ECHELON®

# FTXL User's Guide

078-0363-01A

Echelon Corporation
www.echelon.com

# Welcome

Echelon's FTXL™ products enable any product that contains an Altera® Nios® II processor to quickly and inexpensively become a networked smart device. An FTXL device includes a complete ANSI/CEA 709.1-B (EN14908.1) implementation that runs on the Nios II embedded processor. Thus, the FTXL 3190™ Free Topology Smart Transceiver Chip provides a simple way to add LONWORKS® networking to smart devices. The FTXL Transceiver is easy to use because it has a simple host application programming interface (API), a pre-built link-layer driver, a simple hardware interface, and comprehensive tool support.

This document describes how to develop an application for a LONWORKS device using Echelon's FTXL Transceiver. It describes the architecture of an FTXL device and how to develop the software for an FTXL device. Development of a FTXL device includes creating a model file, running the LonTalk® Interface Developer utility, and using the FTXL API functions to program your FTXL application for the Nios II processor.

See the *FTXL Hardware Guide* for a description of the hardware interfaces for an FTXL device, the development boards for which the FTXL Developer's Kit provides reference designs, and FPGA design requirements for an FTXL device.

# Audience

This document assumes that the reader has a good understanding of the LONWORKS platform and programming for the Altera Nios II processor.

# Related Documentation

In addition to this manual and the *FTXL Hardware Guide* (078-0364-01A), the FTXL Developer's Kit includes the following manuals:

- *Neuron C Programmer's Guide* (078-0002-02G). 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* (078-0140-02E). This manual provides reference information for writing programs that use the Neuron C language.
- *NodeBuilder Errors Guide* (078-0193-01B). This manual describes error codes issued by the Neuron C compiler.

The FTXL Developer's Kit also includes the reference documentation for the FTXL LonTalk API, which is delivered as a set of HTML files.

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

The following manuals are available from the Echelon Web site (www.echelon.com) and provide additional information that can help you develop applications for an FTXL Transceiver:

- *Introduction to the LONWORKS System* (078-0183-01A). This manual provides an introduction to the ANSI/CEA-709.1 (EN14908) Control Networking 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 Smart Transceiver Data Book* (005-0139-01D). This manual provides detailed technical specifications on the electrical interfaces, mechanical interfaces, and operating environment characteristics for the FT 3120®, FT 3150®, and FTXL 3190 Smart Transceivers.
- *LonMaker User's Guide* (078-0333-01A). This manual describes how to use the Turbo edition of the LonMaker® Integration Tool to design, commission, monitor and control, maintain, and manage a network.

All of the FTXL 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: www.adobe.com/products/acrobat/readstep2.html.

## Related Altera Product Documentation

For information about the Altera Nios II family of embedded processors and associated tools, see the Altera Nios II Literature page: www.altera.com/literature/lit-nio2.jsp.

**Table 1** lists Altera product documents that are particularly useful for the FTXL Developer's Kit.

**Table 1**. Related Altera Documentation

| Product Category | Documentation Titles |
|---|---|
| Quartus® II software | Introduction to Quartus II Software<br>Quartus II Quick Start Guide<br>Quartus II Development Software Handbook v7.2 |

| Product Category | Documentation Titles |
|---|---|
| Nios II processor | Nios II Hardware Development Tutorial<br>Nios II Software Development Tutorial (included in the online help for the Nios II EDS integrated development environment)<br>Nios II Flash Programmer User Guide<br>Nios II Processor Reference Handbook<br>Nios II Software Developer's Handbook |
| Cyclone® II and Cyclone III FPGA and device configuration | Cyclone II Device Handbook<br>Cyclone III Device Handbook<br>Configuration Handbook |
| USB-Blaster™ download cable | USB-Blaster Download Cable User Guide |
| Software licensing | Quartus II Installation & Licensing for Windows<br>AN 340: Altera Software Licensing |

## Related devboards.de Product Documentation

The FTXL Developer's Kit uses the devboards.de DBC2C20 Altera Cyclone II Development Board for its examples and reference designs. For information about the DBC2C20 Altera Cyclone II Development Board, including the most current data sheet for the board, see the DBC2C20 page: www.devboards.de/index.php?mode=products&kategorie=14.

The DBC2C20 development board is also available from EBV Elektronik; see www.ebv.com/en/products/development_boards/dbc2c20.html.

# Table of Contents

# 1

# Introduction to FTXL

This chapter introduces the LonTalk Platform for FTXL Transceivers. It describes the architecture of an FTXL device, including a comparison with other LONWORKS devices. It also describes attributes of an FTXL device, the requirements and restrictions of the FTXL LonTalk protocol stack, and the FTXL products that are available from Echelon.

# Overview

Automation solutions for buildings, homes, and industrial applications include sensors, actuators, and control systems. A *LONWORKS network* is a peer-to-peer network that uses an industry-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 ANSI/CEA 709.1-B (EN14908.1) 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 the seven abstract layers described in **Table 2**. The implementation of these layers in a LONWORKS device provides standardized interconnectivity for devices within a LONWORKS network.

**Table 2**. LONWORKS Network Protocol 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 ANSI/CEA-709.1 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), LNS® Server, LNS remote client, *i*.LON® servers, and the FTXL LonTalk protocol stack. This document refers to

the ANSI/CEA-709.1 (EN14908-1) 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, Echelon's 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 Smart Transceiver runs the device's application, implements the LonTalk protocol, and interfaces with the physical communications media through a transformer. **Figure 1** on page 4 shows the seven-layer LonTalk protocol on a single Neuron Chip or 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 Smart Transceiver) runs both the application and the LonTalk protocol.

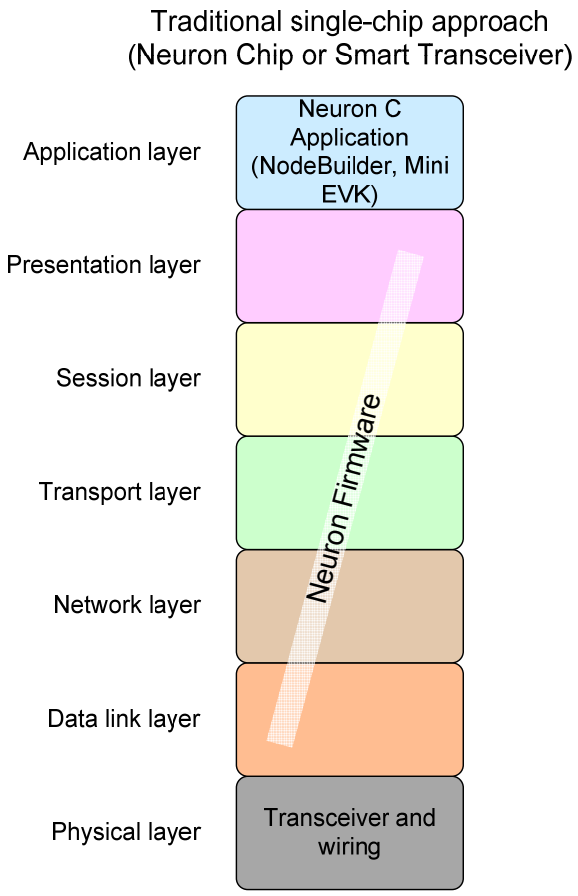



**Figure 1**. A Single-Chip LONWORKS Device

For a Neuron-hosted device that uses a Neuron Chip or Smart Transceiver, the physical layer (layer 1) is handled by the Neuron Chip or 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 create the application program using the Neuron C programming language in either the NodeBuilder® Development Tool or the Mini EVK Evaluation Kit.

## 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 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
- A microprocessor, microcontroller, or embedded processor in a field-programmable gate array (FPGA) device, 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 a Smart Transceiver plus a host processor.

Compared to the single-chip device, the Smart Transceiver implements only a subset of the LonTalk protocol layers. The host processor implements the remaining layers and runs the device's application program. The Smart Transceiver 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 Development Tool or the Mini EVK application, to develop the application.

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

- The LonTalk Platform for ShortStack Micro Servers
- The LonTalk Platform for FTXL Transceivers

# LonTalk Platform for ShortStack Micro Servers

The LonTalk Platform for ShortStack Micro Servers 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 a Smart Transceiver with firmware, the *ShortStack firmware*, that implements layers 2 to 5 (and part of layer 6) of the LonTalk protocol, as shown in **Figure 2** on page 6. The host processor implements the application layer (layer 7) and part of the presentation layer (layer 6).

The ShortStack firmware allows you to use almost any host processor for your device's application and I/O. The Smart Transceiver implements layers 2 to 5 (and part of layer 6) of the LonTalk protocol and provides the physical interface for the LONWORKS communications channel.

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. Currently, example drivers are available for an Atmel® ARM7 microprocessor and an Altera Nios II embedded processor.

For ShortStack device development, you use the C programming language[1]. You use the Echelon LonTalk Interface Developer utility to create the application framework. Your application uses an ANSI C API, the Echelon LonTalk Compact API, to manage communications with the ShortStack Micro Server and devices on the LONWORKS network.

[1] For ShortStack device development, you could alternatively use 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.

Using a ShortStack Micro Server makes it easy to add LONWORKS networks to any existing smart device.

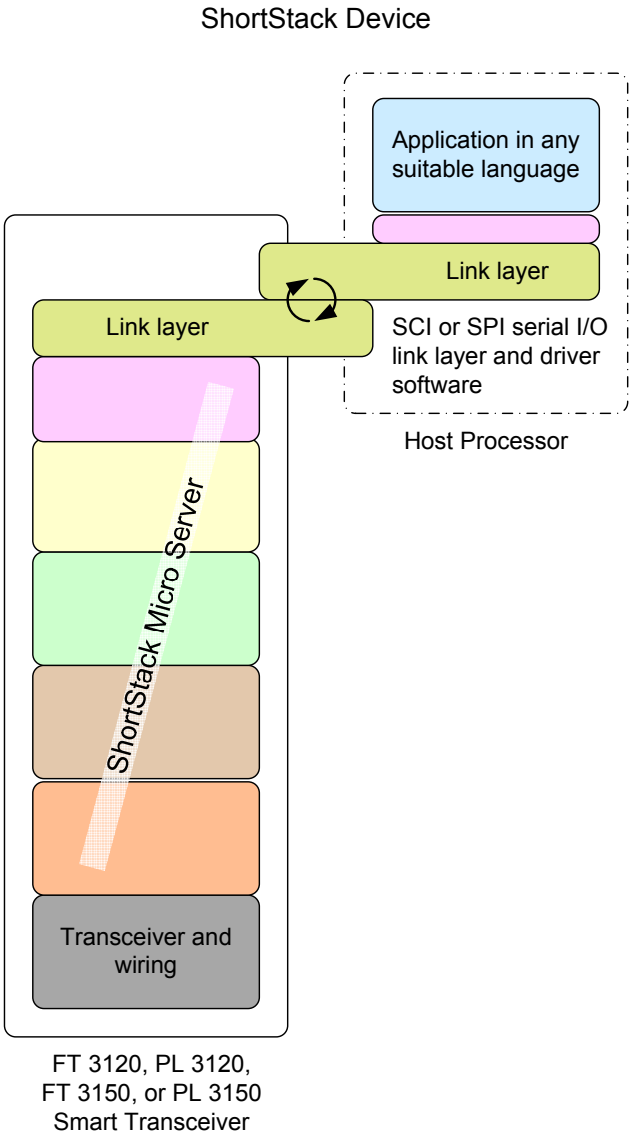



**Figure 2**. The ShortStack Solution for a Host-Based LONWORKS Device

# LonTalk Platform for FTXL Transceivers

The LonTalk Platform for FTXL Transceivers is a set of development tools, APIs, firmware, and chips for developing host-based LONWORKS devices that use the LonTalk API and an FTXL Transceiver.

An FTXL Transceiver is an FT 3190 Smart Transceiver with firmware that implements the data link layer (layer 2) of the LonTalk protocol, as shown in **Figure 3** on page 8. The host processor implements the remaining layers (layers 3 to 7). Included with the FTXL development tools is the FTXL LonTalk protocol stack, which implements layers 3 to 6 of the LonTalk protocol and runs on the host processor. Your application implements the application layer (layer 7).

For an FTXL device, you use an Altera Nios II processor as the host processor for your device's application and I/O. The Nios II processor runs on an Altera Cyclone II or Cyclone III FPGA device. The FTXL LonTalk protocol stack implements layers 3 to 6 of the LonTalk protocol, and the FTXL Transceiver implements layers 1 and 2, including the physical interface for the LONWORKS communications channel.

The FTXL LonTalk protocol stack includes a communications interface driver for the parallel link layer that manages communications between the FTXL LonTalk protocol stack within the Nios II host processor and the FTXL Transceiver. You need to include the physical implementation of the parallel link layer in your FTXL device design. However, you do not need to provide the software implementation of the parallel interface driver because it is included with the FTXL LonTalk protocol stack, nor can you modify the Echelon-provided implementation.

For FTXL device development, you use a C or C++ compiler that supports the Nios II processor. As with ShortStack development, you use the Echelon LonTalk Interface Developer utility to create the application framework. Your application uses an ANSI C API, the Echelon LonTalk API, to manage communications with the FTXL LonTalk protocol stack, FTXL Transceiver, and devices on the LONWORKS network.

Using an FTXL Transceiver, it is easy to add LONWORKS networking to a high-performance FPGA-based smart device.

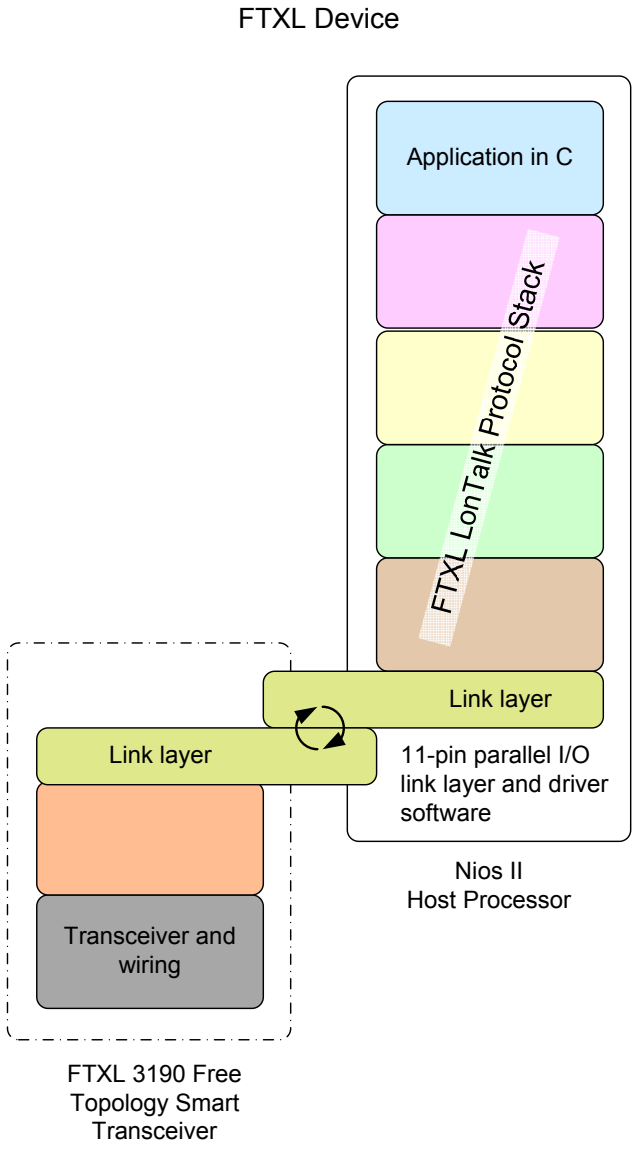



**Figure 3**. An FTXL Device

# *Comparing Neuron-Hosted, ShortStack, and FTXL Devices*

**Table 3** on page 9 compares some of the key characteristics of the Neuron-hosted and host-based solutions for LONWORKS devices.

**Table 3**. Comparing Neuron-Hosted and Host-Based Solutions for LONWORKS Devices

| Characteristic | Neuron-Hosted Solution | ShortStack Solution | FTXL Solution |
|---|---|---|---|
| Maximum number of network variables | 62 | 254 [1] | 4096 |
| Maximum number of aliases | 62 | 127 [2] | 8192 |
| Maximum number of addresses | 15 | 15 | 4096 |
| Maximum number of dynamic network variables | 0 | 0 | 4096 |
| Maximum number of receive transaction records | 16 | 16 | 200 |
| Maximum number of transmit transaction records | 2 | 2 | 2500 |
| Support for the LonTalk Extended Command Set | No | No | Yes [3] |
| File access methods supported | FTP [4] , DMF | FTP [4], DMF | FTP [4], DMF [5] |
| Link-layer type | N/A | 4- or 5-line SCI<br>or<br>6- or 7-line SPI | 11-line parallel I/O [6] |
| Typical host API runtime footprint | N/A | 5-6 KB code with 1 KB RAM (includes serial driver, but does not include optional API or ISI API) | 540 KB (includes LonTalk protocol stack, but does not include the application or operating system) |
| Host processor type | N/A | Any 8-, 16-, 32-, or 64-bit microprocessor or microcontroller | Altera Nios II embedded processor |
| Application development language | Neuron C | Any (typically ANSI C) | ANSI C or C++ for the Nios II processor |

Notes:

1. ShortStack Micro Servers running on FT 3150 or PL 3150 Smart Transceivers support up to 254 network variables. ShortStack Micro Servers running on FT 3120 Smart Transceivers support up to 240 network variables, and ShortStack Micro Servers running on PL 3120 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 Smart Transceivers support up to 127 aliases. ShortStack Micro Servers running on FT 3120 Smart Transceivers support up to 120 aliases. ShortStack Micro Servers running on PL 3120 Smart Transceivers support up to 62 aliases. A custom Micro Server can support up to 127 aliases, depending on available resources.
3. See the *LonTalk Control Network Protocol Specification*, EIA/CEA 709.1-B-2002, for more information about the extended command set (ECS) network management commands. This document is available from the IHS Standards Store: global.ihs.com/doc_detail.cfm?item_s_key=00391891&item_key_date=971131&rid=CEA.
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* on page 96.
6. The FTXL parallel I/O link-layer driver is included with the FTXL LonTalk protocol stack.

The FTXL solution provides the best performance and highest network capacity, but is limited using to an Altera Nios II host processor and the TP/FT-10 channel. The ShortStack solution provides support for any host processor (with available examples for both an Atmel ARM7 host processor and an Altera Nios II host processor), and supports both the TP/FT-10 and PL-20 channels. The ShortStack solution supports fewer network variables and aliases that the FTXL solution, but more network variables and aliases than the Neuron-hosted solution.

Because the ShortStack and FTXL solutions are both built on the LonTalk platform, they share a very similar API (the FTXL LonTalk API and the ShortStack LonTalk Compact API). Thus, migrating applications from one solution to the other is fairly easy. In addition, you can create applications that share a common code base for devices that use both solutions.

# Requirements and Restrictions for FTXL

The FTXL Developer's Kit supports only the FTXL 3190 Free Topology Smart Transceiver. It does not support other transceiver types.

The FTXL LonTalk protocol stack requires that the FTXL application use an embedded operating system. The FTXL Developer's Kit includes an example application that uses the Micrium µC/OS-II operating system, but you can use any embedded operating system that meets your application's requirements. And although the µC/OS-II operating system is a real-time operating system, the FTXL LonTalk protocol stack does not require the operating system to be a real-time operating system.

The FTXL LonTalk protocol stack and API require about 540 KB of program memory on the Nios II 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* on page 96 for more information about when to use FTP or the DMF feature.

