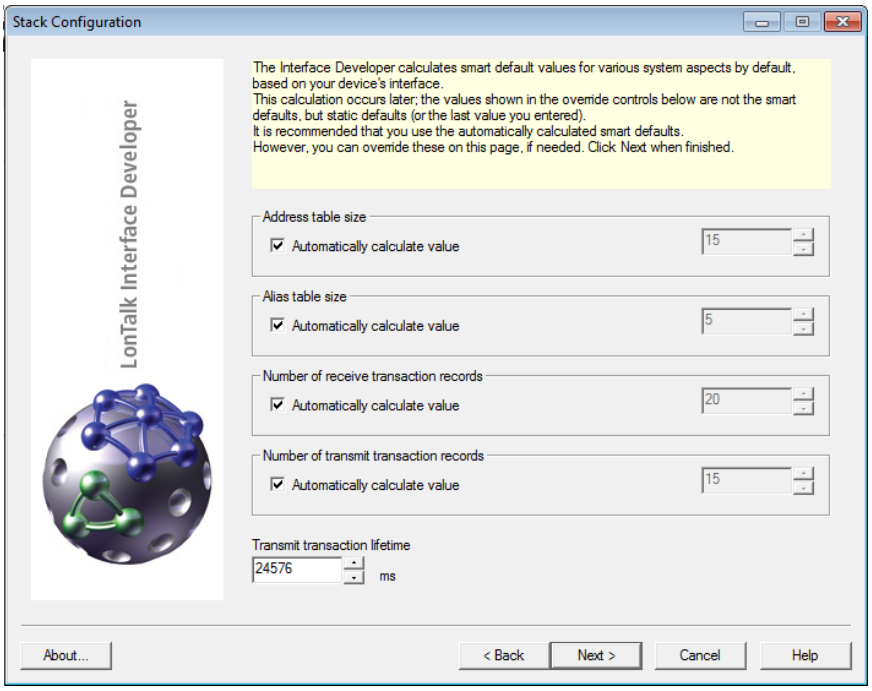


Configuring the LonTalk Stack

From the Stack Configuration page of the utility, you can specify override values for the following system definitions:

- The size of the address table (the number of addresses)
- The size of the alias table (the number of aliases)
- The number of receive transaction records
- The number of transmit transaction records
- The maximum lifetime of a transmit transaction

If you do not specify an override value, the LonTalk Interface Developer utility generates appropriate values based on other preferences that you specify for the project.



You can select the options to automatically calculate values to have the LonTalk Interface Developer utility calculate appropriate values for the stack configuration.

See *Managing Memory* for more information about these values.

Click **Next**.

Configuring the Buffers

From the Buffer Configuration page of the utility, you can specify the number for each of the following application buffer types:

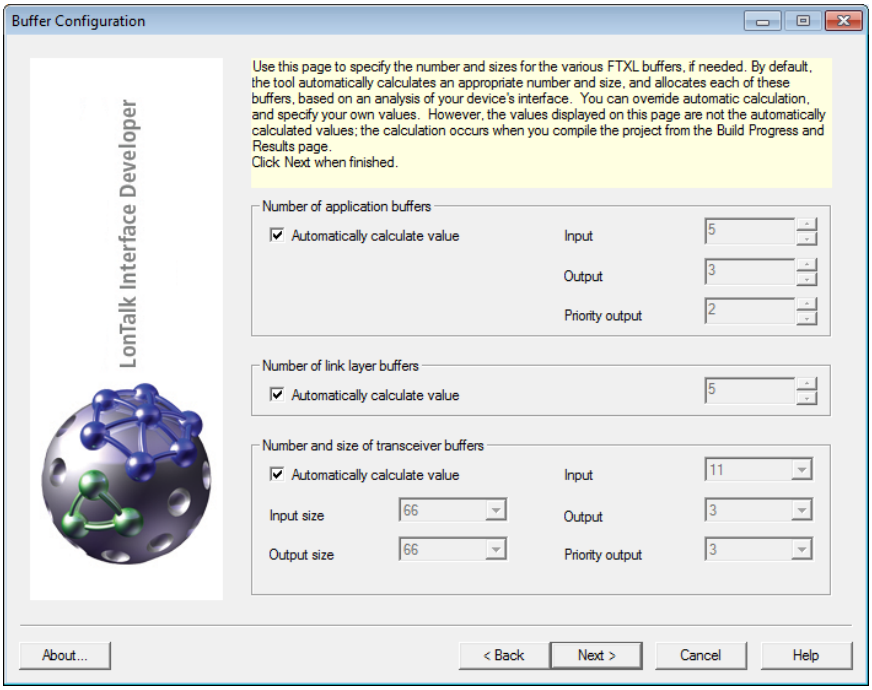
- Input buffers
- Non-priority output buffers
- Priority output buffers

You can also specify the number of link-layer buffers.

In addition, you can specify both the size and number for the transceiver buffers:

- Input buffers

- Non-priority output buffers
- Priority output buffers



You can select the options to automatically calculate values to have the LonTalk Interface Developer utility calculate appropriate values for the buffer configuration.

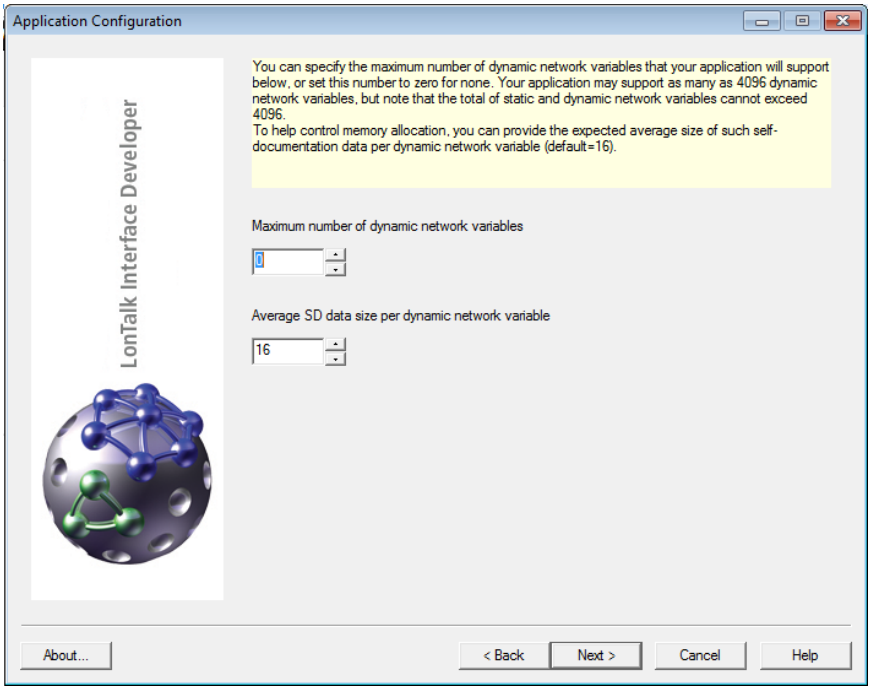
Click **Next**.

Configuring the Application

From the Application Configuration page of the utility, you can specify the following parameters for the application:

- The number of dynamic network variables
- The average amount of memory to reserve for self-documentation data for dynamic network variables

By default, the number of supported dynamic network variables is zero, but you can specify up to 4096. During compilation, the utility verifies that the sum of static and dynamic network variables does not exceed a total of 4096 for the device.



The average amount of memory to reserve for dynamic network variable self-documentation strings is used, along with the number of dynamic network variables, to calculate the maximum amount of non-volatile data that might be required for the LonTalk Stack device. The actual size of a particular dynamic variable's self-documentation string can exceed the specified average, as long as the actual average size is less than or equal to the specified average size.

The default size for the dynamic network variable self-documentation data is 16 bytes, but you can specify up to 128 bytes.

Click **Next**.

Configuring Support for Non-Volatile Data

From the Non-Volatile Data Support page of the utility, you can specify the following parameters for the application:

- Non-volatile data driver model
- Non-volatile data flush guard timeout value
- Name for the top-level root segment for the non-volatile data

The non-volatile data driver model can be one of the following types, depending on your application's requirements:

- Flash file system (such as Linux)
- Flash direct memory (with no file system) if you do not have, or do not want to use, a flash file system for your non-volatile data
- User defined if you have another non-volatile data support model that your application uses

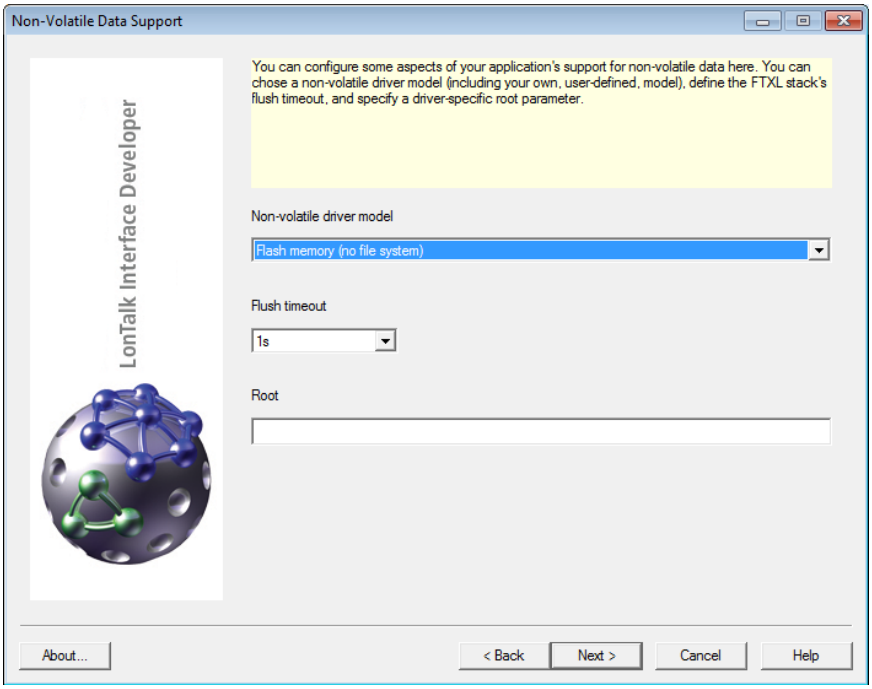
You can only select one driver model for the specified application.

The non-volatile data flush timeout value determines how long the LonTalk host stack waits to receive additional updates before writing them to the non-volatile data.

The non-volatile root name is used to configure the non-volatile data support driver. If you use the flash file system, the non-volatile root name is used as a file system directory name in which to create non-volatile data files. If you use the direct flash model, the name represents a host processor flash device name. If you use unstructured flash memory, leave the **Root** field blank.

Within the host processor development environment, the **system.h** file defines the root name. For the examples that are included with the LonTalk Stack, the root name is **/dev/cfi_flash**, which is the root directory for the flash file system.

The source files that handle non-volatile data (**FtxlNvdFlashDirect.c**, **FtxlNvdFlashFs.c**, and **FtxlNvdUserDefined.c**) use conditional compilation based on the selected model to include the appropriate code. If you select a user-defined model, the related callback handler functions are not defined and cause a linker error if they are not implemented.



Click **Next**.

Specifying the Device Program ID

From the Program ID Selection page of the utility, you specify the device program ID or use the LONMARK Standard Program ID Calculator to specify the device program ID. The program ID is a 16-digit hexadecimal number that uniquely identifies the device interface for your device.

The program ID can be formatted as a standard or non-standard program ID. When formatted as a standard program ID, the 16 hexadecimal digits are organized into six fields that identify the manufacturer, classification, usage, channel type, and model number of the device. The LONMARK Standard Program ID Calculator simplifies the selection of the appropriate values for these fields by

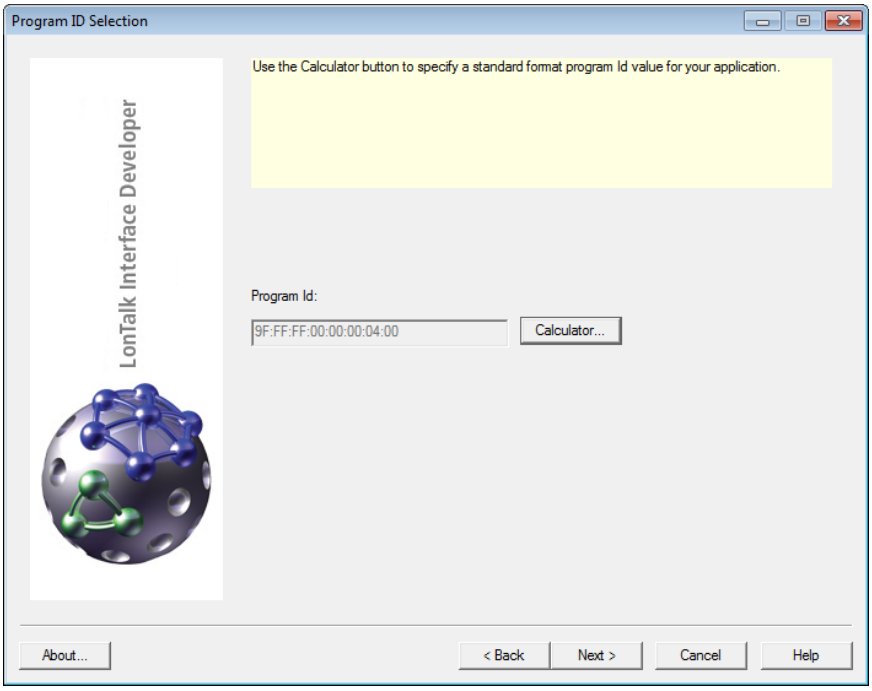
allowing you to select from lists contained in a program ID definition file distributed by LONMARK International. A current version of this list is included with the LonTalk Stack.

Within the device's program ID, you must include your manufacturer ID. If your company is a member of LONMARK International, you have a permanent Manufacturer ID assigned by LONMARK International. You can find those listed within the Standard Program ID Calculator utility, or online at www.lonmark.org/mid.

If your company is not a member of the LONMARK International, you can obtain a temporary manufacturer ID from www.lonmark.org/mid. You do not have to join LONMARK International to obtain a temporary manufacturer ID.

For prototypes and example applications, you can use the F:FF:FF manufacturer ID, but you should not release a device that uses this non-unique identifier into production.

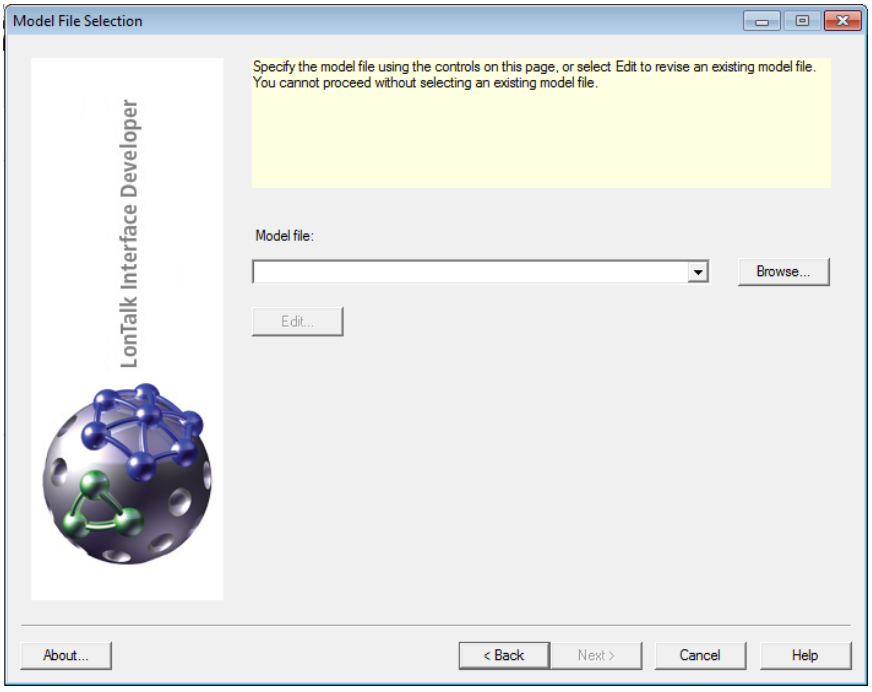
If you want to specify a program ID that does not follow the standard program ID format, you must use the command-line interface for the LonTalk Interface Developer utility. LONMARK International requires all interoperable LONWORKS devices to use a standard-format program ID. Using a non-standard format for the program ID will prevent the use of functional blocks and configuration properties, and will prevent certification.



Click **Next**.

Specifying the Model File

From the Model File Selection page of the utility, you specify the model file for the device. You can also click **Edit** to open the model file in whatever editor is associated with the .nc file type, for example, Notepad or the NodeBuilder Development Tool.



The model file is a simple source file written using a subset of the Neuron C Version 2.1 programming language. The model file contains declarations of network variables, configuration properties, functional blocks, and their relationships.

The LonTalk Interface Developer utility uses the information in the model file, combined with other user preferences, to generate the application framework files and the interface files. You must compile and link the application framework files with the host application.

See Chapter 6, *Creating a Model File*, for more information about the model file.

Click **Next**.

Specifying Neuron C Compiler Preferences

From the Neuron C Compiler Preferences page of the utility, you can specify macros for the Neuron C compiler preprocessor and extend the include search path for the compiler.

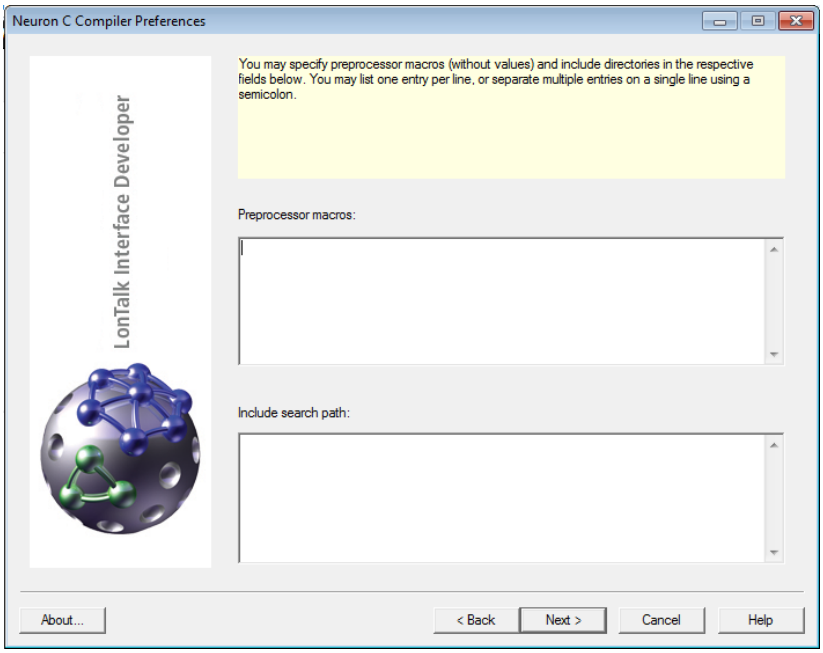
For the preprocessor macros (**#define** statements), you can only specify macros that do not require values. These macros are optional. Use separate lines to specify multiple macros.

The **_FTXL** macro is always predefined by the LonTalk Interface Developer utility, and does not need to be specified explicitly. You can use this macro to control conditional compilation for LonTalk Stack applications. In addition, the utility predefines the **_MODEL_FILE** macro for model file definitions and the **_LID3** macro for LonTalk Interface Developer utility macros.

For the search path, you can specify additional directories in which the compiler should search for user-defined include files (files specified within quotation marks, for example, **#include "my_header.h"**).

Specifying additional directories is optional. Use separate lines to specify multiple directories.

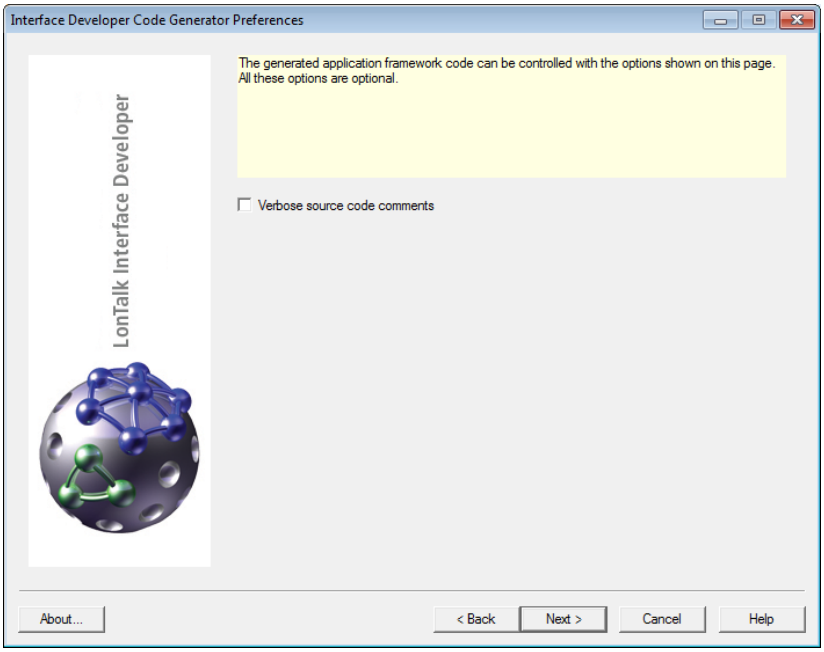
The LonTalk Interface Developer project directory is automatically included in the compiler search path, and does not need to be specified explicitly. Similarly, the Neuron C Compiler system directories (for header files specified with angled brackets, for example, **#include <string.h>**) are also automatically included in the compiler search path.



Click **Next**.

Specifying Code Generator Preferences

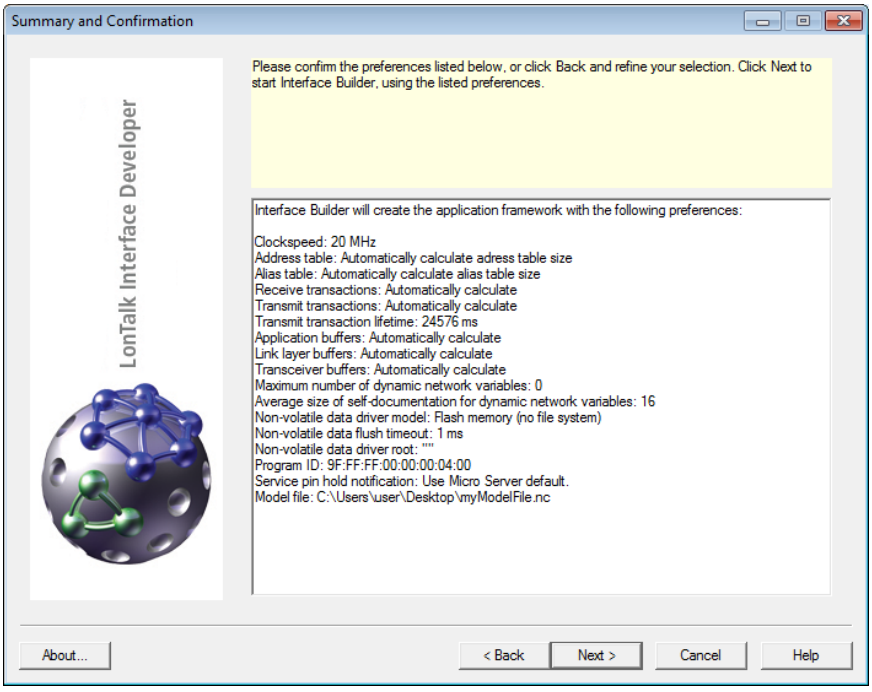
From the Interface Developer Code Generator Preferences page of the utility, you can specify preferences for the LonTalk Interface Developer compiler, such as whether to generate verbose source-code comments.



Click **Next**.

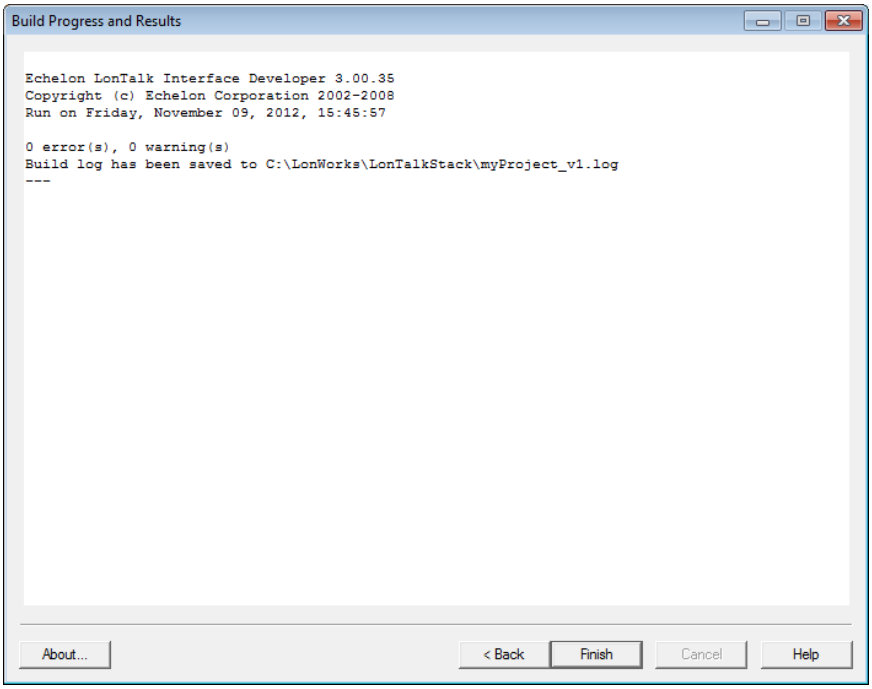
Compiling and Generating the Files

From the Summary and Confirmation page of the utility, you can view all of the information that you specified for the project.



When you click **Next**, the LonTalk Interface Developer utility compiles the model file and generates a number of C source files and header files, as described in *Using the LonTalk Interface Developer Files*.

The Build Progress and Summary page shows the results of compilation and generation of the LonTalk Stack project files.



Any warning or error messages have the following format:

Error-type: Model_file_name Line_number(Column_number): Message

Example: A model file named "tester.nc" includes the following single network variable declaration:

```
network input SNVT_volt nviVolt
```

Note the missing semi-colon at the end of the line. When you use this file to build a project from the LonTalk Interface Developer utility, the compiler issues the following message:

```
Error:   TESTER.NC    1( 32):
        Unexpected END-OF-FILE in source file [NCC#21]
```

The message type is error, the line number is 1, the column number is 32 (which corresponds to the position of the error, in this case, the missing semi-colon), and the compiler message number is NCC#21. To fix this error, add a semi-colon to the end of the line.

See the *Neuron Tools Errors Guide* for information about the compiler messages.

Using the LonTalk Interface Developer Files

The LonTalk Interface Developer utility takes all of the information that you provide and automatically generates the following files that are needed for your LonTalk Stack application:

- **LonNvTypes.h**
- **LonCpTypes.h**
- **FtxlDev.h**

- **FtxlDev.c**
- ***project*.xif**
- ***project*.xfb**

These files form the LonTalk Stack application framework, which defines the LonTalk Stack device initialization data and self-identification data for use in initialization phase, including communication parameters and everything you need to begin device development. The framework includes ANSI C type definitions for network variable and configuration property types used with the application, and implements them as global application variables.

To include these files in your application, include the **FtxlDev.h** file in your LonTalk Stack application using an ANSI C **#include** statement, and add the **FtxlDev.c** file to your project so that it can be compiled and linked.

The following sections describe the copied and generated files.

Copied Files

The LonTalk Interface Developer utility copies the following files into your project directory if no file with the same name already exists:

- **FtxlApi.h**
- **FtxlHandlers.c**
- **FtxlNvdFlashDirect.c**
- **FtxlNvdFlashFs.c**
- **FtxlNvdUserDefined.c**
- **FtxlTypes.h**
- **LonPlatform.h**

Existing files with the same name, even if they are not write-protected, are not overwritten by the utility.

Other than **FtxlDev.h**, you do not normally have to explicitly include any of the header files with your application source, because the **FtxlDev.h** file already includes the required files.

You must ensure that the files in the **Source** directory and the various LID-generated C files are available to your project so that they can be compiled and linked with your application.

LonNvTypes.h and LonCpTypes.h

The **LonNvTypes.h** file defines network variable types, and includes type definitions for standard or user network variable types (SNVTs or UNVTs). See *Using Types* for more information on the generated types.

The **LonCpTypes.h** file defines configuration property types, and includes standard or user configuration property types (SCPTs or UCPTs) for configuration properties implemented within configuration files.

Either of these files might be empty if your application does not use either network variables or configuration properties.

FtxlDev.h

The **FtxlDev.h** file is the main header file that the LonTalk Interface Developer utility produces. This file provides the definitions that are required for your application code to interface with the application framework and the LonTalk API, including C **extern** references to public functions, variables, and constants generated by the LonTalk Interface Developer utility.

You should include this file with all source files that your application uses, but you do not normally have to edit this file. Any manual changes to this file are not retained when you rerun the LonTalk Interface Developer utility. The file contains comments that describe how you can override some of the preferences and assumptions made by the utility.