# Development Tools for FTXL

To develop an application for a device that uses an FTXL Transceiver, you need a development system for the Nios II processor. In addition, you need the FTXL Developer's Kit, which includes:

- The FTXL LonTalk API
- The LonTalk Interface Developer utility for defining the interface for your FTXL device and generating the application framework
- Example FTXL applications
- A reference design for a Nios II processor and associated hardware for the FPGA device

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

You do not need the NodeBuilder Development Tool to use the FTXL Developer's Kit; however, the NodeBuilder Code Wizard that is included with the NodeBuilder tool, version 3 or later, can help you develop your Neuron C model file. The model file is used to define the device's interoperable interface.

# FTXL Architecture

An FTXL device consists of the following components:

- The FTXL 3190 Free Topology Smart Transceiver running the FTXL firmware
- A Nios II embedded processor running the following software:
    - An FTXL host application that uses the FTXL LonTalk API
    - The FTXL LonTalk protocol stack
    - The FTXL hardware abstraction layer (HAL)
    - The FTXL non-volatile data (NVD) driver
    - The FTXL operating system abstraction layer (OSAL)
    - An embedded operating system
    - The Altera SOPC Builder hardware abstraction layer (HAL)

**Figure 4** shows the basic architecture of an FTXL device.

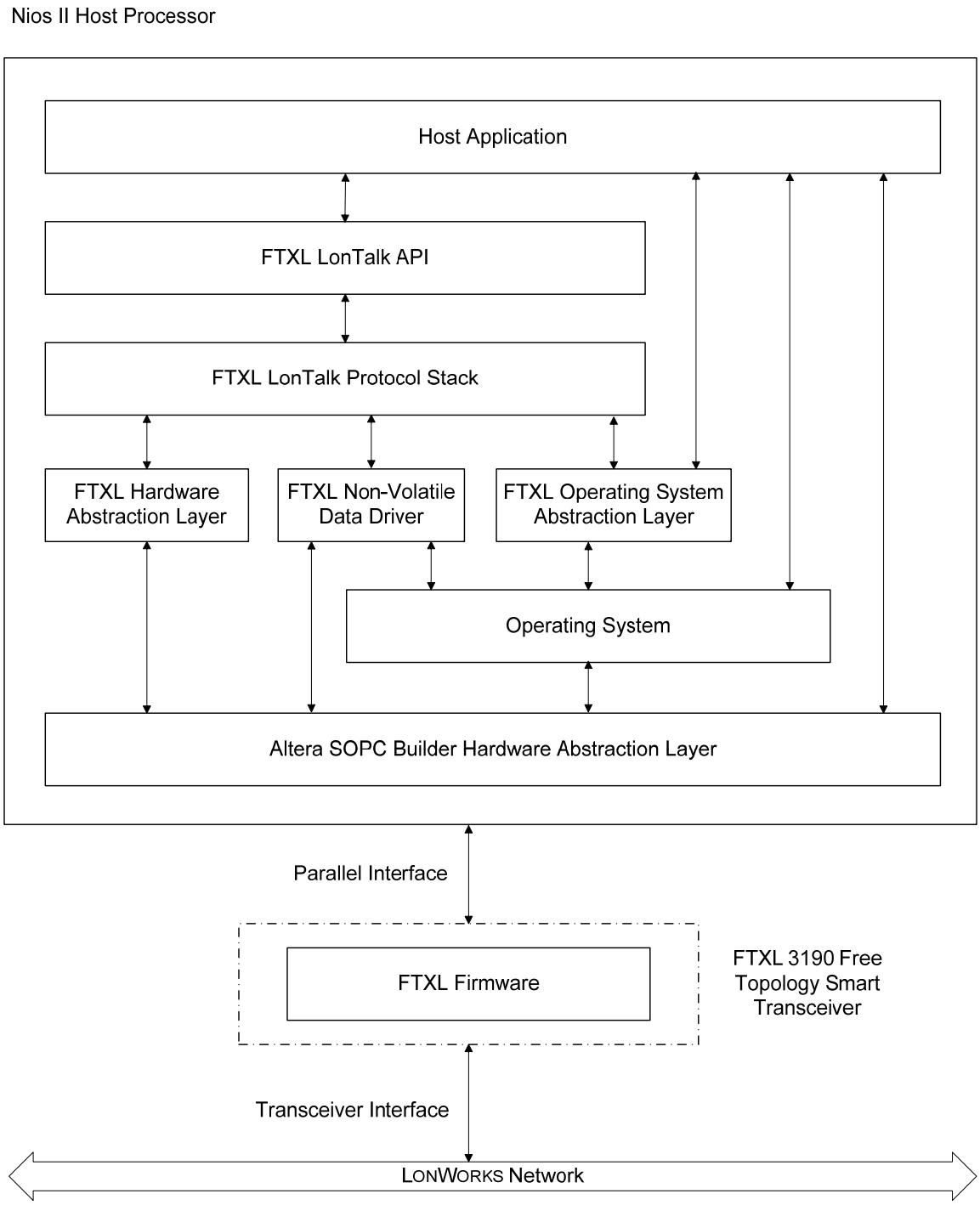



**Figure 4**. FTXL Architecture

The FTXL Developer's Kit includes the FTXL LonTalk API and a precompiled library that implements the FTXL LonTalk protocol stack. The kit also includes source code for additional operating system and hardware APIs that you compile and link with your application. The FTXL 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 FTXL LonTalk API consists of the following types of functions:

- Functions to initialize the FTXL 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.

# The FTXL Developer's Kit

The FTXL Developer's Kit consists of two components: a hardware component and a software component. See the *FTXL Hardware Guide* for information about the hardware component of the FTXL Developer's Kit.

The software component contains the software required to develop FTXL applications that use an FTXL Transceiver:

1. The FTXL LonTalk protocol stack library and FTXL LonTalk API
2. ANSI C source code for event handler functions.
3. Portable ANSI C source code for the reference implementations of the APIs for the operating system and hardware.
4. The LonTalk Interface Developer utility that you use to generate device interface data, device interface files, and a skeletal application framework.
5. Example applications that run on the reference design hardware.

The software component of the FTXL Developer's Kit is available as a free download from the Echelon Web site: www.echelon.com/downloads. You also must acquire a licence from Echelon to use the FTXL Developer's Kit.

# Overview of the FTXL Development Process

**Figure 5** on page 14 shows a high-level overview of the development process for an FTXL application. The basic process includes the following steps:

1. Use the Altera Quartus II software and SOPC Builder tool, with input from FTXL hardware components and your FPGA design, to generate compiled hardware description files.
2. Use the LonTalk Interface Developer utility, with input from a model file that you create, to generate application framework files and interface files.
3. Use the Altera Nios II EDS IDE to create the FTXL application, with input from:
   - The application framework files generated by the LonTalk Interface Developer utility

- The FTXL hardware abstraction layer (HAL) files, which you might need to modify
- The FTXL operating system abstraction layer (OSAL) files, which you might need to modify
- The FTXL non-volatile data (NVD) driver files, which you might need modify
- The FTXL LonTalk protocol stack

Because an FTXL device is comprised of both hardware and software components, 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.

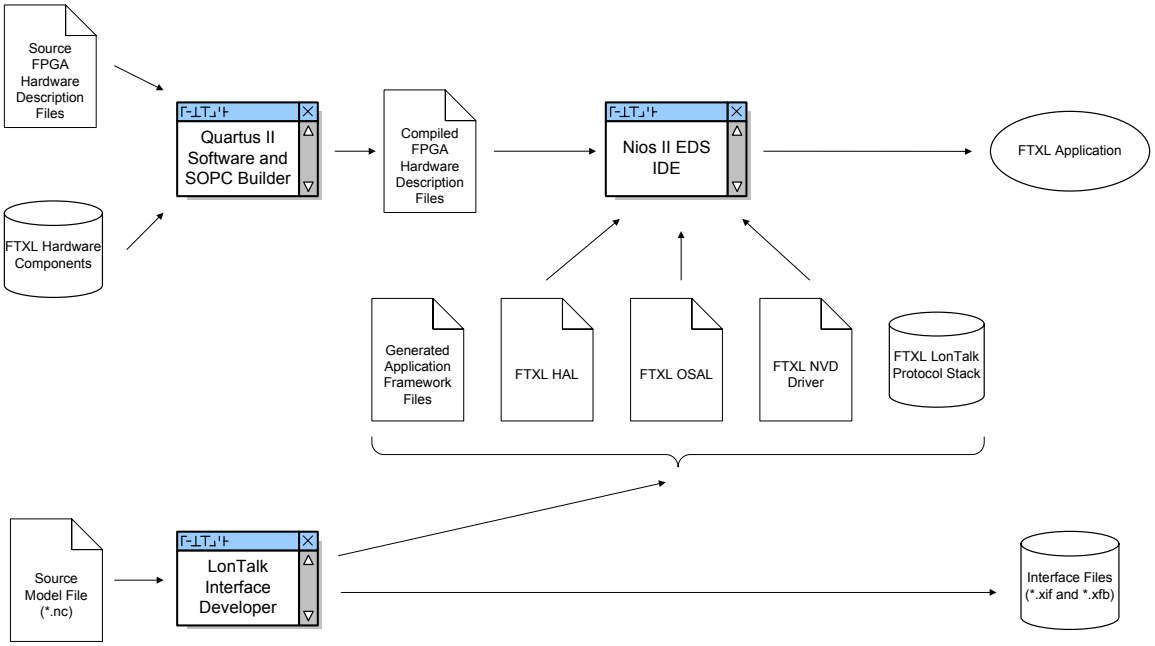



**Figure 5**. Overview of the FTXL Development Process

For more information about hardware development for an FTXL device, see the *FTXL Hardware Guide*.

This manual describes the software development process for creating an FTXL device, which includes the general tasks listed in **Table 4**.

**Table 4**. Tasks for Developing Software for an FTXL Device

| Task | Additional Considerations | Reference |
|---|---|---|
| Install the FTXL Developer's Kit and become familiar with it | | Chapter 2, *Getting Started with FTXL*, on page 17 |

| Task | Additional Considerations | Reference |
|---|---|---|
| Select an FPGA device and load it with Nios II processor and related hardware | The FTXL application runs on a Nios II embedded processor, which is implemented on an FPGA device. You must meet the FTXL hardware and software requirements to ensure that the FTXL device has sufficient RAM and non-volatile memory. | The *FTXL Hardware Guide* |
| Integrate the FTXL application with your device hardware | You integrate the FTXL Transceiver with the device hardware. You can reuse many parts of a hardware design for different applications to create different FTXL devices. | The *FTXL Hardware Guide*<br>Chapter 6, *Working with the Nios II Development Environment*, on page 101 |
| Test and verify your hardware design | You must ensure that the host processor and the FTXL Transceiver can communicate using the parallel interface. The FTXL Developer's Kit includes a Bring-Up application to help test and verify the communications interface. | The *FTXL Hardware Guide* |
| 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 3, *Creating a Model File*, on page 23 |
| Structure the layout and interoperable interface of your FTXL 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 or ShortStack application, or use the NodeBuilder Code Wizard included with the NodeBuilder Development Tool to create the required code using a graphical user interface. | Chapter 3, *Creating a Model File*, on page 23<br>Appendix C, *Neuron C Syntax for the Model File*, on page 125 |

| Task | Additional Considerations | Reference |
|---|---|---|
| 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 4, *Using the LonTalk Interface Developer Utility*, on page 55 |
| Complete the FTXL 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 5, *Developing an FTXL Application*, on page 73<br>Appendix D, *FTXL LonTalk API*, on page 147 |
| Modify the FTXL Operating System Abstraction Layer (OSAL) files for your application's operating system | If you use the Micrium µC/OS-II operating system, you can use the OSAL files that are included with the FTXL Developer's Kit. | *The FTXL Operating System Abstraction Layer* on page 157 |
| 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 FTXL Developer's Kit, make minor modifications to one of these drivers, or implement your own driver. | *Providing Persistent Storage for Non-Volatile Data* on page 77 |
| Modify your application to interface with a LONWORKS network by using the FTXL 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 5, *Developing an FTXL Application*, on page 73<br>Appendix D, *FTXL LonTalk API*, on page 147 |
| Test, install, and integrate your FTXL device using a LONWORKS network tool such as the LonMaker Integration Tool | | The *LonMaker User's Guide* |

# 2

# Getting Started with FTXL

This chapter describes the FTXL Developer's Kit and how to install it.

# FTXL Developer's Kit Overview

The FTXL Developer's Kit is a development toolkit that contains the hardware designs, software designs, and documentation needed for developing applications that use an FTXL Transceiver. The kit includes the following components:

- Hardware and software design files for the FPGA design, including Quartus II files, SOPC Builder files, and Nios IDE files
- Hardware component files for the FPGA development board
- The FTXL LonTalk protocol stack and FTXL LonTalk API, delivered as a C object library
- Software source files for the FTXL LonTalk API
- A set of example programs that demonstrate how to use the FTXL LonTalk API to communicate with a LONWORKS network
- The LonTalk Interface Developer utility, which defines parameters for your FTXL host application program and generates required device interface data for your device
- Documentation, including this *FTXL User's Guide*, the *FTXL Hardware Guide*, and HTML documentation for the FTXL API

The FTXL Developer's Kit also refers to three hardware development boards that are available from devboards GmbH, www.devboards.de. European customers can also obtain these boards from EBV Elektronik GmbH, www.ebv.com. The FTXL Developer's Kit uses these boards for its examples and reference designs. These boards are:

- The *DBC2C20 Altera Cyclone II Development Board*, which provides the FPGA device and peripheral I/O
- The *FTXL Adapter Board*, which primarily provides voltage regulation between the DBC2C20 development board and the FTXL Transceiver Board
- The *FTXL Transceiver Board*, which includes the FTXL 3190 Smart Transceiver Chip and a LONWORKS network connector

Contact your Altera representative for information about acquiring a Nios II development license.

See the *FTXL Hardware Guide* for more information about the hardware development boards and the reference designs for the FTXL Developer's Kit.

The software for the FTXL Developer's Kit is available as a free download from www.echelon.com/ftxl.

# Installing the FTXL Developer's Kit

The FTXL Developer's Kit requires the following software:

- Altera Quartus II software, Version 7.2 or later
- Altera Nios II EDS integrated development environment (IDE), Version 7.2 or later

- Driver software for the Altera USB-Blaster download cable
- FPGA configuration data and software for the DBC2C20 development board (included with the FTXL Developer' Kit)

For more information about the Altera software products, see see Chapter 6, *Working with the Nios II Development Environment*, on page 101, and the Altera Web site for the Nios II processor, www.altera.com/products/ip/processors/nios2/ni2-index.html.

The following sections describe the hardware and software requirements, and how to install the FTXL Developer's Kit.

## Hardware Requirements

For the FTXL Developer's Kit plus the Altera design software, your computer system must meet the following minimum requirements:

- Intel® Pentium® III 866 MHz processor
- 256 MB RAM
- 5 GB available hard disk space (includes the space required for the Altera tools)
- CD-ROM drive
- 1 available Universal Serial Bus (USB) port

The recommended specifications for your computer system include:

- Intel Pentium 4 2.0 GHz processor
- 1 GB RAM
- 5 GB available hard disk space
- CD-ROM or DVD-ROM drive
- 2 available USB ports

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

- LONWORKS compatible network interface, such as a U10 USB Network Interface or an *i*.LON 100 Internet Server
- A LONWORKS TP/FT-10 network cable, with network terminator

## Software Requirements

For the FTXL Developer's Kit plus the Altera design software, your computer system must meet one of the following minimum requirements:

- Microsoft® Windows® XP, plus Service Pack 2 or later
- Microsoft Windows Vista

The following software is optional, depending on your requirements:

- Adobe Reader 7.0.8 or later

## DBC2C20 Software

Although the DBC2C20 Altera Cyclone II Development Board includes a set of software for general FPGA development, you do not need to install any of the DBC2C20 software to work with the FTXL Developer's Kit. All of the necessary FPGA components and other software for the DBC2C20 development board are installed with the FTXL Developer's Kit.

## Installing the FTXL Developer's Kit

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

1. Download the FTXL Developer's Kit from www.echelon.com/ftxl. Although the download is free, you must agree to the licence terms for the FTXL Developer's Kit when you download it.
2. Double click the **FtxlDevKit100.exe** file that you downloaded. The Echelon FTXL Developer's Kit main installer window opens.
3. Follow the installation dialogs to install the FTXL Developer's Kit onto your computer.

After you install the kit, you can integrate it into your Nios II application development environment, as described in Chapter 6, *Working with the Nios II Development Environment*, on page 101.

In addition to the FTXL Developer's Kit, the installation program also installs:

- LONMARK® Resource Files
- LONMARK Standard Program ID Calculator
- NodeBuilder Resource Editor

# FTXL API Files

The FTXL LonTalk protocol stack and FTXL LonTalk API are provided as a C object library. In addition, the FTXL Developer's Kit includes a set of portable ANSI C files that accompany the API, which are listed in **Table 5**. These files are contained in the [*FTXL*]**\Core** directory (where [*FTXL*] is the directory in which you installed FTXL, usually **C:\LonWorks\FTXL**). In addition, there is a backup of these files in a ZIP file in the [*FTXL*]**\SourceArchive** directory.

The LonTalk Interface Developer utility automatically copies these files into the project folder, but does not overwrite existing files with the same names.

**Table 5**. FTXL LonTalk API Files

| File Name | Description |
|---|---|
| **FtxlApi.h** | Function definitions for the FTXL LonTalk API |
| **FtxlHal.h**<br>**FtxlHal.c** | Functions for the FTXL hardware abstraction layer (HAL) |

| File Name | Description |
|---|---|
| **FtxlHandlers.c** | Function definitions for the FTXL event handler functions and callback handler functions |
| **FtxlNvdFlashDirect.c**<br>**FtxlNvdFlashFs.c**<br>**FtxlNvdUserDefined.c** | Functions for managing non-volatile data |
| **FtxlOsal.h**<br>**FtxlOsal.c** | Functions for the FTXL operating system abstraction layer (OSAL) |
| **FtxlTypes.h** | C type definitions that are used by the FTXL LonTalk API |
| **libFtxl100.a** | C library for the FTXL LonTalk protocol stack and FTXL LonTalk API |
| **LonPlatform.h** | Definitions for adjusting your compiler and development environment to the requirements of the FTXL 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 FTXL 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*, on page 111.

For more information about the LonTalk Interface Developer utility, see Chapter 4, *Using the LonTalk Interface Developer Utility*, on page 55.

# Example FTXL Applications

The FTXL Developer's Kit includes two example applications that run on the reference designs for the development boards that are available from devboards.de GmbH. The first example is a simple FTXL application that simulates a voltage amplifier, whereas the second example demonstrates the use of dynamic network variables and changeable-type network variables.

See Appendix G, *Example FTXL Applications*, on page 195, for more information about these examples.

# 3

# 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*, on page 125.

# 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 an FTXL 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 Smart Transceivers. However, you do not need to be proficient in Neuron C to create a model file for an FTXL application because the model file does not include executable code. All tools required to process model files are included with FTXL; you do not need to license another Neuron C development tool to work with an FTXL model file. The model file uses Neuron C Version 2.1 declaration syntax.

The LonTalk Interface Developer utility 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 a 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 Release 3 or later of the NodeBuilder 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 FTXL

Developer's Kit, and must be licensed separately. See the *NodeBuilder User's Guide* for details about using the NodeBuilder Code Wizard.

See Appendix C, *Neuron C Syntax for the Model File*, on page 125, 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 the LonMaker Integration 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 an FTXL 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* on page 132 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 FTXL 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 FTXL 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 the *Defining a Functional Block* on page 27) 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* on page 30 for more information about implementing changeable-type network variables for FTXL applications.