FtxlDev.c

The **FtxlDev.c** file is the main source file that the LonTalk Interface Developer utility produces. This file includes the **FtxlDev.h** file header file, declares the network variables, configuration properties, and configuration files (where applicable).

It defines the device's **LonInit()** function. It also defines variables and constants, including the network variable table, the device's initialization data block, and a number of utility functions.

You must compile and link this file with your application, but you do not normally have to edit this file. Any manual changes to this file are not retained when you rerun the LonTalk Interface Developer utility, but the file contains comments that describe how you can override some of the preferences and assumptions made by the utility.

project.*xif and* project.*xfb*

The LonTalk Interface Developer utility generates the device interface file for your project in two formats:

- ***project*.xif** (a text file)
- ***project*.xfb** (a binary file)

For both files, *project* is the name of the LonTalk Stack project that you specified in the Welcome to LonTalk Interface Developer window of the LonTalk Interface Developer utility. Thus, these files have the same name as the LonTalk Stack project file (.lidprj extension).

These files comply with the LONMARK device interface revision 4.401 format.

If your device is defined with a non-standard program ID, the device interface file cannot contain interoperable LONMARK constructs such as functional blocks and configuration properties.

Using Types

The LonTalk Interface Developer utility produces type definitions for the network variables and configuration properties in your model file. For maximum portability, all types defined by the utility are based on a small set of host-side

equivalents to the built-in Neuron C types. For example, the **LonPlatform.h** file contains a type definition for a Neuron C signed integer equivalent type called **ncsInt**. This type must be the equivalent of a Neuron C signed integer, a signed 8-bit scalar. For most target platforms, the **ncsInt** type is defined as signed char type.

A network variable declared by a Neuron C built-in type does not require a host-side type definition in the **LonNvTypes.h** file, but is instead declared with its respective host-side Neuron C equivalent type as declared in **LonPlatform.h**.

Network variables that use ordinary C types, such as **int** or **long**, are not interoperable. For interoperability, network variables must use types defined within the device resource files. These network variable types include standard network variable types (SNVTs) and user-defined network variable types (UNVTs). You can use the Resource Editor tool to define your own UNVT.

Example:

A model file contains the following declarations:

```
network input  int            nviInteger;
network output SNVT_count     nvoCount;
network output SNVT_switch    nvoSwitch;
```

- The **nviInteger** declaration uses a built-in Neuron-C type, so the LonTalk Interface Developer utility uses the **ncsInt** type defined in **LonPlatform.h**.
- The **nvoCount** declaration uses a type that is not a built-in Neuron C type. The utility produces the following type definition:

  ```
  typedef ncuLong SNVT_count;
  ```

  The **ncuLong** type represents the host-side equivalent of a Neuron C unsigned long, a 16-bit unsigned scalar. It is defined in **LonPlatform.h**, and typically maps to the **LonWord** type. **LonWord** is a platform-independent definition of a 16-bit scalar in big-endian notation:

  ```
  typedef struct {
     LonByte msb;
     LonByte lsb;
  } LonWord;
  ```

  To use this platform-independent type for numeric operations, you can use the optional **LON_GET_UNSIGNED_WORD** or **LON_SET_UNSIGNED_WORD** macros. Similar macros are provided for signed words (16 bit), and for signed and unsigned 32-bit scalars (DOUBLE).

  If a network variable or configuration property is defined with an initializer in your device's model file, and if you change the default definition of multi-byte scalars (such as the **ncuLong** type), you must modify the initializer generated by the LonTalk Interface Developer utility if the type is a multi-byte scalar type.
- The **nvoSwitch** declaration is based on a structure. The LonTalk Interface Developer utility redefines this structure using built-in Neuron C equivalent types:

```
typedef LON_STRUCT_BEGIN(SNVT_switch){
   ncuInt   value;
   ncsInt   state;
} LON_STRUCT_END(SNVT_switch);
```

Type definitions for structures assume a padding of 0 (zero) bytes and a packing of 1 byte. The **LON_STRUCT_BEGIN** and **LON_STRUCT_END** macros enforce platform-specific byte packing and padding. These macros are defined in the **LonPlatform.h** file, which allows you to adjust them for your compiler.

Bit Field Members

For portability, none of the types that the LonTalk Interface Developer utility generates use bit fields. Instead, the utility defines bit fields with their enclosing bytes, and provides macros to extract or manipulate the bit field information.

By using macros to work directly with the bytes of the bit field, your code is portable to both big-endian and little-endian platforms (that is, platforms that represent the most-significant bit in the left-most position and platforms that represent the most-significant bit in the right-most position). The macros also reduce the need for anonymous bit fields to achieve the correct alignment and padding.

Example: The following macros and structure define a simple bit field of two flags, a 1-bit flag alpha and a 4-bit flag beta:

```
typedef LON_STRUCT_BEGIN(Example) {
   LonByte flags_1;     // contains alpha, beta
} LON_STRUCT_END(Example);

#define LON_ALPHA_MASK 0x80
#define LON_ALPHA_SHIFT 7
#define LON_ALPHA_FIELD flags_1
#define LON_BETA_MASK 0x70
#define LON_BETA_SHIFT 4
#define LON_BETA_FIELD flags_1
```

When your program refers to the **flags_1** structure member, it can use the bit mask macros (**LON_ALPHA_MASK** and **LON_BETA_MASK**), along with the bit shift values (**LON_ALPHA_SHIFT** and **LON_BETA_SHIFT**), to retrieve the two flag values. These macros are defined in the **LonNvTypes.h** file. The **LON_STRUCT_*** macros enforce platform-specific byte packing.

To read the alpha flag, use the following example assignment:

```
Example var;
alpha_flag = (var.LON_ALPHA_FIELD & var.LON_ALPHA_MASK) >>
                 var.LON_ALPHA_SHIFT;
```

You can also use the **LON_GET_ATTRIBUTE()** and **LON_SET_ATTRIBUTE()** macros to access flag values. For example, for a variable named *var*, you can use these macros to get or set the attributes:

```
alpha_flag = LON_GET_ATTRIBUTE(var, LON_ALPHA);
…
LON_SET_ATTRIBUTE(var, LON_ALPHA, alpha_flag);
```

These macros are defined in the **FtxlTypes.h** file.

Enumerations

The LonTalk Interface Developer utility does not produce enumerations. The LonTalk Stack requires an enumeration to be of size **byte**. The ANSI C standard requires that an enumeration be an **int**, which is larger than one byte for many platforms.

The LonTalk Stack enumeration uses the **LON_ENUM_BEGIN** and **LON_ENUM_END** macros. For many compilers, these macros can be defined to generate native enumerations:

```
#define LON_ENUM_BEGIN(name)  enum
#define LON_ENUM_END(name)    name
```

Some compilers support a colon notation to define the enumeration's underlying type:

```
#define LON_ENUM_BEGIN(name) enum : signed char
#define LON_ENUM_END(name)
```

When your program refers to an enumerated type in a structure or union, it should not use the enumeration's name, but should use the **LON_ENUM_*** macros.

For those compilers that support byte-sized enumerations, it can be defined as:

```
#define LON_ENUM(name) name
```

For other compilers, it can be defined as:

```
#define LON_ENUM(name) signed char
```

The following table shows an example enumeration using the LonTalk Stack **LON_ENUM_*** macros, and the equivalent ANSI C enumeration.

LonTalk Stack Enumeration	Equivalent ANSI C Enumeration

`LON_ENUM_BEGIN(Color) {`
`  red, green, blue`
`} LON_ENUM_END(Color);`
 	
`typedef struct {`
`  …`
`  LON_ENUM(Color) color;`
`} Example;`
	`enum {`
	`  red, green, blue`
	`} Color;`
	
	`typedef struct {`
	`  …`
	`  Color color;`
	`} Example;`

Floating Point Variables

Floating point variables receive special processing, because the Neuron C compiler does not have built-in support for floating point types. Instead, it offers an implementation for floating point arithmetic using a set of floating-point support functions operating on a **float_type** type. The LonTalk Interface Developer utility represents this type as a **float_type** structure, just like any other structured type.

This floating-point format can represent numbers with the following characteristics:

- $\pm 1 * 10^{1038}$ approximate maximum value

- $\pm 1 * 10^{-7}$ approximate relative resolution

The **float_type** structure declaration represents a floating-point number in IEEE 754 single-precision format. This format has one sign bit, eight exponent bits, and 23 mantissa bits; the data is stored in big-endian order. The **float_type** type is identical to the type used to represent floating-point network variables.

For example, the LonTalk Interface Developer utility generates the following definitions for the floating point type **SNVT_volt_f**:

```
/*
 *  Type: SNVT_volt_f
 */
typedef LON_STRUCT_BEGIN(SNVT_volt_f)
{
  LonByte  Flags_1;   /* Use bit field macros, defined
                         below */
  LonByte  Flags_2;   /* Use bit field macros, defined
                         below */
  ncuLong  LS_mantissa;
} LON_STRUCT_END(SNVT_volt_f);


/*
 *  Macros to access the sign bit field contained in
 *  Flags_1
 */
#define LON_SIGN_MASK  0x80
#define LON_SIGN_SHIFT 7
#define LON_SIGN_FIELD Flags_1

/*
 *  Macros to access the MS_exponent bit field contained in
 *  Flags_1
 */
#define LON_MSEXPONENT_MASK  0x7F
#define LON_MSEXPONENT_SHIFT 0
#define LON_MSEXPONENT_FIELD Flags_1

/*
 *  Macros to access the LS_exponent bit field contained in
 *  Flags_2
 */
#define LON_LSEXPONENT_MASK  0x80
#define LON_LSEXPONENT_SHIFT 7
#define LON_LSEXPONENT_FIELD Flags_2

/*
 *  Macros to access the MS_mantissa bit field contained in
 *  Flags_2
 */
#define LON_MSMANTISSA_MASK  0x7F
#define LON_MSMANTISSA_SHIFT 0
#define LON_MSMANTISSA_FIELD Flags_2
```

See the IEEE Standard for Binary Floating-Point Arithmetic (ANSI/IEEE Std 754-1985) documentation for more information.

Network Variable and Configuration Property Declarations

The LonTalk Interface Developer utility generates network variables and configuration properties using the built-in types defined in **LonPlatform.h** along with the types defined in **LonNvTypes.h** and **LonCpTypes.h**. Both network variables and configuration properties are declared in the **FtxlDev.c** file, where input network variables (including configuration network variables) appear as volatile variables of the relevant type, and configuration properties that are not implemented with network variables appear as members of configuration files.

Example:

A model file contains the following Neuron C declarations:

```
SCPTlocation cp_family cpLocation;

network input SNVT_obj_request nviNodeRequest;
network output polled SNVT_obj_status nvoNodeStatus;
const network output polled SNVT_address nvoFileDir;

fblock SFPTnodeObject {
   nviNodeRequest implements nviRequest;
   nvoNodeStatus  implements nvoStatus;
   nvoFileDir     implements nvoFileDirectory;
} NodeObject external_name("NodeObject") fb_properties {
   cpLocation
};
```

The LonTalk Interface Developer utility generates the following variables in the **FtxlDev.c** file for the **nviNodeRequest**, **nvoNodeStatus**, and **nvoFileDir** network variables:

```
volatile SNVT_obj_request nviNodeRequest;
SNVT_obj_status nvoNodeStatus;
SNVT_address nvoFileDir = {
    LON_DMF_WINDOW_START/256u, LON_DMF_WINDOW_START%256u
};
```

The LonTalk API, upon receipt of an incoming network variable update, automatically moves data into the corresponding input network variable and signals this event by calling an event handler function, which allows your application to respond to the arrival of new network variable data. Your application then reads the input variable to obtain the latest value.

To send an update to the **nvoNodeStatus** output network variable, your application writes the new value to the **nvoNodeStatus** variable, and then calls the **LonPropagateNv()** function to propagate the new value onto the network.

See Chapter 8, *Developing a LonTalk Stack Device Application,* for information about the development of a LonTalk Stack application using the LonTalk Interface Developer utility-generated code.

The utility generates the following configuration file in **FtxlDev.c** for the **cpLocation** configuration property:

```
/*
 *
 *  Writable configuration parameter value file
```

```
 */
volatile LonWriteableValueFile lonWriteableValueFile = {
  {{'\x0', '\x0', '\x0', '\x0', '\x0', '\x0', '\x0',
   '\x0', '\x0', '\x0', '\x0', '\x0', '\x0', '\x0', '\x0',
   '\x0', '\x0', '\x0', '\x0', '\x0', '\x0', '\x0', '\x0',
   '\x0', '\x0', '\x0', '\x0', '\x0', '\x0', '\x0', '\x0'}}
};

/*
 *  CP template file
 */
const char lonTemplateFile[] = \
    "1.1;" \
    "1,0,0\x80,17,31;";


#ifndef LON_FILEDIR_USER_DEFINED
/*
 * Variable: File Directory
 */

const LonFileDirectory lonFileDirectory =
{
  LON_FILE_DIRECTORY_VERSION,
  LON_FILE_COUNT,
  {
      LON_REGISTER_FILE("template",
        sizeof(lonTemplateFile), LonTemplateFileType,
        LON_DMF_WINDOW_START+sizeof(lonFileDirectory)),
      LON_REGISTER_FILE("rwValues",
        sizeof(lonWriteableValueFile), LonValueFileType,
        LON_DMF_WINDOW_START+sizeof(lonFileDirectory)
        +sizeof(lonTemplateFile)),
      LON_REGISTER_FILE("roValues", 0, LonValueFileType,
        0)
    }
};
#endif /* LON_FILEDIR_USER_DEFINED */
```

The **LonWriteableValueFile** data structure is defined in the **FtxlDev.h** header file:

```
typedef LON_STRUCT_BEGIN(LonWriteableValueFile)
{
  SCPTlocation cpLocation_1;
  /* sd_string("1,0,0\x80,17,31;") */
} LON_STRUCT_END(LonWriteableValueFile);

extern volatile LonWriteableValueFile
   lonWriteableValueFile;
```

Similarly, a **LonReadOnlyValueFile** type is defined and used to declare a **lonReadOnlyValueFile** variable if the model file declares read-only configuration properties.

The LonTalk Interface Developer utility generates resource definitions for configuration properties and network variables defined with the **eeprom** keyword. Your application must provide sufficient persistent storage for these

resources. You can use any type of non-volatile memory, or any other media for persistent data storage. The template file and the read-only value file would normally be declared as **const**, and can be linked into a code segment, which might relate to non-modifiable memory such as PROM or EPROM (these files must not be changed at runtime). However, writable, non-volatile storage must be implemented for the writable configuration property value file.

The details of such persistent storage are subject to the host platform requirements and capabilities; persistent storage options include: flash memory, EEPROM memory, non-volatile RAM, or storage in a file or database on a hard drive.

You can specify initializers for network variables or configuration properties in the model file. Alternatively, you can specify initializers for configuration properties in the resource file that defines the configuration property type or functional profile. For network variables without explicit initialization, the rules imposed by your host development environment apply. These values might have random content, or might automatically be preset to a well-defined value.

Constant Configuration Properties

In general, a configuration property can be modifiable, either from within the LonTalk Stack application or from a network management tool. However, the LonTalk Interface Developer utility declares constant configuration property files as constants (using the C **const** keyword), so that they are allocated in non-modifiable memory.

A special class of configuration properties is the *device-specific* configuration property. A device-specific configuration property is considered variable to the application (that is, your application can change it), but constant to the external interface. These properties might, for example, be used to store calibration data that is gathered during the device's auto-tuning procedure.

However, a paradox arises because the network manager expects this configuration property within the read-only value file, but the read-only value file must be writable from the local application. This paradox is known as the writeable read-only value file.

The LonTalk Stack presents the following solution to resolve this paradox:

- Before the inclusion of the **FtxlDev.h** header file into the **FtxlDev.c** file, you can define the **LON_READONLY_FILE_IS_WRITEABLE** macro to a value of 1 (one). If you do not define this macro, or define it to equate to zero, the read-only value file is constant. This is the default state. The **LON_READONLY_FILE_IS_WRITEABLE** macro is used within the **FtxlDev.h** header to define the read-only file's storage type with the **LON_READONLY_FILE_STORAGE_TYPE** macro, which in turn is used in declaration and specification of the **lonReadOnlyValueFile** variable.
- Defining the **LON_READONLY_FILE_IS_WRITEABLE** macro to 1 causes the read-only value file to be writeable by the local application. Because it is now allocated in volatile memory, your driver for non-volatile data must also be able to read and write the read-only value file.

For the network management tool, however, the read-only file remains non-writeable. If your application uses the direct memory files feature to access the files, the LonTalk Interface Developer utility generates code that declares this direct memory files window segment as non-modifiable. If your application uses LONWORKS FTP to access the files, your implementation of the LONWORKS file transfer protocol and server must prevent write operations to the read-only value file under all circumstances.

The Network Variable Table

The network variable table lists all the network variables that are defined by your application. It contains a pointer to each network variable and the initial (or declared) length of each network variable, in bytes. It also contains an attribute byte that contains flags which define the characteristics of each network variable.

The network variable table acts as a bridge between your application and the LonTalk API. The LonTalk Interface Developer utility generates the network variable table, along with the **LonInit()** function that reads the table and register the network variables with the LonTalk API.

A LonTalk Stack application typically accesses a network variable value through the C global variable that implements the network variable. However, the LonTalk API also provides a function that returns the pointer to a network variable's value as a function of its index:

```
void* const LonGetNvValue(unsigned index);
```

You can use this function for any network variable, including static network variables, dynamic network variables, and configuration property network variables. The **LonGetNvValue()** function returns NULL for an invalid index, or returns a pointer to the value.

For dynamic network variables, you must use the **LonGetNvValue()** function because there is no global C variable or network variable table entry for a dynamic network variable.

Network Variable Attributes

The network variable table (**nvTable[]**)in the **FtxlDev.c** file includes a bitmask for each network variable to define the network variable's attributes, including, for example, whether the network variable is:

- An output network variable
- Persistent
- Polled
- Synchronous
- Of changeable type

The **FtxlTypes.h** file defines the bitmasks for these attributes. For example, **LON_NV_IS_OUTPUT** is the mask for an output network variable, **LON_NV_POLLED** is the mask for a polled network variable, and so on.

The LonTalk API does not propagate a polled output network variable's value to the network when your application calls the **LonPropagateNv()** function. For

input network variables, the **polled** attribute changes the behavior of the network management tool's binder, which determines how a network variable connection is managed.

See *Developing a LonTalk Stack Device Application* for more information about propagation of network variable updates.

The Message Tag Table

Although the LonTalk Host stack does not use the message tag table, the LonTalk Interface Developer utility declares the message tag table in **FtxlDev.c** if you declare one or more message tags in the model file.

The message tag table lists all the message tags that are defined by your application. It contains a flag for each message tag which indicates that the message tag is not associated with an address table entry and therefore can only be used for sending explicitly addressed application messages. This flag is set for all message tags declared with the **bind_info(nonbind)** modifier in the model file.

See *Communicating with Other Devices Using Application Messages* for more information about using message tags.

8

Developing a LonTalk Stack Device Application

This chapter describes how to develop a LonTalk Stack device application. It also describes the various tasks performed by the application.

Overview of a LonTalk Stack Device Application

This chapter describes how to use the LonTalk API and the application framework produced by the LonTalk Interface Developer utility to perform the following tasks:

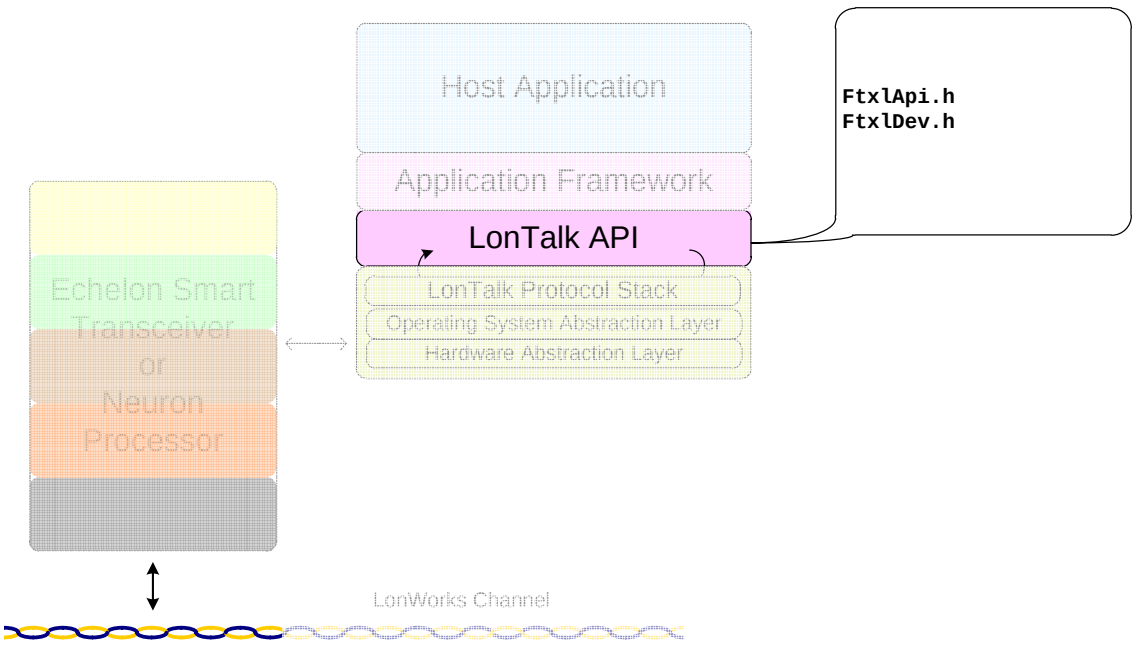
- Use the LonTalk API and LonTalk Host stack
- Integrate the application with an operating system
- Provide persistent storage for non-volatile data
- Initialize the LonTalk Stack device
- Periodically call the LonTalk Stack event pump
- Send information to other devices using network variables
- Receive information from other devices using network variables
- Handle network variable poll requests from other devices
- Handle updates to changeable-type network variables
- Handle dynamic network variables
- Communicate with other devices using application messages
- Handle management tasks and events
- Handle local network management commands
- Handle reset events
- Query the error log
- Use the direct memory files feature
- Shut down the LonTalk Stack device

Most LonTalk Stack applications need to perform only the tasks that relate to persistent storage, initialization, calling the event pump, and sending and receiving network variables.

This chapter shows you the basic control flow for each of the above tasks. It also provides a simple code example to illustrate some of the basic tasks.

Using the LonTalk API

Within the seven-layer OSI Model protocol, the LonTalk API forms the majority of Layer 6 (the Presentation layer), and provides the interface between the LonTalk host stack in Layer 5 (the Session layer) and the host application in Layer 7 (the Application layer), as shown in the following figure.



The [*LonTalkStack*]**\Source** folder contains the LonTalk Host stack, the LonTalk API, and the serial interface driver example, which together allow your LonTalk Stack application to handle network events, propagate network variables, respond to network variable poll requests, and so on.

A LonTalk Stack application must include the **FtxlDev.h** file to be able to use the LonTalk API. This file is generated by the LonTalk Interface Developer utility, and is located in your application project directory. The **FtxlDev.h** file includes the [*LonTalkStack*]**\Source\FtxlApi.h** file, which contains definitions for accessing the LonTalk API.

The [*LonTalkStack*]**\Source\FtxlHandlers.c** source file contains stubs for the event handler functions and callback handler functions that the LonTalk API calls. You must add code to these stubs to respond to specific events. For example, the **LonNvUpdateOccurred()** event handler function could inform the application of the arrival of new data for a set-point value, and the related code could re-calculate the device's response, assign output values to peripheral I/O devices, update the appropriate network variables, and propagate the changes to the network.

The following recommendations can help you manage your LonTalk Stack application project:

- Keep edits to LonTalk Interface Developer utility-generated files to a minimum, that is, do not edit the **LonNvTypes.h**, **LonCpTypes.h**, **FtxlDev.h** or **FtxlDev.c** files unless necessary
- Add **#include "FtxlDev.h"** to your application source files to provide access to network variable types and instantiations and the LonTalk API
- Keep changes to the **FtxlHandlers.c** file to a minimum
    - Add calls to your own functions in files that you create and maintain

- Future versions or fixes to the LonTalk Stack might affect these API files

Callbacks and Events

The LonTalk API uses two types of notifications for occurrences within the system: callbacks and events.

The LonTalk API uses a *callback* when the API needs a return value from the application immediately. A callback can occur in one of the LonTalk host stack contexts (tasks or threads).

When you implement a callback handler function to process a callback, you must ensure that the function completes its work as quickly as possible. Generally, a callback handler function must not call LonTalk API functions or perform time-intensive operations.

The LonTalk API uses an *event* to deliver a one-way notification to the application. The protocol stack does not wait for the processing of the event to complete before continuing.

The LonTalk host stack holds events in an internal queue for processing. Thus, the application program must periodically call the **LonEventPump()** function to process the event queue. This function also calls the related event handler functions.

Because event processing in the event handler functions is not tied to the context of the protocol stack, an event handler function can call LonTalk API functions or perform time-intensive operations. An event handler function runs within the same context (task or thread) as its caller (the **LonEventPump()** function).

See Appendix D, *LonTalk API*, for a list of the callback handler functions and event handler functions.

Integrating the Application with an Operating System

The LonTalk host stack requires a LonTalk Stack application to use an operating system or include code that implements key operating system services. The LonTalk Host stack does not require the operating system to be a real-time operating system.

To allow the LonTalk Host stack to use any operating system, the LonTalk Host stack library is linked with the Operating System Abstraction Layer (OSAL) files, **Osal.h** and **Osal.c** (Windows) or **PosixOsal.c** (Linux). The OSAL files provide macros and C functions for general operating system functions, such as creating semaphores and waiting for events. The OSAL functions also include error handling and basic debug tracing for the operating system functions.

Your LonTalk Stack application can call the OSAL functions when it needs to call operating system functions, or it can call the operating system functions directly. By calling OSAL functions, your LonTalk Stack application can be more easily ported to another operating system, if needed.

The OSAL function prototypes are generic, and do not depend on the operating system's syntax. For example, to create a binary semaphore, your application can call the **OsalCreateBinarySemaphore()** function, which in turn calls the operating system's function to create the semaphore. The OSAL function assigns

a pointer to the created semaphore and returns a status variable that indicates whether the function was successful.

The LonTalk Stack includes source code for OSAL files that interface with Windows and Linux. To use a different operating system or provide your own operating system services, you must modify the OSAL files to implement the API for that operating system.

For more information about the OSAL functions, see *The Operating System Abstraction Layer*. For information about configuring the operating system, see *Configuring the Operating System*.

Providing Persistent Storage for Non-Volatile Data

The LonTalk host stack provides an API for managing non-volatile data (NVD). Because non-volatile data is stored and managed by the host processor rather than the Echelon Smart Transceiver, the LonTalk Stack application must implement the API's functions so that both the LonTalk host stack and the application can read and write NVD to non-volatile memory (typically, flash memory). Two example implementations, one using a flash file system, and one using raw flash access (through the HAL flash access routines) are provided in the **FtxlNvdFlashDirect.c** and **FtxlNvdFlashFs.c** files.

The implementations of the NVD-management functions are contained in one of the following files (all of which are copied to the project directory by the LonTalk Interface Developer utility):

- **FtxlNvdFlashDirect.c** for direct-access flash memory management
- **FtxlNvdFlashFs.c** for file-system flash memory management
- **FtxlNvdUserDefined.c** for your own flash memory management

Typically, if you select either the direct flash model or the flash file system model, you need only specify the appropriate value for the non-volatile root in the LonTalk Interface Developer Utility. This section describes how the LonTalk API uses the non-volatile memory driver, in case you need to implement your own user-defined non-volatile data driver or modify one of the provided drivers.

Non-volatile data is stored in segments. Two of the segments are used to store data maintained by the LonTalk host stack, and the third segment is used to store data maintained by the application. Examples of data maintained by the LonTalk host stack include network variable configuration and address tables. Examples of data maintained by the application include configuration network variable values and persistent memory files (used for configuration property value files and user files). Each data segment is identified by an enumeration of type **LonNvdSegmentType**, defined in the **FtxlTypes.h** file.

The LonTalk host stack reads non-volatile data (loads it into RAM) only during device initialization. Included with the data is a header that the LonTalk host stack uses for validation. Within this header is an application identifier, generated by the LonTalk Interface Developer utility, that allows the LonTalk host stack to ensure that the data belongs to the current application. The header also includes a checksum to ensure that the data is free of errors. If any of these validations fails, the LonTalk host stack deletes all non-volatile data in the segment and sets the device to the unconfigured state.

When data that must be stored persistently is updated in RAM, the LonTalk host stack does not immediately update the corresponding persistent memory. Instead, the LonTalk host stack defers writing the data to persistent memory so that it can continue to respond to network management commands in a timely fashion. The reasons for deferred writing of persistent memory include:

- Flash sectors sizes tend to be large and can take a long time to write.
- Each network management update generally affects only a small amount of data, and typically, a single logical operation consists of many messages (commissioning of the device generally being the most common and most extensive).
- The LonTalk host stack supports large configurations.

If the LonTalk host stack has not received any updates to a particular segment for a short (configurable) time (for example, 1 second), it uses the application callback handler functions to write the data to persistent memory. If the LonTalk host stack is shut down by calling the **LonExit()** function, the LonTalk host stack completes the write process before returning from the function. However, a sudden power outage or an unexpected CPU reset can prevent an orderly shutdown. The LonTalk host stack maintains a set of flags (one for each segment) that survive an unorderly shutdown so that the LonTalk host stack can detect the unorderly shutdown at the next restart.

The LonTalk host stack checks the flag, by calling the **LonNvdIsInTransaction()** callback handler function, during device startup before it reads the non-volatile data. If the flag is set, integrity of the non-volatile data has been compromised. Even if the configuration is internally consistent, the LonTalk Stack device has likely lost updates from a network manager that it has already acknowledged. If the LonTalk Stack device reverted to the last known configuration, this inconsistency would likely be undetected and could result in errors that are difficult to isolate. Instead, the LonTalk host stack deletes the configuration data, logs a configuration checksum error, and goes unconfigured. You can restore the configuration by recommissioning the device from network management tool.

If you use either of the standard non-volatile drivers, you can enable tracing by setting the global variable **nvdTraceEnabled** to a non-zero value. If you create your own custom non-volatile data driver, be sure to add some tracing capability to it.

Restoring Non-Volatile Data

During device startup, the LonTalk host stack reads the non-volatile data for each segment and initializes the corresponding data structures stored in RAM by performing the following steps:

1. Calling the **LonNvdIsInTransaction()** callback handler function. The application returns whether an NVD transaction for this segment was in progress when the LonTalk host stack was stopped. Typically, this function returns **FALSE**, but if the device was reset while a transaction was in progress, this function returns **TRUE** and the non-volatile data segment is considered corrupt, so the restore fails.
2. Calling the **LonNvdOpenForRead()** callback handler function to open the segment that corresponds to the specified type.

3. Calling the **LonNvdRead()** callback handler function to read the header of the NVD image. This function verifies the header and, if it is valid, uses the size information in the header to allocate the appropriate buffers.
4. Calling the **LonNvdRead()** callback handler function again (perhaps many times) to read the entire configuration and de-serialize the image.
5. Deserializing the image and updating the LonTalk host stack's control structures.
6. Calling the **LonNvdClose()** callback handler function to close the file.

If, at any time during this process any error occurs, the LonTalk host stack sets the device to the unconfigured state, generates a configuration checksum error, and calls the **LonNvdDelete()** callback handler function.

The LonTalk host stack handles the deserialization of the data for the **LonNvdSegNetworkImage** and **LonNvdSegNodeDefinition** segments, but not for the application-defined **LonNvdSegApplicationData** segment. Instead, the LonTalk host stack calls the **LonNvdDeserializeSegment()** callback handler function during step 5 above when it processes the **LonNvdSegApplicationData** segment. The **LonNvdDeserializeSegment()** callback handler function is generated by the LonTalk Interface Developer utility.

Writing Non-Volatile Data

When the LonTalk host stack processes a network management message that affects any of its configuration data, the LonTalk host stack checks whether there is an NVD transaction for the affected segment. If not, LonTalk host stack starts a timer and calls the **LonNvdEnterTransaction()** callback handler function for the segment. If there is already a transaction pending, the LonTalk host stack simply resets the timer.

When the timer expires, the LonTalk host stack writes the data to persistent memory by performing the following steps:

1. Determining the size of the serialized image.
2. Allocating a buffer large enough to hold the serialized image.
3. Serializing the data.
4. Calling the **LonNvdOpenForWrite()** callback handler function to open the segment with write access. If the segment does not already exist, this function must create it. If the segment exists, but is the wrong size, the application might need to delete it before writing to it.
5. Calling the **LonNvdWrite()** callback handler function one or more times to write the image.
6. Calling the **LonNvdClose()** callback handler function to close the file.
7. Calling the **LonNvdExitTransaction()** callback handler function to clear the transaction.
8. Freeing the buffer that contains the serialized image.

The LonTalk host stack determines the size of the serialized image and handles the serialization of the data for the **LonNvdSegNetworkImage** and

LonNvdSegNodeDefinition segments, but not for the application-defined **LonNvdSegApplicationData** segment. Instead, the LonTalk host stack calls the **LonNvdGetApplicationSegmentSize()** callback handler function in step 1 above, and the **LonNvdSerializeSegment()** callback handler function during step 3 above when it processes the **LonNvdSegApplicationData** segment. Both of these callback handler functions are generated by the LonTalk Interface Developer utility.

The LonTalk host stack uses a low-priority operating system task or thread (typically lower than the application task) to write NVD to persistent memory. By using a low-priority task or thread, writing NVD should not block the running of the application or the LonTalk host stack. In addition, LonTalk host stack ensures that these NVD-management functions are never called by more than one task or thread at a time.

The application can update configuration network variables (CPNVs) and user files directly, without the LonTalk host stack's knowledge. The application must inform the LonTalk host stack when this occurs so that the LonTalk host stack can manage the write transaction. Thus, the application should call the **LonNvdAppSegHasBeenUpdated()** function to initiate an NVD transaction for the application segment.

Tasks Performed by a LonTalk Stack Application

The **main()** function of a LonTalk Stack application typically performs only the following actions:

1. Creates one or more operating system contexts (tasks or threads)
2. Starts the operating system (if it is not already started)

Within one of the newly created tasks, the application life cycle includes two phases:

- Initialization
- Normal processing

The initialization phase of a LonTalk Stack application includes a call to the **LonInit()** API function to initialize the LonTalk host stack and the Echelon Smart Transceiver or Neuron Chip. The initialization phase defines basic parameters for LONWORKS network communication, such as the communication parameters for the physical transceiver in use, and defines the application's external interface: its network variables, configuration properties, and self-documentation data. Successful completion of the initialization phase causes the Echelon Smart Transceiver or Neuron Chip to leave Quiet mode, after which it can send and receive messages over the network. During the initialization phase, the application also creates at least one operating system event (or other protected shared resource).

During normal processing, which is often implemented within an infinite loop, the application waits for an operating system event whenever it is not busy. When the event occurs, the application calls the **LonEventPump()** API function to process LonTalk Stack events. This function then calls event handler functions (such as **LonNvUpdateOccurred()** or **LonNvUpdateCompleted()**).

The following sections describe the tasks that a LonTalk Stack application performs during its life cycle.

Initializing the LonTalk Stack Device

Before your application initializes the LonTalk host stack, it must initialize the C runtime environment and the operating system.

If your LonTalk Stack device uses a native LonTalk interface, your application must implement the **LonGetMyNetworkInterface()** function in the **FtxlHandlers.c** file to specify the name of the network interface to be used by the driver. If your LonTalk Stack device uses an IP-852 interface, your application must implement the **LonGetMyIpAddress()** fucntion in the **FtxlHandlers.c** file to return the IP address and port to be used by the IP-852 interface.

Your application must call the **LonInit()** function once during device startup. The implementation of this function is generated by the LonTalk Interface Developer utility, and is included in the **FtxlDev.c** file. This function initializes the LonTalk API, the LonTalk host stack, and the Echelon Smart Transceiver or Neuron Chip. The main application thread must call this function before it calls any other LonTalk API functions.

LonInit() registers the LonTalk Stack device interface data with the LonTalk host stack. This data defines the network parameters and device interface. If your application needs to change the network parameters or change the device interface, it can call the **LonExit()** function to shut down the LonTalk host stack, and then call the **LonInit()** function to restart the LonTalk host stack with the updated interface.

Add a call the **LonInit()** function to the beginning of the application's main thread. If this function is successful, your application can begin normal operations, including calling the event pump, as described in *Periodically Calling the Event Pump*.

Example:

```
void myMainThread(void) {
   LonApiError sts;
   sts = LonInit();
   if (sts == LonApiNoError) {
      // begin normal operations
   }
}
```

Periodically Calling the Event Pump

As described in *Callbacks and Events*, your LonTalk Stack application must periodically call the **LonEventPump()** function to check if there are any LONWORKS events to process. This function calls specific API functions based on the type of event, then calls event handler functions to notify the application layer of these network events. You can call this function from the idle loop within the main application thread or from any point in your application that is processed periodically. However, you must call this function from the same application context (task or thread) that called the **LonInit()** function.

The LonTalk API calls the **LonEventReady()** callback handler function whenever an event has been posted. This function is typically called from a LonTalk host stack task or thread, and you must not call the **LonEventPump()** function directly from the callback. However, your application could define an

operating system event which is signaled by the **LonEventReady()** callback handler function. From within your application's main thread, the application should implement an infinite loop that waits on this operating system event. Whenever the event is signaled, the application should call the **LonEventPump()** API function to process LonTalk Stack events.

You can signal this same operating system event to schedule your main application thread to perform other functions as well. For example, you could signal the operating system event from within an interrupt handler to signal the main application task to process application I/O. Calling the **LonEventPump()** function when there are no LonTalk Stack events is acceptable.

The host application should be prepared to process the maximum rate of LONWORKS traffic delivered to the device. Although events are enqueued within the LonTalk host stack, your application should call the **LonEventPump()** function frequently to process events. Use the following formula to determine the minimum call rate for the **LonEventPump()** function:

$$rate = \frac{MaxPacketRate}{InputBufferCount - 1}$$

where *MaxPacketRate* is the maximum number of packets per second arriving for this device, and *InputBufferCount* is the number of input buffers defined for your application (that is, buffers that hold incoming data until your application is ready to process it). The formula subtracts one from the number of available buffers to allow new data to arrive while other data is being processed. However, the formula also assumes that your application has more than one input buffer; having only one input buffer is generally not recommended.

If the application expects periods of inactivity, it can simply wait for the LonTalk host stack to post an event. If the application expects periods where it is busy for several milliseconds at a time, it should call the **LonEventPump()** function during the busy time to ensure that events are processed. Use the formula above to determine a baseline for how often to call the **LonEventPump()** function.

If you do not have measured data for your typical network and you are developing a device for the TP/FT-10 channel, assume 90 packets per second arriving for the device. This packet rate meets the TP/FT-10 channel's throughput figures, assuming that most traffic uses acknowledged or request/response service. Use of other service types will increase the required packet rate, but not every packet on the network is necessarily addressed to this device.

Using the formula, devices that implement two input buffers and are attached to a TP/FT-10 network that expect high throughput should call the **LonEventPump()** function approximately once every 10 ms.

When an event occurs, the **LonEventPump()** function calls the appropriate event function for your host application to handle the event. Your event handler functions must be designed for this minimum call rate, and should defer time-consuming operations (such as lengthy flash writes) whenever possible, or manage them in separate contexts (tasks or threads).

See Appendix D, *LonTalk API*, for a list of the available event handler and callback handler functions.

Example:

```
while (1) {
   // process application-specific data
   ...
   if (OsalWaitForEvent(readyHandle, OSAL_WAIT_FOREVER) ==
         OSALSTS_SUCCESS)
      LonEventPump();
}

...

void LonEventReady(void)
{
    OsalSetEvent(readyHandle);
}
```

In the example, the **readyHandle** variable is the handle to an OSAL event; this handle is defined using the **OsalCreateEvent()** function during the application's initialization phase, and is signaled by the **LonEventReady()** callback handler function whenever an event is ready to be processed.

Sending a Network Variable Update

Your LonTalk Stack device typically communicates with other LONWORKS devices by sending and receiving network variables. Each static network variable is represented by a global variable declared by the LonTalk Interface Developer utility in the **FtxlDev.c** file, with **extern** declarations provided in the **FtxlDev.h** file. To send an update for a static output network variable, first write the new value to the network variable declared in **FtxlDev.c**, and then call the **LonPropagateNv()** function to send the network variable update. The **LonPropagateNv()** function uses the index of the network variable, which is defined in the **LonNvIndex** enumeration in **FtxlDev.h**. The index names use the following format:

LonNvIndex*Name*

Example: A network variable that is named **nviRequest** has the index name **LonNvIndexNviRequest**.

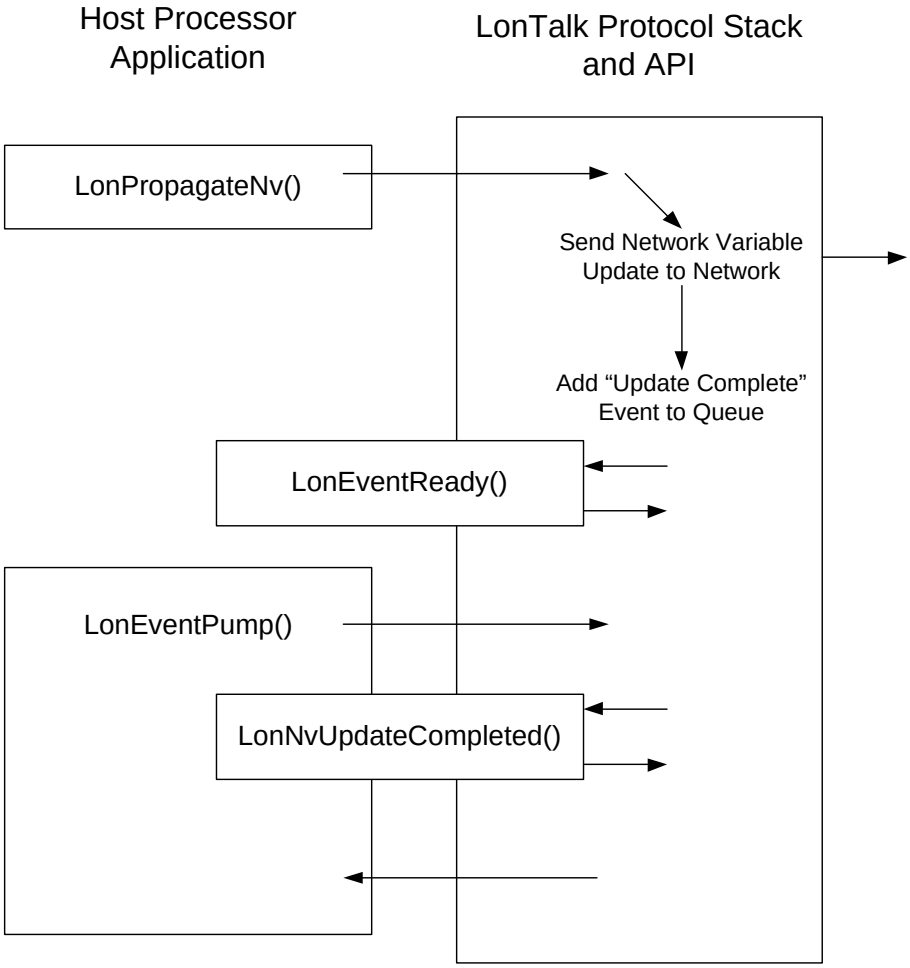
For dynamic network variables, the application must call the **LonGetNvValue()** function to retrieve the address of the value of a dynamic network variable.

The **LonPropagateNv()** function forwards the update to the LonTalk host stack, which in turn transmits the update to the network. This function returns an error status that indicates whether the update was delivered to the LonTalk host stack, but does not indicate successful completion of the update itself.

The LonTalk Stack device must be configured and online to be able to propagate a network variable value. If the **LonPropagateNv()** function is called when the LonTalk Stack device is not configured or not online, the function returns **LonApiOffline**.

After the update is complete, the LonTalk host stack informs the **LonEventReady()** callback handler function in the LonTalk Stack application. The application then calls the **LonEventPump()** function, which in turn calls your **LonNvUpdateCompleted()** callback handler function, to notify your application of the success or failure of the update. You can use this function for

any application-specific processing of update completion. The following figure shows the control flow for processing a network variable update.



In the case of an unacknowledged or repeated service type, the LonTalk host stack considers the update complete when it has finished sending the update to the network. In the case of an acknowledged service type, the LonTalk host stack considers the update complete when it receives acknowledgements from all receiving devices, or when the retry timer expires *n* times (where *n* is the retry count for the network variable + 1).

To process an update failure, edit the **LonNvUpdateCompleted()** callback handler function in the **FtxlHandlers.c** file. This function is passed the network variable index (the same one that you passed to the **LonPropagateNv()** function), and is also passed a success flag. The function is initially empty, but you can edit it to add your application-specific processing. The function initially appears as:

```
void LonNvUpdateCompleted(const unsigned index, const
                          LonBool success)
{
   /* TBD */
}
```

Do not handle an update failure with a repeated propagation; the LonTalk host stack automatically retries a number of times based on the network variable's retry count. A completion failure generally indicates a problem that should be signaled to the user interface (if any), flagged by an error or alarm output network variable (if any), or by signaled as a **comm_failure** error through the **nvoStatus** network variable of the Node Object functional block (if there is one).

Example: The following model file defines the device interface for a simple power converter. This converter accepts current and voltage inputs on its **nviAmpere** and **nviVolt** input network variables. It computes the power and sends the value on its **nvoWatt** output network variable:

```
network input  SNVT_amp       nviAmpere;
network input  SNVT_volt      nviVolt;
network output SNVT_power     nvoWatt;

fblock UFPTpowerConverter {
   nvoWatt     implements nvoPower;
   nviAmpere   implements nviCurrent;
   nviVolt     implements nviVoltage;
} powerConverter;
```

The following code fragment, implemented in your application's code, uses the data most recently received by either of the two input network variables, computes the product, and stores the result in the **nvoWatt** output network variable. It then calls the **LonPropagateNv()** function to send the computed value.

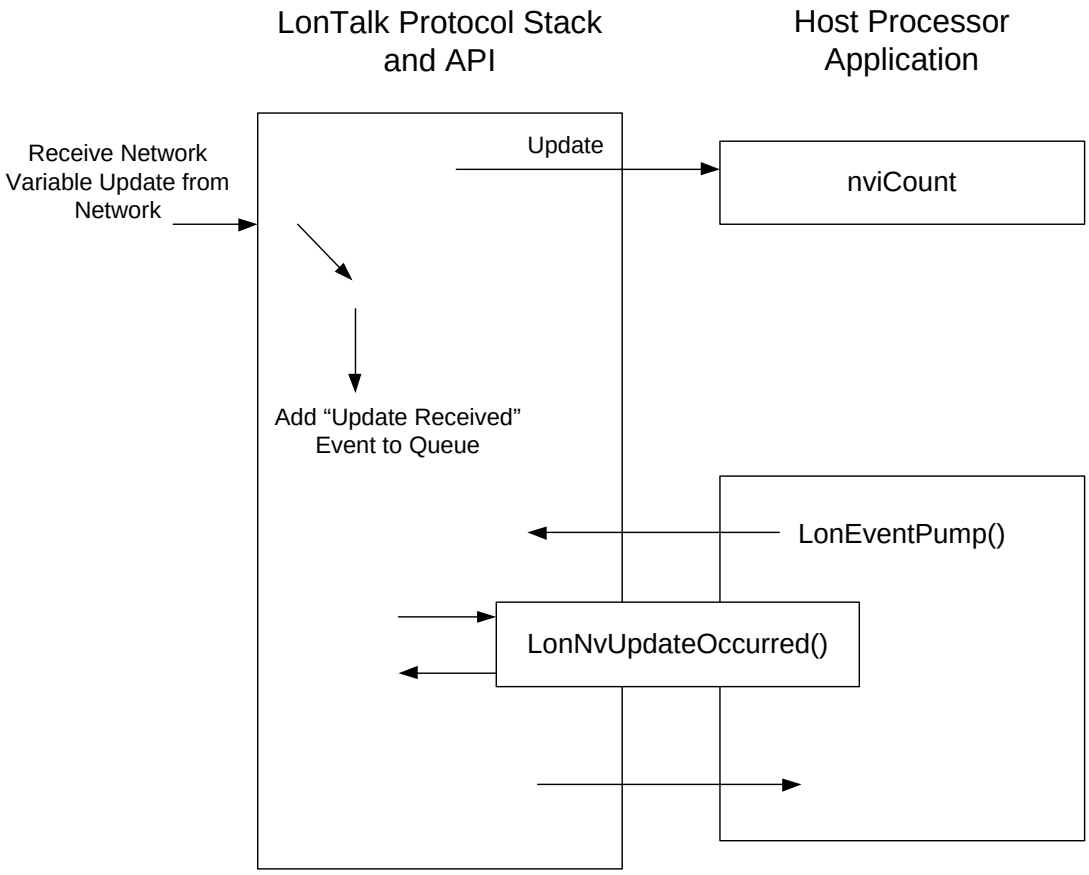
```
#include "FtxlDev.h"

void myController(void)
{
   nvoWatt = nviAmpere * nviVolt;
   if (LonPropagateNv(LonNvIndexNvoWatt)!= LonApiNoError) {
      // handle propagation error here
      // such as lack of buffers or validation
      ...
   }
}
```

Receiving a Network Variable Update from the Network

When the LonTalk host stack receives a network variable update from the network, it enqueues the event and signals the arrival of the event by calling the **LonEventReady()** callback handler function. When the application calls the **LonEventPump()** function, the LonTalk host stack writes the update to your network variable (by using the variable's address stored in the network variable table), and then calls the **LonNvUpdateOccurred()** event handler function to inform your application that the update occurred. The application can read the current value of any input network variable by reading the value of the variable declared in the **FtxlDev.c** file.

If a network variable update is received while the LonTalk Stack device is offline, the value of the network variable is updated, but the **LonNvUpdateOccurred()** event handler function is not called.

To process notification of a network variable update, modify the **LonNvUpdateOccurred()** event handler function (in the **FtxlHandlers.c** file) to call the appropriate functions in your application. The API calls this function with the index of the updated network variable. The following figure shows the control flow for receiving a network variable update.



Configuration network variables are used much in the same way as input network variables, with the exception that the values must be kept in persistent storage, and the application does not always respond to changes immediately. Example 1, below, shows the processing flow for regular network variable updates, and example 2 shows the same flow but with the addition of a configuration network variable.

Example 1:

This example uses the same power converter model file from the example in the previous section, *Sending a Network Variable Update*. That example demonstrated how to read the network variable inputs asynchronously by reading the latest values from the network variables declared in the **FtxlDev.c** file.

This example extends the previous example and shows how your application can be notified of an update to either network variable. To receive notification of a network variable update, modify the **LonNvUpdateOccurred()** callback function.

In **FtxlHandlers.c**:

```
extern void myController(void);

void LonNvUpdateCompleted(unsigned index, const LonBool
                          success) {

   switch (index) {
      case LonNvIndexNviAmpere:   /* fall through */
      case LonNvIndexNviVolt:
         myController();
         break;
      default:
         /* handle other NV updates (if any) */
   }
}
```

In your application source file:

```
#include "FtxlDev.h"

void myController(void) {
   // nvoWatt = nviAmpere * nviVolt;
   LON_SET_UNSIGNED_WORD(nvoWatt,
      LON_GET_UNSIGNED_WORD(nviAmpere)
      * LON_GET_UNSIGNED_WORD(nviVolt));
   if (LonPropagateNv(LonNvIndexNvoWatt) != LonApiNoError)
   {
      // handle propagation error here
      ...
   }
}
```

This modification calls the **myController()** function defined in the example in the previous section, *Sending a Network Variable Update*. Because network variable types are defined as type **LonWord**, this example uses the **LON_GET_UNSIGNED_WORD** macros to get the **nviAmpere** and **nviVolt** network variable values, and **LON_SET_UNSIGNED_WORD** to set the value for the **nvoWatt** network variable.

Example 2:

This example adds a configuration network variable to Example 1. A **SCPTgain** configuration property is added to the device interface in the model file:

```
network input  SNVT_amp       nviAmpere;
network input  SNVT_volt      nviVolt;
network output SNVT_power     nvoWatt;

network input cp SCPTgain nciGain;

fblock UFPTpowerConverter {
   nvoWatt     implements nvoPower;
   nviAmpere   implements nviCurrent;
   nviVolt     implements nviVoltage;
} powerConverter fb_properties {
   nciGain
};
```

You can enhance the **myController()** function to implement the new gain factor:

```
void myController(void)
```

```
{
   // nvoWatt = nviAmpere * nviVolt * nciGain.multiplier;
   LON_SET_UNSIGNED_WORD(nvoWatt,
      LON_GET_UNSIGNED_WORD(nviAmpere)
      * LON_GET_UNSIGNED_WORD(nviVolt)
      * LON_GET_UNSIGNED_WORD(nciGain.multiplier));
   // nvoWatt /= nciGain.divider;
   LON_SET_UNSIGNED_WORD(nvoWatt,
      LON_GET_UNSIGNED_WORD(nvoWatt)
      / LON_GET_UNSIGNED_WORD(nciGain.  divider));
   if (LonPropagateNv(LonNvIndexNvoWatt) != LonApiNoError)
   {
      // handle propagation error here
      ...
   }
}
```

Configuration network variables must be persistent, that is, their values must withstand a power outage.

Handling a Network Variable Poll Request from the Network

Devices on the network can request the current value of a network variable on your device by polling or fetching the network variable. The LonTalk host stack responds to poll of fetch requests by sending the current value of the requested network variable.

Handling Changes to Changeable-Type Network Variables

When a network management tool plug-in or the OpenLNS CT Browser changes the type of a changeable-type network variable, it informs your application of the change by describing the new type in the **SCPTnvType** configuration property that is associated with the network variable.

When your application detects a change to the **SCPTnvType** value:

- It determines if the change is valid.
- If the change is valid, it processes the change.
- If the change is not valid, it reports an error.

Valid type changes are those that the application can support. For example, an implementation of a generic PID controller might accept any numerical floating-point typed network variables (such as **SNVT_temp_f**, **SNVT_rpm_f**, or **SNVT_volt_f**), but can reject other types of network variables. Or a data logger device might support all types that are less than 16 bytes in size, and so on.

See *The Dynamic Interface Example Application* for an example application that handles changeable-type network variables.

Validating a Type Change

The **SCPTnvType** configuration property is defined by the following structure:

```
typedef LON_STRUCT_BEGIN(SCPTnvType) {
    ncuInt type_program_ID[8];
    ncuInt type_scope;
    ncuLong type_index;
    ncsInt type_category;
    ncuInt type_length;
    ncsLong scaling_factor_a;
    ncsLong scaling_factor_b;
    ncsLong scaling_factor_c;
} LON_STRUCT_END(SCPTnvType);
```

When validating a change to a network variable, an application can check five of the fields in the **SCPTnvType** configuration property:

- The program ID template of the resource file that contains the network variable type definition (**type_program_ID[8]**)
- The scope of the resource file that contains the network variable type definition (**type_scope**)
- The index within the specified resource file of the network variable type definition (**type_index**)
- The category of the network variable type (**type_category**)
- The length of the network variable type (**type_length**)

The **type_program_ID** and **type_scope** values specify a program ID template and a resource scope that together uniquely identify a resource file set. The **type_index** value identifies the network variable type within that resource file set. If the **type_scope** value is 0, the **type_index** value is a SNVT index. For example, checking the **type_scope** and **type_program_ID** fields lets you accept only types that you created.

The **type_category** enumeration is defined in the **<snvt_nvt.h>** include file as:

```
typedef enum nv_type_category_t {
   NVT_CAT_INITIAL = 0,       // Initial (default) type
   NVT_CAT_SIGNED_CHAR,       // Signed Char
   NVT_CAT_UNSIGNED_CHAR,     // Unsigned Char
   NVT_CAT_SIGNED_SHORT,      // 8-bit Signed Short
   NVT_CAT_UNSIGNED_SHORT,    // 8-bit Unsigned Short
   NVT_CAT_SIGNED_LONG,       // 16-bit Signed Long
   NVT_CAT_UNSIGNED_LONG,     // 16-bit Unsigned Long
   NVT_CAT_ENUM,              // Enumeration
   NVT_CAT_ARRAY,             // Array
   NVT_CAT_STRUCT,            // Structure
   NVT_CAT_UNION,             // Union
   NVT_CAT_BITFIELD,          // Bitfield
   NVT_CAT_FLOAT,             // 32-bit Floating Point
   NVT_CAT_SIGNED_QUAD,       // 32-bit Signed Quad
   NVT_CAT_REFERENCE,         // Reference
   NVT_CAT_NUL = -1           // Invalid Value
} nv_type_category_t;
```

This enumeration describes the type (signed short or floating-point, for example), but does not provide information about structure or union fields. To support all scalar types, test for a **type_category** value between **NVT_CAT_SIGNED_CHAR** and **NVT_UNSIGNED_LONG**, plus **NVT_CAT_SIGNED_QUAD**.

The **type_length** field provides the size of the type in bytes.

Multiple changeable-type network variables can share the **SCPTnvType** configuration property. In this case, the application must process all network variables from the property's application set, because just as the **SCTPnvType** configuration property applies to all of these network variables, so does the type change request. The application should accept the type change only if all related network variables can perform the required change.

If one or more type-inheriting configuration properties apply to changing configuration network variables, these type-inheriting configuration NVs also change their type at the same time. If this type-inheriting configuration NV is shared among multiple network variables, all related network variables must change to the new type. Sharing a type-inheriting configuration property among both changeable and non-changeable network variables is not supported.

Processing a Type Change

After validating a type change request, the application performs the type change. The type-dependent part of your application queries these details when required and processes the network variable data accordingly.