You can also define your own nonstandard data types. The NodeBuilder Resource Editor utility, which is included with the FTXL Development 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 to look through the standard functional profiles, as described in *Defining a Resource File* on page 40. You can find detailed documentation for each of the standard functional profiles at types.lonmark.org[2].

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* on page 40.

**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**

[2] Use the Windows Internet Explorer browser to view this site.

- 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* on page 40, 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 the LonMaker Integration Tool, to connect these two devices, allowing the switch to control the lighting device, as shown in **Figure 6**.

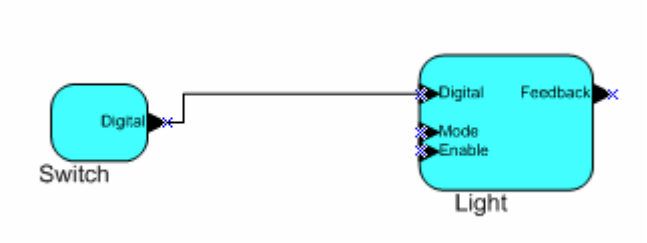



**Figure 6**. Simple Switch Controlling a Single Light

A single network variable can be connected to multiple network variables of the same type but opposite direction. The example in **Figure 7** shows the same switch being used to control three lights.

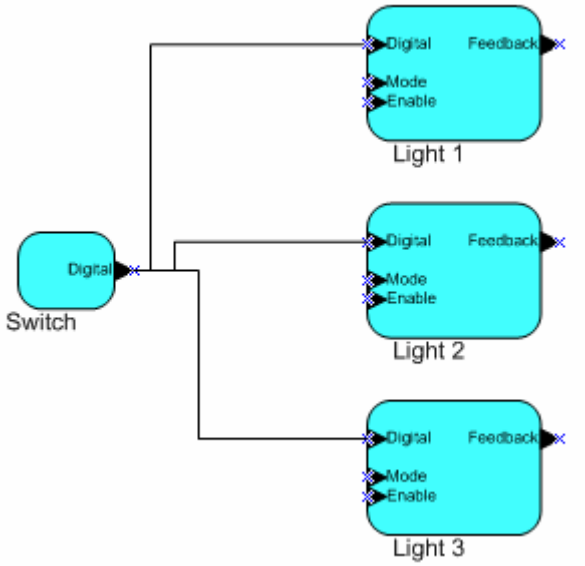



**Figure 7**. Simple Switch Controlling Three Lights

The FTXL 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 FTXL application updates a value for an output network variable, it uses a simple API function call to have the FTXL LonTalk protocol stack propagate it.

Through a process called *binding* that takes place during network design and installation, the FTXL 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 FTXL stack assembles and sends the appropriate packets to these devices. Similarly, when the FTXL 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 the LonMaker Integration tool, can change the type of a changeable-type network variable. The FTXL device does not initiate type changes.

To create a changeable-type network variable for an FTXL 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 FTXL 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 LonMaker browser provides integrators with a user interface to change network variable types. However, you might want to 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.

The LonMaker Integration tool, Turbo Edition, supports changeable-type network variables. However, if you use LonMaker 3.0 or earlier to manage an FTXL device with changeable-type network variables, you must explicitly set the CP value in the LonMaker browser (or in a device plug-in) to inform the FTXL device of the type changes in addition to using the "Change Network Variable Type" facility that is provided with LonMaker 3.0 or earlier to change the type of a network variable in the LNS database.

See *Handling Changes to Changeable-Type Network Variables* on page 88 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* on page 40, 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 CPNV) 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 CPNV can also provide your application with detailed notification of updates to the configuration property. However, a CPNV is limited to a maximum of 31 bytes, and an FTXL application is limited to a maximum of 4096 network variables, including CPNVs. Use the **network … config_prop** syntax described in *Declaring a Configuration Network Variable* on page 143 to implement a configuration property as a configuration network variable. By convention, CPNV 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 an FTXL 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 (CPNVs) 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* on page 140 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 for more information about the LONWORKS file transfer protocol; see *Using Direct Memory Files* on page 96 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 feature 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 (CPNVs), 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 (CPNV), of type **SCPToffset**. This CPNV 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.

**Table 6** 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.

**Table 6**. Functional-Block Members for the Four-Channel ADC

| 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 (CPNV) 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 property network variable can be type-inheriting; however, for configuration network variable arrays and arrays of configuration network variables (CPNVs), 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, you should specify the configuration property initializer option in the property list rather than in the declaration. Likewise, you should 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 proprietary interfaces to LONWORKS devices as described in Chapter 5, *Developing an FTXL Application*, on page 73.

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 LonMaker Integration 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 FTXL Developer's Kit 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 LNS tools, including the LonMaker tool assume a default scope of 3 for all user resources. LNS automatically sets the scope to the highest (most specific) applicable scope level. However, if you use LNS 3.0 or earlier with scope 4, 5, or 6 resource files, you must explicitly set the scope in LNS so that LNS uses the appropriate scope. See the *NodeBuilder User's Guide* for information about developing a plug-in to set the scope, or see the *LonMaker 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*, on page 119, 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 an FTXL device, or because it refers to attributes that are determined by the FTXL 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 FTXL model file.

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

## *Anonymous Top-Level Types*

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

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

This statement is not acceptable 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 shown in **Figure 8** on page 46. 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 LonMaker Integration 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 *How Authentication Works* on page 45. 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. **Recommended**: Set the device's domain to an invalid 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

**Figure 8** on page 46 illustrates the process of authentication:

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 FTXL LonTalk protocol 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.

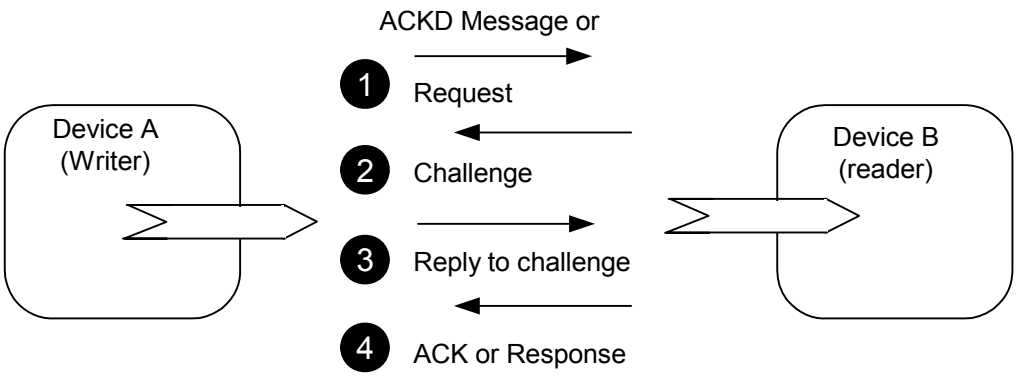



**Figure 8**. Authentication Process

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, an FTXL 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 FTXL LonTalk protocol 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:

- The address table
- The alias table
- The domain table
- The network variable configuration table

See the *LonTalk Control Network Protocol Specification*, EIA/CEA 709.1-B-2002, for more information about these tables. This document is available from the IHS Standards Store:

http://global.ihs.com/doc_detail.cfm?item_s_key=00391891&item_key_date=971131&rid=CEA.

See Appendix E, *Determining Memory Usage for FTXL Applications*, on page 181, for information about how to calculate the memory requirements for you FTXL 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.

**Recommendation**: Whenever possible, use the LonTalk Interface Developer utility-generated size for the address table.

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 FTXL LonTalk protocol 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.

**Recommendation**: Whenever possible, use the LonTalk Interface Developer utility-generated size for the alias table.

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.

**Recommendation**: Whenever possible, use the LonTalk Interface Developer utility-generated number of domain table entries.

## 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* on page 68 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 FTXL LonTalk protocol 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 FTXL LonTalk protocol 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* on page 64.

**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 (CPNV). 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* on page 96 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 for more information about the LONWORKS file transfer protocol.

# 4

# Using the LonTalk Interface Developer Utility

You use the model file, described in Chapter 3, 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 FTXL 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 FTXL 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 3, *Creating a Model File*, on page 23.
2. Start the LonTalk Interface Developer utility: from the Windows **Start** menu, select **Programs → Echelon FTXL Developer's Kit → 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* on page 61.
4. Add the **FtxlDev.h** ANSI C header file to your FTXL 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 FTXL LonTalk API files and FTXL LonTalk protocol stack library, **libFtxl100.a**) to your project directory.

In general, you should limit changes to the LonTalk Interface Developer utility-generated files. 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 FTXL LonTalk API files.

After you have created the LonTalk Interface Developer utility-generated files, you need to modify and add code to your application, using the FTXL LonTalk API, to implement desired LONWORKS functionality into your FXTL application. See Chapter 5, *Developing an FTXL Application*, on page 73, for information about how to use the FXTL LonTalk API calls to implement LONWORKS tasks.

# *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 FTXL 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.

**Recommendation**: 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.

## Specifying the FTXL Transceiver Configuration

From the FTXL Transceiver Configuration page of the utility, you can specify the clock speed for the FTXL Transceiver.

## Specifying Service Pin Held Events

When you press the local service pin on the device, the FTXL LonTalk protocol stack sends a service-pin message on the LONWORKS network and signals a service-pin event to the application. However, when you press *and hold* the local service pin on the device, whether the FTXL LonTalk protocol stack sends a service-pin-held event depends on how you configure it.

From the System Preferences page of the utility, you can configure the following behavior for how the FTXL LonTalk API handles a service-pin-held event:

- Whether the FTXL LonTalk API sends a notification of a service-pin-held event to the application.
- How long the local service pin must be pressed and held before the FTXL LonTalk API notifies the application of the event. You can specify from 1 to 30 seconds, or accept a default of 10 seconds.

Receiving service-pin-held events is optional, and how the application processes the events depends on the requirements of the application. For example, many devices support an emergency recovery feature that is triggered by pressing and holding the service pin for a prolonged amount of time (typically 10 or 20s). Then the device moves to the unconfigured state (that is, calls **LonGoUnconfigured()**) or uses another method to return to a factory state.

## Configuring the FTXL LonTalk Protocol 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.

**Recommendation**: Allow the LonTalk Interface Developer utility to calculate appropriate values for the stack configuration.

See *Managing Memory* on page 46 for more information about these values.

## 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

**Recommendation**: Allow the LonTalk Interface Developer utility to calculate appropriate values for the buffer configuration.

## 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-documation 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 FTXL 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.

## 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:

- The non-volatile data driver model

- The non-volatile data flush guard timeout value
- The 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 the Micrium µC/FS embedded file system)
- 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 FTXL LonTalk protocol 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 Nios II flash device name. If you use unstructured flash memory, leave the **Root** field blank.

Within the Nios development environment, the **system.h** file defines the root name. For the examples that are included with the FTXL Developer's Kit, the root name is **/dev/cfi_flash**, which is the root directory for the flash file system.

The FTXL 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.

## 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 FTXL Developer's Kit.

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.

## 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 3, *Creating a Model File*, on page 23 for more information about the model file.

## 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 FTXL 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.

## *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.

## *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* on page 61.

The Build Progress and Summary page shows the results of compilation and generation of the FTXL 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 *NodeBuilder 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 FTXL application:

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

- *project*.xif
- *project*.xfb

These files form the FTXL application framework, which defines the FTXL 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 FTXL 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**
- **FtxlHal.c**
- **FtxlHal.h**
- **FtxlHandlers.c**
- **FtxlNvdFlashDirect.c**
- **FtxlNvdFlashFs.c**
- **FtxlNvdUserDefined.c**
- **FtxlOsal.c**
- **FtxlOsal.h**
- **FtxlTypes.h**
- **libFtxl100.a**
- **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 **libFtxl100.a** library and the various 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* on page 64 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 FTXL 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 FTXL 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 FTXL project file (.lidprj extension).

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

**Important**: If your device is defined with a non-standard program ID, the device interface file cannot contain interoperable LONMARK constructs.

# 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**.

**Important**: 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).

  **Important**: 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 multibyte scalars (such as the **ncuLong** type), you must modify the initializer generated by the LonTalk Interface Developer utility if the type is a multibyte 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. FTXL 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.