Some type changes require additional processing, while others do not. For example, if your application supports changing between different floating-point types, perhaps no additional processing is required. But if your application supports changing between different scalar types, it might require the use of scaling factors to convert the raw network variable value to a scaled value. You can use the three scaling factors defined in the **SCPTnvType** configuration property (**scaling_factor_a**, **scaling_factor_b**, and **scaling_factor_c**) to convert from raw data to scaled fixed-point data according to the following formula:

$$scaled = \left(a * 10^{b} * \left(raw + c\right)\right)$$

where *raw* is the value before scaling is applied, and *a*, *b*, and *c* are the values for **scaling_factor_a**, **scaling_factor_b**, and **scaling_factor_c**.

To convert the scaled data back to a raw value for an output network variable, use the following inverted scaling formula:

$$raw = \left(\frac{scaled}{a * 10^{b}}\right) - c$$

For example, the **SNVT_lev_cont** type is an unsigned short value that represents a continuous level from 0 to 100 percent, with a resolution of 0.5%. The actual data values (the raw values) are in the variable range from 0 to 200. The scaling factors for **SNVT_lev_cont** are defined as a=5, b= -1, c=0.

If the network variable is a member of an inheriting configuration property's application set that implements the property as a configuration network variable,

then the application must process the type changes for both the network variable and the configuration network variable.

If the network variable is a member of a configuration property's application set where the configuration property is shared among multiple network variables, the application must process the type and length changes for all network variables involved.

However, if the configuration property is implemented within a configuration file, no change to the configuration file is required. The configuration file states the configuration property's initial and maximum size (in the CP documentation-string *length* field), and network management tools derive the current and actual type for type-inheriting CPs from the associated network variable.

Your application must always support the **NVT_CAT_INITIAL** type category. If the requested type is of that category, your application must ignore all other content of the **SCPTnvType** configuration property and change the related network variable's type back to its initial type. The network variable's initial type is the type declared in the model file.

Processing a Size Change

If a supported change to the **SCPTnvType** configuration property results in a change in the size of a network variable type, your application must provide code to inform the LonTalk host stack about the current length of the changeable-type network variable. The current length information must be kept in non-volatile memory.

The LonTalk API provides a callback handler function, **LonGetNvSize()**, that allows you to inform the API of the network variable's current size. The following code shows an example implementation for the callback handler function.

```
unsigned LonGetNvSize(const unsigned index) {
   const LidNvDefinition* const nvTable = LonGetNvTable();
   unsigned size = LonGetDeclaredNvSize(index);

   if (index < LonNvCount &&
       nvTable[index].Definition.Flags & LON_NV_CHANGEABLE)
   {
      const SCPTnvType* pNvType = myGetNvTypeCp(index);
      // if the NV uses the initial type, its size is
      // the declared size set above
      if (pNvType->type_category != NVT_CAT_INITIAL) {
         size = pNvType->type_length;
      }
   }
   return size;
}
```

The example uses a **myGetNvTypeCp()** function (that you provide) to determine the type of a network variable, based on your knowledge of the relationships between the network variables and configuration properties implemented.

If the changeable-type network variable is member of an inheriting configuration property that is implemented as a configuration property network variable, the type information must be propagated from the changeable-type network variable to the type-inheriting configuration property, so that the **LonGetNvSize()**

callback handler function can report the correct current size for any implemented network variable. Your **myGetNvTypeCp()** function could handle that mapping.

For the convenience of network management tools, you can also declare a **SCPTmaxNVLength** configuration property to inform the tools of the maximum type length supported by the changeable-type network variable. For example:

```
network input cp SCPTnvType nciNvType;
const SCPTmaxNVLength cp_family nciNvMaxLength;

network output changeable_type SNVT_volt_f nvoVolt
   nv_properties {
      nciNvType,
      nciNvMaxLength=sizeof(SNVT_volt_f)
};
```

Rejecting a Type Change

If a network management tool attempts to change the type of a changeable-type network variable to a type that is not supported by the application (or is an unknown type), your application must do the following:

- Report the error within a maximum of 30 seconds from the receipt of the type change request. The application should signal an **invalid_request** through the Node Object functional block and optionally disable the related functional block. If the application does not include a Node Object functional block, the application can set an application-specific error code and take the device offline (use the offline parameter with the **LonSetNodeMode()** function).
- Reset the **SCPTnvType** value to the last known good value.
- Reset all other housekeeping data, if any, so that the last known good type is re-established.

Handling Dynamic Network Variables

To define the maximum number of supported dynamic network variables for your LonTalk Stack device, you use the LonTalk Interface Developer utility (the Application Configuration page) to specify the total number of dynamic variables that the application supports. This number represents the application's capacity for dynamic network variables; the actual dynamic network variables are created or deleted when the application is running. The process of managing dynamic network variables is handled by the LonTalk host stack and the API, but to use the dynamically created network variables, your application must respond to related events.

The application must be able to handle the addition, modification, or deletion of dynamic network variables. Dynamic network variable requests can come from a network management tool or from another LONWORKS device on the network. You must add code to the following event handler functions to support dynamic network variables:

- **LonNvAdded()**

  The LonTalk host stack calls this function when a dynamic network

variable is added. On device startup, it calls this function for each dynamic network variable that had been previously defined.

- **LonNvTypeChanged()**

  The LonTalk host stack calls this function when a dynamic network variable definition is changed.

- **LonNvDeleted()**

  The LonTalk host stack calls this function when a dynamic network variable is deleted.

For the **LonNvAdded()** and **LonNvTypeChanged()** event handler functions, the LonTalk host stack passes the index value for the dynamic network variable, and a pointer to the network variable's attributes, such as direction, size, name, and self-documentation string.

When a dynamic network variable is first added, the name and the self-documentation string for the network variable might be blank. A network management tool can update the name or the self-documentation string in a subsequent network management message, for which the LonTalk host stack calls the **LonNvTypeChanged()** event handler.

Communicating with Other Devices Using Application Messages

Application messages are used to create both standard and proprietary (that is, non-interoperable) interfaces for a device. You can use application messages for standard interfaces such as LONWORKS FTP or LonMark data log transfer, and you can use application messages if your device needs a proprietary interface that does not need to interoperate with devices from other manufacturers, for example, to implement a manufacturing-test interface that is only used during manufacturing test of your device. You can also use the same mechanism that is used for application messaging to create foreign-frame messages (for proprietary gateways) and explicitly addressed network variable messages.

One interoperable use for application messages is to implement the LONWORKS file transfer protocol. This protocol is used to exchange large blocks of data between devices or between devices and tools, and is also used to access configuration files on some devices.

The content of an application message is defined by a *message code* that is sent as part of the message. The message codes that are available for use by your application are *standard application messages* and *user-defined application messages*. User-defined application messages use message codes 0 to 47 (0x0 to 0x2F). Your application must define the meaning of each user-defined message code. Standard application messages are defined by LONMARK International, and use message codes 48 to 62 (0x30 to 0x3E).

The message code is followed by a variable-length data field, that is, a message code could have one byte of data in one instance and 25 bytes of data in another instance.

Sending an Application Message to the Network

Call the **LonSendMsg()** function to send an application message. This function forwards the message to the LonTalk host stack, which in turn transmits the message on the network. After the message is sent, the LonTalk host stack calls the **LonEventReady()** callback handler function to inform the application that an event has been enqueued. When the application calls the **LonEventPump()** function, the LonTalk API calls your **LonMsgCompleted()** event handler function. This function notifies your application of the success or failure of the transmission. You can use this function for any application-specific processing of message transmission completion.

To be able to send an application message, the LonTalk Stack device must be configured and online. If the application calls the **LonSendMsg()** function when the device is either not configured or not online, the function returns the **LonApiOffline** error code.

You can send an application message as a request message that causes the generation of a response by the receiving device or devices. If you send a request message, the receiving device (or devices) sends a response (or responses) to the message. When the Echelon Smart Transceiver or Neuron Chip receives a response, it enqueues the response and calls the **LonEventReady()** callback handler function to inform that application that an event has been enqueued. When the application calls the **LonEventPump()** function, the LonTalk API calls your **LonResponseArrived()** event handler function for each response it receives.

Receiving an Application Message from the Network

When the LonTalk host stack receives an application message from the network, it forwards the message to the **LonEventPump()** function in the LonTalk API, which in turn calls your **LonMsgArrived()** callback handler function. Your implementation of this function must process the application message.

The LonTalk host stack does not call the **LonMsgArrived()** callback handler function if an application message is received while the LonTalk Stack device is either unconfigured or offline.

If the message is a request message, your implementation of the **LonMsgArrived()** callback handler function must determine the appropriate response and send it using the **LonSendResponse()** function.

Handling Management Commands

LONWORKS installation and maintenance tools use network management commands to set and maintain the network configuration for a device. The LonTalk host stack automatically handles most network management commands that are received from these tools. A few network management commands might require additional application-specific processing, so the LonTalk API forwards the request to your application through the network management callbacks. These commands are requests for your application to wink, go offline, go online,

or reset, and are handled by your **LonWink()**, **LonOffline()**, **LonOnline()**, and **LonReset()** callback handler functions.

Handling Local Network Management Tasks

There are various network management tasks that a device can choose to initiate on its own. These are local network management tasks, which are initiated by the LonTalk Stack application and implemented by the LonTalk host stack. Local network management commands are never propagated to the network. The Extended LonTalk APIs allow you to include handling of these local network management commands if your LonTalk Stack application requires it.

Handling Reset Events

A network management tool can send a reset message to the LonTalk Stack device for a variety of reasons. For example, to reset the device after changing the communication parameters (including setting the priority), or following an update to a configuration property that is declared with a restriction flag which indicates that the network manager must reset the device after an update. The LonTalk host stack processes reset messages and manages everything that is required by the protocol. It also calls the **LonReset()** event handler function to inform the application, so that the application can perform any application specific processing.

The **LonReset()** callback handler function returns a pointer to the **LonResetNotification** structure, but this pointer is always NULL. The pointer is included for code compatibility with ShortStack applications. Whenever the LonTalk Stack device is reset, the state of the device is set to configured, and the mode of the device is changed to **online**, but no **LonOnline()** event is generated.

Resetting a LonTalk Stack device from the network affects only the LonTalk Stack, and does not cause a processor or application software reset.

Querying the Error Log

The LonTalk host stack writes application errors to the system error log. The **LonStatus** structure, which is returned by the **LonQueryStatus()** function contains complete statistics information, such as the number of transmit errors, transaction timeouts, missed and lost messages, and so on.

Working with ECS Devices

A LonTalk Stack device is an extended command set (ECS) device (that is, the **ver_nm_max** field of the Capability Info Record in the device's self-identification string is greater than 0). A LonTalk Stack device supports both the extended command set and legacy network management commands. However, after a device receives any extended commands, it operates in the extended mode, and returns a negative response to legacy commands.

Any OpenLNS-based tool communicates with a LonTalk Stack device using ECS commands (for example, during device commissioning), and thus places the device in extended mode. Some tools that are not based on OpenLNS, such as the NodeUtil utility, might not be able to communicate with a device that is in the extended mode.

To return a LonTalk Stack device to the legacy mode, rather than the extended mode, perform one of the following tasks:

- Re-run the LonTalk Interface Developer utility to generate a new signature for the device, and rebuild and load the application image.
- Send the **NM_NODE::NM_INITIALIZE** extended network management command to the device.
- Erase the non-volatile memory for the device.
- If the device is currently commissioned in an OpenLNS database, de-commission it.

You should not need to perform any of these tasks often because most network management tools use OpenLNS or are compatible with ECS.

For more information about the LonTalk extended command set (ECS) network management commands, see the *ISO/IEC 14908-1 Control Network Protocol Specification*. This document is available from ISO: www.iso.org/iso/home/store/catalogue_tc/catalogue_detail.htm?csnumber=60203

Using Direct Memory Files

To use configuration properties in files, your host application program must implement a method that allows the network management tool to access those files. You can support either one of the following:

- The LONWORKS FTP protocol
- The host direct memory files (DMF) feature

The FTP protocol is appropriate when large amounts of data need to be transferred between the host processor and Echelon Smart Transceiver or Neuron Chip. The host DMF feature is appropriate for most other cases.

By supporting direct memory files, your application allows the network management tool to use standard memory read and write network messages to access configuration property files located on the host. Direct memory files appear to the network management tool as if they were located within the Echelon Smart Transceiver or Neuron Chip's native address space, but the LonTalk host stack routes memory read and write requests within the DMF memory window to the **LonMemoryRead()** and **LonMemoryWrite()** callback handler functions provided in the **FtxlHandlers.c** file. These functions use the **LonTranslateWindowArea()** support function, which is generated by the LonTalk Interface Developer utility to translate between Echelon Smart Transceiver or Neuron Chip addresses and host addresses.

If the model file contains a network variable of type **SNVT_address**, the LonTalk Interface Developer utility automatically generates all necessary code and data for the memory read and write requests, including code in the **LonInit()** function to register the virtual memory window with the LonTalk Host stack.

You do not generally need to modify the code that the LonTalk Interface Developer utility generates (in **FtxlDev.c**) or the **LonMemoryRead()** and **LonMemoryWrite()** callback handler functions (in **FtxlHandlers.c**).

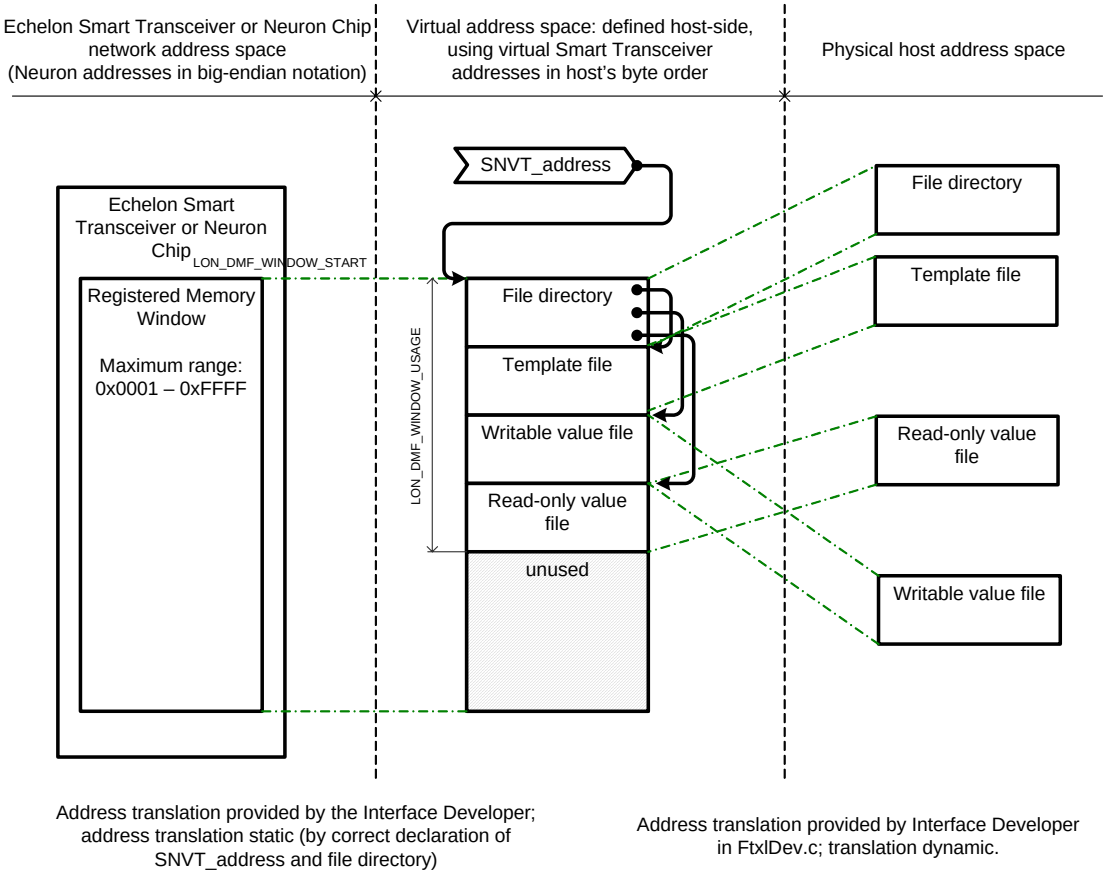
The DMF Memory Window

To the network management tool, all content of the DMF memory window is presented as a continuous area of RAM memory in the virtual DMF memory space. The DMF memory space is virtual because it appears to the network management tool to be located within the Echelon Smart Transceiver or Neuron Chip's native address space, even though it is not. In the code that the LonTalk Interface Developer utility generates, the content of the DMF memory window, which can be physically located in different parts, or even types, of the host processor's memory, is presented as a continuous area of memory. Another part of the generated code identifies the actual segment within the host memory that is shown at a particular offset within the virtual address space of the DMF memory window, and allows the DMF memory driver to correctly access the corresponding data within the host processor's address space.

Data that appears in the DMF memory window includes the following:

- File directory
- Template file
- Writeable CP value files (if any)
- Read-only CP value files (if any)

The following figure shows how the different memory address spaces relate to each other.



The LonTalk Interface Developer utility defines three macros in the generated **FtxlDev.h** file for working with the DMF window:

- **LON_DMF_WINDOW_START**
- **LON_DMF_WINDOW_SIZE**
- **LON_DMF_WINDOW_USAGE**

The **LON_DMF_WINDOW_USAGE** macro helps you keep track of the DMF window fill level. The LonTalk Interface Developer utility uses this value when it registers the actual window, whereas **LON_DMF_WINDOW_SIZE** defines only the maximum window size.

You can modify the DMF framework that the LonTalk Interface Developer utility generates to include support for user-defined files. However, all of the data must fit within the DMF memory window.

When your data exceeds the size of the DMF memory window, you must perform one of the following tasks:

- Reduce the amount of data
- Implement the LONWORKS File Transfer Protocol

File Directory

The LonTalk Interface Developer utility produces a configurable file directory structure, which supports:

- Using named or unnamed files (the DMF framework uses unnamed files by default, whereas FTP uses named files)
- Up to 64 KB of data for each file
- For the DMF framework: Up to a total of 64 KB for all files plus the file directory itself
- For FTP: unlimited size

The utility initializes the file directory depending on the chosen access method. The directory can be used with an FTP server implementation or the file access host memory window implementation. The initialization that the utility provides works for both little-endian and big-endian host processors.

The **FtxlDev.h** header file allows you to customize the file directory structure, if needed.

Shutting Down the LonTalk Stack device

To perform an orderly shutdown of a LonTalk Stack device, your application can call the **LonExit()** API function. The implementation of this function is generated by the LonTalk Interface Developer utility, and is included in the **FtxlDev.c** file. This function calls the **LonLidDestroyStack()** API function to stop the LonTalk Host stack and free its resources. In addition, the **LonExit()** function can perform any clean-up for the application, such as deleting operating system events and other resources.

After your application calls the **LonExit()** function, it can call the **LonInit()** function again. However, if you want to change the LonTalk host stack's interface, you must reboot the device.

9

Developing an IP-852 Router Application

This chapter describes how to develop a LonTalk Stack IP-852 router application.

Developing an IP-852 Router Application

You can develop an IP-852 router application using the LonTalk Stack. The IP-852 router application does not require model files, a code framework generated by the LonWorks Interface Developer, or the LonTalk API. To create an IP-852 router application you use the following two C++ classes:

Class	Description	File
LtLogicalChannel	Represents the network interface used to access a native LonWorks channel.	**LonTalkStack\Source\ Shared\include\ LtChannel.h**
LtIp852Router	Represents the router object.	**LonTalkStack\Source\ Shared\include\ LtRourterApp.h**

LtLogicalChannel

The **LtLogicalChannel** class represents the network interface used to access a native LonWorks channel. You must instantiate and open this class before starting the IP-852 router.

You can call its constructor with either no parameters (if there is only one possible native network interface supported by the platform), or with a single parameter representing the name of the network interface.

LtIp852Router

The **LtIp852Router** class represents the router object. Its constructor takes no parameters. This class has the following functions:

Function	Syntax	Description
Start	`LtErrorType Start( int ltAppIndex, LtUniqueId &ltUid, LtLtLogicalChannel *pLtChannel, int ipAppIndex, LtUniqueId &ipUid, int ipAddress, int ipPort)`	Creates the IP-852 channel and starts both sides of the router. The following describes this method's parameters: **ltAppIndex**. The application index of the native LonWorks side of the router. The LonTalk Stack requires an application index whenever there is more than one stack. It is mainly used to store the unique ID of the stack and to

Function	Syntax	Description
		name persistence files. The index is arbitrary, but it may need to be between 0 and *n*, where *n* is platform dependent (depending on how unique IDs and other items are stored). **LtUniqueId**. The uniqueId of the native LonWorks side of the router. **pLtChannel**. The **LtLogicalChannel** representing the native LonWorks channel **ipAppIndex**. The application index of the IP-852 router side. **ipUid**. The unique ID of the IP-852 router side. **ipAddress**. The 32-bit representation of the IP address of the IP-852 router side in network order. **ipPort**. The port number on which the IP-852 router side will be listening.
Shutdown	`Shutdown()`	Closes the router.
SendServicePin Message	`SendServicePin Message()`	Sends a service pin message from both sides of the router.

10

Porting a LonTalk Stack Application

This chapter describes how to port a LonTalk Stack device or IP-852 router application to your platform.

Porting Overview

You can port your LonTalk Stack device or IP-852 router application to your own platform. Porting your application may require one to all of the following components:

1. A version of the OSAL for your operating system.
2. A LON-link driver.
3. Socket interfaces.
4. The LonTalk Stack source files.
5. Application-specific files for LonTalk Stack devices.
6. Your own application-specific files.

The following sections provide tips for creating these components.

OSAL

This LonTalk Stack Developer's Kit contains the following two example implementations of the OSAL in the **Source\VxLayer** directory:

File	Description
Osal.c	The Windows OSAL used by the LonTalk Stack example applications.
PosixOsal.c	The Linux version of the OSAL that uses p-threads.

LonLink Driver

If your application uses a native LonWorks interface, you need to create a class that is derived from the LonLink class to enable the stack to communicate with the network interface. The LonLink class is defined in the **Source\Shared\LonLink.h** file.

The LonTalk Stack Developer's Kit contains the two example LonLink base classes.

File/Folder	Description
Shared\LonLinkWin.cpp	The Windows version, which uses OpenLDV to communicate with any OpenLDV compatible network interface that includes a Layer 2 MIP.
Source\Target\Drivers \Linux\SMIP	The Linux version which communicates with an Echelon Smart Transceiver or Neuron Chip running the Layer 2 Serial

File/Folder	Description
	MIP. The LonLink derived class is implemented in **LonLinkDcx.cpp** and **LonLinkDcx.h**.

By default, Linux will automatically use the provided Linux driver for the Serial MIP. If you are using a different operating system, you need to modify the **#define LtLinkDefault** statement in the **Source\Shared\include\LtLinkDefault.h** file to reference your LonLink derived class.

Service LED

To control the service LED, you need to override the virtual **setServicePinState()** method of the **LtLink** object. The **LonLink** class is derived from the **LtLink** base class.

Socket Interfaces

If your platform supports IP-852 interfaces, you need to provide target-specific code to interface with sockets. The LonTalk Stack Developer's Kit contains two socket interface examples:

Socket Interface Example Files
Source\ShareIp\VxSockets.c
Source\VxLayer\VxWinSockets.c

LonTalkStack Source Files

The LonTalk Stack source files you need depends on whether your LonTalk Stack device is an application device using a native LonWorks interface, an application device using an IP-852 interface, an IP-852 Router, or a combination of these. The follow table lists the files required for each device type (files are relative to the **Source** directory):

File	App Device (Native LonWorks Interface)	App Device (IP-852 Interface)	IP-852 Router
FtxlApi\FtxlApi.cpp	X	X	
FtXlApi\FtXlStack.cpp	X	X	
Lre\LtIpPortClient.cpp	X	X	X
Lre\LtLre.cpp	X	X	X
Router\LtRouterApp.cpp			X
Shared\LonLink.cpp	X	X	X
Shared\LtBlob.cpp	X	X	X

Shared\LtChannel.cpp	X	X	X
Shared\LtCUtil.c	X	X	X
Shared\LtDomain.cpp	X	X	X
Shared\LtFailSafeFile.cpp		X	X
Shared\LtHashTable.cpp	X	X	X
Shared\LtLinkBase.cpp	X	X	X
Shared\LtNetwork.cpp	X	X	X
Shared\LtNvRam.cpp		X	X
Shared\LtPersistence.cpp	X	X	X
Shared\LtPktAllocator.cpp	X	X	X
Shared\LtPktAllocatorOne.cpp	X	X	X
Shared\LtPktInfo.cpp	X	X	X
Shared\LtProgramId.cpp	X	X	X
Shared\LtTaskOwner.cpp	X	X	X
Shared\LtUniqueId.cpp	X	X	X
Shared\LtVector.cpp	X	X	X
Shared\RefQues.cpp	X	X	X
ShareIp\iLonSntp.cpp		X	X
ShareIp\IpLink.cpp		X	X
ShareIp\LtIpBase.cpp		X	X
ShareIp\LtIpChannel.cpp		X	X
ShareIp\LtIpEchPackets.cpp		X	X
ShareIp\LtIpMaster.cpp		X	X
ShareIp\LtIpPackets.cpp		X	X
ShareIp\LtIpPersist.cpp		X	X
ShareIp\LtIpPlatform.cpp		X	X
ShareIp\LtLreIpClient.cpp		X	X
ShareIp\LtMD5.cpp		X	X
ShareIp\LtPktReorderQue.cpp		X	X
ShareIp\md5c.c		X	X
ShareIp\Segment.cpp		X	X
ShareIp\SegSupport.cpp		X	X
ShareIp\sntpcLib.c		X	X
ShareIp\vxlTarget.c	X	X	X
Stack\DynamicNvs.cpp	X	X	X
Stack\LonTalkde.cpp	X	X	X
Stack\LonTalkStack.cpp	X	X	X
Stack\LtaBase.cpp	X	X	X
Stack\LtAddressConfiguration.cpp	X	X	X
Stack\LtAddressConfigurationTable.cpp	X	X	X
Stack\LtApdu.cpp	X	X	X
Stack\LtBitMap.cpp	X	X	X
Stack\LtConfigData.cpp	X	X	X
Stack\LtConfigurationEntity.cpp	X	X	X
Stack\LtDescription.cpp	X	X	X
Stack\LtDeviceStack.cpp	X	X	X
Stack\LtDomainConfiguration.cpp	X	X	X
Stack\LtDomainConfigurationTa	X	X	X

ble.cpp			
Stack\LtLayer4.cpp	X	X	X
Stack\LtLayer6.cpp	X	X	X
Stack\LtMip.cpp	X	X	X
Stack\LtMipApp.cpp	X	X	X
Stack\LtMsgOverride.cpp	X	X	X
Stack\LtNetworkImage.cpp	X	X	X
Stack\LtNetworkManager.cpp	X	X	X
Stack\LtNetworkManager2.cpp	X	X	X
Stack\LtNetworkStats.cpp	X	X	X
Stack\LtNetworkVariable.cpp	X	X	X
Stack\LtNetworkVariableConfiguration.cpp	X	X	X
Stack\LtNetworkVariableConfigurationTable.cpp	X	X	X
Stack\LtOutgoingAddress.cpp	X	X	X
Stack\LtPlatform.cpp	X	X	X
Stack\LtProXy.cpp	X	X	X
Stack\LtReadOnlyData.cpp	X	X	X
Stack\LtRouteMap.cpp	X	X	X
Stack\LtStackClient.cpp	X	X	X
Stack\LtStatus.cpp	X	X	X
Stack\LtTransactions.cpp	X	X	X
Stack\LtXcvrId.cpp	X	X	X
Stack\NdNetworkVariable.cpp	X	X	X
Stack\dedef.cpp	X	X	X
VXLayer\VxLayer.c	X	X	X
VXLayer\VxLQues.c	X	X	X
VXLayer\VxMsgQ.c	X	X	X
VXLayer\VxSemaph.c	X	X	X
VXLayer\VxTimers.c	X	X	X

Application-Specific Files for LonTalk Stack Devices

If you are building an application device that uses a native LonWorks or IP-852 interface, you need to include the LID-generated files that customize your application interface, and modify the templates as appropriate.

Application-Specific Code for IP-852 Interfaces

If you define an IP-852 interface, you must include a valid, unique ID for the IP-852 interface. If you are defining an application device, you must register this unique ID (prior to calling the **LonInit()** method) using the **LonResgisterUniqueId()** method. This method is defined in the **Source\FtxlApi\FtxlApi.h** file.

Selecting the Device Type

Your build process must define one of the following preprocessor definitions that are used by LonTalk Stack to control the type of device that is being built:

Preprocessor Definition	Description
LONTALK_STACK_PLATFORM	An application device using a native LonWorks network interface.
LONTALK_IP852_STACK_PLATFORM	An application device using an IP-852 network interface.
LONTALK_ROUTER_PLATFORM	An IP-852 to native LonWorks router.

These preprocessor definitions control which features to be included in the LonTalk Stack by defining other preprocessor definitions that are stored in the **Source\Shared\include\LtaDefine.h** file. You generally use these definitions with the **FEATURE_INCLUDED** or **PRODUCT_IS** macros, which are also defined in the **LtaDefine.h** file.