An FTXL 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
```

**Example**: **Table 7** shows an example enumeration using the FTXL **LON_ENUM_*** macros, and the equivalent ANSI C enumeration.

**Table 7**. Enumerations in FTXL

| **FTXL Enumeration** | **Equivalent ANSI C Enumeration** |
|---|---|
| `LON_ENUM_BEGIN(Color) {`<br>`  red, green, blue`<br>`} LON_ENUM_END(Color);`<br><br>`typedef struct {`<br>`  …`<br>`  LON_ENUM(Color) color;`<br>`} Example;` | `enum {`<br>`  red, green, blue`<br>`} Color;`<br><br>`typedef struct {`<br>`  …`<br>`  Color color;`<br>`} 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 FTXL 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 *Developing an FTXL Application* on page 73 for information about the development of a FTXL application using the LonTalk Interface Developer utility-generated code.

The utility generates a 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 FTXL 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.

FTXL 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 FTXL 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 FTXL LonTalk API.

An FTXL application typically accesses a network variable value through the C global variable that implements the network variable. However, the FTXL 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 FTXL 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 an FTXL Application* on page 73 for more information about propagation of network variable updates.

# The Message Tag Table

Although the FTXL LonTalk protocol 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* on page 93 for more information about using message tags.

# 5

# Developing an FTXL Application

This chapter describes how to develop an FTXL application. It also describes the various tasks performed by the application.

# Overview of an FTXL Application

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

- Use the FTXL LonTalk API and FTXL LonTalk protocol stack
- Integrate the application with an operating system
- Provide persistent storage for non-volatile data
- Initialize the FTXL device
- Periodically call the FTXL 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 FTXL device

Most FTXL 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 FTXL LonTalk API*

Within the seven-layer OSI Model protocol, the FTXL LonTalk API forms the majority of the Presentation layer, and provides the interface between the FTXL LonTalk protocol stack in the Session layer and the host application in the Application layer, as shown in **Figure 9** on page 75.

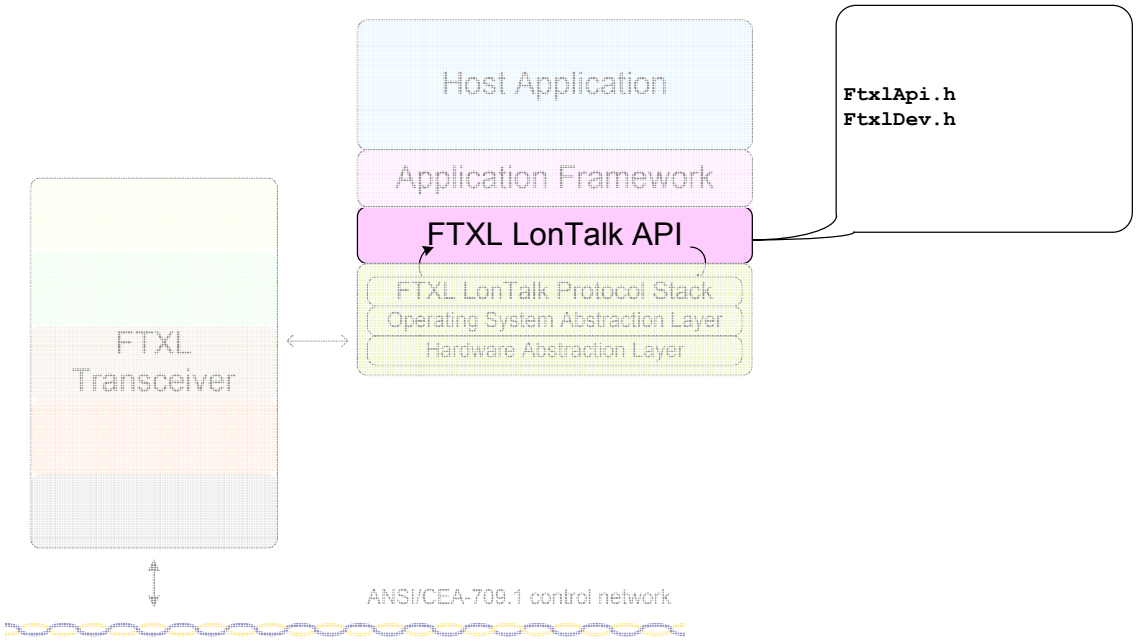



**Figure 9**. The FTXL LonTalk API within the OSI Model

The [*FTXL*]**\Core\libFtxl100.a** file contains the FTXL LonTalk protocol stack, the FTXL LonTalk API, and the parallel interface driver, which together allow your FTXL application to handle network events, propagate network variables, respond to network variable poll requests, and so on.

An FTXL application must include the **FtxlDev.h** file to be able to use the FTXL 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 [*FTXL*]**\Core\FtxlApi.h** file, which contains definitions for accessing the FTXL LonTalk API.

The [*FTXL*]**\Core\FtxlHandlers.c** source file contains stubs for the event handler functions and callback handler functions that the FTXL 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 FTXL 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 FTXL 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 FTXL product might affect these API files

## Callbacks and Events

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

The FTXL LonTalk API uses a *callback* when the API needs a return value from the application immediately. A callback can occur in one of the FTXL LonTalk protocol 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 FTXL LonTalk API functions or perform time-intensive operations.

The FTXL 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 FTXL LonTalk protocol 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 FTXL 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, *FTXL LonTalk API*, on page 147, for a list of the callback handler functions and event handler functions.

## Integrating the Application with an Operating System

The FTXL LonTalk protocol stack requires an FTXL application to use an operating system. The FTXL Developer's Kit includes example applications that use the Micrium µC/OS-II operating system, but you can use any embedded operating system that meets your application's requirements. Although the µC/OS-II operating system is a real-time operating system, the FTXL LonTalk protocol stack does not require the operating system to be a real-time operating system.

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

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

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

The FTXL Developer's Kit includes source code for FTXL OSAL files that use the syntax of the Micrium µC/OS-II operating system. To use a different operating system, you must modify the OSAL files to implement the API for that operating system.

For more information about the FTXL OSAL functions, see *The FTXL Operating System Abstraction Layer* on page 156. For information about configuring the operating system, see *Configuring the Operating System* on page 160 and see *Configuring the Micrium µC/OS-II Operating System* on page 165.

## Providing Persistent Storage for Non-Volatile Data

The FTXL LonTalk protocol 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 FTXL Transciever, the FTXL application must implement the API's functions so that both the FTXL LonTalk protocol 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 FTXL 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 FTXL LonTalk protocol stack, and the third segment is used to store data maintained by the application. Examples of data maintained by the FTXL LonTalk protocol 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 FTXL LonTalk protocol stack reads non-volatile data (loads it into RAM) only during device initialization. Included with the data is a header that the FTXL LonTalk protocol stack uses for validation. Within this header is an application identifier, generated by the LonTalk Interface Developer utility, that allows the FTXL LonTalk protocol 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 FTXL LonTalk protocol 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 FTXL LonTalk protocol stack does not immediately update the corresponding persistent memory. Instead, the FTXL LonTalk protocol 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 FTXL LonTalk protocol stack supports large configurations.

If the FTXL LonTalk protocol 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 FTXL LonTalk protocol stack is shut down by calling the **LonExit()** function, the FTXL LonTalk protocol 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 FTXL LonTalk protocol stack maintains a set of flags (one for each segment) that survive an unorderly shutdown so that the FTXL LonTalk protocol stack can detect the unorderly shutdown at the next restart.

The FTXL LonTalk protocol 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 FTXL device has likely lost updates from a network manager that it has already acknowleged. If the FTXL 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 FTXL LonTalk protocol stack deletes the configuration data, logs a configuration checksum error, and goes unconfigured. You can restore the configuration by recommissiong 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 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 FTXL LonTalk protocol 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 FTXL LonTalk protocol 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 FTXL LonTalk protocol 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 FTXL LonTalk protocol stack sets the device to the unconfigured state, generates a configuration checksum error, and calls the **LonNvdDelete()** callback handler function.

The FTXL LonTalk protocol stack handles the deserialization of the data for the **LonNvdSegNetworkImage** and **LonNvdSegNodeDefinition** segments, but not for the application-defined **LonNvdSegApplicationData** segment. Instead, the FTXL LonTalk protocol 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 FTXL LonTalk protocol stack processes a network management message that affects any of its configuration data, the FTXL LonTalk protocol stack checks whether there is an NVD transaction for the affected segment. If not, FTXL LonTalk protocol stack starts a timer and calls the **LonNvdEnterTransaction()** callback handler function for the segment. If there is already a transaction pending, the FTXL LonTalk protocol stack simply resets the timer.

When the timer expires, the FTXL LonTalk protocol 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 hanlder 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 FTXL LonTalk protocol 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. Intstead, theFTXL LonTalk protocol 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 FTXL LonTalk protocol 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 FTXL LonTalk protocol stack. In addition, FTXL LonTalk protocol 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 FTXL LonTalk protocol stack's knowledge. The application must inform the FTXL LonTalk protocol stack when this occurs so that the FTXL LonTalk protocol 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 an FTXL Application

The **main()** function of an FTXL 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 an FTXL application includes a call to the **LonInit()** API function to initialize the FTXL LonTalk protocol stack and the FTXL Transceiver. 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 FTXL Transceiver 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 FTXL events. This function then calls event handler functions (such as **LonNvUpdateOccurred()** or **LonNvUpdateCompleted()**).

The following sections describe the tasks that an FTXL application performs during its life cycle.

# Initializing the FTXL Device

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

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 FTXL LonTalk API, the FTXL LonTalk protocol stack, and the FTXL Transceiver. The main application thread must call this function before it calls any other FTXL LonTalk API functions.

The **LonInit()** registers the FTXL device interface data with the FTXL LonTalk protocol 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 FTXL LonTalk protocol stack, and then call the **LonInit()** function to restart the protocol 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* on page 76, your FTXL 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 the **LonInit()** function.

The FTXL LonTalk API calls the **LonEventReady()** callback handler function whenever an event has been posted. This function is typically called from an FTXL LonTalk protocol 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 FTXL 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 FTXL 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 FTXL LonTalk protocol 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 FTXL LonTalk protocol 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.

**Recommendation**: In the absence of measured data for the network, 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, *FTXL LonTalk API*, on page 147, 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 FTXL 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 FTXL LonTalk protocol 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 FTXL LonTalk protocol stack, but does not indicate successful completion of the update itself.

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

After the update is complete, the FTXL LonTalk protocol stack informs the **LonEventReady()** callback handler function in the FTXL 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. **Figure 10** shows the control flow for processing a network variable update.

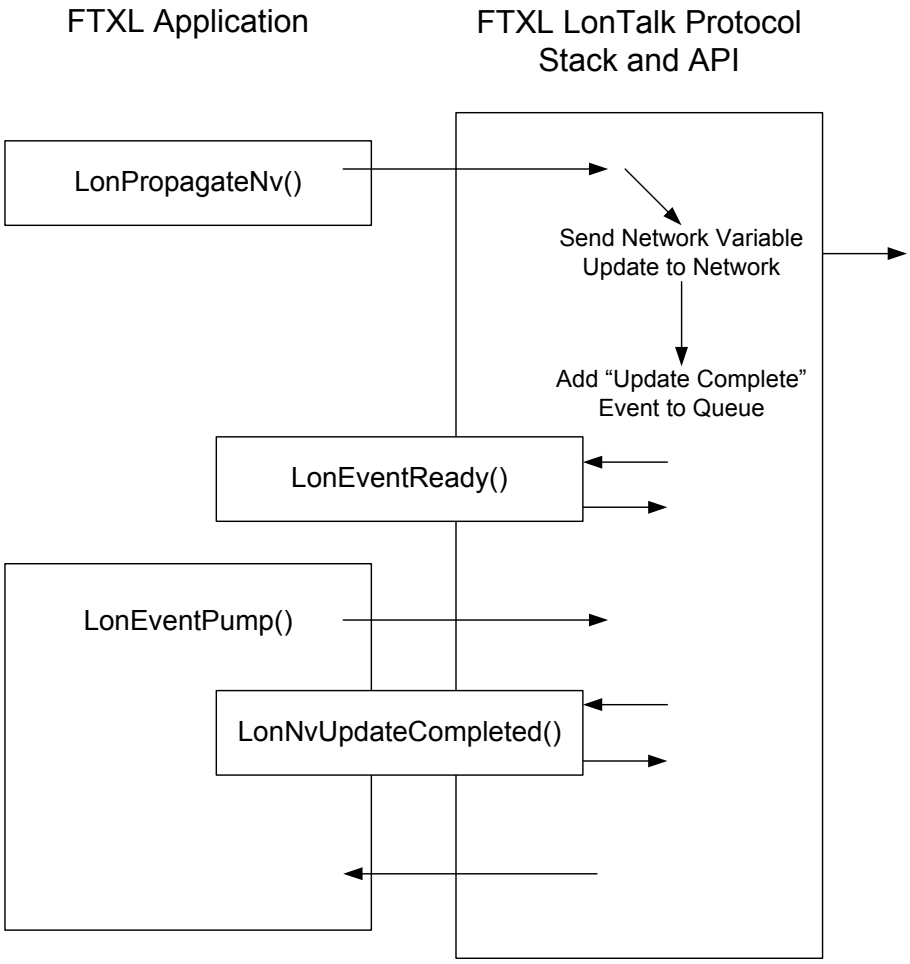



**Figure 10**. Control Flow for Sending a Network Variable Update to the Network

In the case of an unacknowledged or repeated service type, the FTXL LonTalk protocol stack considers the update complete when it has finished sending the update to the network. In the case of an acknowledged service type, the FTXL LonTalk protocol 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 FTXL LonTalk protocol 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 FTXL LonTalk protocol 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 FTXL LonTalk protocol 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 FTXL 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. **Figure 11** shows the control flow for receiving a network variable update.

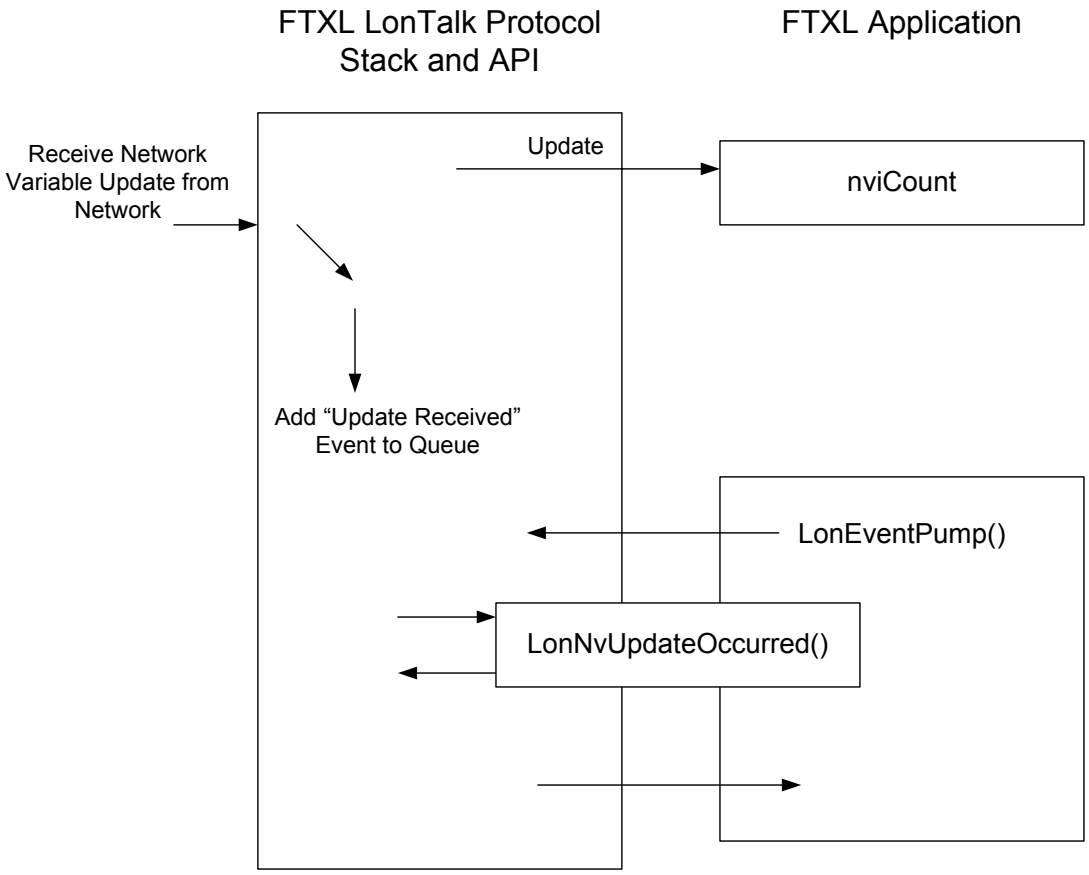



**Figure 11**. Control Flow for Receiving a Network Variable Update

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*, on page 83. 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*, on page 83. 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 FTXL LonTalk protocol 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 LonMaker 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* on page 202 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 (CPNVs), these type-inheriting CPNVs also change their type at the same time. If this type-inheriting CPNV 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 LNS derives 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 FTXL LonTalk protocol stack about the current length of the changeable-type network variable. The current length information must be kept in non-volatile memory.

The FTXL 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 FTXL 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 FTXL LonTalk protocol 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 FTXL LonTalk protocol 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 FTXL LonTalk protocol stack calls this function when a dynamic network variable definition is changed.

- **LonNvDeleted()**

  The FTXL LonTalk protocol stack calls this function when a dynamic network variable is deleted.

For the **LonNvAdded()** and **LonNvTypeChanged()** event handler functions, the FTXL LonTalk protocol 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 FTXL LonTalk protocol stack calls the **LonNvTypeChanged()** event handler.

# *Communicating with Other Devices Using Application Messages*

Application messages are used to create a proprietary (that is, non-interoperable) interface for a device. 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 FTXL LonTalk protocol stack, which in turn transmits the message on the network. After the message is sent, the FTXL LonTalk protocol 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 FTXL 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 FTXL 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 FTXL Transceiver 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 FTXL LonTalk API calls your **LonResponseArrived()** event handler function for each response it receives.

## Receiving an Application Message from the Network

When the FTXL LonTalk protocol stack receives an application message from the network, it forwards the message to the **LonEventPump()** function in the FTXL LonTalk API, which in turn calls your **LonMsgArrived()** callback handler function. Your implementation of this function must process the application message.

The FTXL LonTalk protocol stack does not call the **LonMsgArrived()** callback handler function if an application message is received while the FTXL 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 FTXL LonTalk protocol 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 FTXL 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 FTXL application and implemented by the FTXL LonTalk protocol stack. Local network management commands are never propagated to the network. The FTXL Extended LonTalk APIs allow you to include handling of these local network management commands if your FTXL application requires it.

## Handling Reset Events

A network management tool can send a reset message to the FTXL 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 FTXL LonTalk protocol 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 FTXL 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 an FTXL device from the network affects only the FTXL stack, and does not cause a processor or application software reset.

## Querying the Error Log

The FTXL LonTalk protocol 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

An FTXL 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). An FTXL device supports both the extended command set and legacy network management commands. However, after the device receives any extended commands, it operates in the extended mode, and returns a negative response to legacy commands.

Any LNS-based tool communicates with an FTXL device using ECS commands (for example, during device commissioning), and thus places the device in extended mode. Some tools that are not based on LNS, such as the NodeUtil

utility, might not be able to communicate with a device that is in the extended mode.

To return an FTXL 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 LNS database, de-commission it.

You should not need to perform any of these tasks often because most network management tools use LNS or are compatible with ECS.

For more information about the LonTalk extended command set (ECS) network management commands, see the *LonTalk Control Network Protocol Specification*, EIA/CEA 709.1-B-2002. This document is available from the IHS Standards Store: http://global.ihs.com/doc_detail.cfm?item_s_key=00391891&item_key_date=971131&rid=CEA.

# 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 FTXL Transceiver. 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 FTXL Transceiver's native address space, but the FTXL LonTalk protocol 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 FTXL Transceiver 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 FTXL LonTalk protocol 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 FTXL Transceiver'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 file directory
- The template file
- The writeable CP value files (if any)
- The read-only CP value files (if any)

**Figure 12** on page 98 shows how the different memory address spaces relate to each other.

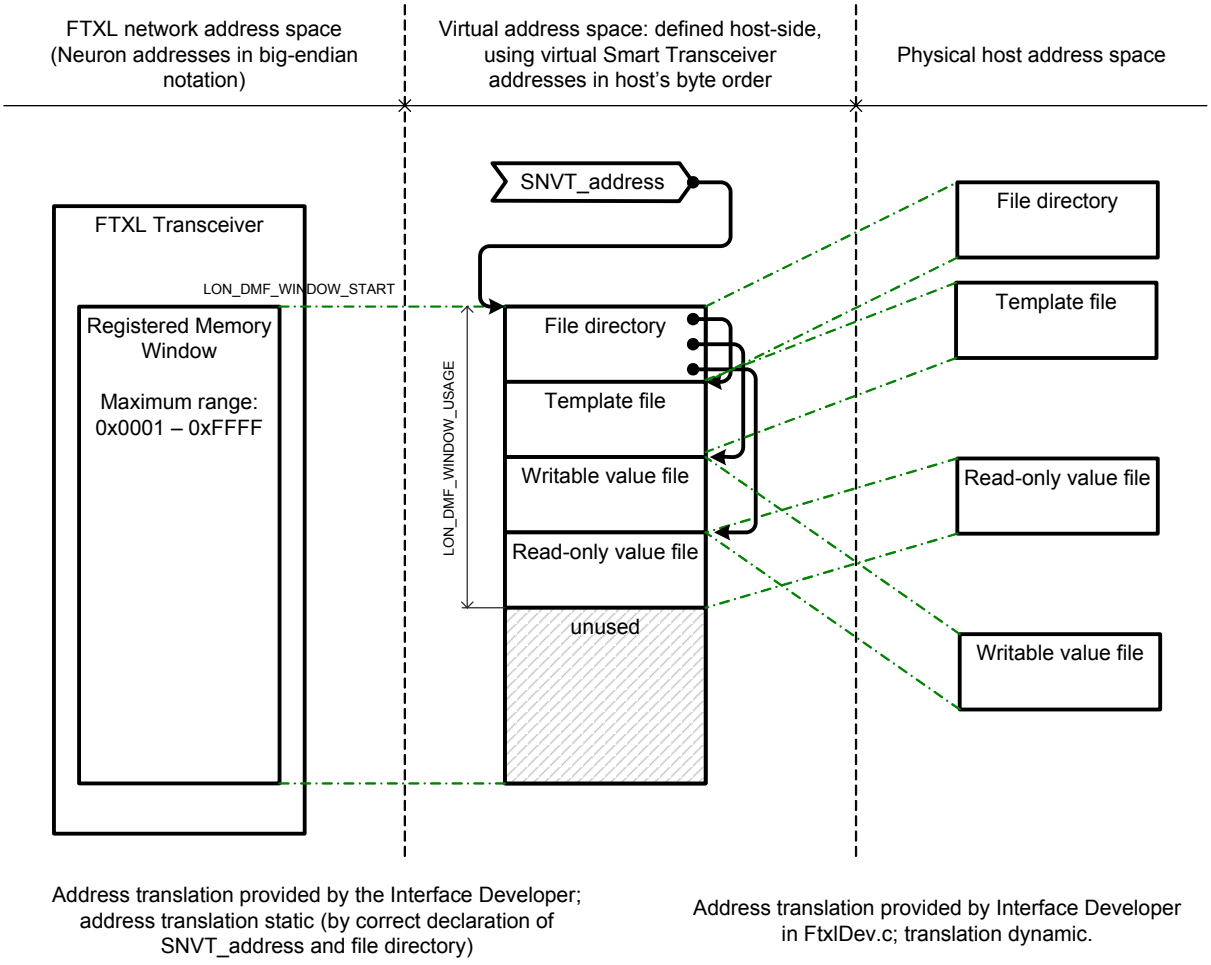



**Figure 12**. Relationship between Different Memory Spaces

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 FTXL Device

To perform an orderly shutdown of an FTXL 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 FTXL LonTalk protocol 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 FTXL LonTalk protocol stack's interface, you must reboot the device.

# 6

# Working with the Nios II Development Environment

This chapter describes how to set up the Nios II IDE for building and running your application for FTXL. It also describes how to import or create the application, and how to customize it.

See Appendix G, *Example FTXL Applications*, on page 195, for information about working with the Nios II IDE for the example applications.

# Development Tools

To develop your FTXL application, you use version 7.2 or later of the Altera Complete Design Suite, as listed in **Table 8**. You can obtain the Altera Complete Design Suite on DVD-ROM from Altera Corporation, or you can download the Web Edition of the tools from https://www.altera.com/support/software/download/nios2/dnl-nios2.jsp.

**Table 8**. Altera Complete Design Suite

| Quartus II Design Software for Windows |
|---|
| The Quartus II design software provides a suite of tools for system-level design, embedded software programming, FPGA and CPLD design, synthesis, place-and-route, verification, and device programming. Quartus II software supports all of Altera's current device families.<br><br>The Quartus II Web Edition is a subset of the Quartus II design software that provides support for selected Altera processors.<br><br>Both the Quartus II design software and the Quartus II Web Edition include the SOPC Builder tool, which is an automated system development tool that dramatically simplifies the task of creating high-performance system-on-a-programmable-chip (SOPC) designs. |
| ModelSim®-Altera VHDL & Verilog HDL Simulation Tool |
| The ModelSim-Altera software is an Altera-specific version of the Model Technology™ ModelSim simulation software, which supports behavioral simulation and testbenches for VHDL or Verilog hardware description languages (HDLs). The ModelSim-Altera software is included with Altera software subscriptions. |
| MegaCore IP Library |
| The MegaCore IP library includes some of Altera's most popular intellectual property (IP) cores, including a finite impulse response (FIR) compiler, a numerically controlled oscillator (NCO) compiler, a fast Fourier transform (FFT) compiler, several DDR SDRAM controllers, a QDRII SDRAM controller, an RLDRAM II controller, and a lightweight serial interconnect protocol. The MegaCore IP library is included with Altera software subscriptions. |
| Nios II Embedded Design Suite |
| The Nios II integrated development environment (IDE) is a graphical user interface (GUI) within which you can accomplish all Nios II embedded processor software development tasks, including editing, building, managing, and debugging embedded software programs. The Nios II IDE is included with Altera software subscriptions. |

For more information about installing the Altera Complete Design Suite, see *Quartus II Installation & Licensing for Windows*, available from the Quartus II Development Software Literature page at www.altera.com/literature/lit-qts.jsp.

# Using a Device Programmer for the FPGA Device

To load your hardware design, software application, and the FTXL LonTalk protocol stack, into the FPGA device, you can use a device programmer, such as the Altera USB-Blaster download cable, as described in **Table 9**.

**Table 9**. Device Programmer for the Nios II Processor

| Altera USB-Blaster Download Cable |
| --- |
| The USB-Blaster download cable interfaces to a standard PC USB port. This cable drives configuration or programming data from the PC to the device. For more information about the USB-Blaster, see the *USB-Blaster Download Cable User Guide*. |

The Windows driver for the USB-Blaster is in the [*Altera*]**\quartus\drivers\usb-blaster** directory, where [*Altera*] is the directory in which you installed the Altera Complete Design Suite, usually **C:\altera\72**.

To set up the programming hardware in the Nios II IDE:

1. Start the Nios II IDE.
2. Select **Tools → Quartus Programmer** to open the Chain Description File (*.cdf) for the project.
3. Click **Hardware Setup** to open the Hardware Setup window.
4. If you have already installed the Windows drivers for the USB-Blaster, it should appear in the **Available hardware items** area of the Hardware Setup window.
5. If the programming hardware that you want to use does not appear in the **Available hardware items** area of the Hardware Setup window, click the **Add Hardware** button to open the Add Hardware window.
    a. Select the appropriate programming cable or programming hardware from the **Hardware Type** dropdown list box.
    b. Select the appropriate port, baud rate, and server information, if necessary.
    c. Click **OK**.
6. Select the programming hardware that you want to use from the **Currently selected hardware** dropdown list box.
7. Click **Close** to close the Hardware Setup window.
8. Select **JTAG** from the **Mode** dropdown list box of the Chain Description File view for the project.
9. Select **File → Close** to close the Chain Description File.

You can save the Chain Description File (*.cdf) for use with other projects.

# Setting up the Nios II IDE

The development environment for FTXL applications is the Altera Nios II Embedded Design Suite integrated development environment (IDE). An FTXL

application consists of the following components, all of which need to be added to the Nios II IDE for FTXL application development:

- Your application code
- The application framework files generated by the LonTalk Interface Developer utility
- The FTXL LonTalk protocol stack library, which includes the FTXL LonTalk API
- The FTXL system library, which includes the system description for the Nios II processor and associated hardware

To set up the Nios II IDE to use the example FTXL applications, perform the following general steps:

1. Optional: Create a new workspace for each example application project.
2. Create a new application project (optionally based on one of the FTXL project templates). This step creates both the application project and the system library project for the application.
3. Run the LonTalk Interface Developer utility to generate and copy the necessary files for the project.
4. Refresh the Nios II C/C++ Projects pane.
5. Build the project.
6. As necessary, customize the system library and operating system settings.
7. Specify the properties for the application.

The following sections describe these steps. After you build the project, you can load it into the Nios II processor and run it.

See Appendix G, *Example FTXL Applications*, on page 195, for information about working with the Nios II IDE for the example applications.

## Creating a New FTXL Application Project

You can create each example project in a new workspace or use an existing workspace. To work in a new workspace, select **File → Switch Workplace** to open the Workspace Launcher window, from which you can select a new or existing workspace.

To create a new application project for the FTXL simple example application:

1. Select **File → New → Nios II C/C++ Application** to open the New Project window.
2. From the New Project window's **Select Project Template** selection box, select the **FTXL Simple** project.
3. Optional: Enter a project name in the **Name** field.
4. Specify a location for this project by selecting the **Specify Location** checkbox and specifying the location in the **Location** field. The directory name must not contain spaces. If you use the default location, your source files will be placed in the project workspace directory.

5. Specify the target hardware. Click **Browse** in the Select Target Hardware area to open the Select Target Hardware dialog.
    a. In the Select Target Hardware dialog, browse to the directory that contains your project's hardware description files and select the appropriate SOPC Builder system file (*.ptf).
    b. Click **Open** to select the file and close the Select Target Hardware dialog.
6. Do not modify the **CPU** field in the Select Target Hardware area; the name of the CPU is contained in the project's *.ptf file. However, if this file specifies more than one Nios II processor, you need to select which one the application project should use.
7. Click **Finish** to create the project and generate the project's system library.

## Running the LonTalk Interface Developer Utility

Before you can compile the newly created project, you must run the LonTalk Interface Developer utility to generate the application framework files and copy other required files to the project directory.

To run the LonTalk Interface Developer utility for the project:

1. Start the LonTalk Interface Developer utility from the Windows Start menu: **Start → Programs → Echelon FTXL Developer's Kit → LonTalk Interface Developer**.
2. From the Welcome to LonTalk Interface Developer page of the utility, specify a name and directory for a new project file, or click **Browse** to open an existing project file.
3. For each page of the LonTalk Interface Developer utility, specify the required project settings. For many of the settings, you can accept the utility's default settings. See Chapter 4, *Using the LonTalk Interface Developer Utility*, on page 55, or the utility's online help, for more information about using the LonTalk Interface Developer utility.
4. Click **Finish** on the Build Progress and Summary page of the utility to close the LonTalk Interface Developer utility.

Within the Nios IDE, right-click within the Nios II C/C++ Projects pane and select **Refresh** to see the newly generated and copied files for the project. You can now compile and build the project.

After you have generated the application framework files, you can add to and modify them to develop your own FTXL application. See Chapter 5, *Developing an FTXL Application*, on page 73, for more information about developing an FTXL application.

## Customizing the FTXL System Library

After you create the project, you can customize the project's system library, which includes specifying the settings for the operating system. See *Configuring the Operating System* on page 160 for information about determing the settings for the operating system.

To customize the system library properties for the operating system:

1. Right-click the system library from the Nios II C/C++ Projects pane and select **Properties** to open the Properties window.
2. In the Properties window, click **RTOS Options** to open the Options dialog for your operating system.
3. Within the Options dialog, set the appropriate values. If you use the Micrium µC/OS-II operating system, see *Configuring the Micrium µC/OS-II Operating System* on page 165 for information about the settings that are required for an FTXL application.

## Specifying the Properties for the Application

After you create the application project, you must customize its properties:

1. Right-click the application project folder from the Nios II C/C++ Projects pane and select **Properties** to open the Properties window.
2. In the Properties window, select **Associated System Library** in the left-hand pane to display the Associated System Library page. Ensure that the **System Library** field displays the correct system library for the project.
3. In the Properties window, select **C/C++ Build** in the left-hand pane to display the C/C++ Build page.
4. For all configurations (such as Debug and Release), set the following values from the Tool Settings tab of the C/C++ Build page:
    a. Expand **Nios II Compiler** in the left-hand pane and select **Preprocessor**.
    b. In the Defined Symbols box, click the **Add** button to add a new symbol with a value of **GCC_NIOS**. This symbol is used in the **LonPlatform.h** file to specify macros for the GNU Compiler Collection (GCC) compiler.
    c. Expand **Linker** in the left-hand pane and select **General**.
    d. In the Libraries box, click the **Add** button to add a new library with a value of **Ftxl100**. This is the name of the FTXL LonTalk protocol library.
    e. In the Library Paths box, click the **Add** button to add a new library path to open the Add directory path dialog.
        i. In the Library Path dialog, click **Workspace** to open the Folder Selection dialog.
        ii. In the Folder Selection dialog, select your FTXL application project and click **OK**.
        iii. In the Add directory path dialog, the Directory field should display a relative path for the project similar to `${workspace_loc:/FTXL/MyApplication}`. Alternatively, you can specify the project's parent directory as "`..`". Click **OK**.

f. In the Properties window, click **Apply** to save these settings, then click **OK** to close the dialog.

Other options and properties can be set to their default values.

# Building the Application Image

After you first create your application project in the Nios II IDE, you should perform a clean build to remove obsolete files and to fix any problems from a previously built state.

To clean the FTXL software image:

1. Select **Project** → **Clean** to open the Clean window.
2. In the Clean window, ensure that the **Clean all projects** radio button is selected and that the **Start a build immediately** checkbox is selected and click **OK**.

For subsequent builds, you can perform automatic or manual builds.

To build the FTXL software image automatically:

1. Select **Window** → **Preferences** to open the Preferences window.
2. In the Preferences window, expand **General** and select **Workspace** to display the Workspace page.
3. In the Workspace page, select **Build automatically**. For automatic builds, you should also select **Save automatically before build** to ensure that your most recent changes are included in the build.
4. In the Preferences window, click **Apply** to save these settings, then click **OK** to close the dialog.

To build the FTXL software image manually, select **Project** → **Build Project** or **Project** → **Build All**. You can also right-click the project folder from the Nios II C/C++ Projects pane and select **Build Project**.

After you build the project, you can run it in RAM, as described in *Running the Application* on page 108, or you can load the software image into persistent memory, as described in *Loading the Application Image into Persistent Memory*.

# Loading the Application Image into Persistent Memory

To load the software image into persistent memory (such as flash memory):

1. Ensure that the FPGA device board is powered on and that a device programmer (such as a USB-Blaster download cable) is connected to the board's JTAG header connector.
2. Start the Nios II IDE.
3. Select the FTXL application project from the Nios II C/C++ Projects pane.
4. Ensure that the FPGA device contains the hardware image in RAM:
    a. Select **Tools** → **Quartus II Programmer** to open the open the Chain Description File view for the project.

b. Load the hardware image for the Nios II processor into the FPGA device, as described in the *FTXL Hardware Guide*. Leave Quartus II Programmer window open.

5. Select **Tools → Flash Programmer** to open the Flash Programmer window.
6. In the Flash Programmer window, right-click **Flash Programmer** in the left-hand pane and select **New** to create a configuration for the selected project.
7. Select **Program software project into flash memory**. Click **Browse** to open the Project Selection dialog, select the FTXL application project, and click **OK**.
8. Select **Program FPGA configuration data into hardware-image region of flash memory** to load the hardware design into flash memory along with the software design. You can skip this step if you have already loaded the hardware design into flash memory.
    a. Click **Browse** to open the Choose an FPGA Configuration file dialog, select the SRAM object file (*.sof) for the hardware design, and click **Open**.
    b. Select the appropriate hardware image, memory (serial configuration device controller), and memory offset from the **Hardware Image**, **Memory**, and **Offset** dropdown list boxes.
9. Select **Validate Nios II system ID before software download**.
10. Click **Program Flash** to load the software image into the Nios II processor.
11. If the Program Flash Now? dialog appears, click **Yes**.
12. After the software is loaded, perform a reset for the FPGA device.
13. Close the Quartus II Programmer window. You can also close the Nios II IDE window.

The Nios II processor runs the loaded software as soon as the processor completes restart processing.

# Running the Application

If you loaded the application image into persistent memory (such as flash memory), the application runs automatically as soon as the Nios II processor is properly programmed and reset.

You can also run the application from the Nios II IDE:

1. Ensure that the FPGA device board is powered on and that a device programmer (such as a USB-Blaster download cable) is connected to the board's JTAG header connector.
2. Start the Nios II IDE.
3. Right-click the FTXL application project from the Nios II C/C++ Projects pane and select **Run As → Nios II Hardware**. The Nios II IDE recompiles the project.

4. If you have a valid Nios II development license, and have already loaded the configuration data (the JTAG Indirect Configuration (*.jic) file or SRAM object file (*.sof)) into the FPGA device, proceed to step 6.
5. If you do not have a valid Nios II development license, or have not loaded the configuration data into the FPGA device:
    a. The Nios II IDE displays the following text in the Console window:

       There are no Nios II CPUs with debug modules available which match the values specified. Please check that your PLD is correctly configured, downloading a new SOF file if necessary.
    b. The Quartus II Programmer window opens.
    c. In the Quartus II Programmer window, click **Add File** to open the Select Programming File dialog.
    d. In the Select Programming File dialog, select the appropriate SRAM object file (*.sof) file for the project, and click **Open**.
    e. Ensure that a device programmer (such as a USB-Blaster download cable) is defined in the Chain Description File for the project.

       If you have not defined the USB-Blaster download cable in the Chain Description File for the project, click **Hardware Setup**. See *Using a Device Programmer for the FPGA Device* on page 103 for more information about setting up the device programmer.
    f. Select the **Program/Configure** checkbox for the *.sof file.
    g. Click **Start** to load the selected *.sof file into the Nios II processor.
    h. Do not close the Quartus II Programmer window. You must leave this window open while you are running the application.
    i. Return to the Nios II IDE, right-click the FTXL application project from the Nios II C/C++ Projects pane, and select **Run As → Nios II Hardware**. The Nios II IDE recompiles the project.
6. Your application should run. It should include some visual indication within the Nios II IDE that it is running, such as by displaying text in the Console window or blinking an LED on the device board.

To verify that the application runs as expected, connect the FTXL device to a LONWORKS network, and commission it using a network management tool, such as the LonMaker Integration tool.

**Recommendation**: Use the generated *.xif file when you commission your FTXL device, rather than performing ad-hoc network installation. Ad-hoc network installation for a device can require much more time to commission the device than when you use a *.xif file.

# Debugging the Application

You debug an FTXL application in the same way as any other C application for the Nios II processor. There are four main areas to consider while debugging an FTXL application:

- The application itself, including implementations of the event handler functions and callback handler functions of the FTXL LonTalk API.
- The FTXL non-volatile data driver. You can modify the driver from the example applications.
- The FTXL operating system abstraction layer. You can modify the FTXL OSAL from the example applications to provide support for an operating system other than the Micrium µC/OS-II operating system or to change the settings for the operating system.
- The FTXL hardware abstraction layer. If your FTXL device's hardware differs from the design recommendations in the *FTXL Hardware Guide*, you might need to modify the FTXL HAL. To debug the FTXL HAL, you should run the FTXL Bring-Up application, which is described in the *FTXL Hardware Guide*.

To debug the application from the Nios II IDE:

1. Ensure that the FPGA device board is powered on and that a device programmer (such as a USB-Blaster download cable) is connected to the board's JTAG header connector.
2. Start the Nios II IDE.
3. Right-click the FTXL application project from the Nios II C/C++ Projects pane and select **Debug As → Nios II Hardware**. The Nios II IDE recompiles the project, and opens the Debug perspective.
4. If you have a valid Nios II development license, and have already loaded the configuration data (the JTAG Indirect Configuration (*.jic) file or SRAM object file (*.sof)) into the FPGA device, proceed to step 6.
5. If you do not have a valid Nios II development license, or have not loaded the configuration data into the FPGA device, follow the procedure listed under step 5 on page 109, described in *Running the Application* on page 108.
6. The Nios II IDE halts the application at the first executable statement. You can step into the code, step over functions, or run the application to a breakpoint.

To verify that the application runs as expected, connect the FTXL device to a LONWORKS network, and commission it using a network management tool, such as the LonMaker Integration tool.

# 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 Interface Builder, 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 Interface Builder 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 Interface Builder tool for FTXL, 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

**Table 10** 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.

**Table 10**. Command Switches for the libf Command

| Command Switch | | Description |
|---|---|---|
| **Long Form** | **Short Form** | |
| --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 FTXL Transceiver 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 |
| --silent | | Suppress banner message display |

| Command Switch | | Description |
|---|---|---|
| Long Form | Short Form | |
| --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 **Table 11** on page 116, 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.

Note that 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
```