File System Requirements

The LonTalk API defines callback handlers for various forms of non-volatile memory. For example, a native LonWorks application device may store its non-volatile data directly to flash without a file system.

The IP-852 portions of the LonTalk Stack source, however, do require a file system. If your application is an IP-852 device or IP-852 router, you must either provide a file system or port the file system references used by the IP-852 source as necessary. For more information, you can view the **Source\Shared\LtFailSafe.cpp** and **Source\Stack\LtPlatform.cpp** files.

Appendix A

LonTalk Interface Developer Command Line Usage

This appendix describes the command-line interface for the LonTalk Interface Developer utility. You can use this interface for script-driven or other automation uses of the LonTalk Interface Developer utility.

Overview

The LonTalk Interface Developer utility consists of two main components:

- The LonTalk Interface Developer graphical user interface (GUI), which collects your preferences and displays the results
- The LonTalk Stack Interface Developer, which processes the data from the GUI and generates the required output files

If you plan to run the LonTalk Interface Developer utility in an unattended mode, for example as part of an automated build process, you can use the command-line interface to the LonTalk Stack Interface Developer part of the LonTalk Interface Developer utility.

All commonly used project preferences are available through either the GUI or the command line interface. However, a few less common preferences (such as specifying the number of domain table entries, or setting the DMF window size or starting address) are available only through the command line interface.

To run the LonTalk Stack Interface Developer, open a Windows command prompt (**Start → Programs → Accessories → Command Prompt**), and enter the following command from the **[*LonWorks*]\InterfaceDeveloper** directory:

```
libf
```

Command Usage

The following command usage notes apply to running the **libf** command:

- If no command switches or arguments follow the command name, the tool responds with usage hints and a list of available command switches.
- Most command switches come in two forms: A short form and a long form.

  The short form consists of a single, case-sensitive, character that identifies the command, and must be prefixed with a single forward slash '/' or a single dash '-'. Short command switches can be separated from their respective values with a single space or an equal sign. Short command switches do not require a separator; the value can follow the command identifier immediately.

  The long form consists of the verbose, case-sensitive, name of the command, and must be prefixed with a double dash '--'. Long command switches require a separator, which can consist of a single space or an equal sign.

  Examples:
  Short form: `libf -n …`

  Long form: `libf --source …`
- Multiple command switches can be separated by a single space.

- Commands of a Boolean type need not be followed by a value. In this case, the value **yes** is assumed. Possible values for Boolean commands are **yes**, **on**, **1**, **+**, **no**, **off**, **0**, **-** (a minus sign or dash).

  Examples:
  ```
  libf --verbosecomments=yes
  libf --verbosecomments
  ```

- Commands can be read from the command line or from a command file (script file). A command file contains empty lines, lines starting with a semicolon (comment lines), or lines containing one command switch on each line (with value as applicable). The file extension can be any characters, but it is recommended that you use the ".libf" extension.

  Example command file:

  ```
  ; LIBF command file for myProject
  --source=myModelFile.nc
  --basename=myProjectVer1
  --clock=10
  --pid=9F:FF:FF:00:00:00:04:00
  --out=C:\myFolder\ProjectVer1
  ```

- Command switches can appear at any location within the command line or in any order (on separate lines) within a script.

Command Switches

The following table lists the available command switches for the **libf** command. Only the following switches are required for the command:

- --source (–n)
- --pid (-i)
- --basename (-b)
- --clock (-c)

Other command switches are optional.

Command Switch		
Long Form	**Short Form**	**Description**
--addresses	-A	Implement address table with the specified number of entries
--aliases	-L	Implement alias table with specified number of entries
--avgdynsd	-g	Set the average dynamic network variable self-documentation string size (0..128)
--basename	-b	Set the project's base name

Command Switch		Description
Long Form	**Short Form**	
--buffer	-B	Implement specified number of buffers of the specified type
--clock	-c	Set the Echelon Smart Transceiver or Neuron Chip clock rate (in MHz)
--define	-D	Define a specified preprocessor symbol (without value)
--defloc		Location of an optional default command file
--dmfsize	-z	Override size of the direct memory file memory window
--dmfstart	-a	Override start address of the direct memory file memory window
--domains	-d	Implement domain table with specified number of entries
--dynamicnvs	-y	Provide support for specified number of dynamic network variables
--file	-@	Include a command file
--help	-?	Display usage hint for command
--include	-I	Add the specified folder to the include search path
--mkscript		Generate command script in specified location
--nodefaults		Disable processing of default command files
--nvdflush	-N	Flush non-volatile date after specified timeout period (1, 5, 10, or 20 seconds)
--nvdmodel	-M	Use specified model for non-volatile data (flash, file, or user)
--nvdroot	-R	Use the specified root for the non-volatile driver
--out	-o	Generate all output files in the specified location
--pid	-i	Use the specified program ID (in colon-separated format)
--rxdb	-r	Manage specified number of receive transaction records

Command Switch		Description
Long Form	**Short Form**	
--silent		Suppress banner message display
--source	-n	Use the specified model file
--spdelay	-p	Set the service pin notification delay (255=default, 0=off)
--txdb	-t	Manage specified number of transmit transaction records
--txttl	-T	Let transmit transactions expire after specified number of microseconds
--verbose	-v	Run with verbosity level 0 (normal), 1 (verbose), or 2 (trace)
--verbosecomments	-V	Generate verbose comments
--warning		Display specified message type as a warning

Specifying Buffers

The **--buffer** (**-B**) command switch specifies a number of buffers of a specified type. The supported types of buffers are:

- Application input buffers
- Application output buffers
- Application output priority buffers
- Link-layer buffers
- Network input buffers
- Network output buffers
- Network input buffer size
- Network output buffer size
- Network output priority buffers

For each of these buffer types, you can specify a number of buffers using the following syntax:

```
--buffer=buffer_type.number
```

where *buffer_type* can be any of the specifications listed in the following table, and *number* is the number of that type of buffer. Each of the specifications has several allowable values; the table lists the primary specification and allowable alternate specifications.

The type and number for the **--buffer** switch are separated by a period. You can include several buffer specifications within a single **--buffer** switch, separated by commas, or with multiple **--buffer** switches. For example:

```
--buffer=ai.5,ao.3
--buffer=ai.5 --buffer=ao.3
```

Buffer Type	Primary Specification	Alternate Specifications	Valid Values
Application input buffers	ai	appinput appin application-input	1 to 100 Default: 5
Application output buffers	ao	appoutput appout application-output	1 to 100 Default: 3
Application output priority buffers	aop	appoutputprio appoutprio appprio application-priority-output	1 to 100 Default: 2
Link-layer buffers	ll	linklayer link-layer link	1 to 100 Default: 2
Network input buffers	nis	netinsize network-input-size	1 to 100 Default: 11
Network output buffers	nos	netoutsize network-output-size	1 to 100 Default: 3
Network input buffer size	ni	netinput netin network-input	1 to 100 Default: 11
Network output buffer size	no	netoutput netout network-output	1 to 100 Default: 3

Buffer Type	Primary Specification	Alternate Specifications	Valid Values
Network output priority buffers	nop	netoutputprio netoutprio netprio network-priority-output	1 to 100 Default: 3

The application buffers (ai, ao, and aop) all have a range of 1 to 100 for the allowable number of buffers.

You can set priority buffers to a count of 0 (zero), but you must specify at least one non-priority buffer in both directions (input and output). However, LONMARK International requires all interoperable LONWORKS devices to have at least one priority buffer. Eliminating priority buffers will prevent certification.

The LonTalk Interface Developer utility issues messages that relate to the buffer configuration. For example, the utility issues messages for the following situations:

- If the configuration exceeds the available buffer space, the utility issues error LID#62 (Insufficient buffer space).
- If additional netin or netout buffers of the currently configured size could be added to the configuration, the utility issues warning LID#4026 (Unused buffer space).
- If at least one 20-byte buffer could be added to the configuration, the utility issues hint LID#8005 (Unused buffer space).

Appendix B

Model File Compiler Directives

This Appendix lists the compiler directives that can be included in a model file. Model files are described in Chapter 6, *Creating a Model File*.

Using Model File Compiler Directives

ANSI C permits compiler extensions through the **#pragma** directive. These directives are implementation-specific. The ANSI standard states that a compiler can define any sort of language extensions through the use of these directives. Unknown directives can be ignored or discarded. The Neuron C compiler issues warning messages for unrecognized directives.

In the Neuron C compiler, pragmas can be used to set certain Neuron firmware system resources and device parameters such as code generation options, debugging options, error reporting options, and other miscellaneous features. In general, these directives can appear anywhere in the model file.

Any compiler directive that is not described in this appendix is not accepted by the LonTalk Interface Developer utility, and causes an error if included in a model file. You can use conditional compilation to exclude unsupported directives.

Acceptable Model File Compiler Directives

You can specify the following compiler directives in a model file. These directives can appear anywhere in the model file, and control the output produced by the LonTalk Interface Developer utility.

#pragma codegen *option*

This pragma allows control of certain features in the compiler's code generator. Application timing and code size might be affected by use of these directives. Valid values for *option* include:

- **cp_family_space_optimization**
- **no_cp_template_compression**

The Neuron C compiler can attempt to compact the configuration property template file by merging adjacent family members that are scalars into elements of an array. Any CP family members that are adjacent in the template file and value file, and that have identical properties, except for the item index to which they apply, are merged. Using optional *configuration property re-ordering and merging* can achieve additional compaction beyond what is normally provided by automatic merging of whatever CP family members happen to be adjacent in the files. To enable this re-ordering feature, specify **#pragma codegen cp_family_space_optimization** in your model file. With this feature enabled, the Neuron C compiler optimizes the layout of CP family members in the value and template files to make merging more likely.

You can specify **#pragma codegen no_cp_template_compression** in your program to disable the automatic merging and compaction of the configuration property template file. Use of this directive can cause your program to consume more of the device's memory, and is intended only to provide compatibility with the NodeBuilder 3.0 Neuron C compiler.

You cannot use both the **no_cp_template_compression** option and the **cp_family_space_optimization** option in the same model file.

Important: Configuration property re-ordering and merging can reduce the memory required for the template file, but can also result in slower access to the application's configuration properties by network management tools. This can potentially cause a significant increase in the time required to commission your device, especially on low-bandwidth channel types. You should typically only use configuration property re-ordering and merging if you must conserve memory. If you use configuration property re-ordering and merging, be sure to test the effect on the time required to commission and configure your device.

#pragma enable_sd_nv_names

Causes the LonTalk Interface Developer utility to include the network variable names in the self-documentation (SD) information when self-identification (SI) data is generated. This pragma can only appear once in the model file.

#pragma fyi_off
#pragma fyi_on

Controls the compiler's printing of informational messages. Informational messages are less severe than warnings, yet can indicate a problem in the model file. Informational messages are off by default at the start of compilation. These pragmas can be intermixed multiple times throughout a program to turn informational message printing on and off as needed.

#pragma hidden

This pragma is for use only in the **<echelon.h>** standard include file.

#pragma ignore_notused *symbol*

Requests that the compiler ignore the symbol-not-referenced flag for the named symbol. The compiler normally prints warning messages for any variables, functions, I/O objects, and so on, that are declared but are never used in the model file. This pragma can be used one or more times to suppress the warning on a symbol-by-symbol basis.

The pragma should appear after the variable declaration. A good coding convention is to place this pragma on the line that immediately follows the variable's declaration. For automatic scope variables, the pragma must appear no later than the line preceding the close brace character '**}**', which terminates the scope containing the variable. There is no terminating brace for any variable declared at file scope.

#pragma no_hidden

This pragma is for use only in the **<echelon.h>** standard include file.

#pragma relaxed_casting_off
#pragma relaxed_casting_on

These pragmas control whether the compiler treats a cast that removes the **const** attribute as an error or as a warning. The cast can be explicit or implicit (for example, an automatic conversion due to assignment). Normally, the compiler considers any conversion that removes the **const** attribute to be an error. Turning on the relaxed casting feature causes the compiler to treat this condition as a warning instead. These pragmas can be intermixed throughout a program to enable and disable the relaxed casting as needed.

#pragma set_guidelines_version *string*

The Neuron C 2.1 compiler generates LONMARK information in the device's XIF file and in the device's SIDATA (stored in device program memory). By default, the compiler uses "3.3" as the string to identify the LONMARK guidelines version to which the device conforms. To override this default, specify the overriding value in a string constant following the pragma name, as shown. For example, a program could specify **#pragma set_guidelines_version "3.2"** to indicate that the device conforms to the 3.2 guidelines. This directive is useful for backward compatibility with older versions of the Neuron C compiler.

This directive can be used to state compatibility with a guidelines version that is not actually supported by the compiler. Future versions of the guidelines that require a different syntax for SI/SD data are likely to require an update to the compiler. This directive has only the effect described above, and does not change the syntax of SD strings generated.

#pragma set_id_string "*ssssssss*"

Provides a legacy mechanism for setting the device's 8-byte program ID. This directive is allowed for legacy application support, and should not be used in a model file. Use the LonTalk Interface Developer utility to set the program ID.

#pragma set_node_sd_string *C-string-const*

Specifies and controls the generation of a comment string in the self-documentation (SD) data in a device's application image. Most devices have an SD string. The first part of this string documents the functional blocks for the device. This part is automatically generated by the LonTalk Interface Developer utility. This first part is followed by a comment string that documents the purpose of the device. This comment string defaults to a NULL string and can have a maximum of 1023 bytes, minus the first part of the SD string generated by the LonTalk Interface Developer utility, including the zero termination character. This pragma explicitly sets the comment string. Concatenated string constants are not allowed. This pragma can only appear once in the model file.

#pragma set_std_prog_id *hh:hh:hh:hh:hh:hh:hh:hh*

Provides a legacy mechanism for setting the device's 8-byte program ID. This directive is allowed for legacy application support, and should not be used in a model file. Use the LonTalk Interface Developer utility to set the program ID.

#pragma warnings_off
#pragma warnings_on

Controls the compiler's printing of warning messages. Warning messages generally indicate a problem in the model file, or a place where the code could be improved. Warning messages are on by default. These pragmas can be intermixed multiple times throughout a model file to turn informational message printing on and off as needed.

#pragma disable_warning *number*
#pragma enable_warning *number*

Controls the compiler's printing of individual warning messages. Warning messages generally indicate a problem in the model file, or a place where the code could be improved. Warning messages are on by default. These pragmas can be intermixed multiple times throughout a model file to turn informational message printing on and off as needed.

The *number* parameter refers to a specific warning number, for example **#pragma disable_warning 123**. Alternatively, you can use an asterisk to select all warnings, for example **#pragma enable_warning ***. This pragma is ignored if you specify **#pragma warnings_off** or **#pragma fyi_off**.

Appendix C

Neuron C Syntax for the Model File

This Appendix lists the Neuron C syntax for the allowable statements of a model file.

Functional Block Syntax

fblock *FPT-identifier* **{** *fblock-member-list* **}** *identifier* [*array-bounds*]
[*ext-name*] [*fb-property-list*] **;**

fblock-member-list : *fblock-member-list ; fblock-member*
fblock-member

fblock-member : *nv-reference* **implements** *member-name*
nv-reference impl-specific

impl-specific : **implementation_specific (** *const-expr* **)** *member-name*

nv-reference : *nv-identifier array-index*
nv-identifier

array-index : [*const-expr*]

array-bounds : [*const-expr*]

ext-name : **external_name (** *concatenated-string-const* **)**
external_resource_name (*concatenated-string-const* **)**
external_resource_name (*const-expr : const-expr* **)**

fb-property-list : See *Functional Block Properties Syntax*.

Keywords

fblock

Declares the functional block for the *FPT-identifier* functional-profile-type identifier and the *identifier* functional block identifier.

The functional block declaration begins with the **fblock** keyword, followed by the name of a functional profile from a resource file. The functional block is an implementation of the functional profile. The functional profile defines the network variable and configuration property members, a unique key called the *functional profile key*, and other information. The network variable and configuration property members are divided into mandatory members and optional members. Mandatory members must be implemented, and optional members may or may not be implemented.

The functional block declaration then includes a member list. In this member list, network variables are associated with the abstract member network variables of the profile. These network variables must have been previously declared in the model file. The association between the members of the functional block declaration and the abstract members of the profile is performed with the **implements** keyword.

After the member list, the functional block declaration continues with the name of the functional block itself. A functional block can be a single declaration, or it can be a singly-dimensioned array.

If you do not specify an external name for the functional block, the functional block identifier is limited to 16 characters.

If the **fblock** is implemented as an array, each network variable that is to be referenced by the **fblock** must be declared as an array of at least the same size. When implementing an **fblock** array's member with an array network variable element, the *starting index* of the first network variable array element in the range of array elements must be provided in the **implements** statement. The Neuron C compiler automatically adds the following network variable array elements to the **fblock** array elements, distributing the elements consecutively.

external_name

Defines an optional external name for the functional block.

The external name is part of the device interface that is exposed to network management tools. The external name is limited to 16 characters. You can specify an external name using either the **external_name** or **external_resource_name** keyword. If you do not specify either keyword, the functional block identifier (supplied in the declaration) is used as the default external name.

The **external_name** keyword is used to specify an external name as a string. The string must follow the **external_name** keyword, and must be enclosed in parentheses.

external_resource_name

Defines an optional external name for the functional block. This external name is defined in a language file that is part of a resource file set.

The **external_resource_name** keyword is followed by a scope and index pair (the first number is a scope, followed by a colon character, and the second number is an index) enclosed in parentheses. The scope and index pair identifies a language string in a resource file, which a network management tool can access for a language-dependent name of the functional block. You can use the scope and index pair to reduce memory requirements and to provide language-dependent names for your functional blocks.

Alternatively, you can specify a string argument for the **external_resource_name** keyword. The LonTalk Interface Developer utility uses this string to look up the appropriate string in the resource files that apply to the device. The string must exist in an accessible resource file.

Whether you specify a scope and index pair or a string name, the device interface information uses the scope and index pair rather than the string.

implements

Defines the association between the members of the functional block declaration and the abstract members of the profile.

At a minimum, every *mandatory* abstract member network variable of the profile must be implemented by an actual network variable in the model file. Each network variable (or, in the case of a network variable array, each array element) can implement no more than one profile member, and can be associated with at most one functional block.

implementation_specific

Defines additional network variables in the functional block that are not in the list of optional members of the profile. Such additional network variable members beyond the profile are called *implementation-specific members*.

These extra members are declared in the member list using the **implementation_specific** keyword, followed by a unique index number, and a unique name. Each network variable in a functional profile assigns an index number and a member name to each abstract network variable member of the profile, and the implementation-specific member cannot use any of the index numbers or member names that the profile has already used.

Examples

Example 1: The following example declares a functional block with a single network variable.

```
network output SNVT_amp nvoAmpere;

fblock SFPTopenLoopSensor {
   nvoAmpere implements nvoValue;
} fbAmpereMeter;
```

Example 2: The following example implements the **nvoValue** mandatory network variable of the **SFPTopenLoopSensor** functional profile, and adds an implementation-specific **SNVT_time_stamp** network variable with a member name of **nvoInstall**.

If you include the compiler directive **#pragma enable_sd_nv_names**, the name of the network variable, **nvoInstallDate**, is exposed to the network integrator by means of network variable self-documentation (SD) data and device interface files. In a network management tool, the name **nvoInstall** appears as the member of the functional block, wherever the network tool uses the profile definition.

```
network output SNVT_amp nvoAmpere;
network output polled SNVT_time_stamp nvoInstallDate;

fblock SFPTopenLoopSensor {
   nvoAmpere implements nvoValue;
   nvoInstallDate implementation_specific(128)
         nvoInstall;
} fbAmpereMeter;
```

Example 3: The following example declares a functional block array, and defines an external name for the functional block.

```
#define  NUM_AMMETERS   4

network output SNVT_amp nvoAmpere[NUM_AMMETERS];

fblock SFPTopenLoopSensor {
   nvoAmpere[0] implements nvoValue;
} fbAmpereMeter[NUM_AMMETERS] external_name("AmpereMeter");
```

Functional Block Properties Syntax

fb_properties { *property-reference-list* **}**

property-reference-list :

property-reference-list , property-reference

property-reference

property-reference : *property-identifier* [= *initializer*] [*range-mod*]

property-identifier [*range-mod*] [= *initializer*]

range-mod : **range_mod_string** (*concatenated-string-constant*)

property-identifier : [*property-modifier*] *identifier* **[** *constant-expression* **]**

[*property-modifier*] *identifier*

property-modifier : **static** | **global**

Keywords

fb_properties

Declares a functional block property list.

The functional block property list begins with the **fb_properties** keyword. It contains a list of property references, separated by commas, exactly like the device property list and the network variable property list. Each property reference must be the name of a previously declared CP family or the name of a previously declared configuration network variable.

Following the *property-identifier*, there can be an optional *initializer*, and an optional *range-mod*. These optional elements can occur in either order if both are given. If present, the instantiation initializer for a CP family member overrides any initializer provided at the time of declaration of the family; thus, using this mechanism, some CP family members can be initialized specially, with the remaining family members having a more generic initial value. If a network variable is initialized in multiple places (in other words, in its declaration as well as in its use in a property list), the initializations must match.

range_mod_string

Defines an optional range modification string following the property identifier.

The *range-mod* modifier allows you to specify a range-modification string that modifies the valid range for the configuration property defined by the resource file. The range-modification string can only be used with fixed-point and floating-point types, and consists of a pair of either fixed-point or floating-point numbers delimited by a colon. The first number is the lower limit while the second number is the high limit. If either the high limit or the low limit is the maximum or minimum specified in the configuration property type definition, then the field is empty to specify this.

In the case of a structure or an array, if one member of the structure or array

has a range modification, then all members must have a range modification specified. In this case, each range modification pair is delimited by the ASCII vertical bar character '**|**'. To specify no range modification for a member of a structure (that is, revert to the default for that member), encode the field as '**|**'. Use the same encoding for structure members that cannot have their ranges modified due to their data type. The '**|**' encoding is only allowed for members of structures.

Whenever a member of a structure is not a fixed or floating-point number, its range cannot be restricted. Instead, the default ranges must be used.

In the case of an array, the specified range modifications apply to all elements of the array. For example, to specify a range modification for a 3-member structure where the second member has the default ranges, and the third member only has an upper limit modification, the range modification string is encoded as: **"n:m||:m;"**. Positive values for range modifications and their exponents (if any) are implicit, while negative numbers and negative exponents must be explicitly designated as such with a preceding negative sign '**-**' character. Floating-point numbers use a decimal point '.' character for the decimal point. Fixed-point numbers must be expressed as a signed 32-bit integer. Floating-point numbers must be within the range of an IEEE 32-bit floating-point number. To express an exponent, precede the exponent by an '**e**' or an '**E**' and then follow with an integer value.

A range modification string provided in the instantiation of a CP family member overrides any range modification string provided in the declaration of the CP family.

static | global

The elements of an **fblock** array all share the same set of configuration properties as listed in the associated *fb-property-list*. Without special keywords, each element of the **fblock** array obtains its own set of configuration properties.

Special modifiers can be used to share individual properties among members of the same **fblock** array (through use of the **static** keyword), or among all the functional blocks on the device that have the particular property (through use of the **global** keyword).

Like network variable properties, functional block properties can be shared between two or more functional blocks. The use of the **global** keyword creates a CP family member that is shared among two or more functional blocks. (This global member is a *different* member than a global member that would be shared among network variables, because no single configuration property can apply to both network variables and functional blocks.)

The use of the **static** keyword creates a CP family member that is shared among all the members of a functional block array, but not with any other functional blocks outside the array. See the discussion of functional block properties in the *Neuron C Programmer's Guide* for more information on this topic.

Examples

Example 1: The following example instantiates four heartbeat (**SCPTminSndT**) and four throttle (**SCPTmaxSndT**) CP family members (one

pair for each member of the **nvoData** network variable array), and four offset CP family members (**SCPToffset**), one for each member of each **fblock** array.

It also instantiates a total of two gain control CP family members (**SCPTgain**), one for MyFb1, and one for MyFb2. Finally, it instantiates a single location CP family member (**SCPTlocation**) that is shared by MyFb1 and MyFb2.

```
// CP Family Declarations:
SCPTgain cp_family cpGain;
SCPTlocation cp_family cpLocation;
SCPToffset cp_family cpOffset;
SCPTmaxSndT cp_family cpMaxSendT;
SCPTminSndT cp_family cpMinSendT;

// NV Declarations:
network output SNVT_lev_percent nvoData[4]
               nv_properties {
   cpMaxSendT, // throttle interval
   cpMinSendT // heartbeat interval
};

// Four open loop sensors, implemented as two arrays of
// two sensors, each.  This might be beneficial in that
// this software layout might meet the hardware design
// best, for example with regards to shared and individual
// properties.

fblock SFPTopenLoopSensor {
   nvoData[0] implements nvoValue;
} MyFb1[2]
   fb_properties {
      cpOffset, // offset for each fblock
      static cpGain, // gain shared in MyFb1
      global cpLocation // location shared in all 4
   };

fblock SFPTopenLoopSensor {
   nvoData[2] implements nvoValue;
} MyFb2[2]
   fb_properties {
      cpOffset, // offset for each fblock
      static cpGain, // gain shared in MyFb2
      global cpLocation // location shared in all 4
   };
```

Example 2: This example implements an open loop sensor as an ammeter. The **nvoValue** mandatory network variable is implemented, but no optional network variables are. The **SCPTdefOutput** optional configuration property is implemented, and a second, implementation-specific, **SCPTbrightness** configuration property is also implemented.

The names in the example for the CP families (**cpDefaultOutput** and **cpDisplayBrightness**) have no external relevance; these names are only used within the device's source code in order to reference the configuration property.

```
SCPTdefOutput  cp_family cpDefaultOutput;
SCPTbrightness cp_family cpDisplayBrightness;
```

```
network output SNVT_amp nvoAmpere;
network output polled SNVT_time_stamp nvoInstallDate;

fblock SFPTopenLoopSensor {
   nvoAmpere implements nvoValue;
   nvoInstallDate implementation_specific(128)
         nvoInstall;
} fbAmpereMeter external_name("AmpereMeter")
   fb_properties {
      cpDefaultOutput,        // optional CP
      cpDisplayBrightness = {50.0, 1} // impl-specific
   };
```

Network Variable Syntax

The syntax for declaring a single network variable object is:

network input | **output** [*netvar-modifier*] [*storage-class*] *type*

[*connection-info*] *identifier*

[= *initial-value*] [*nv-property-list*] **;**

The syntax for declaring an array of network variables is:

network input | **output** [*netvar-modifier*] [*storage-class*] *type*

[*connection-info*] *identifier* **[** *array-bound* **]**

[= *initializer-list*] [*nv-property-list*] **;**

The brackets around *array-bound* are shown in **bold** type. The brackets do not, in this case, indicate an optional field. They are a required part of the syntax for declaring an array, and must be entered into the program code.

Network variable arrays can only be single dimension. The *array-bound* must be a constant. Each element of the array is treated as a separate network variable for purposes of events, transmissions on the network, and so on. Therefore, each element counts individually towards the maximum number of network variables on a given device. Each element of the array is a separately bindable network variable.

Keywords

network

Declares a network variable of a specific *type* and with a specific *identifier*.

input | **output**

Defines the direction (input or output) for the network variable, from the point of view of the Echelon Smart Transceiver or Neuron Chip.

The Network Variable Modifier

The optional *netvar-modifier* specification for a network variable includes the following keywords:

sync | synchronized

Specifies that all values assigned to this network variable must be propagated, and in their original order. This flag is passed on to your LonTalk Stack application, and must be enforced by your application.

This keyword is mutually exclusive with the **polled** keyword.

polled

For an output network variable, specifies that the value of the network variable is to be sent *only* in response to a poll request from a device that reads this network variable. When this keyword is omitted for an output network variable, its value is propagated over the network every time the variable is assigned a value. However, any reader device can always poll the outputs of writer devices to which it is connected, whether or not the output is declared as **polled**.

Unlike for native Neuron C, the **polled** network modifier is permitted for input network variables (as well as output network variables) in model files.

The **polled** modifier, when used with the declaration of an input network variable, indicates that the application uses the **LonPollNv()** LonTalk API function with this network variable.

If you use the NodeBuilder Code Wizard to generate your model file, the code wizard does not insert the **polled** modifier for input network variables. You can edit the code produced by the code wizard to add the **polled** modifier.

You can perform all normal network variable operations with a polled input network variable; however, the **LonPollNv()** function requires the network variable to be connected to one or more output network variables. If you call **LonPollNv()** without having made such a connection, you will not receive any data. If you call **LonPollNv()** for a network variable that is not an input network variable and that has not been declared with the **polled** modifier, the **LonPollNv()** function returns an error.

The **polled** modifier can cause an address table entry to be used to allow the input to poll a group connection to the input.

This keyword is mutually exclusive with the **sync** keyword.

changeable_type

Declares that the network variable can have its type changed by a network management tool. The **changeable_type** modifier can only appear once per network variable declaration, and must appear after the **sync** or **polled** modifiers, if either is used.

sd_string (*C-string-const*)

Sets a network variable's self-documentation (SD) string of up to 1023 characters. This modifier can only appear once per network variable declaration. If any of the **sync, polled, or changeable_type** keywords is used, then the **sd_string** must follow these other keywords. Concatenated string constants are permitted. Each variable's SD string can have a maximum length of 1023 bytes.