**Table 11**. Buffer-Type Specifications for the --buffer Command Switch

| Buffer Type | Primary Specification | Alternate Specifications | Valid Values |
|---|---|---|---|
| Application input buffers | ai | appinput<br>appin<br>application-input | 1 to 100<br>Default: 5 |
| Application output buffers | ao | appoutput<br>appout<br>application-output | 1 to 100<br>Default: 3 |
| Application output priority buffers | aop | appoutputprio<br>appoutprio<br>appprio<br>application-priority-output | 1 to 100<br>Default: 2 |
| Link-layer buffers | ll | linklayer<br>link-layer<br>link | 1 to 100<br>Default: 2 |
| Network input buffers | nis | netinsize<br>network-input-size | 1 to 100<br>Default: 11 |
| Network output buffers | nos | netoutsize<br>network-output-size | 1 to 100<br>Default: 3 |
| Network input buffer size | ni | netinput<br>netin<br>network-input | 1 to 100<br>Default: 11 |
| Network output buffer size | no | netoutput<br>netout<br>network-output | 1 to 100<br>Default: 3 |

| Buffer Type | Primary Specification | Alternate Specifications | Valid Values |
|---|---|---|---|
| Network output priority buffers | nop | netoutputprio<br>netoutprio<br>netprio<br>network-priority-output | 1 to 100<br>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).

# 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 3, *Creating a Model File*, on page 23.

# 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.

Note that 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**.

# 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* on page 129.

# *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 FTXL Transceiver.

# 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 FTXL 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()** FTXL 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.

**Recommendation**: 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 an FTXL 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* on page 44 for more information about authentication.

The default is **nonauth (config)**.

**Recommendation**: Use 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*<br>or<br>*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* on page 68 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 FTXL 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* on page 143.

# The Configuration Property Type

The type for a CP family cannot be 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* on page 68 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* on page 38 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 *LonMaker 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* on page 132, 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* on page 140.

**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.

# D

# FTXL LonTalk API

This Appendix describes the API functions, event handler functions, and callback handler functions that are included with the FTXL LonTalk API. It also describes the FTXL operating system abstraction layer (OSAL) and hardware abstraction layer (HAL) functions.

# Introduction

The FTXL LonTalk API provides the functions that you call from your FTXL 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 FTXL application must provide to handle LONWORKS events from the network and FTXL LonTalk protocol stack. Because each FTXL 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 FTXL LonTalk API includes the FTXL operating system abstraction layer (OSAL) API. To provide hardware independence, the FTXL LonTalk API provides the FTXL hardware abstraction layer (HAL) API. To provide non-volatile data independence, the FTXL LonTalk API provides two complete (and one skeletal) non-volatile data driver (NVD) APIs.

Typically, you use the FTXL LonTalk API functions with the event handler functions for FTXL device initialization and for sending and receiving network variable updates. See Chapter 5, *Developing an FTXL Application*, on page 73, for more information about using these functions.

The implementations of the FTXL LonTalk API are contained in the following files:

- **libFtxl100.a**, the FTXL library – implements the FTXL LonTalk protocol stack, FTXL LonTalk API, and parallel interface driver
- **FtxlHandlers.c**, stub functions for the FTXL event handler functions and callback handler functions
- **FtxlOsal.c**, the Micrium µC/OS-II implementation of the FTXL OSAL
- **FtxlHal.c**, the FTXL Developer's Kit hardware implementation of the FTXL HAL
- **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 *FTXL API Files* on page 20 for a list of the files that are included with the FTXL Developer's Kit.

# The FTXL LonTalk API, Event Handler Functions, and Callback Handler Functions

This section provides an overview of the FTXL 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 FTXL Developer's Kit → Documentation → API Reference**

- API source code for the example applications: **Start → Programs → Echelon FTXL Developer's Kit → Source Code**

# *FTXL LonTalk API Functions*

The FTXL LonTalk API includes functions for managing network data, the FTXL device, and non-volatile data.

## Commonly Used FTXL LonTalk API Functions

**Table 12** lists API functions that you will most likely use in your FTXL application.

**Table 12**. Commonly Used FTXL LonTalk API Functions

| Function | Description |
|---|---|
| **LonEventPump()** | Processes any messages received by the FTXL LonTalk protocol stack. If messages are received, it calls the appropriate event handler functions.<br>See *Periodically Calling the Event Pump* on page 81 for more information about this function. |
| **LonExit()** | Stops the FTXL LonTalk protocol stack for an orderly shutdown of the FTXL device. |
| **LonInit()** | Initializes the FTXL LonTalk API and the FTXL LonTalk protocol stack. This function downloads FTXL device interface data from the FTXL application to the FTXL Transceiver.<br>The FTXL application must call **LonInit()** once on startup. |
| **LonPropagateNv()** | Propagates a network variable value to the network.<br>This function propagates a network variable if *all* of the following conditions are met:<br>• The network variable is declared with the **output** modifier<br>• The network variable must be bound to the network<br>• The network variable must not be declared with the **polled** modifier. |

## Other FTXL LonTalk API Functions

**Table 13** on page 150 lists other FTXL LonTalk API functions that you can use in your FTXL application. These functions are not typically used by most FTXL applications, or are used only for specific application functionality (for example, including support for changeable-type network variables).