The use of any of the following Neuron C keywords causes the compiler to take control over the generation of self-documentation strings: **fblock**, **config_prop**, **cp**, **device_properties**, **nv_properties**, **fblock_properties**, or **cp_family**.

In an application that uses compiler-generated SD data, you can still specify additional SD data with the **sd_string()** modifier. The compiler appends this additional SD information to the compiler-generated SD data, but it will be separated from the compiler-generated information with a semicolon. SD data that appears after the semicolon is treated as a comment and is not included in the device's interoperable interface.

The Network Variable Storage Class

Network variables constitute one of the storage classes in Neuron C. The optional *storage-class* specification for a network variable includes the following keywords:

const

Specifies a network variable that cannot be changed by the application program. Output network variables declared with **const** can be placed in PROM or EPROM. Input network variables declared with **const** can be updated over the network, and should therefore be placed in RAM.

When **const** is used with output network variables, the **polled** modifier should also be considered.

Important: If specified, the **const** keyword must appear as the first keyword for the network variable declaration in a model file. For example:

```
const network output polled SNVT_address nvoFileDir;
```

eeprom

Allows the application program to indicate network variables whose values are stored in non-volatile memory and therefore are preserved across power outages.

config

This modifier is obsolete and has been replaced by the **config_prop** keyword.

config_prop | cp

This keyword declares the network variable to be a configuration property.

If no class is specified for a network variable, the network variable is a global variable. Global variables should be stored in RAM and need not be preserved across power outages.

The Network Variable Type

Network variable types serve two purposes. First, typing ensures proper use of the variable in the device's application. Second, typing ensures proper connection of network variables so that a sending device and a receiving device can agree on the representation of data within the network variable. A network variable can be declared using any of the following types:

- A standard network variable type (SNVT) or standard configuration property type (SCPT) defined in the standard resource file. You can use the NodeBuilder Resource Editor to view all available SNVTs and SCPTs, along with their definitions.

Use a SNVT or SCPT if one is available that matches your data because SNVTs and SCPTs can provide interoperability with other devices.

- A user network variable type (UNVT) or user configuration property type (UCPT) defined in a user resource file. You can use the NodeBuilder Resource Editor to create custom UNVTs and UCPTs, and to view the available UNVTs and UCPTs in your resource files. Use a UNVT or UCPT if you cannot find an appropriate SNVT or SCPT for your data.
- Any of the following built-in types (including single-dimension arrays, unions, structures, or named types of the following types):

  [signed] long int
  unsigned long int
  signed char
  [unsigned] char
  [signed] [short] int
  unsigned [short] int
  enum (an **enum** is **int** type)

  In general, built-in types should not be used because they cannot be verified by network management tools when creating connections. Network variables based on built-in types are not interoperable.

The Network Variable Connection Information

The optional *connection-info* specification for a network variable defines options in the network variable table and the SI and SD data for a LonTalk Stack application. If the **nonconfig** keyword is not specified, these connection information assignments can be overridden by a network management tool when a device is installed.

The syntax for the *connection-info* specification is:

bind_info (

[**expand_array_info**]
[**offline**]
[**unackd** | **unackd_rpt** | **ackd** [**(config** | **nonconfig)**]]
[**authenticated** | **nonauthenticated** [**(config** | **nonconfig)**]]
[**priority** | **nonpriority** [**(config** | **nonconfig)**]]
[**rate_est (** *const-expr* **)**]
[**max_rate_est (** *const-expr* **)**]

)

The following keywords can be specified in any order:

expand_array_info

Includes individual names for each element of an array in the device's SI and SD data, and in the device interface file. The names of the array elements have unique identifying characters postfixed. These identifying characters are typically the index of the array element. For example, an **xyz[4]** network variable array becomes four separate **xyz__0**, **xyz__1**, **xyz__2**, and **xyz__3**

network variables.

This keyword is not required for model files. Names of array elements are automatically expanded by the LonTalk Interface Developer compiler.

offline

Specifies that a network management tool must take this device offline, or ensure that the device is already offline, before updating the network variable.

Do not use this feature in the **bind_info** for a configuration network variable that is declared using the **config_prop** or **cp** keyword. Instead, use the **offline** option in the **cp_info**.

unackd | unackd_rpt | ackd [(config | nonconfig)]

Selects the LonTalk protocol service to use for updating this network variable. The allowed types are:

unackd — unacknowledged service; the update is sent once and no acknowledgment is expected.

unackd_rpt — repeated service; the update is sent multiple times and no acknowledgments are expected.

ackd (the default) — acknowledged service with retry; if acknowledgments are not received from all receiving devices before the layer 4 retransmission timer expires, the message is sent again, up to the retry count.

An unacknowledged (**unackd**) network variable uses minimal network resources to propagate its values to other devices. As a result, propagation failures are more likely to occur, and failures are not detected by the device. This class might be used for variables that are updated on a frequent, periodic basis, where loss of an update is not critical, or in cases where the probability of a collision or transmission error is extremely low.

The repeated (**unackd_rpt**) service is typically used when a message is propagated to many devices, and a reliable delivery is required. This service reduces the network traffic caused by a large number of devices sending acknowledgements simultaneously and can provide the same reliability as the acknowledged service by using a repeat count equal to the retry count.

The **config** keyword indicates that this service type can be changed by a network management tool. This option allows the tool to change the service specification during installation. **config** is the default.

The **nonconfig** keyword indicates that this service cannot be changed by a network management tool.

authenticated | nonauthenticated [(config | nonconfig)]

Specifies whether the network variable update requires authentication. With authentication, the identity of the sending device is verified by all receiving devices. Abbreviations for **authenticated** and **nonauthenticated** are **auth** and **nonauth**.

The **config** keyword indicates that this service type can be changed by a network management tool. This option allows the tool to change the service specification during installation. **config** is the default

The **nonconfig** keyword indicates that this service cannot be changed by a network management tool.

A network variable connection is authenticated only if the readers and writers have the authenticated keywords specified. However, if only the originator of a network variable update or poll uses the keyword, the connection is authenticated (although the update does take place). See *Using Authentication for Network Variables* for more information about authentication.

The default is **nonauth (config)**.

You must the acknowledged service with authenticated updates. Do not use the unacknowledged or repeated services.

priority | nonpriority [(config | nonconfig)]

Specifies whether the network variable update has priority access to the communications channel. This field specifies the default value.

All priority network variables in a device use the same priority time slot because each device is configured to have no more than one priority time slot.

The **config** keyword indicates that this service type can be changed by a network management tool. This option allows the tool to change the service specification during installation. **config** is the default

The **nonconfig** keyword indicates that this service cannot be changed by a network management tool.

The default is **nonpriority (config)**.

The **priority** keyword affects output or polled input network variables. When a priority network variable is updated, its value is propagated on the network within a bounded amount of time as long as the device is configured to have a priority slot by a network management tool. The exact bound is a function of the bit rate and priority. The delay before propagation for a **nonpriority** network variable update is unbounded.

rate_est (const-expr)

The estimated sustained update rate, in tenths of updates per second, that the associated network variable is expected to transmit. The allowable value range is from 0 to 18780 (0 to 1878.0 updates per second).

max_rate_est (const-expr)

The estimated maximum update rate, in tenths of messages per second, that the associated network variable is expected to transmit. The allowable value range is from 0 to 18780 (0 to 1878.0 updates per second).

It might not always be possible to determine **rate_est** and **max_rate_est**. For example, update rates are often a function of the particular network where the device is installed. These values can be used by a network management tool to perform network load analysis and are optional.

Although you can specify any value in the range 0 to 18780, not all values are used. The values are mapped into encoded values in the range 0 to 127. Only the encoded values are stored in the device's self-identification (SI) data. The actual value can be reconstructed from the encoded value. If the encoded value is zero, the actual value is undefined. If the encoded value is in the range 1 to 127, the actual value is $a = 2^{(n/8)-5}$, rounded to the nearest tenth. The value a, produced by the formula, is in units of messages per second.

The Network Variable Initializer

initial-value or *initializer-list*	Specifies an initial value (or values) for the network variable. All network variables, especially input network variables, should be initialized to a reasonable default value.

The initial value should be chosen such that if a device is reset, the initial value can be used for subsequent calculations prior to the variable's being updated from the network, and these calculations will not cause the device to create a hazardous condition or to create an error condition. Initializers should not be propagated over the network, regardless of whether the network variables are declared input or output. See *Network Variable and Configuration Property Declarations* for more information about initializers.

Example:

```
network input SNVT_temp nv_temp = 2960;   // 23 C, 73.4 F
```

The Network Variable Property List

A network variable property list declares instances of configuration properties defined by CP family declarations and configuration network variable declarations that apply to a network variable.

The syntax for the *nv-property-list* specification is:

nv_properties { *property-reference-list* **}**

property-reference-list :

property-reference-list , property-reference

property-reference

property-reference :

property-identifier [= *initializer*] [*range-mod*]

property-identifier [*range-mod*] [= *initializer*]

range-mod : **range_mod_string (** *concatenated-string-constant* **)**

property-identifier : [*property-modifier*] *identifier* **[** *constant-expression* **]**

[*property-modifier*] *identifier*

property-modifier : **static** | **global**

The network variable property list begins with the *nv_properties* keyword. It then contains a list of property references, separated by commas, exactly like the device property list and functional block property lists. Each property reference must be the name of a previously declared CP family or the name of a previously declared configuration network variable. The rest of the syntax is very similar to the device property list and functional block property list syntax.

Following the *property-identifier*, there can be an optional *initializer*, and an optional *range-mod*.

You cannot have more than one configuration property of any given SCPT or UCPT type that applies to the same network variable.

Network variable properties can be shared between two or more network variables. The use of the global keyword creates a CP family member that is shared between two or more network variables. The use of the static keyword creates a CP family member that is shared between all the members of a network variable array, but not with any other network variables outside the array.

Example:

```
// CP for heartbeat and throttle (default 1 min each)
SCPTmaxSndT cp_family cpMaxSendT = { 0, 0, 1, 0, 0 };
SCPTminSndT cp_family cpMinSendT = { 0, 0, 1, 0, 0 };

// NV with heartbeat and throttle:
network output SNVT_lev_percent nvoValue
nv_properties {
   cpMaxSendT,
   // override default for minSendT to 30 seconds:
   cpMinSendT = { 0, 0, 0, 30, 0 }
};
```

Configuration Property Syntax

[**const**] *type* **cp_family** [*cp-modifiers*] *identifier*

[**[** *array-bound* **]**] [**=** *initial-value*] **;**

The declaration for a configuration property is similar to a C language `typedef` declaration because no actual variables are created as a result of the declaration. In the case of a type definition, variables are instantiated when the type definition is used in a later declaration that is not, itself, another `typedef`. At that time, variables are instantiated, which means that variables are declared and memory is allocated for and assigned to the variables. The variables can then be used in later expressions in the executable code of the program.

The instantiation of CP family members occurs when the CP family declaration's identifier is used in a property list. However, a configuration network variable is already instantiated at the time it is declared. For a configuration network variable, the property list serves only to identify the association between the configuration property and the object or objects to which it applies.

Configuration properties can apply to a device, one or more functional blocks, or one or more network variables. In each case, a configuration property is made to apply to its respective objects through a property list.

The brackets around *array-bound* are shown in **bold** type. The brackets do not, in this case, indicate an optional field. They are a required part of the syntax for declaring an array, and must be entered into the program code.

Keywords

const

Declares the configuration property as a constant, so that it is allocated in non-modifiable memory.

In general, a configuration property can be modifiable, either from within the LonTalk Stack application or from a network management tool, and thus is not declared with this keyword.

cp_family

Declares the configuration property as part of a configuration file.

The **cp_family** declaration is repeatable. The declaration can be repeated two or more times, and, as long as the duplicated declarations match in every regard, the compiler treats these as a single declaration.

The alternative to declaring a configuration property as part of a configuration file is to declare a configuration network variable, as described in *Declaring a Configuration Network Variable*.

The Configuration Property Type

The type for a CP family cannot be a built-in Neuron C type such as **int** or **char**. Instead, the declaration must use a standard configuration property type (SCPT) or a user configuration property type (UCPT) defined in a resource file. There are several hundred SCPT definitions available, and you can create your own types using UCPTs. The SCPT definitions are stored in the **standard.typ** file, which is part of the standard resource file. There can be many similar resource files containing UCPT definitions, and these are managed by the NodeBuilder Resource Editor.

In contrast to an ANSI C **typedef**, a configuration property type also defines a standardized semantic meaning for the type. The configuration property definition in a resource file contains information about the default value, minimum and maximum valid values, a designated (optional) invalid value, and language string references that permit localized descriptive information, additional comments, and units strings to be associated with the configuration property type.

The Configuration Property Modifiers

The configuration property modifiers are an optional part of the CP family and configuration network variable declarations.

The syntax for the *cp-modifiers* specification is:

cp-modifiers :	[**cp_info (** *cp-option-list* **)**] [*range-mod*]
cp-option-list :	*cp-option-list* , *cp-option*
	cp-option
cp-option :	**device_specific** | **manufacturing_only** | **object_disabled**
	| **offline** | **reset_required**
range-mod :	**range_mod_string (** *concatenated-string-constant* **)**

The *cp-option* keywords can occur in any order. There must be at least one keyword. For multiple keywords, a keyword must not appear more than once, and keywords must be separated by commas.

The cp-modifiers begin with the **cp_info** keyword followed by a parenthesized list of one or more of the following option keywords:

device_specific

Specifies a configuration property that is always read from the device instead

of relying upon the value in the device interface file or a value stored in a network database. This specification is used for configuration properties that must be managed by the device, such as a setpoint that is updated by a local operator interface on the device. This option requires the CP family or configuration property network variable to be declared as **const**.

manufacturing_only

Specifies a factory setting that can be read or written when the device is manufactured, but is not normally (or ever) modified in the field. In this way, a standard network management tool can be used when a device is manufactured to calibrate the device, whereas a field installation tool would observe the flag in the field and prevent updates or require a password to modify the value.

object_disabled

Specifies that a network management tool must disable the functional block containing the configuration property, take the device offline, or ensure that the functional block is already disabled or the device is already offline, before modifying the configuration property.

After the network management tool modifies the configuration property, the application might have to take some action based on the modified value. The application should check the configuration property value in the **LonResetOccurred()** and **LonOnline()** callback handler functions.

offline

Specifies that a network management tool must take this device offline before modifying the configuration property.

After the network management tool modifies the configuration property, the application might have to take some action based on the modified value. The application should check the configuration property value in the **LonResetOccurred()** and **LonOnline()** callback handler functions.

reset_required

Specifies that a network management tool must reset the device after changing the value of the configuration property.

After the network management tool modifies the configuration property, the application might have to take some action based on the modified value. The application should check the configuration property value in the **LonResetOccurred()** callback handler function.

range_mod_string

Defines an optional range modification string following the property identifier.

The *range-mod* modifier allows you to specify a range-modification string that modifies the valid range for the configuration property defined by the resource file. The range-modification string can only be used with fixed-point and floating-point types, and consists of a pair of either fixed-point or floating-point numbers delimited by a colon. The first number is the lower limit while the second number is the high limit. If either the high limit or the low limit is the maximum or minimum specified in the configuration property type definition, then the field is empty to specify this.

In the case of a structure or an array, if one member of the structure or array

has a range modification, then all members must have a range modification specified. In this case, each range modification pair is delimited by the ASCII vertical bar character '**|**'. To specify no range modification for a member of a structure (that is, revert to the default for that member), encode the field as '**|**'. Use the same encoding for structure members that cannot have their ranges modified due to their data type. The '**|**' encoding is only allowed for members of structures.

Whenever a member of a structure is not a fixed or floating-point number, its range cannot be restricted. Instead, the default ranges must be used.

In the case of an array, the specified range modifications apply to all elements of the array. For example, to specify a range modification for a 3-member structure where the second member has the default ranges, and the third member only has an upper limit modification, the range modification string is encoded as: **"n:m||:m;"**. Positive values for range modifications and their exponents (if any) are implicit, while negative numbers and negative exponents must be explicitly designated as such with a preceding negative sign '**-**' character. Floating-point numbers use a decimal point '.' character for the decimal point. Fixed-point numbers must be expressed as a signed 32-bit integer. Floating-point numbers must be within the range of an IEEE 32-bit floating-point number. To express an exponent, precede the exponent by an '**e**' or an '**E**' and then follow with an integer value.

A range modification string provided in the instantiation of a CP family member overrides any range modification string provided in the declaration of the CP family.

The Configuration Property Initializer

The *initial-value* in the declaration of a CP family is optional. If *initial-value* is not provided in the declaration, the default value specified by the resource file is used. The *initial-value* given is an initial value for a single member of the family, but the LonTalk Interface Developer utility replicates the initial value for each instantiated family member. See *Network Variable and Configuration Property Declarations* for more information about initializers.

Initialization for a CP family member is performed according to the following rules:

1. If the configuration property is initialized explicitly in the instantiation, then this is the initial value that is used.
2. If the configuration property is initialized explicitly in the CP family declaration, then the family initializer is used.
3. If the configuration property applies to a functional block, and the functional profile that defines the functional block specifies a default value for the associated configuration property member, then the functional profile default is used.
4. If the configuration property type for the configuration property defines a default value, then that default value is used as the initial value. This rule does not apply for a configuration property type that is type-inheriting; see *Inheriting a Configuration Property Type* for more information.

5. If no initial value is available from any of the preceding rules, a value of all zeros is used.

The compiler uses the first rule in this list that applies to the configuration property.

These initialization rules are used to set the initial value that are loaded in the value file from the linked image, as well as the value file stored in the device interface file. A network management tool can use the initial value as a default value, and might at times reset the configuration properties (or a subset of them) back to the default values. Consult the documentation of the particular network management tool, for example, the *OpenLNS Commissioning Tool User's Guide*, for more information on the use of configuration property default values.

Declaring a Configuration Network Variable

The configuration network variable declaration syntax is similar to the declaration syntax of a non-configuration network variable.

The declaration of a configuration network variable is distinct from other network variable declarations by the inclusion of the **config_prop** keyword following the type of the network variable declaration. The **config_prop** keyword can be abbreviated as **cp**.

The syntax for declaring a configuration network variable is:

network input [*netvar-modifier*] [*storage-class*] *type*

config_prop [*cp-modifiers*]

[*connection-info*] *identifier* [**[** *array-bound* **]**]

[= *initial-value*] **;**

The *netvar-modifier*, *storage-class*, *connection-info*, *array-bound*, and *initial-value* portions of this syntax are described in *Network Variable Syntax*, and they apply equally to a configuration network variable as they do to any other network variable.

Similar to the configuration CP family members, configuration network variables must be declared with a *type* that is defined by a standard configuration property type (SCPT) or a user configuration property type (UCPT) defined within a resource file.

The *cp-modifiers* clause that can optionally follow the **config_prop** keyword is described in *The Configuration Property Modifiers*.

Example:

```
network input SCPTupdateRate config_prop nciUpdateRate;
network input SCPTbypassTime cp nciBypassTime = ...
```

Defining a Device Property List

A device property list declares instances of configuration properties defined by CP family declarations and configuration network variables declarations that apply to a device.

The syntax for declaring a device property list is:

device_properties { *property-reference-list* **} ;**

property-reference-list :

property-reference-list , *property-reference*

property-reference

property-reference :

property-identifier [= *initializer*] [*range-mod*]

property-identifier [*range-mod*] [= *initializer*]

range-mod : **range_mod_string (** *concatenated-string-constant* **)**

property-identifier : *identifier* [*constant-expression*]

identifier

The device property list begins with the **device_properties** keyword. It then contains a list of property references, separated by commas. Each property reference must be the name of a previously declared CP family or the name of a previously declared configuration network variable. If the network variable is an array, only a single array element can be chosen as the device property, so an array index must be given as part of the property reference in that case.

Following the *property-identifier*, there can be an optional *initializer*, and an optional *range-mod*.

The device property list appears at file scope. This is the same level as a function declaration, a task declaration, or a global data declaration. A model file can have multiple device property lists. These lists are merged together by the LonTalk Interface Developer utility to create one combined device property list. However, you cannot have more than one configuration property of any given SCPT or UCPT type that applies to the device.

Example 1:

```
SCPTlocation cp_family cpLocation;

device_properties {
   cpLocation = { "Unknown" }
};
```

Example 2:

```
network input SCPTlocation cp cpLocation[5];

device_properties {
   cpLocation[0] = { "Unknown" }
};
```

Example 3:

```
UCPTsomeDeviceCp cp_family cpSomeDeviceCp;
SCPTlocation cp_family cpLocation = {""};

device_properties {
   cpSomeDeviceCp,
   cpLocation = { "Unknown" }
      // This instantiation overrides the
```

```
        // empty string initializer with its own
};
```

Message Tag Syntax

msg_tag [*connection-info*] *tag-identifier* [, *tag-identifier* ...] **;**

Keywords

The *connection-info* field is an optional specification for connection options, and includes the following keywords:

msg_tag

Declares a message tag with the specified *tag-identifier*.

bind_info (*options* **)**

The following connection *options* apply to message tags:

nonbind

Specifies a message tag that carries no addressing information and does not consume an address table entry. It is used as a destination tag when creating explicitly addressed messages.

rate_est (*const-expr* **)**

The estimated sustained message rate, in tenths of messages per second, that the associated message tag is expected to transmit. The allowable value range is from 0 to 18780 (0 to 1878.0 messages/second).

max_rate_est (*const-expr* **)**

The estimated maximum message rate, in tenths of messages per second, that the associated message tag is expected to transmit. The allowable value range is from 0 to 18780 (0 to 1878.0 messages/second).

It might not always be possible to determine **rate_est** and **max_rate_est**. For example, update rates are often a function of the particular network where the device is installed. These values can be used by a network management tool to perform network load analysis and are optional.

Although you can specify any value in the range 0 to 18780, not all values are used. The values are mapped into encoded values in the range 0 to 127. Only the encoded values are stored in the device's self-identification (SI) data. The actual value can be reconstructed from the encoded value. If the encoded value is zero, the actual value is undefined. If the encoded value is in the range 1 to 127, the actual value is $a = 2^{(n/8)-5}$, rounded to the nearest tenth. The value a, produced by the formula, is in units of messages per second.

Appendix D

LonTalk API

This Appendix describes the API functions, event handler functions, and callback handler functions that are included with the LonTalk API. It also describes the operating system abstraction layer (OSAL) functions.

Introduction

The LonTalk API provides the functions that you call from your LonTalk Stack application to send and receive information to and from a LONWORKS network. The API also defines the event handler functions and callback handler functions that your LonTalk Stack application must provide to handle LONWORKS events from the network and LonTalk Host stack. Because each LonTalk Stack application handles these events and callbacks in its own specific way, you need to modify the event and callback handler functions.

To provide operating system independence, the LonTalk API includes the operating system abstraction layer (OSAL) API. To provide non-volatile data independence, the LonTalk API provides two complete (and one skeletal) non-volatile data driver (NVD) APIs.

Typically, you use the LonTalk API functions with the event handler functions for LonTalk Stack device initialization and for sending and receiving network variable updates. See Chapter 8, *Developing a LonTalk Stack Device Application*, for more information about using these functions.

The implementations of the LonTalk API are contained in the following files:

- **FtxlHandlers.c**, stub functions for the LonTalk API event handler functions and callback handler functions
- **FtxlNvdFlashDirect.c**, the direct-access model for working with flash memory
- **FtxlNvdFlashFs.c**, the file-system model for working with flash memory
- **FtxlNvdUserDefined.c**, a skeletal implementation for a user-defined non-volatile memory model

See *LonTalk API Files* for a list of the files that are included with the LonTalk Stack.

The LonTalk API, Event Handler Functions, and Callback Handler Functions

This section provides an overview of the LonTalk API functions, event handler functions, and callback handler functions. For detailed information about these functions, see the HTML API documentation and the API source code:

- HTML API documentation: **Start → Programs → Echelon LonTalk Stack Developer's Kit → Documentation → API Reference**
- API source code for the example applications: **Start → Programs → Echelon LonTalk Stack Developer's Kit→ LonTalk Stack Example Applications**

LonTalk API Functions

The LonTalk API includes functions for managing network data, the LonTalk Stack device, and non-volatile data.

Commonly Used LonTalk API Functions

The following table lists API functions that you will most likely use in your LonTalk Stack application.

Function	Description
LonEventPump()	Processes any messages received by the LonTalk host stack. If messages are received, it calls the appropriate event handler functions. See *Periodically Calling the Event Pump* for more information about this function.
LonExit()	Stops the LonTalk host stack for an orderly shutdown of the LonTalk Stack device.
LonInit()	Initializes the LonTalk API and the LonTalk host stack. This function downloads LonTalk Stack device interface data from the LonTalk Stack application to the Echelon Smart Transceiver or Neuron Chip. The LonTalk Stack application must call **LonInit()** once on startup.
LonPropagateNv()	Propagates a network variable value to the network. This function propagates a network variable if *all* of the following conditions are met: • The network variable is declared with the **output** modifier • The network variable must be bound to the network • The network variable must not be declared with the **polled** modifier.

Other LonTalk API Functions

The following table lists other LonTalk API functions that you can use in your LonTalk Stack application. These functions are not typically used by most LonTalk Stack applications, or are used only for specific application functionality (for example, including support for changeable-type network variables).

Function	Description
LonFreeNvTypeData()	Frees internal buffers that were allocated by a call to the **LonQueryNvType()** function.
LonGetDeclaredNvSize()	Gets the declared size for a network variable.
LonGetNvValue()	Gets a pointer to the value for a network variable. This function is required for dynamic network variables, but could be used for any network variable.

Function	Description
LonGetUniqueId()	Gets the unique ID (Neuron ID) value of the Echelon Smart Transceiver or Neuron Chip.
LonGetVersion()	Gets the version number of the LonTalk API.
LonPollNv()	Requests a network variable value from the network. A LonTalk Stack application can call **LonPollNv()** to request that another LONWORKS device (or devices) send the latest value (or values) for network variables that are bound to the specified input variable. To be able to poll a network variable, it must be declared in the model file as an input network variable and include the **polled** modifier.
LonQueryNvType()	Queries the information about a network variable.
LonSendServicePin()	Broadcasts a service-pin message to the network. The service-pin message is used during configuration, installation, and maintenance of a LONWORKS device. The LonTalk host stack automatically broadcasts service-pin messages when needed.

Application Messaging API Functions

The following table lists the LonTalk API functions that are used for implementing application messaging and for responding to an application message. Application messages can be used to implement a standard interface or a proprietary interface that does not need to interface to devices from other manufacturers. Support for application messaging is optional.

Function	Description
LonReleaseCorrelator()	Releases a request correlator for an application message without sending a response.
LonSendMsg()	Sends an application message.
LonSendResponse()	Sends an application message response to a request message. The LonTalk Stack application calls **LonSendResponse()** in response to a **LonMsgArrived()** event handler function.

Non-Volatile Data API Functions

The following table lists the LonTalk API functions that are used for implementing support for non-volatile data.

Function	Description
LonNvdAppSegmentHasBeenUpdated()	Indicates that the application data segment in non-volatile data memory has been updated.
LonNvdFlushData()	Requests that the LonTalk host stack flush all non-volatile data to persistent memory.
LonNvdGetMaxSize()	Gets the number of bytes required to store persistent data.

Extended API Functions

The LonTalk API includes an extended API that provides additional local network management commands listed in the following table.

Function	Description
LonClearStatus()	Clears the status statistics on the LonTalk Stack device.
LonGoConfigured()	Sets the state for the LonTalk Stack device as configured.
LonGoOffline()	Sets the state for the LonTalk Stack device as offline.
lonGoOnline()	Sets the state for the LonTalk Stack device as online.
LonGoUnconfigured()	Sets the state for the LonTalk Stack device as unconfigured.
LonMtIsBound()	Queries whether a message tag is bound.
LonNvIsBound()	Queries whether a network variable is bound.
LonQueryAddressConfig()	Queries configuration data for the LonTalk Stack device's address table.
LonQueryAliasConfig()	Queries configuration data for the LonTalk Stack device's alias table.
LonQueryConfigData()	Queries local configuration data on the LonTalk Stack device.
LonQueryDomainConfig()	Retrieves a copy of the local domain table record from the LonTalk Stack device.
LonQueryNvConfig()	Queries configuration data for theLonTalk Stack device's network variable table.
LonQueryStatus()	Requests local status and statistics.

Function	Description
LonQueryTransceiverStatus()	Requests the local status of the Echelon Smart Transceiver or Neuron Chip.
LonSetNodeMode()	Sets the operating mode for the LonTalk Stack device: • Online: An online device executes its application and responds to all network messages. • Offline: An offline device does not execute its application or respond to network messages. It will respond to network management messages. • Configured: The device is ready for network operation. • Unconfigured: The device is not ready for network operation.
LonUpdateAddressConfig()	Sets configuration data for the LonTalk Stack device's address table.
LonUpdateAliasConfig()	Sets configuration data for the LonTalk Stack device's alias table.
LonUpdateConfigData()	Sets configuration data on the LonTalk Stack device.
LonUpdateDomainConfig()	Sets a domain table record on the LonTalk Stack device.
LonUpdateNvConfig()	Sets configuration data for the LonTalk Stack device's network variable table.