**Table 13**. Other FTXL LonTalk API Functions

| 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. |
| **LonGetUniqueId()** | Gets the unique ID (Neuron ID) value of the FTXL Transceiver. |
| **LonGetVersion()** | Gets the version number of the FTXL LonTalk API. |
| **LonPollNv()** | Requests a network variable value from the network. An FTXL 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 FTXL LonTalk protocol stack automatically broadcasts service-pin messages when needed. |

# Application Messaging API Functions

**Table 14** lists the FTXL 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 proprietary interface that does not need to interface to devices from other manufacturers. Support for application messaging is optional.

**Table 14**. Application Messaging FTXL LonTalk API Functions

| Function | Description |
|---|---|
| **LonReleaseCorrelator()** | Releases a request correlator for an application message without sending a response. |
| **LonSendMsg()** | Sends an application message. |

| Function | Description |
|---|---|
| LonSendResponse() | Sends an application message response to a request message.<br>The FTXL application calls LonSendResponse() in response to a LonMsgArrived() event handler function. |

# Non-Volatile Data API Functions

Table 15 lists the FTXL LonTalk API functions that are used for implementing support for non-volatile data.

Table 15. Non-Volatile Data FTXL LonTalk API Functions

| Function | Description |
|---|---|
| LonNvdAppSegmentHasBeenUpdated() | Indicates that the application data segment in non-volatile data memory has been updated. |
| LonNvdFlushData() | Requests that the FTXL LonTalk protocol stack flush all non-volatile data to persistent memory. |
| LonNvdGetMaxSize() | Gets the number of bytes required to store persistent data. |

# Extended API Functions

The FTXL LonTalk API includes an extended API that provides additional local network management commands listed in Table 16.

Table 16. Extended API Functions

| Function | Description |
|---|---|
| LonClearStatus() | Clears the status statistics on the FTXL device. |
| LonGoConfigured() | Sets the state for the FTXL device as configured. |
| LonGoOffline() | Sets the state for the FTXL device as offline. |
| lonGoOnline() | Sets the state for the FTXL device as online. |
| LonGoUnconfigured() | Sets the state for the FTXL 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 FTXL device's address table. |
| LonQueryAliasConfig() | Queries configuration data for the FTXL device's alias table. |

| Function | Description |
|---|---|
| LonQueryConfigData() | Queries local configuration data on the FTXL device. |
| LonQueryDomainConfig() | Retrieves a copy of the local domain table record from the FTXL device. |
| LonQueryNvConfig() | Queries configuration data for theFTXL device's network variable table. |
| LonQueryStatus() | Requests local status and statistics. |
| LonQueryTransceiverStatus() | Requests the local status of the FTXL Transceiver. |
| LonSetNodeMode() | Sets the operating mode for the FTXL device:<br>• Online: An online device executes its application and responds to all network messages.<br>• Offline: An offline device does not execute its application or respond to network messages. It will respond to network management messages.<br>• Configured: The device is ready for network operation.<br>• Unconfigured: The device is not ready for network operation. |
| LonUpdateAddressConfig() | Sets configuration data for the FTXL device's address table. |
| LonUpdateAliasConfig() | Sets configuration data for the FTXL device's alias table. |
| LonUpdateConfigData() | Sets configuration data on the FTXL device. |
| LonUpdateDomainConfig() | Sets a domain table record on the FTXL device. |
| LonUpdateNvConfig() | Sets configuration data for the FTXL device's network variable table. |

# *FTXL Event Handler Functions*

The FTXL LonTalk API provides event handler functions for managing network and device events.

## Commonly Used Event Handler Functions

**Table 17** on page 153 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.

**Table 17**. Commonly Used FTXL Event Handler Functions

| Function | Description |
|---|---|
| **LonNvUpdateCompleted()** | Indicates that either an update network variable or a poll network variable call is completed. |
| **LonNvUpdateOccurred()** | Indicates that a network variable update request from the network has been processed by the FTXL 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.<br>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 FTXL Transceiver is already offline and the FTXL application need only take application-specific action. |
| **LonOnline()** | A request from the network that the device go online.<br>Installation tools use this message to enable application processing in a device. When this function is called, the FTXL Transceiver is already online and the FTXL 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.<br>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

**Table 18** lists the event handler functions that are called by the FTXL LonTalk API to process dynamic network variables. See *Handling Dynamic Network Variables* on page 92 for more information about using these functions.

**Table 18**. Dynamic Network Variable Event Handler 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

**Table 19** lists the event handler functions that are called by the FTXL LonTalk API for application messaging transactions. Customize these functions if you use application messaging in your FTXL device. Application messaging is optional and only recommended for implementing the LONWORKS file transfer protocol and for proprietary interfaces.

If you choose not to support application messaging, you do not need to customize these functions.

**Table 19**. Application Messaging Event Handler 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.<br>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.<br>If a request message has been sent, this event handler is called only after all responses have been reported by the **LonResponseArrived()** event handler. |

| Function | Description |
|---|---|
| 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 FTXL LonTalk API provides the event handler function listed in **Table 20** to support non-volatile data.

**Table 20**. FTXL Non-Volatile Data Event Handler Function

| Function | Description |
|---|---|
| LonNvdStarvation() | Indicates that a write request to non-volatile data has taken more than 60 seconds.<br>The application should call the **LonNvdFlushData()** API function to ensure that non-volatile data is written. |

# *FTXL Callback Handler Functions*

In addition to providing event handler functions, the FTXL LonTalk API also provides callback handler functions, mainly for managing memory on the FTXL device.

## Commonly Used Callback Handler Functions

In addition to processing events, the FTXL LonTalk API provides the callback handler functions listed in **Table 21**.

**Table 21**. FTXL Callback Handler Functions

| Function | Description |
|---|---|
| LonGetCurrentNvSize() | Indicates a request for the network variable size.<br>The FTXL LonTalk protocol stack calls this callback handler function to determine the current size of a changeable-type network variable.<br>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. |

| Function | Description |
|---|---|
| LonEventReady() | Indicates that a network event is ready to be processed.<br>The FTXL LonTalk protocol 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. |

# Direct Memory Files Callback Handler Functions

The FTXL LonTalk API provides the callback handler functions listed in Table 22 to support the direct memory files (DMF) feature. These functions rely on utility functions generated by the LonTalk Interface Developer utility.

Table 22. FTXL DMF Callback Handler Functions

| Function | Description |
|---|---|
| LonMemoryRead() | Indicates a request to read memory in the FTXL device's memory space. |
| LonMemoryWrite() | Indicates a request to write memory in the FTXL device's memory space. |

# Non-Volatile Data Callback Handler Functions

Table 23 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 (also listed in Table 23):

- LonNvdDeserializeSegment()
- LonNvdGetApplicationSegmentSize()
- LonNvdSerializeSegment()

The remaining non-volatile data callback handler functions are implemented in the FtxlFlashDirect.c and FtxlFlashFs.c files.

Table 23. FTXL Non-Volatile Data Callback Handler Functions

| Function | Description |
|---|---|
| LonNvdClose() | Indicates a request to close a non-volatile data segment. |

| Function | Description |
|---|---|
| LonNvdDelete() | Indicates a request to delete a non-volatile data segment. |
| LonNvdDeserializeSegment() | Indicates a request to update the FTXL 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 FTXL Operating System Abstraction Layer

The FTXL Developer's Kit includes an operating system abstraction layer (OSAL), which allows the FTXL LonTalk protocol stack and FTXL applications to be ported to any operating system that is supported for the Nios II processor.

The example applications that are included with the FTXL Developer's Kit implement the FTXL OSAL for the Micrium µC/OS-II operating system. The FTXL OSAL is provided as source code so that you can modify this implementation to support other operating systems.

For detailed information about the FTXL OSAL, see the HTML API documentation and the API source code:

- HTML API documentation: **Start → Programs → Echelon FTXL Developer's Kit → Documentation → API Reference**
- API source code for the example applications: **Start → Programs → Echelon FTXL Developer's Kit → Source Code**

The following sections provide an overview of the functions that the FTXL OSAL provides.

## Managing Critical Sections

To manage critical sections, the FTXL OSAL provides the functions listed in **Table 24**.

**Table 24**. FTXL OSAL Critical Section Functions

| 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 FTXL OSAL provides the functions listed in **Table 25**.

**Table 25**. FTXL OSAL Binary Semaphore Functions

| 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 FTXL OSAL provides the functions listed in **Table 26**.

**Table 26**. FTXL OSAL Event Functions

| Function | Description |
|---|---|
| **OsalCreateEvent()** | Creates an event. |

| Function | Description |
|---|---|
| OsalDeleteEvent() | Deletes an event. |
| OsalSetEvent() | Sets an event. |
| OsalWaitForEvent() | Waits for an event. |

## Managing System Timing

To manage system timing, the FTXL OSAL provides the functions listed in **Table 27**.

**Table 27**. FTXL OSAL Timing Functions

| 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 FTXL OSAL provides the functions listed in **Table 28**.

An application should not use the FTXL OSAL functions for creating a task; if the application needs to create tasks, it should call operating system functions directly. The OSAL functions for creating a task are designed for creating FTXL LonTalk protocol stack tasks only.

**Table 28**. FTXL OSAL Task Functions

| 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 FTXL OSAL provides the functions listed in **Table 29** on page 160.

**Table 29**. FTXL OSAL Debug Functions

| 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. |

## *Configuring the Operating System*

The FTXL OSAL defines resources for an operating system. Most of the resources for the FTXL OSAL have fixed definitions or allocations. However, you can specify the relative priorities of system contexts (tasks or threads). In addition, within your operating system, you can modify the allocations for the following resource types:

- Crtitical sections
- Binary semaphores
- System events

For some operating systems, such as the Micrium µC/OS-II operating system, the allocations for critical sections and binary semaphores are combined into a single definition.

This section describes how to determine how many of each resource your FTXL application requires. The following section, *Configuring the Micrium µC/OS-II Operating System,* on page 165, describes how to allocate these resources for the Micrium µC/OS-II operating system that is used by the example applications.

# Determining Resource Requirements

**Table 30** lists the basic resource requirements for an operating system running with the FTXL OSAL.

**Table 30**. Operating System Resource Requirements

| Resource | Number | Notes |
|---|---|---|
| Tasks (or threads) | Up to 10 | The number of tasks does not include application tasks. An FTXL application must have at least one application task. |
| Critical sections | 82 (default number) | The number of critical sections depends on the application buffer configuration. |
| Binary semaphores | 10 | Modify the number of binary semaphores, if needed, within the operating-system settings. |

| Resource | Number | Notes |
|---|---|---|
| Events | 10 | Modify the number of events, if needed, within the operating-system settings. |

To calculate the maximum number of critical sections that are required for your FTXL application, use the following formula:

$$CritSect = 50 + \left[(4 * (AppInputBuff)) + (2 * (AppOutputBuff))\right]$$

where:

- *CritSect* is the number of critical sections required
- *AppInputBuff* is the number of application input buffers
- *AppOutputBuff* is the number of application output buffers, including priority buffers and non-priority buffers

You specify the number of application buffers on the Buffer Configuration page of the LonTalk Interface Developer utility.

**Table 31** lists the default values for the number of application buffers and the number of critical sections defined within the FTXL OSAL.

**Table 31**. Default Number of Buffers and Critical Sections

| Buffer Type | Number | Critical Sections |
|---|---|---|
| Application input | 5 | 20 |
| Application output, non-priority | 5 | 10 |
| Application output, priority | 1 | 2 |

Thus, using the formula, the default number of critical sections is 82:

50 + 20 + 10 + 2 = 82

You can modify the number of critical sections defined for the application within the settings for the operating system, but the number should not be less than the default number calculated according to the formula.

# Specifying Task Priorities

An FTXL application program must include at least one application task (or thread), but can include additional tasks (or threads). When assigning task priorities, you must assign application task priorities so that they do not conflict with the required priorties for the tasks defined for the FTXL LonTalk protocol stack. In addition, some operating systems, such as the Micrium µC/OS-II operating system, require that each task be assigned a unique priority.

The FTXL OSAL defines three abstract priorties: high, medium, and low, as shown in **Figure 13** on page 162.

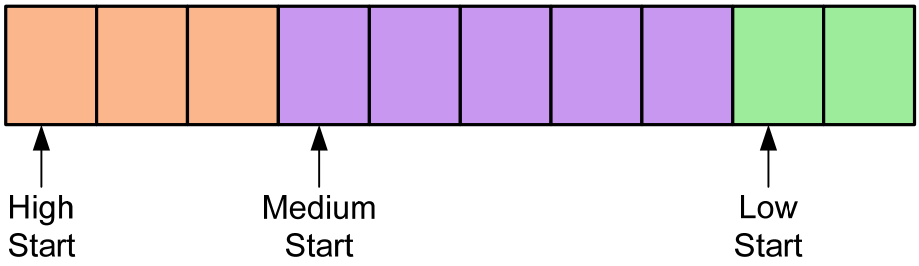



**Figure 13**. FTXL Abstract Priorities

Before you can instantiate a task within an application, you must map the FTXL OSAL abstract priorities to the operating system's priorities. The abstract priorities are defined in the **FtxlOsal.h** file that is copied to your project directory by the LonTalk Interface Developer utility.

To map the FTXL OSAL abstract priorities to the operating system's priorities, use the macros that are defined in the **FtxlOsal.h** file, as listed in **Table 32**. The FTXL OSAL reserves room between the high and medium FTXL tasks for high priority application tasks, and between the medium and low FTXL tasks for medium priority tasks. The relationships between these macros are shown in **Figure 14** on page 163.

**Table 32**. Macros for Operating System Task Priorities

| Macro | Description |
|---|---|
| OS_HIGH_PRIORITY_BASE | Defines the OS priority number of the highest priority task used by the FTXL OSAL. |
| OS_APPLICATION_HIGH_PRIORITY_BASE | Defines the highest OS priority used for high priority application tasks.<br>These tasks should have a higher priority than the FTXL OSAL medium priority tasks, but lower than the OSAL high priority tasks. |
| OS_MEDIUM_PRIORITY_BASE | Defines the highest OS priority used for medium priority FTXL tasks.<br>These tasks should have a higher priority than normal application tasks, but lower than the application high priority tasks. |
| OS_APPLICATION_PRIORITY_BASE | Defines the highest OS priority used for normal application tasks.<br>These tasks should have a higher priority than FTXL OSAL low priority tasks, but lower than the FTXL OSAL medium priority tasks. |
| OS_LOW_PRIORITY_BASE | Defines the highest OS priority used for low priority tasks. |

| | |
|---|---|
| OS_LOW_APPLICATION_PRIORITY_BASE | Defines the highest OS priority used for application tasks running at a lower priority than any FTXL LonTalk protocol stack tasks. |

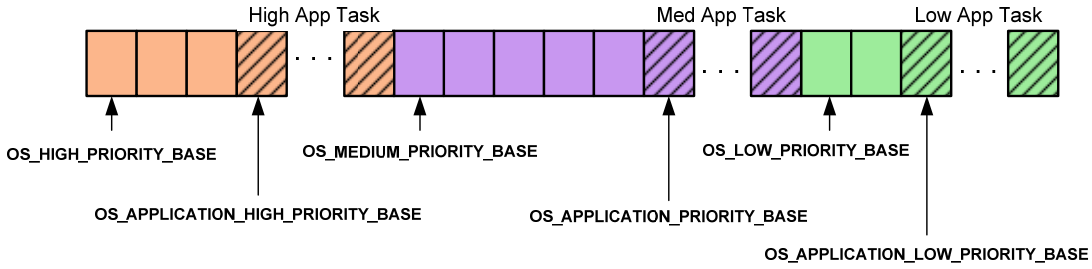



**Figure 14**. Relationships among the Macros for Operating System Task Priorities

The number of reserved priorities is controlled by the following macros:

- **NUM_RESERVED_HIGH_PRIORITY_APPLICATION_TASKS** (defines the number of high priority application tasks, and defaults to 0)
- **NUM_RESERVED_APPLICATION_PRIORITIES** (defines the number of normal application tasks, and defaults to 1)

The number of application tasks that can run at lower priority than any of the FTXL tasks is limited only by the number of priorities supported by the operating system (or for some some operating systems, such as the Micrium µC/OS-II operating system, the lowest assignable priority).

You must define at least one application task, typically with priority **OS_APPLICATION_PRIORITY_BASE**. To support more than one application task of medium priority, you must override the definition of **NUM_RESERVED_APPLICATION_PRIORITIES**. Modify this definition by performing either of the following tasks:

- Modify the definition of the **NUM_RESERVED_APPLICATION_PRIORITIES** macro in the **FtxlOsal.h** file
- Define the **NUM_RESERVED_APPLICATION_PRIORITIES** macro in the defined symbols section of your application project's preprocessor definitions, as shown in **Figure 15** on page 164.

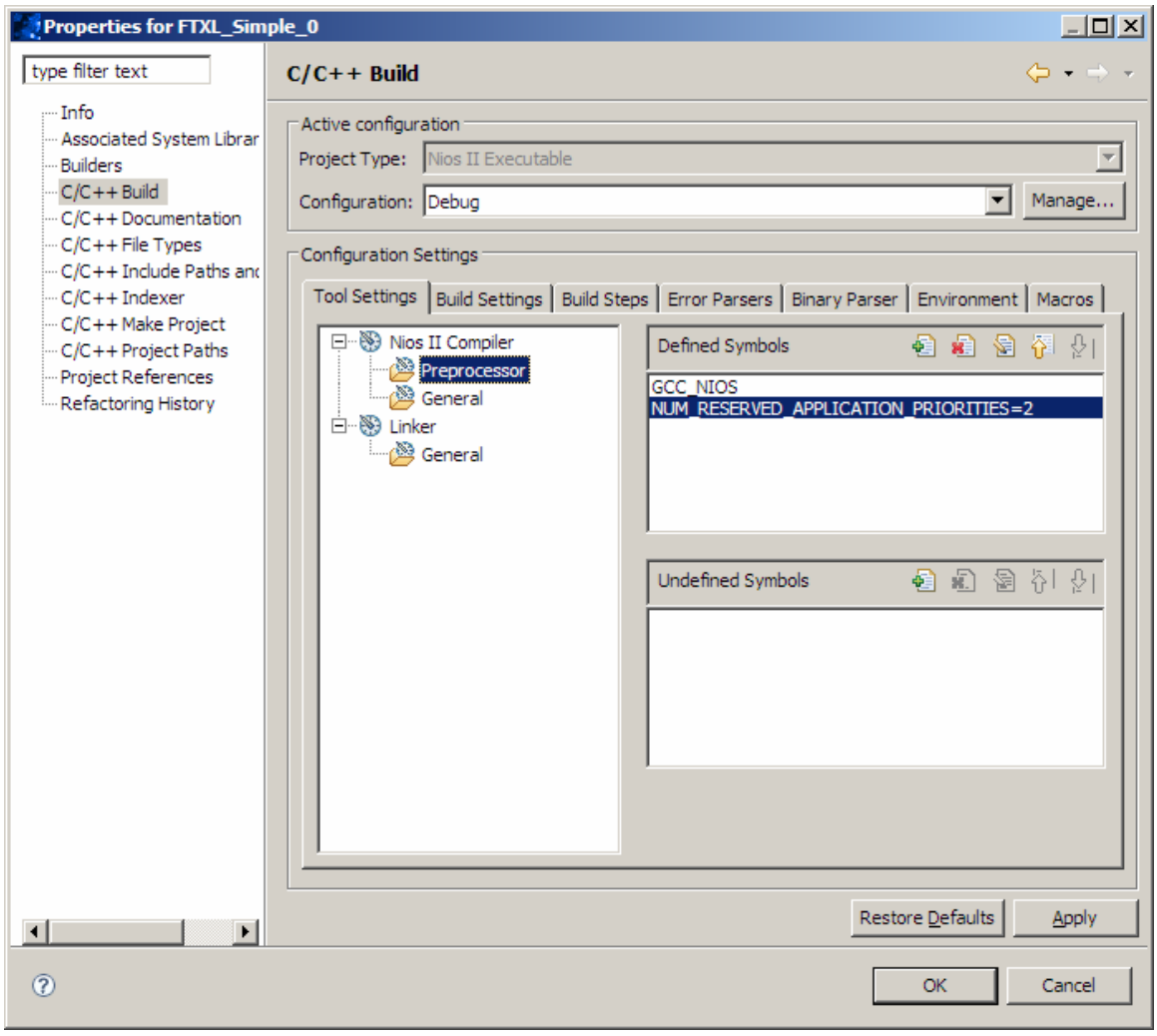



**Figure 15**. Defined Symbols in the Project Properties Dialog

Similarly, if you want to support any application tasks in the high priority class, you must override the definition of the **NUM_RESERVED_HIGH_PRIORITY_APPLICATION_TASKS** macro.