Event Handler Functions

The LonTalk API provides event handler functions for managing network and device events.

Commonly Used Event Handler Functions

The following table lists the event handler functions that you will most likely need to define so that your application can perform application specific processing for certain LONWORKS events. You do not need to modify these callback functions if you have no application-specific processing requirements.

Function	Description
LonNvUpdateCompleted()	Indicates that either an update network variable or a poll network variable call is completed.

Function	Description
LonNvUpdateOccurred()	Indicates that a network variable update request from the network has been processed by the LonTalk API. This call indicates that the network variable value has already been updated, and allows your host application to perform any additional processing, if necessary.
LonOffline()	A request from the network that the device go offline. Installation tools use this message to disable application processing in a device. An offline device continues to respond to network management messages, but the interaction between the application and the control network is suspended. When this function is called, the Echelon Smart Transceiver or Neuron Chip is already offline and the LonTalk Stack application need only take application-specific action.
LonOnline()	A request from the network that the device go online. Installation tools use this message to enable application processing in a device. When this function is called, the Echelon Smart Transceiver or Neuron Chip is already online and the LonTalk Stack application need only take application-specific action.
LonReset()	A notification that the device has been reset.
LonServicePinHeld()	An indication that the service pin on the device has been held for some number of seconds (default is 10 seconds). Use it if your application needs notification of the service pin's being held.
LonServicePinPressed()	An indication that the service pin on the device has been pressed. Use it if your application needs notification of the service pin's being pressed.
LonWink()	A wink request from the network. Installation tools use the Wink message to help installers physically identify devices. When a device receives a Wink message, it should provide some visual, audio, or other indication for an installer to be able to physically identify this device.

Dynamic Network Variable Event Handler Functions

The following lists the event handler functions that are called by the LonTalk API to process dynamic network variables. See *Handling Dynamic Network Variables* for more information about using these functions.

Function	Description
LonNvAdded()	Indicates that a dynamic network variable has been added.
LonNvDeleted()	Indicates that a dynamic network variable has been deleted.
LonNvTypeChanged()	Indicates that one or more attributes of a dynamic network variable have changed.

Application Messaging Event Handler Functions

The following table lists the event handler functions that are called by the LonTalk API for application messaging transactions. Customize these functions if you use application messaging in your LonTalk Stack device. Application messaging is optional.

If you choose not to support application messaging, you do not need to customize these functions.

Function	Description
LonMsgArrived()	Indicates that an application message has arrived from the network to be processed. This function performs any application-specific processing required for the message. If the message is a request message, the function must deliver a response using the **LonSendMsgResponse()** function. Application messages are always delivered to the application, regardless of whether the message passed authentication. The application decides whether authentication is required for a message.
LonMsgCompleted()	Indicates that message delivery, initiated by a **LonSendMsg()** call, was completed. If a request message has been sent, this event handler is called only after all responses have been reported by the **LonResponseArrived()** event handler.
LonResponseArrived()	Indicates that an application message response has arrived from the network. This function performs any application-specific processing required for the message.

Non-Volatile Data Event Handler Functions

The LonTalk API provides the event handler function listed in the following table to support non-volatile data.

Function	Description
LonNvdStarvation()	Indicates that a write request to non-volatile data has taken more than 60 seconds. The application should call the **LonNvdFlushData()** API function to ensure that non-volatile data is written.

LonTalk Stack Callback Handler Functions

In addition to providing event handler functions, the LonTalk API also provides callback handler functions, mainly for managing memory on the LonTalk Stack device.

Commonly Used Callback Handler Functions

In addition to processing events, the LonTalk API provides the callback handler functions listed in the following table.

Function	Description
LonGetCurrentNvSize()	Indicates a request for the network variable size. The LonTalk Host stack calls this callback handler function to determine the current size of a changeable-type network variable. For non-changeable-type network variables, this function should return the value of the **LonGetDeclaredNvSize()** function. For changeable-type network variables, you must modify this function in the **FtxlHandlers.c** file.
LonEventReady()	Indicates that a network event is ready to be processed. The LonTalk Host stack calls this callback handler function to indicate that a network event is ready to be processed, and that the main application should call the **LonEventPump()** function. However, the **LonEventReady()** function should not call the **LonEventPump()** function directly. Typically, the **LonEventReady()** callback signals an operating system event that the main application task waits upon. When the main application task wakes up, it should call the **LonEventPump()** function.
LonGetMyIpAddress()	Gets the IP address and port number of an IP-852 interface. This method has the following syntax: void **LonGetMyIpAddress**(int *pAddress, int *pPort);

Function	Description
LonGetMyNetworkInterface()	Gets the name of the network interface used to open the native LonTalk interface. This method returns the name of the network interface, such as the serial port name. This method has the following syntax: const char ***LonGetMyNetworkInterface**(void);

Direct Memory Files Callback Handler Functions

The LonTalk API provides the callback handler functions listed in the following table to support the direct memory files (DMF) feature. These functions rely on utility functions generated by the LonTalk Interface Developer utility.

Function	Description
LonMemoryRead()	Indicates a request to read memory in the LonTalk Stack device's memory space.
LonMemoryWrite()	Indicates a request to write memory in the LonTalk Stack device's memory space.

Non-Volatile Data Callback Handler Functions

The following table lists the callback handler functions that support non-volatile data. For the functions listed in the table, the LonTalk Interface Developer utility generates the following callback handler functions, which are also listed in the table:

- **LonNvdDeserializeSegment()**
- **LonNvdGetApplicationSegmentSize()**
- **LonNvdSerializeSegment()**

The remaining non-volatile data callback handler functions are implemented in the **FtxlFlashDirect.c** and **FtxlFlashFs.c** files.

Function	Description
LonNvdClose()	Indicates a request to close a non-volatile data segment.
LonNvdDelete()	Indicates a request to delete a non-volatile data segment.

Function	Description
LonNvdDeserializeSegment()	Indicates a request to update the LonTalk Stack device's control structures from the serialized application's data segment.
LonNvdEnterTransaction()	Indicates a request to begin a transaction for the non-volatile data segment.
LonNvdExitTransaction()	Indicates a request to complete a transaction for the non-volatile data segment.
LonNvdGetApplicationSegmentSize()	Indicates a request to determine the number of bytes required to store the application's non-volatile data segment.
LonNvdIsInTransaction()	Indicates a request to determine if a transaction for the non-volatile data segment was in progress during the device's previous shutdown.
LonNvdOpenForRead()	Indicates a request to open a non-volatile data segment for reading.
LonNvdOpenForWrite()	Indicates a request to open a non-volatile data segment for writing.
LonNvdRead()	Indicates a request to read a section of a non-volatile data segment.
LonNvdWrite()	Indicates a request to write a section of a non-volatile data segment.
LonNvdSerializeSegment()	Indicates a request to create a serialized image of the application's non-volatile data segment.

The Operating System Abstraction Layer

The LonTalk Stack includes an operating system abstraction layer (OSAL), which allows the LonTalk host stack and LonTalk Stack applications to be ported to any operating system that is supported for the embedded processor.

Example OSAL implementations are included for Linux and Windows. The OSAL is provided as source code so that you can modify either of the implementations to support other operating systems.

For detailed information about the OSAL, see the HTML API documentation and the API source code:

- HTML API documentation: **Start → Programs → Echelon LonTalk Stack → Documentation → API Reference**
- API source code for the example applications: **Start → Programs → Echelon LonTalk Stack → Source Code**

The following sections provide an overview of the functions that the OSAL provides.

Managing Critical Sections

To manage critical sections, the OSAL provides the functions listed in the following table:

Function	Description
OsalCreateCriticalSection()	Creates a critical section.
OsalDeleteCriticalSection()	Deletes a critical section.
OsalEnterCriticalSection()	Enters a critical section.
OsalLeaveCriticalSection()	Leaves a critical section.

Managing Binary Semaphores

To manage binary semaphores, the OSAL provides the functions listed in the following table:

Function	Description
OsalCreateBinarySemaphore()	Creates a binary semaphore.
OsalDeleteBinarySemaphore()	Deletes a binary semaphore.
OsalReleaseBinarySemaphore()	Releases a binary semaphore.
OsalWaitForBinarySemaphore()	Waits for binary semaphore.

Managing Operating System Events

To manage operating system events, the OSAL provides the functions listed in the following table:

Function	Description
OsalCreateEvent()	Creates an event.
OsalDeleteEvent()	Deletes an event.
OsalSetEvent()	Sets an event.
OsalWaitForEvent()	Waits for an event.

Managing System Timing

To manage system timing, the OSAL provides the functions listed in the following table:

Function	Description
OsalGetTickCount()	Gets the current system tick count.
OsalGetTicksPerSecond()	Gets the number of ticks in a second.

Managing Operating System Tasks

To manage operating system tasks or threads, the OSAL provides the functions listed in the following table.

Function	Description
OsalCreateTask()	Creates a task.
OsalCloseTaskHandle()	Closes the handle for a task.
OsalGetTaskId()	Gets the task ID of the current task.
OsalGetTaskIndex()	Gets the task index of the current task.
OsalSleep()	Causes a task to sleep for a specified number of ticks.
OsalTaskEntryPoint()	Sets the entry point for a task.

Debugging Operating System Functions

To provide debugging capability for the OSAL, including tracing and statistics, the OSAL provides the functions listed in the following table.

Function	Description
OsalClearStatistics()	Clears the current operating system statistics.
OsalGetLastOsError()	Gets the most recent error from the operating system.
OsalGetStatistics()	Gets operating system statistics.
OsalGetTraceLevel()	Gets the current OSAL tracing level.
OsalSetTraceLevel()	Sets the OSAL tracing level.

Appendix E

Determining Memory Usage for LonTalk Stack Applications

This Appendix describes how much volatile and non-volatile memory a LonTalk Stack application requires, and how to determine the application's memory requirements.

Overview

The LonTalk Host stack allocates memory dynamically, so a direct measurement of the memory usage might lead to an underestimate for memory usage, especially for peak usage conditions. This appendix provides both static code analysis and runtime measurements so that you can calculate more reliable memory usage estimates.

Memory Use for Code

The following table lists the estimated memory required for the LonTalk Stack code. The values will vary depending on the processor, operating system, and compiler that you use.

Code Type	**Bytes Required**
Native LonTalk protocol stack and LonTalk API	750 KB
IP-852 LonTalk protocol stack and LonTalk API	855 KB
Native LonTalk to IP-852 Router	880 KB
Serial MIP driver	70 KB
SimpleDevice and *SimpleIp852Device* example applications	30 KB
Ip852Router example application	15 KB

Memory Use for Transactions

The LonTalk host stack allocates memory for transactions at runtime, as they are needed. On the Stack Configuration page of the LonTalk Interface Developer utility, you can specify maximum allowed values for the number of simultaneous receive transactions and for the number of simultaneous transmit transactions. These values limit the amount of memory that the LonTalk host stack allocates for transactions.

The following table lists the estimated amount of memory required for each type of transaction. .

Transaction Type	**Bytes Required**
Transmit transaction	196
Receive transaction	400

Memory Use for Buffers

The Buffer Configuration page of the LonTalk Interface Developer utility allows you to specify the number of input, output, and priority output application buffers that your LonTalk Stack application should use. The values that you specify in the utility are defined in the **FtxlDev.h** file that the utility generates.

The LonTalk host stack uses the number of application buffers that you specify to allocate memory for both the application buffers and related internal buffers. Some of the internal buffers are allocated in advance, and some are allocated on an as-needed basis.

The following table lists the estimated amount of memory required for each type of application buffer.

Application Buffer Type	Bytes Required
Input buffer	1710
Output nonpriority buffer	1118
Output priority buffer	1118

The default numbers for each type of buffer are: 5 input buffers, 5 output nonpriority buffers, and 1 output priority buffer. The RAM usage for the default number of application buffers is approximately 15 KB.

Memory for LONWORKS Resources

Each LonTalk Stack device uses LONWORKS resources, such as network variables defined for the device, address table entries, and aliases supported by the device.

The LonTalk host stack allocates memory only for resources that are in use. For example, it allocates memory for address table entries only if the address is bound. To calculate maximum memory requirements, assume that all resources are in use.

The following table lists the estimated amount of memory required for each type of LONWORKS resource. For example, network variables can vary in their actual sizes, so the table uses an average value.

Resource Type	Bytes Required
Static network variable	320 + *SD_length* + *NV_length*
Dynamic network variable	331 + *SD_length* + *NV_length*
Alias	220
Address table entry	67

Resource Type	Bytes Required
Notes: • *SD_length* is the length of the self-documentation string for the network variable • *NV_length* is the declared size of the network variable (for changeable-type network variables, *NV_length* is the maximum size of the network variable)	

In addition to RAM, LONWORKS resources also require memory for constant data. This constant data must be included in both the total RAM size and the total flash memory size, because all of the constant data is typically loaded from flash memory into RAM.

The following table lists the estimated amount of flash memory required for each type of LONWORKS resource.

Resource Type	Bytes Required
Static network variable	24 + *SD_length* + *NV_name_length*
Dynamic network variable	*Dyn_NV_count*
Alias	*Alias_count*
Address table entry	*Address_count*
Notes: • *SD_length* is the length of the self-documentation string for the network variable • *NV_name_length* is the length of the network variable's name, as defined in the device's model file • *Dyn_NV_count* is the number of dynamic network variables that are defined for the application • *Alias_count* is the number of aliases that are defined for the application • *Address_count* is the number of address table entries that are defined for the application	

In addition to storing constant data, flash memory stores non-volatile data for the application, as described in *Memory for Non-Volatile Data*.

Memory for Non-Volatile Data

A LonTalk Stack application typically has some non-volatile data that it must maintain across device reset (see *Providing Persistent Storage for Non-Volatile Data*). The LonTalk host stack stores only non-volatile data that is in use. For example, it does not store address table and alias table entries that are not used. Therefore, the actual amount of non-volatile memory used can be smaller than the maximum amount required. The example direct flash implementation of the

non-volatile data functions calculates the maximum use configuration, and reserves flash memory space so that if one segment grows, it does not interfere with other segments.

This section describes the amount of non-volatile data space required for the following application elements:

- Network image (**LonNvdSegNetworkImage**)
- Node definition (**LonNvdSegNodeDefinition**)
- Application data (**LonNvdSegApplicationData**)

The flash memory implementation in the **FtxlNvdFlashDirect.c** file requires that each data segment begin on a flash sector boundary. Depending on the flash sector size, this requirement can increase the total flash memory needed for the application.

The following table describes the amount of non-volatile memory required for the network image.

Network Data	**Bytes Required**
Header	16
Overhead	102
Domain	21 (for each domain)
Network variables and aliases	15 (for each network variable [static or dynamic] and each alias)
Address table	11 (for each address table entry)

The following table describes the amount of non-volatile memory required for the node definition.

Node Data	**Bytes Required**
Header	16
Overhead	100
Node self-documentation string length	*Node_SD_length*
Static network variable self-documentation string length	*NV_SD_length*
Network variables	37 (for each network variable [static or dynamic])

Node Data	Bytes Required
Notes: • *Node_SD_length* is the length of the self-documentation string for the node • *NV_SD_length* is the length of the self-documentation string for all network variables (both static and dynamic)	

The following table describes the amount of non-volatile memory required for the application data.

Application Data	Bytes Required
Header	16
Configuration Network Variables	$\sum_j \left(CPNVlen_j \right)$
File-based CPs	*File_length*
Application-specific data	*Data_length*
Notes: • File-based CPs are configuration properties that are defined in configuration files • $CPNVlen_j$ is the configuration network variable length of a specific configuration NV value – the application data includes the sum of the configuration NV lengths of all configuration NV values • *File_length* is the size of the writeable configuration file for the configuration properties • *Data_length* is the length of any addition application-specific data	

Memory Usage Examples for Data

The following table shows the approximate amount of RAM that is required for various example LonTalk Stack applications. Each row of the table represents a different application by varying the number of network variables, transmit transactions, receive transactions, aliases, and address table entries. The values for all columns except the network variable column represent values calculated by the LonTalk Interface Developer utility.

Note: The listed amounts of memory are based on FTXL device application requirements. These numbers may vary for LonTalk Stack device applications, and the may differ based on the host processor and the compliler.

The table assumes that each network variable has a length of 2 bytes, and has a 5-byte self-documentation string associated with it. The table also assumes the default number of application buffers (5 input buffers, 5 output nonpriority buffers, and 1 output priority buffer). Varying the number of application buffers

does not siginificantly alter the amount of RAM that the application requires. The number of buffers can affect the application's performance.

You can observe that as the number of network variables for the LonTalk Stack application grows, the RAM requirement grows significantly. These memory requirements do not include the requirements for application-specific data.

Number of Network Variables	Number of Transmit Transactions	Number of Receive Transactions	Number of Aliases	Number of Address Table Entries	RAM Required for Data (in KB)
10	15	20	3	15	202
100	20	20	33	20	228
250	50	20	83	50	277
500	101	20	166	101	361
1000	203	25	333	203	528
2000	407	50	666	407	870
4000	814	100	1333	814	1552

Appendix F

Downloading a LonTalk Stack Application Over the Network

This Appendix describes considerations for designing a LonTalk Stack application that allows application updates over the network.

Overview

For a Neuron-hosted device, you can update the application image over the network using OpenLNS or another network management tool. However, you cannot use the same tools or technique to update a LonTalk Stack application image over the network. Many LonTalk Stack devices do not require application updates over the network, but for those that do, this appendix describes considerations for adding this capability to the device.

If a LonTalk Stack device has sufficient non-volatile memory, it can hold two (or more) application images: one image for the currently running application, and the other image to control downloaded updates to the application. The device then switches between these images as necessary. Because neither the LonTalk API nor the LonTalk Host stack directly supports updating the LonTalk Stack application over the network, you must:

1. Define a custom application download protocol.
2. Implement an application download utility.
3. Implement application download capability within your LonTalk Stack application.

For the application download process:

- The application must be running and configured for the duration of the download.
- There must be sufficient volatile and non-volatile memory to store the new image without affecting the current image.
- The application must be able to boot the new image at the end of the download. During this critical period, the application must be able to tolerate device resets and boot either the old application image or the new one, as appropriate.

This appendix decribes some of the considerations for designing a LonTalk Stack application download function.

Important: This appendix does not describe how to download updates to the firmware image into the Ecehlon Free Topology Echelon Smart Transceiver. It only describes updates to the application image running on the host processor.

Custom Application Download Protocol

The custom LonTalk Stack application protocol that you define for downloading a LonTalk Stack application over the network should support the following steps:

1. Prepare for application download.

   When the application download utility informs the current LonTalk Stack application that it needs to start an application download, the application should respond by indicating whether it is ready for the utility to begin the download. The utility must be able to wait until the application is ready, or abort download preparation after a timeout period. The utility should also inform the user of its state.

During this stage, the LonTalk Stack device should verify that the application to be downloaded can run on the device platform (using the FPGA ID or similar mechanism), and verify that the application image is from a trusted source (for example, by using an encrypted signature).

2. Download the application.

   A reliable and efficient data transfer mechanism should be used. The interoperable file transfer protocol (FTP) can be used, treating the entire application image as a file.

   The download utility and the application must support long flash write times during this portion of the download process. The LonTalk Stack application should update the flash in the background (see *Download Capability within the Application*), however, it might be necessary for the protocol to define additional flow control to allow the LonTalk Stack application to complete flash writes before accepting new data.

3. Complete download.

   The application download utility informs the current application that the download is complete. The LonTalk Stack application should verify the integrity of the image, and either:

   a. Accept the image, and proceed to the final steps below.

   b. Request retransmission of some sections of the image.

   c. Reject the download.

4. Boot the new application.

   To boot the new application, you must implement a custom boot loader (or boot copier) so that the embedded processor can load the new application and restart the processor with the new image.

Important: For the duration of the first three steps, the application must be running, the LonTalk Host stack must be started, and the LonTalk Stack device must be configured.

Application Download Utility

This tool needs to read the application image to be loaded, and run the application download protocol described in *Custom Application Download Protocol*. You can write the utility as an OpenLNS Plugin or as any type of network-aware application.

Download Capability within the Application

Your application must implement the custom application protocol, and provide sufficient non-volatile storage for the new application image. The application also must tolerate time consuming writes to flash during the transfer. At a minimum, the LonTalk Stack application should reserve enough RAM to buffer two flash sectors. When one sector has been completely received, the application should write it to flash in a background process. If the write is not complete when the second buffer is filled, the LonTalk Stack application must tell the

application download utility to delay additional updates until the application is ready to receive the data.

After the transfer is complete and all data has been written to non-volatile memory, the application must prepare the image so that the boot loader can reboot the embedded processor from the new image. This preparation must be defined so that a device or processor reset at any point will result in a functioning LonTalk Stack device. For example, the reset could always cause a boot from the old application image, or from the new application image, or from some temporary boot application that can complete the transition (possibly with user intervention).

Another issue to consider is whether the entire image will be loaded or only a partial image. It is far simpler, and more flexible, if the entire image, including the LonTalk host stack and the operating system can be replaced. However, loading the entire image can take several minutes (for example, loading an application such as the simple example application could require 10 minutes or longer). Loading only the application portion of the image is possible if you structure your application very carefully. For example, you might need to provide patchable linkage stubs that allow your loaded application image to interact with the pre-loaded LonTalk host stack library and operating system.

Appendix G

Example LonTalk Stack Applications

This Appendix describes the example applications that are included in the LonTalk Stack. This Appendix describes each application's design, **main()** and event handler functions, and model file. It also describes how to build and load the application images and run the example applications.

Overview of the Example Applications

The LonTalk Stack Developer's Kit includes three example applications that are stored in the **LonWorks\LonTalkStack\Examples** directory. You can build these example applications with Microsoft Visual Studio 2008, and then run them on Windows. To run the examples, you must install OpenLDV 4.0, which you can download for free from the Echelon Web site at www.echelon.com/support/downloads. The following table describes these three example applications:

Function	Description
SimpleLtDevice	Simulates a voltage amplifier device. This device receives an input voltage value, multiplies the value by 2, and outputs the new value. This simulated device connects to a native LonWorks channel via OpenLDV 4.0 (or later), using a standard LonTalk network interface. This example requires a Layer 2 network interface such as the Echelon U10 USB Network Interface or PCC-10, PCLTA-20, or PCLTA-21 network interface card. This network interface cannot be shared with the OpenLNS Commissioning Tool or any other application.
SimpleIp852Device	Identical to the **SimpleLtDevice** example, but it connects to an IP-852 channel. To connect the **SimpleLtDevice** to an IP-852 channel, you need to install the Echelon IP-852 Configuration Server (you can download this app for free from the Echelon Web site at www.echelon.com/support/downloads). In the IP-852 Configuration Server, you will define the IPv4 address and port number of the **SimpleIp852Device** and any other IP-852 devices that it communicates with. For example, to test the **SimpleIp852Device** with OpenLNS CT, you can define an IP-852 channel that consists of two IP-852 devices: the **SimpleIp852Device** and OpenLNS CT.

Function	Description
Ip852Router	A router that connects an IP-852 channel to a native LonTalk channel. For the native LonWorks channel attached to this router, this example requires OpenLDV 4.0 (or later) and a Layer 2 network interface such as the Echelon U10 USB Network Interface or PCC-10, PCLTA-20, or PCLTA-21 network interface card. This network interface cannot be shared with the OpenLNS Commissioning Tool or any other application. For the IP-852 channel, you need to install the Echelon IP-852 Configuration Server (you can download this app for free from the Echelon Web site at www.echelon.com/support/downloads). In the IP-852 Configuration Server, you will define the IPv4 address and port number of the **Ip852Router** and any other IP-852 devices that it communicates with. For example, to test the **Ip852Router** with the OpenLNS Commissioning Tool, you can define an IP-852 channel that consists of two IP-852 devices: the **Ip852Router** and the OpenLNS Commissioning Tool.

The following sections describe the three example applications, including their design, how to build them, and how to run them.

Building the Example Applications

To build the example applications, start Visual Studio 2008, open the **Examples\Examples.sln** file, and then click **Build** and click **Build Solution**. This will build a project for each of the three example applications.

Running the Examples

Each example application is implemented as a simple Windows console application with a command line parameters to specify the network interfaces. Invoking the application with no parameters prints help information.

Starting an example application automatically runs the application and displays the following options on the console:

Command	Description
S	Sends service pin message.
E	Exit.
Q	Quit.
?	Print this screen.

To test the example application, you can use the OpenLNS Commissioning Tool or another OpenLNS installation tool to install the device or router.

Running the SimpleLtDevice Example

The console application takes one argument: **niName** (the network interface name). If you run the console application without entering any parameters, the application prints the following message and then exits:

```
SimpleLtDevice <niName>
<niName> is the name of the native LonTalk network interface
```

Running the SimpleIp852Device Example

The command takes two arguments: **ipAddress** (the IPV4 address in decimal dotted format) and **port** (the port used by the device). If you run the console application without entering any parameters, the application prints the following message and then exits:

```
SimpleIp852Device <ipAddress> <ipPort>

  <ipAddress>    is the IPv4 dotted-decimal address to use
                 for the IP852 interface

   <port>        is a decimal port number to use for the
                 IP852 interface
```

Running the Ip852Router Example

The command takes three arguments: **niName** (the network interface name), **ipAddress** (the IPV4 address in decimal dotted format), and **port** (the port used by the device). If you run the console application without entering any parameters, the application prints the following message and then exits:

```
Ip852Router <niName> <ipAddress> <ipPort>

 <niName>    is the name of the native LonWorks network
             Interface

 <ipAddress> is the IPv4 dotted-decimal address to use for
             the IP-852 interface

 <port>      is a decimal port number to use for the IP-852
             interface
```

SimpleLtDevice and SimpleIp852Device Example Application Details

The **SimpleLtDevice** example is a simple voltage amplifier application that simulates a voltage actuator with a built-in gain of 2. This device receives an input voltage value, multiplies the value by 2, and updates the simulated output feedback value. For a real voltage actuator device, the input value would be used to set a voltage level. After the device updated the voltage level, the application would read the actual level and use that value to set the feedback value.

The model file for this example includes a single **SFPTclosedLoopActuator** functional block for the two network variables. It does not include a Node Object functional block.

The **SimpleLtDevice** project is stored in the **Examples\SimpleLtDevice** directory.

The **SimpleIp852** example is the same as the **SimpleLtDevice** example except that it communicates over an IP-852 channel instead of a native LONWORKS channel.

The **SimpleIp852** project is stored in the **Examples\SimpleIp852** directory. The

The **SimpleLtDevice** and **SimpleIp852** example applications use a single C source file (**main.c**), the LonTalk API files generated by the LonTalk Interface Developer utility (**FtxlDev.c**, **FtxlDev.h**, **LonCpTypes.h**, and **LonNvTypes.h**), and a version of **FtxlHandlers.c** that has been customized for this example application.

The files used by the **SimpleLtDevice** and **SimpleIp852** projects are the same; therefore, they are stored in the **Examples\SimpleDevice** directory.

The following sections describe the **main()** function, the application task (**appTask()**) function, event handler functions, callback handler functions, and model file used by these example applications.

Main Function

The **main()** function is in the **main.c** file. The **main()** function performs the following tasks:

1. Processes the network interface name (**niName**) and stores it in static variables. These variables are later retrieved by the LonTalk Stack using the **GetMyNetworkInterface()** or **GetMyIpAddress()** callback methods. If the arguments are invalid, it displays the help and returns.

2. Creates an **Osal** event that is used to signal the application task that there is something to do.

3. Creates an application task that initializes and runs the LonTalk Stack

4. Runs a simple command console signaling the app task to send a service pin message or exit.

5. On exit, waits for the app task to complete, and then destroys the **Osal** event.

The **main()** function is shown below.

```
/* The main function processes command parameters, creates the application
 * task to the main stack loopand then runs a simple console to
 * allow sending a service pin messages and shutting down the application
 */
int main(int argc, char* argv[])
{
    #if FEATURE_INCLUDED(IP852)
        if (argc < 3 || !SetMyIpAddress(argv[1], argv[2]))
        {
            printf("Run a simple LonTalk Device using an IP-852 interface\n\n"
                "Syntax:\n"
                "    SimpleIp852Device <ipAddress> <ipPort>\n"
```

```
                "\n"
                "        <ipAddress> is the IPv4 dotted-decimal address to use
for the IP-852\n"
                "                    interface\n"
                "        <port>      is a decimal port number to use for the
IP852 interface\n"
                "\n");
            return 1;
        }
    #else
        if (argc < 2)
        {
            printf("Run a simple LonTalk Device using a standard LonTalk
network interface\n\n"
                   "Syntax:\n"
                   "    SimpleLtDevice <niName>"
                   "\n"
                   "        <niName>    is the name of the native LonTalk
network interface\n"
                   "\n");
            return 1;
        }
        else
        {
            SetMyNetworkInterface(argv[1]);
        }
    #endif

    if (OsalCreateEvent(&eventReadyHandle) == OSALSTS_SUCCESS)
    {
        OsalHandle taskHandle;
        OsalTaskId taskId;
        appTaskRunning = TRUE;
        if (OsalCreateTask(appTask, 0 /* Not used */, 64*1024, 11,
&taskHandle, &taskId) == OSALSTS_SUCCESS)
        {

            printHelp();
            printf("> ");
            while (appTaskRunning)
            {
                char c = toupper(getchar());
                if (c == 'S')
                {
                    sendServicPin = TRUE;
                    printf("Sending service pin...\n");
                    OsalSetEvent(eventReadyHandle);
                }
                else if (c == 'Q' || c == 'E')
                {
                    shutdownApp = TRUE;
                    printf("Exiting...\n");
                    OsalSetEvent(eventReadyHandle);
                    break;
                }
                else if (c == 0x0a)
                {
                    printf("> ");
                    ;
                }
                else
                {
                    if (c != '?')
                    {
                        printf("Unrecognized command\n");
                    }
                    printHelp();
                }
            }
            while (appTaskRunning)
            {
```

```
                OsalSleep(10);
            }
            OsalCloseTaskHandle(taskHandle);
        }
        OsalDeleteEvent(&eventReadyHandle);
    }

    return 0;
}
```

Application Task Function

The **appTask()** function is contained in the **main.c** file, and it performs the following tasks:

1. If the device is an IP-852 device, registers the devices unique ID. The example registers a dummy unique ID. A real IP-852 device must register a valid unique ID obtained from Echelon.

2. Calls the **LonInit()** function to initialize the LonTalk protocol stack. If this function fails, the **appTask()** function prints an error and terminates.

3. If the **LonInit()** function succeeds, the **appTask()** function begins a loop to wait for LonTalk Stack network events or for the main task to signal that it is time to shutdown. When a LonTalk Stack network event occurs, it does the following:

    a. Calls the **LonEventPump()** function to process the event.

    b. If the **sendServicePin** variable was set by the main task, calls **LonSendServicePin()** to send a service pin message.

4. Upon termination, the **appTask()** function does the following:

    a. Calls the **LonExit()** function to destroy the LonTalk protocol stack.

    b. Sets the **appTaskRunning** flag to false to signal the main task that it is done.

    c. Returns.

You can use the same basic algorithmic approach with the **main()** and **appTask()** functions for a production-level application.

The **appTask()** function is shown below.

```
/* The application task initializes the LonTalk protocol stack and
 * implements the main control loop.  The bulk of the application processing
 * is performed in the myNvUpdateOccurred event handler.
 */
void appTask(int taskIndex)
{
    /* Create the "event ready" event, which is signaled by the myEventReady
     * callback to wake this task up to process LonTalkStack events.
     */
    /* Initialize the LonTalk Stack */
    LonApiError sts;
#if FEATURE_INCLUDED(IP852)
    #pragma message ("Warning:  TBD - Must set a valid UniqueID for IP-852
interface!!!")
    // TBD - Set the unique ID.
    LonUniqueId uid = { 0xBA, 0xDB, 0xAD, 0xBA, 0xDB, 0xAD };
```

```
    LonResgisterUniqueId(&uid);
#endif

    sts = LonInit();
    if (sts == LonApiNoError)
    {
        /* This is the main control loop, which runs forever.  */

        while (!shutdownApp)
        {
            /* Whenever the ready event is fired, process events by calling
             * LonEventPump.  The ready event is fired by the myEventReady
             * callback.
             */
            if (OsalWaitForEvent(eventReadyHandle, OSAL_WAIT_FOREVER) ==
OSALSTS_SUCCESS)
            {
                LonEventPump();
            }
            if (sendServicPin)
            {
                LonSendServicePin();
                sendServicPin = FALSE;
            }
        }
    }
    else
    {
        printf("Error: LonInit failed with error #%d\n", sts);
    }
    LonExit();
    appTaskRunning = FALSE;
```

Event Handler Function

To signal to the main application the occurrence of certain types of events, the LonTalk API calls specific event handler functions. For the simple voltage amplifier example application, only one of the API's event handler functions has been implemented to provide application-specific behavior.

The **FtxlHandlers.c** file contains the modified **LonNvUpdateOccurred()** function, which is called when the host processor receives a network-variable update. This function simply calls the **myNvUpdateOccurred()** function in the **main.c** file that provides the application-specific behavior. This functional separation approach keeps changes to the LonTalk Interface Developer utility-generated files to a minimum. For a production-level application, you can place application-specific code wherever your application design requires it.

The **myNvUpdateOccurred()** function contains a C **switch** statement, which contains a single **case** statement because the **VoltActuator** functional block includes only a single input network variable, **nviVolt**.

The **case** statement for the **nviVolt** network variable (specified by the **LonNvIndexNviVolt** network variable index) calls the **ProcessNviVoltUpdate()** utility function to perform the following tasks:

- Perform range checking for the network variable
- Set the output network variable to double the value of the input network variable
- Propagate the output network variable to the network

The two network variables are defined in the model file, which is described in *Model File*.

The **myNvUpdateOccurred()** function is shown below.

```
/*
 * This function is called by the FTXL LonNvUpdateOccurred
 * event handler, indicating that a network variable input
 * has arrived.
 */
void myNvUpdateOccurred(const unsigned nvIndex,
                const LonReceiveAddress* const pNvInAddr) {
  switch (nvIndex) {
    case LonNvIndexNviVolt:
    {
      /* process update to nviVolt.  */
      ProcessNviVoltUpdate();
      break;
    }
    /* Add more input NVs here, if any */

    default:
      break;
  }
}
```

Application-Specific Utility Functions

The simple example application includes the following application-specific utility functions:

- **ProcessNviVoltUpdate()**: Performs range checking for the network variables, sets the output network variable to double the value of the input network variable, and propagates the output network variable to the network.
- **ProcessOnlineEvent()**: Calls the **ProcessNviVoltUpdate()** function when the device goes online.

These functions are defined in the **main.c** file.

Callback Handler Function

To signal to the main application the occurrence of certain types of events, the LonTalk API calls specific callback handler functions. For the simple voltage actuator example application, only one of the API's callback handler functions has been implemented to provide application-specific behavior.

The **FtxlHandlers.c** file contains the modified **LonEventReady()** function, which is called when the LonTalk Host stack receives a network event. This function simply calls the **myEventReady()** function in the **main.c** file that provides the application-specific behavior. This functional separation approach keeps changes to the LonTalk Interface Developer utility-generated files to a minimum. For a production-level application, you can place application-specific code wherever your application design requires it.

The **myEventReady()** function calls the OSAL **OsalSetEvent()** function to signal the application task so that it can process the network event.

The **myEventReady()** function is shown below.

```
/* This function is called by the FTXL LonEventReady
 * callback, signaling that an event is ready to be
 * processed.
 */
void myEventReady(void) {
  /* Signal application task so that it can process the
   * event.  */
  OsalSetEvent(eventReadyHandle);
}
```

Model File

The model files for the **SimpleLtDevice** and **SimpleIp852Device** example application define the LONWORKS interface for the example LonTalk Stack devices.

The model file defines one functional block, **VoltActuator**. The **VoltActuator** functional block includes two network variables, **nviVolt** and **nvoVoltFb**. The functionality for these network variables is implemented in the **myNvUpdateOccurred()** function described in *Event Handler Function*.

The model file is shown below.

```
#pragma enable_sd_nv_names

network input SNVT_volt nviVolt;
network output SNVT_volt bind_info(unackd) nvoVoltFb;

fblock SFPTclosedLoopActuator {
   nviVolt  implements nviValue;
   nvoVoltFb   implements nvoValueFb;
} VoltActuator
external_name("VoltActuator");
```

For more information about creating and using a model file, see *Creating a Model File*.

Extending the SimpleLtDevice and SimpleIp852 Examples

You can configure the LONWORKS interface and functionality of the **SimpleLtDevice** and **SimpleIp852** example applications. To do this, perform the following steps:

1. Copy the files in the **Examples\SimpleDevice** directory to your project's directory. This ensures that you have a backup of the examples' original source files.
2. Clear the read-only flag for each of the copied files.
3. Modify the device interface by adding network variables to the **Simple Example.nc** model file.

4. Run the LonWorks Interface Developer by double-clicking the **Simple Example.lidprj** file. This generates an updated application framework.
5. Modify the callback handler functions in the **FtxlHandlers.c** and **main.c** files as required.
6. Rebuild the project.
7. Optionally, load the generated XIF file into the Echelon Smart Transceiver or Neuron Chip.
8. Load the new executable file into the host processor.

IP-852 Router Example Application Details

The IP-852 router example implements an IP-852 to native LonWorks router. Router applications do not implement functional blocks, network variables, or configuration properties or other objects used by device applications; therefore, the IP-852 router project does not include a model file or a LID project.

The **Ip852Router** project is stored in the **Examples\Ip852Router** directory. This folder contains the main application file (**main.cpp**), the Visual Studio project file (**Ip852Router.vcproj**), and a ReadMe (**Readme.txt**).

The **main.cpp** file contains the **main()** function. The **main()** function performs the following tasks:

1. Processes the command parameters: **niName** (the network interface name), **ipAddress** (the IPV4 address in decimal dotted format), and **port** (the port used by the device). Prints an error and returns if the parameters are invalid.

2. Creates a **LtLogicalChannel** object, passing in the name of the network interface. Essentially, this opens the native LonTalk network interface.

3. Reads the unique ID from the native LonWorks interface.

4. Creates a **LtIp852Router** object and starts the router by calling the object's **Start()** method with the following parameters:

   - An **ltAppIndex** of **0**. This the application index of the native LonWorks side of the router. The LonTalk Stack requires an application index whenever there is more than one stack. It is mainly used to store the unique ID of the stack and to name persistence files.
   - **LtUniqueId**. The uniqueId of the native LonWorks side of the router.
   - **pLtChannel**. The pointer to the native LonWorks channel
   - An **ipAppIndex** of **1**. This is the application index of the IP-852 router side.
   - **ipUid**. The unique ID of the IP-852 router side. This example uses a dummy unique ID.
   - **ipAddress**. A 32-bit integer representing the IP address of the IP-852 router side.
   - **ipPort**. The port number on which the IP-852 router side is listening.

5. Runs a simple command console:

   - When the **S** command is pressed, sends a service pin messages from both router halves by calling the **sendServicePinMsg()** method of the **LtIp852Router** object.

   - When the **E** or **Q** command is pressed, breaks out of the console loop

6. On exit, does the following:

   - Shuts down the router by calling the **Shutdown()** method of the **LtIp852Router** object.

   - Closes the native LonWorks network interface by deleting the **LtLogicalChannel** object.

The **main()** function is shown below.

```
/* The main function just initializes the router and then runs a simple
command line interface.  */
int main(int argc, char* argv[])
{
    const char *pNiName = NULL;
    int ipAddress;
    int ipPort;
    LtIp852Router router;
    LtLtLogicalChannel *pLtChannel = NULL;

    LtErrorType sts;
    if (argc < 4 || sscanf(argv[3], "%d", &ipPort) != 1)
    {
        printf("Run an IP-852 to native LonTalk Router\n\n"
               "Syntax:\n"
               "    Ip852Router <niName> <ipAddress> <ipPort>\n"
               "\n"
               "        <niName>    is the name of the native LonTalk network
interface\n"
               "        <ipAddress> is the IPv4 dotted-decimal address to use
for the IP-852\n"
               "                    interface\n"
               "        <port>      is a decimal port number to use for the
IP-852 interface\n"
               "\n");
        return 1;
    }
    else
    {
        pNiName = argv[1];
        ipAddress = htonl(inet_addr(argv[2]));
    }

    pLtChannel = new LtLtLogicalChannel(pNiName);

    sts = pLtChannel->getStartError();
    if (sts == LT_NO_ERROR && !pLtChannel->getLonLink()->isOpen())
    {
        sts = LT_CANT_OPEN_PORT;
    }

    if (sts == LT_NO_ERROR)
    {
#pragma message ("Warning:  TBD - Must set a valid unique ID for IP-852
interface!!!")
        const byte data[6] = { 0xBA, 0xDB, 0xAD, 0xBA, 0xD0, 0x00 };
        LtUniqueId ltUid;
        LtUniqueId ipUid(data);
        pLtChannel->getLonLink()->getUniqueId(ltUid);
```

```
            sts = router.Start(0, ltUid, pLtChannel,
                               1, ipUid,  ipAddress, ipPort);
        }

        if (sts == LT_NO_ERROR)
        {

            printHelp();
            printf("> ");
            while (TRUE)
            {
                char c = toupper(getchar());
                if (c == 'S')
                {
                    printf("Sending service pin...\n");
                    router.sendServicePinMsg();
                }
                else if (c == 'Q' || c == 'E')
                {
                    printf("Exiting...\n");
                    break;
                }
                else if (c == 0x0a)
                {
                    printf("> ");
                    ;
                }
                else
                {
                    if (c != '?')
                    {
                        printf("Unrecognized command\n");
                    }
                    printHelp();
                }
            }
        }
        else
        {
            printf("Initialization error %d\n", sts);
        }

        router.Shutdown();

        delete pLtChannel;

        return 0;
    }
```

Appendix H

LonTalk Interface Developer Utility Error and Warning Messages

This Appendix lists the LonTalk Interface Developer utility error and warning messages, and offers suggestions on how to correct the indicated problems.

Introduction

All messages, errors, and warnings, come with a standard Echelon message identifier LID#*zzz*, where *zzz* is a unique decimal number.

All messages shown below are not actually given with the precise language shown at runtime. Instead, a summary of the message meaning is given for each message, followed by a brief discussion of possible reasons and remedies. In all cases, make sure to consult the actual message as produced by the tool at runtime, as the actual message is likely to contain more details (for example, the name of the offending file, or more detailed language about the precise failure reason).

See the *Neuron Tools Errors Guide* for information about errors issued by the Neuron C compiler (warning and error messages with NCC#*zzz* identifiers).

Error Messages

LID#	Description
1	An NV, CP, or MT item was expected but not present – internal error Remove the device interface files (.xif and .xfb extension), and re-run the LonTalk Interface Developer utility to see if the problem persists. Use the **Trace** verbosity level to help track down the problem.
2	A file cannot be opened for read access See the error message received for details of the offending file. Make sure the file is available and readable and the path is accessible.
3	A file cannot be opened for write access See the error message received for details of the offending file. Make sure the file is available and writable and the path is accessible.
4	A property value is required but has not been obtained from any data source This is an internal error, probably a result of an earlier failure. A non-fatal error during the creation of the device interface file might lead to this error. Re-run the LonTalk Interface Developer utility in **Trace** verbosity mode and carefully examine the LonTalk Interface Developer utility Summary window to determine the root cause of the failure

LID#	Description
5	An error occurred when reading a device interface file This is an internal error, probably a result of an earlier failure. A non-fatal error during the creation of the device interface file might lead to this error. Re-run the LonTalk Interface Developer utility in **Trace** verbosity mode and carefully examine the LonTalk Interface Developer utility Summary window to determine the root cause of the failure.
6	An error occurred when reading a device interface file This is an internal error, probably a result of an earlier failure. A non-fatal error during the creation of the device interface file might lead to this error. Re-run the LonTalk Interface Developer utility in **Trace** verbosity mode and carefully examine the LonTalk Interface Developer utility Summary window to determine the root cause of the failure. (This error is similar to LID#5, but refers to a different internal component recognizing the error.)
7	A device interface file appears malformed This is an internal error, probably a result of an earlier failure. A non-fatal error during the creation of the device interface file might lead to this error. Re-run the LonTalk Interface Developer utility in **Trace** verbosity mode and carefully examine the LonTalk Interface Developer utility Summary window to determine the root cause of the failure.
8	An unrecognized escape character has been detected in a file or NVVAL data record This is an internal error, probably a result of an earlier failure during the creation of an intermediate file with a .bif file extension. Re-run the LonTalk Interface Developer utility in **Trace** verbosity mode and carefully examine the LonTalk Interface Developer utility Summary window to determine the root cause of the failure. After the build, make sure the file with the .bif extension exists and can be read.
9	A FILE or NVVAL value record cannot be read due to an unsupported construct This is an internal error, probably a result of an earlier failure during the creation of an intermediate file with a .bif file extension. Re-run the LonTalk Interface Developer utility in **Trace** verbosity mode and carefully examine the LonTalk Interface Developer utility Summary window to determine the root cause of the failure. After the build, make sure the file with the .bif extension exists and can be read.

LID#	Description
10	Failure to attach to LONUCL32 service DLL The LonTalk Interface Developer utility or one of its components failed to locate a file by name of "LONUCL32.DLL." This file usually resides in the same folder that contains the LID.exe application, but can be in any folder in your current user search path. This file is typically installed into the LonWorks **Bin** folder.
18	An error occurred when composing the application XIF file: the data merge target is ill-chosen (must be the BIF file) This is an internal error that should not normally occur. However, it could be a result of an earlier failure. For example, a non-fatal error during the creation of the device interface file might lead to this error. Re-run the LonTalk Interface Developer utility in **Trace** verbosity mode and carefully examine the LonTalk Interface Developer utility Summary window to determine the root cause of the failure.
19	File I/O error when writing XIF file Refer to the error message for details about the failure cause. The error message contains details such as "disk full," or "file access denied".
20	Error (non-file I/O) when writing XIF file Refer to the error message for details about the failure cause. The error message contains details such as "disk full," or "file access denied".
21	The xif32bin.exe utility returned an error, indicating failure when converting XIF to XFB The binary device interface file (.xfb extension) could not be created. Make sure a previously existing binary device interface file is not write-protected. Also make sure the XIF32Bin.exe utility, which is used to create the binary device interface file, is available in a folder that is part of the system or current user search path. By default, the utility can be found in your LONWORKS **Bin** folder.
22	An error occurred when reading a type info file (.NCT) This is an internal error, possibly resulting from an earlier failure. The .nct file is an intermediate file used by the ShortStack Wizard. Re-run the LonTalk Interface Developer utility in **Trace** verbosity mode and carefully examine the LonTalk Interface Developer utility Summary window to determine the root cause of the failure.

LID#	Description
23	An error occurred when reading a type info file (.NCT) This is an internal error, possibly resulting from an earlier failure. The .nct file is an intermediate file used by the ShortStack Wizard. Re-run the LonTalk Interface Developer utility in **Trace** verbosity mode and carefully examine the LonTalk Interface Developer utility Summary window to determine the root cause of the failure. This error is similar to LID#22, but refers to different internal software components.
24	Type info (.NCT) file seems corrupted This is an internal error, possibly resulting from an earlier failure. The .nct file is an intermediate file used by the LonTalk Interface Developer utility. Re-run the LonTalk Interface Developer utility in **Trace** verbosity mode and carefully examine the LonTalk Interface Developer utility Summary window to determine the root cause of the failure.
25	Unexpected end of type info file (.NCT) This is an internal error, possibly resulting from an earlier failure. The .nct file is an intermediate file used by the LonTalk Interface Developer utility. Re-run the LonTalk Interface Developer utility in **Trace** verbosity mode and carefully examine the LonTalk Interface Developer utility Summary window to determine the root cause of the failure.
27	Unexpected file I/O error when reading a file Refer to the error message for details of the failure cause.
28	Unexpected error (not a file I/O error) when reading a file Refer to the error message for details of the failure cause.
29	Unexpected file I/O error when writing a file Refer to the error message for details of the failure cause.
30	Unexpected error (not a file I/O error) when writing a file Refer to the error message for details of the failure cause. The error message contains details such as “disk full” or “file access denied”.

LID#	Description
31	A type definition cannot be generated: the type is referenced but not defined A type that you have referenced is missing from the NCT file, and intermediate file used by the LonTalk Interface Developer utility. This is an internal error. Delete all intermediate files. Re-run the LonTalk Interface Developer utility in **Trace** verbosity mode and carefully examine the LonTalk Interface Developer utility Summary window to determine the root cause of the failure. If the problem persists, contact Echelon technical support, submitting all files produced by the LonTalk Interface Developer utility when running in **Trace** verbosity level.
32	A type definition is provided but seems incomplete –an element is missing This is an internal error. Delete all intermediate files. Re-run the LonTalk Interface Developer utility in **Trace** verbosity mode and carefully examine the LonTalk Interface Developer utility Summary window to determine the root cause of the failure. If the problem persists, contact Echelon technical support, submitting all files produced by the LonTalk Interface Developer utility when running in **Trace** verbosity level.
33	Anonymous types are not supported Any type used for network variables or configuration properties must have a name. The use of constructs such as, "network input struct { int a, b; } nviZorro;" is not permitted.
34	A compiler feature cannot be selected Refer to the error message for details of the failure cause. This error might be the result of conflicting preferences in the default command file, LonNCC32.def, located in the LonTalk Interface Developer utility's project file. Refer to the *Neuron C Programmer's Guide* and *Neuron C Reference Guide* for more details about the command line tools and script files.
35	Configuration parameters are in use, but no template file has been found This might be the result of an earlier error. Delete all intermediate files. Re-run the LonTalk Interface Developer utility in **Trace** verbosity mode and carefully examine the LonTalk Interface Developer utility Summary window to determine the root cause of the failure. If the problem persists, contact Echelon technical support, submitting all files produced by the LonTalk Interface Developer utility when running in **Trace** verbosity level.

LID#	Description
36	The program ID found in the XIF file seems malformed and cannot be used to produce the niAppinit data Use the LonTalk Interface Developer utility and the Standard Program ID calculator to produce a good program ID record. Delete all intermediate files. Re-run the LonTalk Interface Developer utility in **Trace** verbosity mode and carefully examine the LonTalk Interface Developer utility Summary window to determine the root cause of the failure. If the problem persists, contact Echelon technical support, submitting all files produced by the LonTalk Interface Developer utility when running in **Trace** verbosity level.
42	A type definition cannot be generated –the type definition has more elements than expected Delete all intermediate files. Re-run the LonTalk Interface Developer utility in **Trace** verbosity mode and carefully examine the LonTalk Interface Developer utility Summary window to determine the root cause of the failure. If the problem persists, contact Echelon technical support, submitting all files produced by the LonTalk Interface Developer utility when running in **Trace** verbosity level.
46	One or more configuration properties implemented within a file are present, FTP or DMF must be implemented Alternatively, you can declare configuration properties as configuration network variables.
47	The file transfer protocol (FTP) and direct memory files (DMF) access mechanisms are mutually exclusive
49	The FTP server interface is partially implemented, missing the specified member of the node object
50	Data files and file directory are too big for the available space. Available: <n> bytes, required: <m> bytes (missing: <p> bytes) [LID#50] Possible remedies: reduce the size of files by removing extraneous data files, or by sharing CP, or implement FTP.
51	Malformed XML data (cannot convert to expected type)
52	The specified application framework type is unknown
53	No target framework has been supplied, or the requested framework is not registered with, or not known to, the Builder
54	No code generator found for the selected target framework

LID#	Description
55	The specified framework is not yet supported This is an internal error.
57	Required source file missing
59	Too many network variables. The sum of static and dynamic variables cannot exceed 4096.
60	Insufficient number of addresses This message includes how many addresses are require for the application, and how many were specified.
61	The DMF window specification is invalid, as it exceeds the 64 KB address range
62	Insufficient buffer space The message includes the total number of bytes available for transceiver buffers and how many additional bytes your selected configuration requires.

Warning Codes

LID#	Description
4001	An XIF file contains more fields than expected Refer to the warning message for line # and filename. This might result in an automatic downgrading of the device interface file to the version supported by the LonTalk Stack or ShortStack tools. Check *www.echelon.com* for available updates.
4002	An intermediate file cannot be removed in the sweep-phase. See message for details Refer to the warning message for details about the warning cause. The sweep occurs when the utility's operation is complete and the utility did not run in the **Trace** verbosity level. The warning indicates that an intermediate file cannot be removed.

LID#	Description
4006	A file cannot be copied This is possibly, but not necessarily, fatal. When the LonTalk Interface Developer utility creates the host framework, it produces several files based on input provided by the user. It also copies the necessary files into the destination folder. The utility-generated files refer to these files, which are required to build the host application. Thus, this issue is non-fatal for the LonTalk Interface Developer utility, but probably fatal when building the host application. See also warning LID#4017.
4011	The .NCT file references a built-in type with no host equivalent known to LonTalk Interface Developer utility This condition is unlikely to occur and does report an internal error. Check www.echelon.com for available software updates that address the problem, or contact LonSupport@Echelon.com. This message is a warning rather than an error because the condition does not prevent your application from working. Check the type definitions provided in **LonNvTypes.h** and **LonCpTypes.h** (both generated by LonTalk Interface Developer utility) and correct the offending type. Continue using these files and build your LonTalk Stack device.
4014	Explicit addressing specified but not required This warning reminds you that you have requested support for explicit addressing, although it does not seem to be required. Explicit addressing requires larger buffers on the host, therefore support for explicit addressing is advisable only when needed. Message tag declarations that are intended for use with explicit addressing should be marked with the **bind_info(nobind)** modifier to signal the use of explicit messaging. See also the LID#4013 and LID#4015 warnings.
4015	Explicit addressing specified but neither supported nor required Although support for explicit addressing has been requested, it does not appear to be required. See also the LID#4013 and LID#4014 warnings.
4016	FTP implementation suspect –no message tag but SNVT_file_* implemented The implementation of the file transfer protocol is suspect, as the FTP-related network variables are present but no message tag has been declared.

LID#	Description
4017	Files cannot be made writable When the LonTalk Interface Developer utility creates the host framework, it produces several files based on input provided by the user. It copies the necessary files into the destination folder. These files are made writable after they are copied, unless this warning indicates it is not possible. See also the LID#4006 warning.
4023	Insufficient addresses are implemented for the specified number of network variables For more robust device behavior, increase the number of addresses.
4025	The program ID's channel identifier should be set to 0x04 (TP/FT-10)
4026	Your transceiver buffer configuration leaves a number of bytes unused

Hint Codes

LID#	Description
8001	Your device supports the file transfer protocol, but no configuration property files are available. This is not an error if your application has other uses for the file transfer protocol.
8005	Your transceiver buffer configuration leaves a number of bytes unused

Appendix I

Glossary

This appendix defines many of the common terms used for LonTalk Stack device development.

D

downlink

Link-layer data transfer from the host to the Echelon Smart Transceiver or Neuron Chip.

E

Echelon Smart Transceiver or Neuron Chip

A chip that is used as a transceiver to attach a host processor to a LONWORKS network; the Echelon Smart Transceiver or Neuron Chip runs the Neuron Firmware and implements layers 1 and 2 of the ISO/IEC 14908-1 Control Network Protocol.

execution context

A general term for a thread of execution for an operating system. Depending on the operating system and hardware, this could be a process, task, thread, or fiber.

F

Firmware

The firmware embedded within an Echelon Smart Transceiver or Neuron Chip.

H

host processor

A microcontroller, microprocessor, or embedded processor that is integrated with the LonTalk API and an Echelon Smart Transceiver or Neuron Chip to create a LONWORKS device.

L

link layer

A protocol and interface definition for communication between a host processor and either an Echelon Smart Transceiver or Neuron Chip or ShortStack Micro Server.

link layer protocol

The protocol that is used for data exchange across the link layer.

LonTalk API

A C language interface that can be used by a LonTalk application to send and receive network variable updates and LonTalk messages. Two implementations are available: a full version for LonTalk Stack devices and a compact version for ShortStack devices.

LonTalk application

An application for a LONWORKS device that communicates with other devices using the ISO/IEC 14908-1 Control Network Protocol and is based on the LonTalk API or the LonTalk Compact API.

LonTalk application framework

Application code and device interface data structures created by the LonTalk Interface Developer based on a model file.

LonTalk Compact API

A compact version of the LonTalk API for ShortStack devices with support for up to 254 network variables.

LonTalk host stack

A high-performance implementation of layers 3 through 6 of the ISO/IEC 14908-1 Control Network Protocol that runs on an embedded processor.

LonTalk Interface Developer

A utility that generates an application framework for a LonTalk application; the LonTalk Interface Developer is included with both the LonTalk Stack Developer's Kit and the ShortStack Developer's Kit.

LonTalk Platform

Development tools, APIs, firmware, and chips for developing LONWORKS devices that use the LonTalk API or LonTalk Compact API. Two versions are available as part of the LonTalk Stack Developer's Kit and the ShortStack Developer's Kit.

LonTalk Stack application

An application for a LONWORKS device based on the LonTalk API and Echelon Smart Transceiver or Neuron Chip.

LonTalk Stack

Software required todevelop high-performance LonTalk applications for a host processor with an Echelon Smart Transceiver or Neuron Chip. The LonTalk Stack includes a simple host application programming interface (API), a complete ISO/IEC 14908-1 protocol stack implementation, a link-layer driver, a simple hardware interface, and comprehensive tool support.

LonTalk Stack device

A LONWORKS device based on the LonTalk API and an Echelon Smart Transceiver or Neuron Chip.

M

model file

A Neuron C application that is used to define the network interface for an FTXL or ShortStack application.

N

Neuron C

A programming language based on ANSI C with extensions for control network communication, I/O, and event-driven programming; also used for defining a network interface when used for a model file.

U

uplink

Link-layer data transfer from the Echelon Smart Transceiver or Neuron Chip to the host.

ECHELON®
www.echelon.com