**Example**: Suppose your application requires the following task priority configuration:

- One high-priority application task
- Two medium-priority application tasks
- Three low-priority application tasks

To support this configuration:

- Set **NUM_RESERVED_HIGH_PRIORITY_APPLICATION_TASKS** to 1
- Set **NUM_RESERVED_APPLICATION_PRIORITIES** to 2
- You do not need to specify the number of low-priority application tasks because the FTXL OSAL does not reserve space for low-priority tasks

**Figure 16** on page 165 shows this example configuration. The figure also shows example start values for each type of priority task, with the value for the high-

priority tasks (**OS_HIGH_PRIORITY_BASE**) set to 4. The figure assumes that low numbers represent high priorities. The cross-hatch shaded numbers represent the desired configuration of one high-priority application task (at priority 7), two medium-priority application tasks (at priorities 13 and 14), and three low-priority application tasks (at priorities 17, 18, and 19).

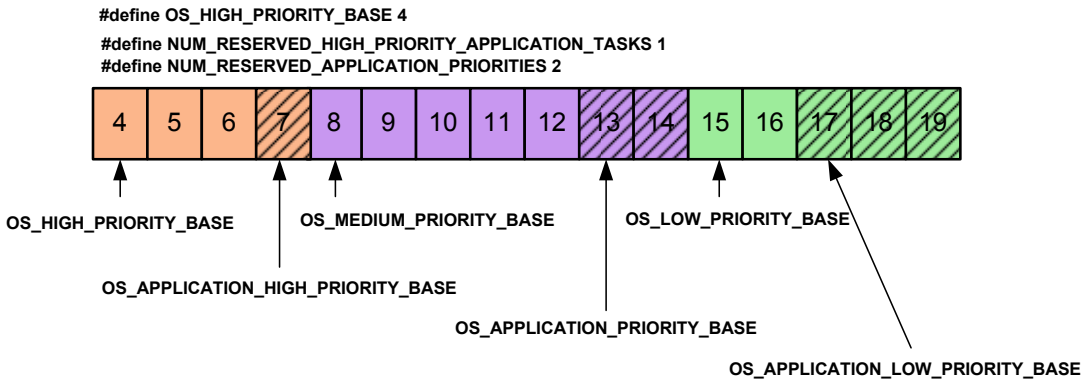



**Figure 16**. Example Priority Configuration

# *Configuring the Micrium µC/OS-II Operating System*

The example applications that are included with the FTXL Developer's Kit provide a working configuration for the Micrium µC/OS-II operating system. However, you can modify this configuration, for example, to increase the number of application tasks or increase the number of event control blocks (for critical sections and binary semaphores).

To configure the Micrium µC/OS-II operating system, you must calculate the resources that are required for both the FTXL LonTalk protocol stack and for your application program. The primary resources that you need to calculate are:

- The maximum number of tasks
- The lowest assignable task priority
- The maximum number of event control blocks

## Maximum Number of Tasks

The µC/OS-II operating system uses at least two tasks of its own (the highest and lowest priority tasks), and also uses a third task if you enable the statistics task. In addition to these operating system tasks, you must add 10 tasks for the FTXL LonTalk protocol stack tasks, and any additional application tasks that your application requires. Be sure to set the **NUM_RESERVED_HIGH_PRIORITY_APPLICATION_TASKS** and **NUM_RESERVED_APPLICATION_PRIORITIES** macros appropriately; see *Specifying Task Priorities* on page 161.

For the default configuration of a single application task and two operating system tasks, you need a total of 13 tasks. If you enable the µC/OS-II statistics task, you need a total of 14 tasks.

# Lowest Assignable Task Priority

The µC/OS-II operating system uses the lowest task priority for the idle task, and it uses the second lowest task priority for the statistics task (if the statistics task is enabled).

The documentation for the µC/OS-II operating system recommends that the application not use the highest four priorities (0 through 3) or the lowest four priorities (**OS_LOWEST_PRIO**-3 through **OS_LOWEST_PRIO**). By default, the definitions in the **FtxlOsal.h** file reserve priorities 0 through 3 for high-priority operating system tasks, and thus the file sets **OS_HIGH_PRIORITY_BASE** to 4.

To determine the values for the low-priority tasks, use the following formula:

$$LowTask = OSTasks + StackTasks + AppTasks$$

where:

- *LowTask* is the value for the lowest priority-task
- *OSTasks* is the number of tasks reserved for the operating system, which by default is 8 tasks
- *StackTasks* is the number of tasks required for the FTXL LonTalk protocol stack, which is 10 tasks
- *AppTasks* is the number of application tasks that your application requires, which is always at least 1 task

Thus, the minimum value (counting from priority 0) for the lowest priority task is 18 (8+10+1, minus 1 to count from priority 0).

**Figure 17** shows the default settings in the **FtxlOsal.h** file for the task-priority macros described in *Specifying Task Priorities* on page 161. The figure shows the four reserved high-priority tasks, the four reserved low-priority tasks, the 10 tasks reserved for the FTXL LonTalk protocol stack, and the one task reserved for the application.

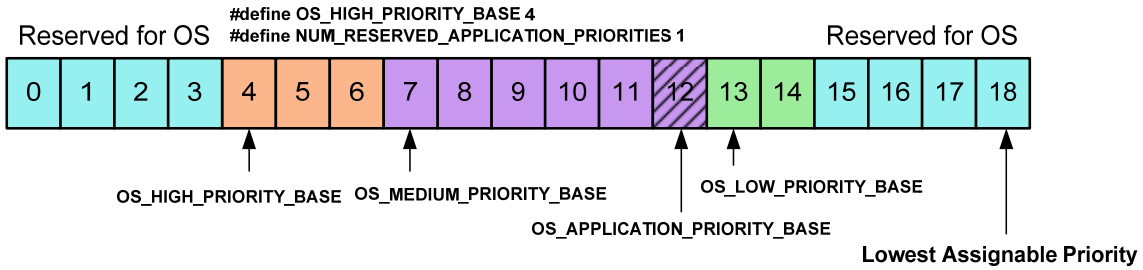



**Figure 17**. Default Task Priority Settings within the FTXL OSAL

If you have no application tasks running in the low priority class, make sure that the lowest assignable task priority is greater than or equal to the value of the **OS_LOW_APPLICATION_PRIORITY_BASE** macro. If your application runs any tasks in the low priority class, you need to set the lowest assignable task priority to the lowest priority that your application uses.

# Maximum Number of Event Control Blocks

The µC/OS-II operating system uses event control blocks for the FTXL OSAL critical sections, binary semaphores, and events. You define the maximum number of event control blocks as the sum of the number of critical sections, binary semaphores, and events. See *Determining Resource Requirements* on page 160 for information about how to calculate the number of each of these resources.

The default number of event control blocks can be calculated based on the values listed in **Table 30** on page 160:

- Default number of critical sections: 82
- Default number of binary semaphores: 10
- Default number of events: 10

Thus, the default number of event control blocks that you need to define is 102.

# Other µC/OS-II Settings

The example applications that are included with the FTXL Developer's Kit provide a working configuration for the Micrium µC/OS-II operating system. If you use the µC/OS-II operating system for your FTXL application, you can accept the default values for all of the options, except the following options:

- The maximum number of tasks (see *Maximum Number of Tasks* on page 165)
- The lowest assignable priority (see *Lowest Assignable Task Priority* on page 166)
- The maximum number of event control blocks (see *Maximum Number of Event Control Blocks*)

For your FTXL application, you might want to customize the operating-system configuration to reduce its memory footprint or provide additional functionality. The following sections describe the settings within the Nios IDE for a µC/OS-II operating system that runs an FTXL application.

To configure the operating system:

1. Right-click the system library for your project and select **Properties** to open the Properties window for the system library.
2. In the Properties window, click **RTOS Options** to open the MicroC/OS-II RTOS Options window.

## *MicroC/OS-II General Options*

**Table 33** on page 168 describes the general options for the µC/OS-II operating system.

**Table 33**. MicroC/OS-II General Options

| Option | Setting for FTXL Applications |
|---|---|
| Maximum number of tasks | 13 or higher<br>See *Maximum Number of Tasks* on page 165. |
| Lowest assignable priority | 18 or higher<br>See *Lowest Assignable Task Priority* on page 166. |
| Thread Safe C Library | Required |
| Enable code for Event Flags | Not required for FTXL applications, but required for debugging |
| Enable code for Mutex Semaphores | Not required |
| Enable code for Mailboxes | Not required |
| Enable code for Queues | Not required |
| Enable code for Memory Management | Not required |
| Enable code for Timers | Not required |

**Figure 18** on page 169 shows the MicroC/OS-II General Options page of the MicroC/OS-II RTOS Options window. The figure shows the options used by the example applications.

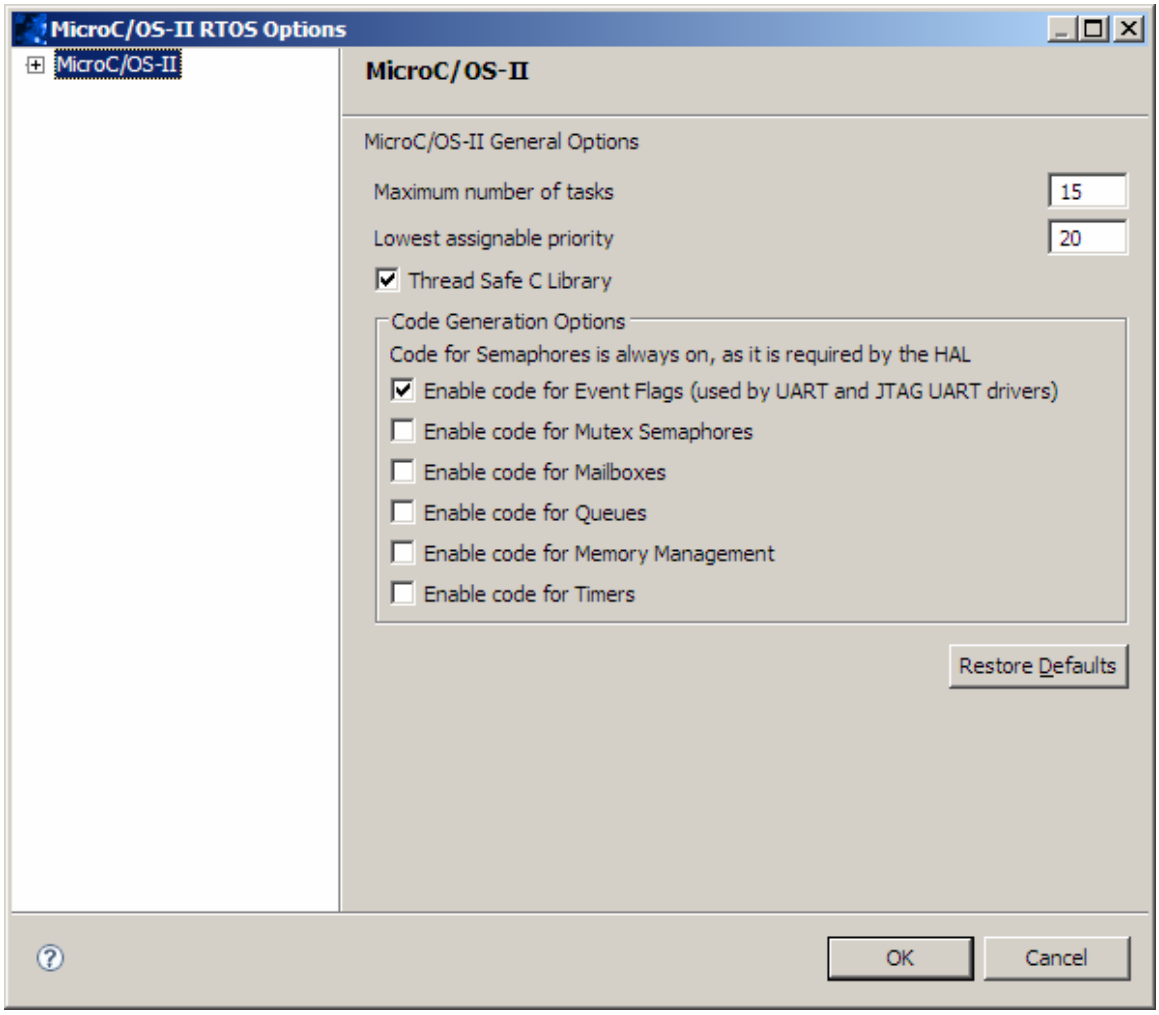



**Figure 18**. MicroC/OS-II General Options Page

## *Event Flags*

**Table 34** describes the event flag options for the µC/OS-II operating system.

**Table 34**. MicroC/OS-II Event Flag Options

| Option | Setting for FTXL Applications |
|---|---|
| Include code for Wait on Clear Event Flags | Not required |
| Include code for OSFlagAccept() | Not required |
| Include code for OSFlagDel() | Not required |
| Include code for OSFlagQuery() | Not required |
| Maximum number of Event Flag groups | 1 or more |

| Option | Setting for FTXL Applications |
|---|---|
| Size of name of Event Flag group | 0 or larger |
| Event flag bits | 8 or more |

**Figure 19** shows the MicroC/OS-II Event Flags page of the MicroC/OS-II RTOS Options window. The figure shows the settings used by the example applications.

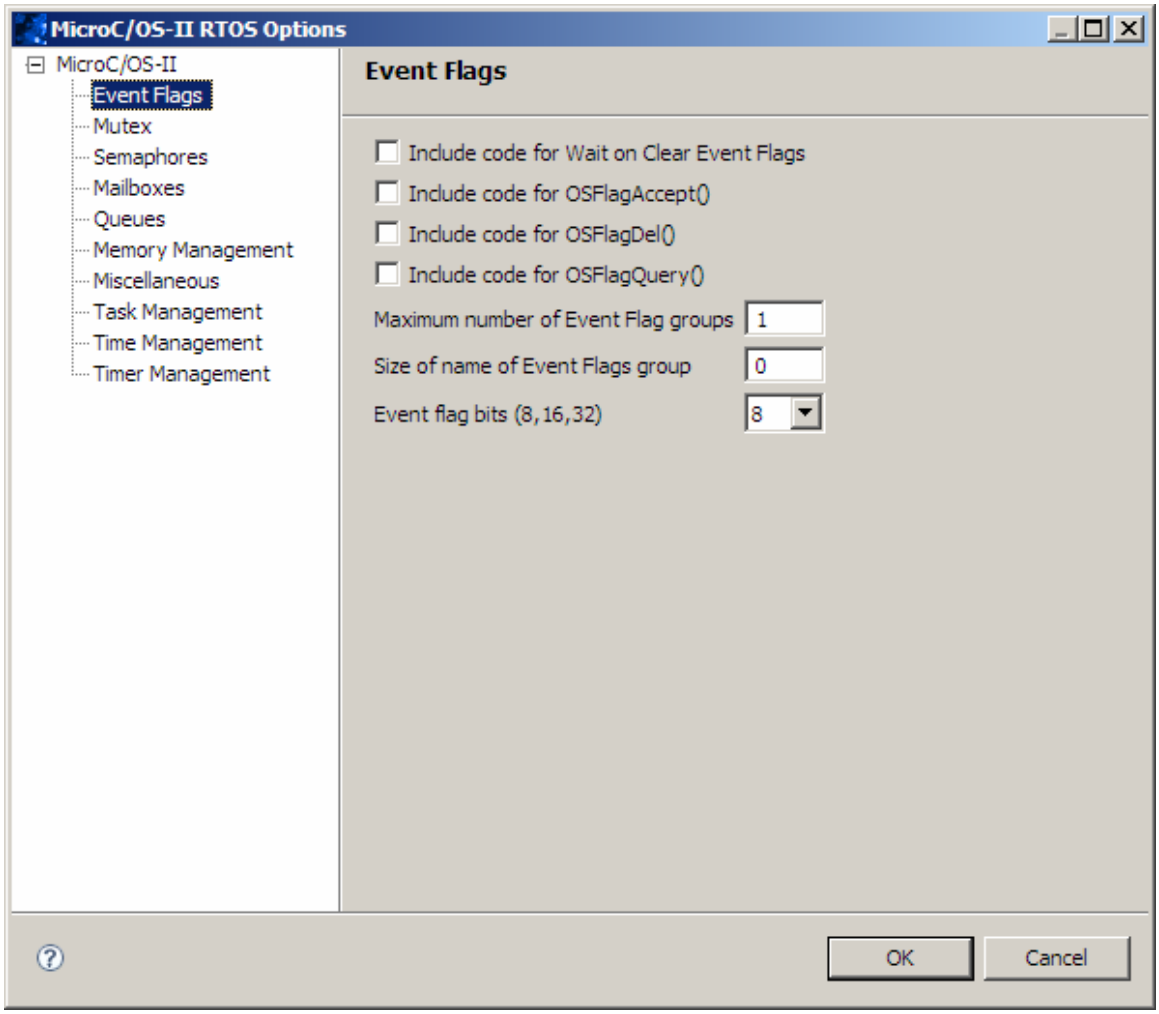



**Figure 19**. MicroC/OS-II Event Flags Page

## *Mutex*

FTXL applications do not use mutex resources. **Table 35** describes the mutex options for the µC/OS-II operating system.

**Table 35**. MicroC/OS-II Mutex Options

| Option | Setting for FTXL Applications |
|---|---|
| Include code for OSMutexAccept() | Not required |

| Option | Setting for FTXL Applications |
|---|---|
| Include code for OSMutexDel() | Not required |
| Include code for OSMutexQuery() | Not required |

## *Semaphores*

**Table 36** describes the semaphore options for the µC/OS-II operating system.

**Table 36**. MicroC/OS-II Semaphore Options

| Option | Setting for FTXL Applications |
|---|---|
| Include code for OSSemAccept() | Required |
| Include code for OSSemSet() | Required |
| Include code for OSSemDel() | Required |
| Include code for OSSemQuery() | Required |

**Figure 20** on page 172 shows the MicroC/OS-II Semaphores page of the MicroC/OS-II RTOS Options window.

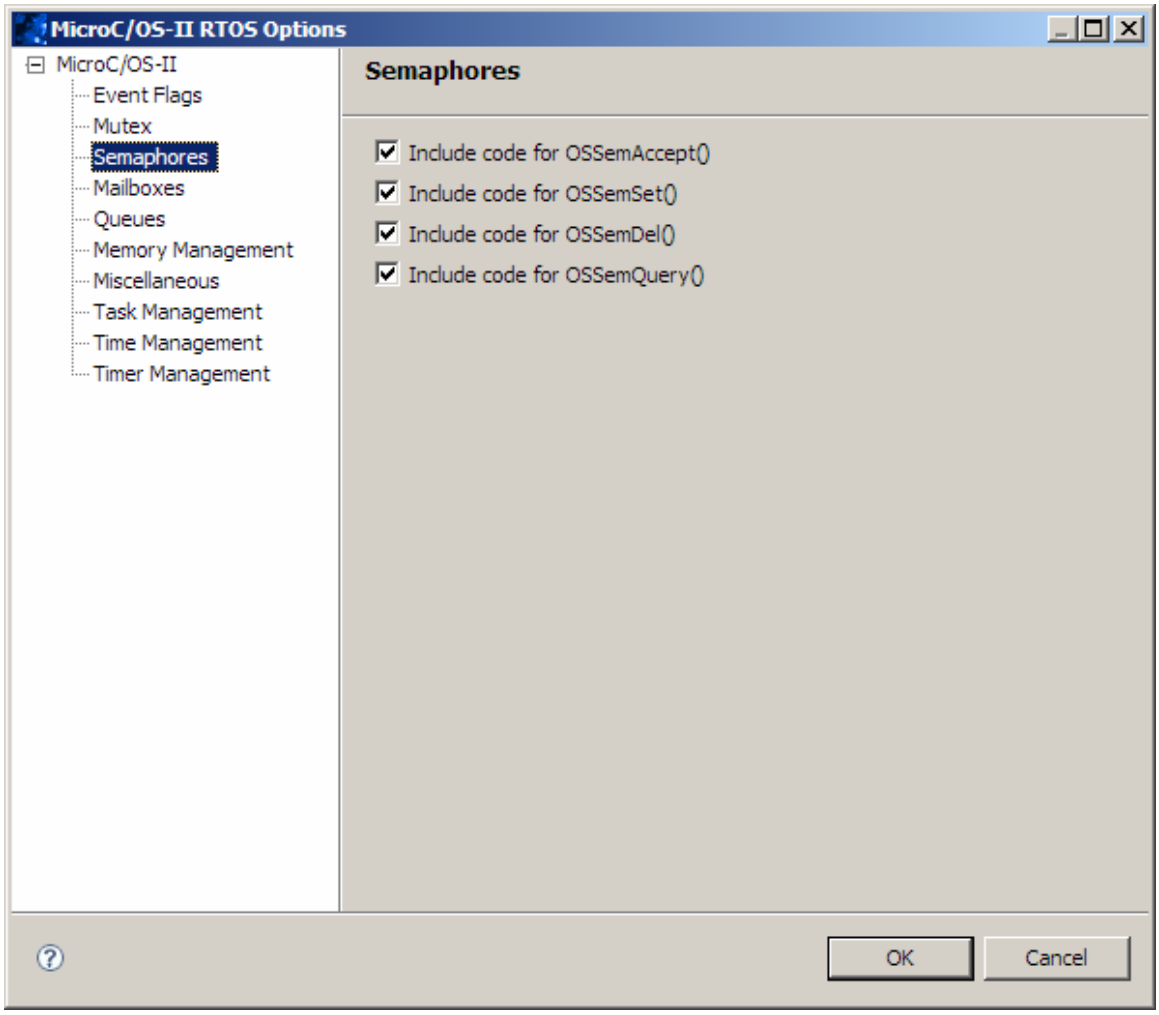



**Figure 20**. MicroC/OS-II Semaphores Page

## *Mailboxes*

FTXL applications do not use mailbox resources. **Table 37** describes the mailbox options for the µC/OS-II operating system.

**Table 37**. MicroC/OS-II Mailbox Options

| Option | Setting for FTXL Applications |
|---|---|
| Include code for OSMboxAccept() | Not required |
| Include code for OSMboxDel() | Not required |
| Include code for OSMboxPost() | Not required |
| Include code for OSMboxPostOpt() | Not required |
| Include code for OSMboxQuery() | Not required |

## Queues

FTXL applications do not use queue resources. **Table 38** describes the queue options for the µC/OS-II operating system.

**Table 38**. MicroC/OS-II Queue Options

| Option | Setting for FTXL Applications |
|---|---|
| Include code for OSQAccept() | Not required |
| Include code for OSQDel() | Not required |
| Include code for OSQFlush() | Not required |
| Include code for OSQPost() | Not required |
| Include code for OSQPostFront() | Not required |
| Include code for OSQPostOpt() | Not required |
| Include code for OSQQuery() | Not required |
| Maximum number of Queue Control blocks | Not required |

## Memory Management

FTXL applications do not use memory-management resources. **Table 39** describes the memory-management options for the µC/OS-II operating system.

**Table 39**. MicroC/OS-II Memory Management Options

| Option | Setting for FTXL Applications |
|---|---|
| Include code for OSMemQuery() | Not required |
| Maximum number of memory partitions | Not required |
| Size of memory partition name | Not required |

## Miscellaneous

**Table 40** describes the miscellaneous options for the µC/OS-II operating system.

**Table 40**. MicroC/OS-II Miscellaneous Options

| Option | Setting for FTXL Applications |
|---|---|
| Enable argument checking | Not required |
| Enable uCOS-II hooks | Required |

| Option | Setting for FTXL Applications |
|---|---|
| Enable debug variables | Not required |
| Include code for OSSchedLock() and OSSchedUnlock() | Not required |
| Enable tick stepping feature for uCOS-View | Not required |
| Enable statistics task | Not required |
| Check task stacks from statistics task | Not required |
| Statistics task stack size | Not required |
| Idle task stack size | 512 |
| Maximum number of Event Control blocks | 102 or more<br>See *Maximum Number of Event Control Blocks* on page 167. |
| Size of Semaphore, Mutex, Mailbox or Queue name | 0 (unless you use one of these resource types) |

**Figure 21** on page 175 shows the MicroC/OS-II Miscellaneous page of the MicroC/OS-II RTOS Options window. The figure shows the settings used by the example applications.

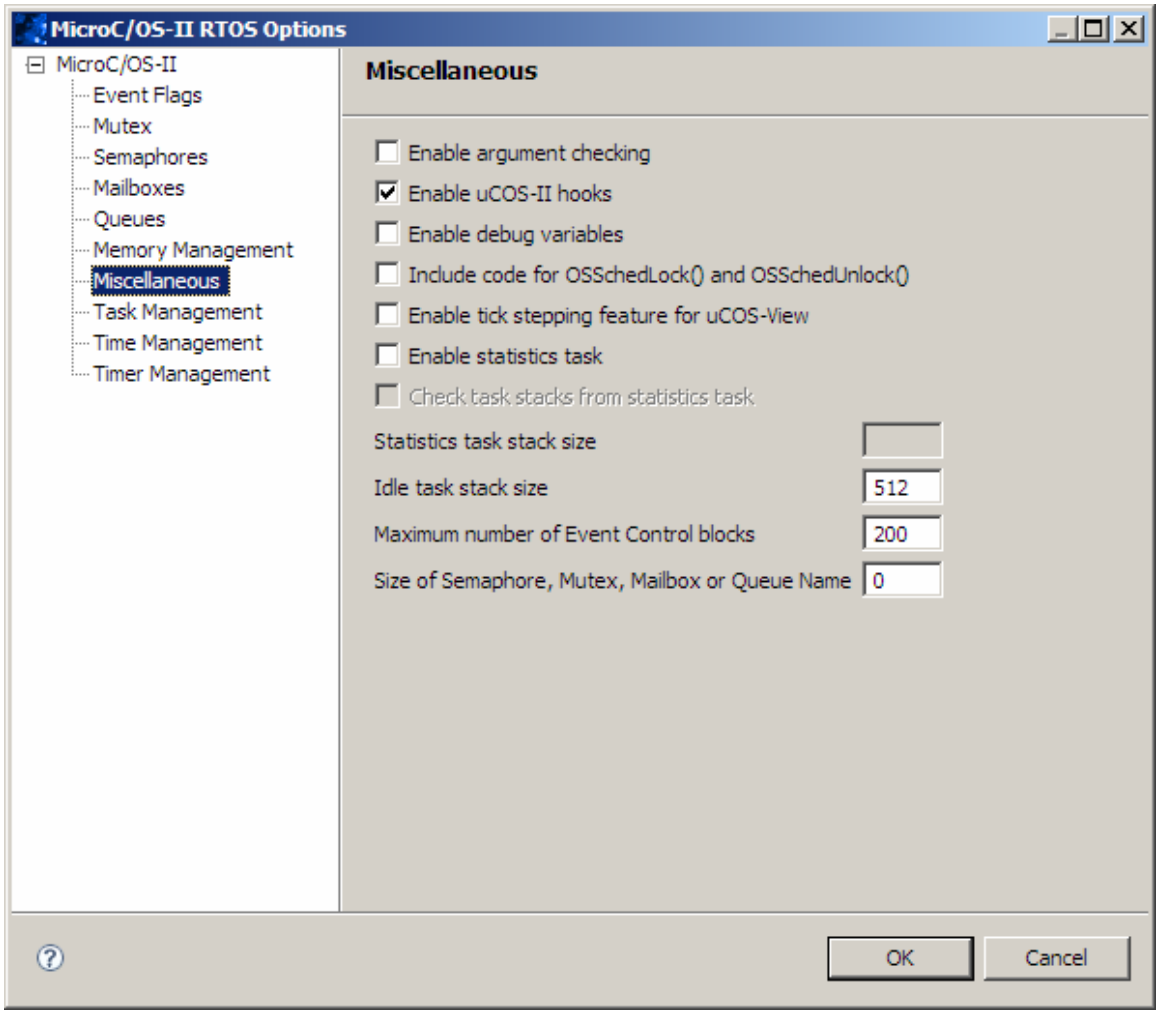



**Figure 21**. MicroC/OS-II Miscellaneous Page

## *Task Management*

**Table 41** describes the task-management options for the µC/OS-II operating system.

**Table 41**. MicroC/OS-II Task Management Options

| Option | Setting for FTXL Applications |
|---|---|
| Include code for OSTaskChangePrio() | Not required |
| Include code for OSTaskCreate() | Not required |
| Include code for OSTaskCreateExt() | Required |
| Include code for OSTaskDel() | Required |
| Include variables in OS_TCB for profiling | Not required |

| Option | Setting for FTXL Applications |
|---|---|
| Include code for OSTaskQuery() | Required |
| Include code for OSTaskSuspend() and OSTaskResume() | Not required |
| Include code for OSTaskSwHook() | Required |
| Size of task name | 0 or larger |

**Figure 22** shows the MicroC/OS-II Task Management page of the MicroC/OS-II RTOS Options window.

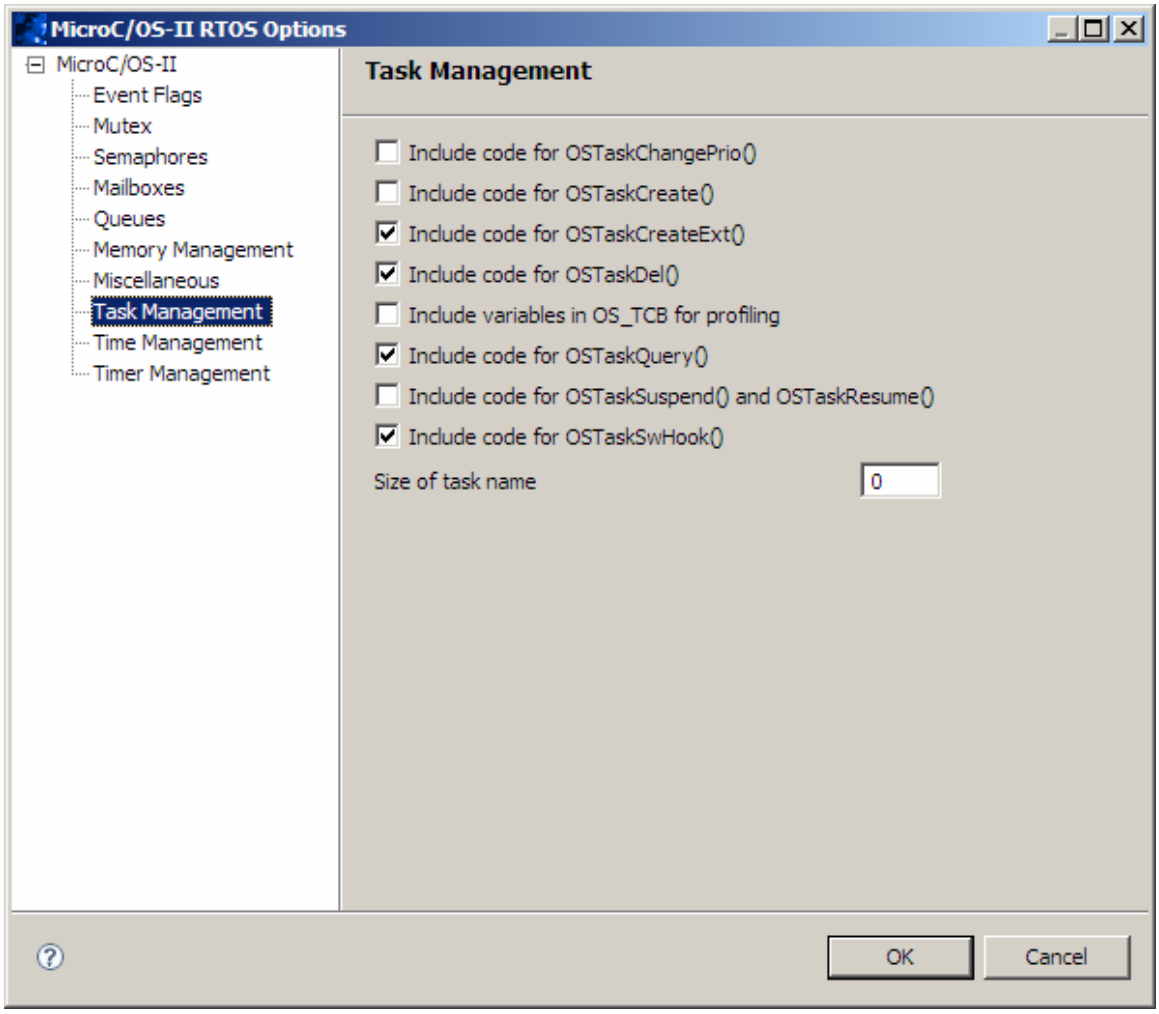



**Figure 22**. MicroC/OS-II Task Management Page

## *Time Management*

**Table 42** on page 177 describes the time-management options for the µC/OS-II operating system.

**Table 42**. MicroC/OS-II Time Management Options

| Option | Setting for FTXL Applications |
|---|---|
| Include code for OSTimeDlyHMSM() | Not required |
| Include code for OSTimeDlyResume() | Not required |
| Include code for OSTimeGet() and OSTimeSet() | Required |
| Include code for OSTimeTickHook() | Not required |

**Figure 23** shows the MicroC/OS-II Time Management page of the MicroC/OS-II RTOS Options window.

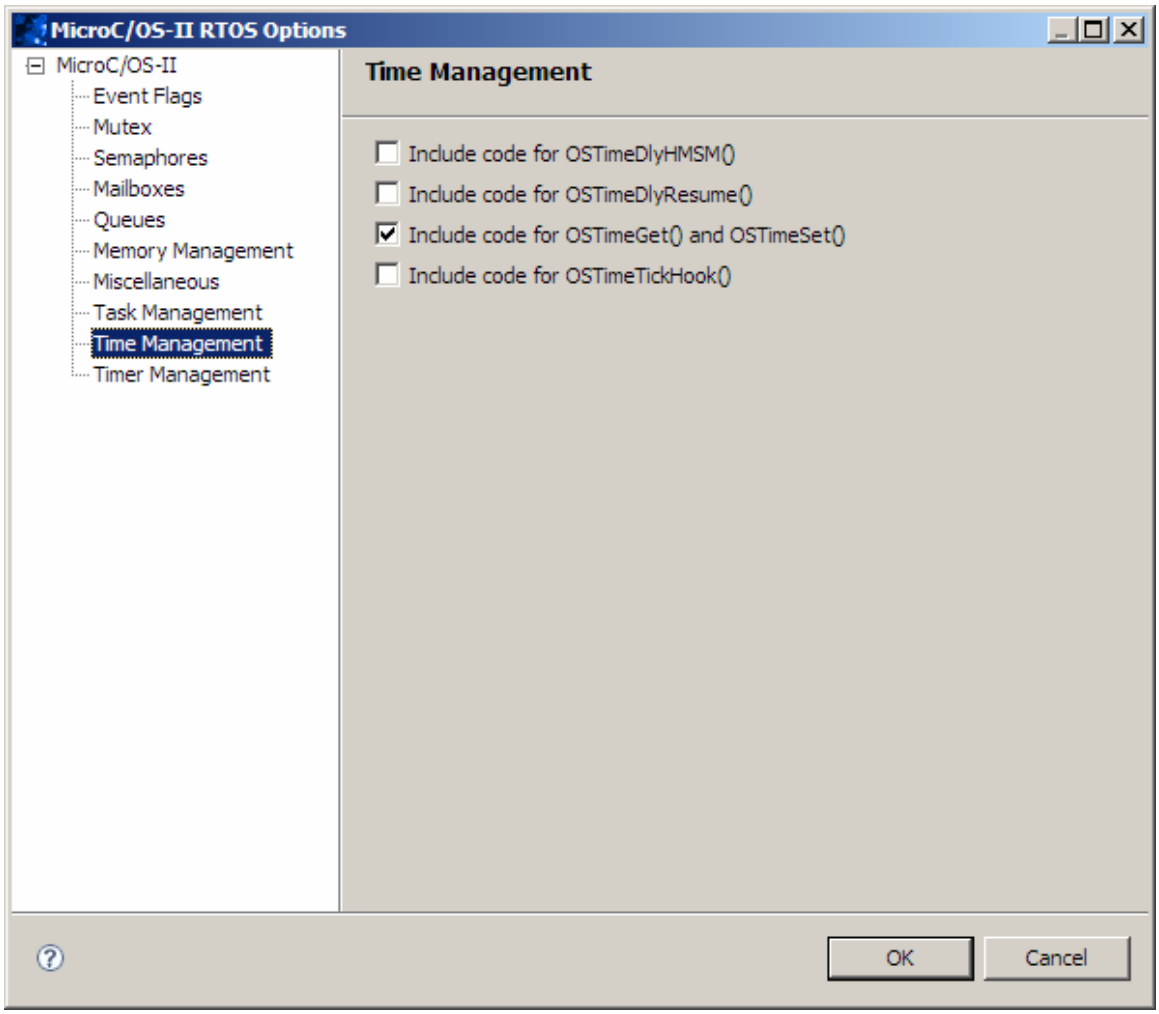



**Figure 23**. MicroC/OS-II Time Management Page

## *Timer Management*

FTXL applications do not use timer-management resources. **Table 43** on page 178 describes the timer-management options for the µC/OS-II operating system.

**Table 43**. MicroC/OS-II Timer Management Options

| Option | Setting for FTXL Applications |
|---|---|
| Maximum number of timers | Not required |
| Determine the size of a timer name | Not required |
| Size of timer wheel | Not required |
| Rate at which timer management task runs | Not required |
| Stack size for timer task | Not required |
| Priority of timer task | Not required |

# The FTXL Hardware Abstraction Layer

The FTXL Developer's Kit includes a hardware abstraction layer (HAL) that provides an abstract interface for the FTXL Transceiver. The FTXL HAL is independent of the Altera HAL, which provides an abstract interface to the Nios II processor and other hardware components for the FPGA device.

The example applications that are included with the FTXL Developer's Kit implement the FTXL HAL for the FTXL Transceiver Board and the DBC2C20 development board. The FTXL HAL is provided as source code so that you can modify this implementation to support other hardware configurations. However, if your design uses the same signal names and the same basic logic as are used in the reference design that is included with the example applications, you can likely use the functions in the **FtxlHal.c** with little or no change.

For detailed information about the FTXL HAL, see the HTML API documentation and the API source code:

- HTML API documentation: **Start → Programs → Echelon FTXL Developer's Kit → Documentation → API Reference**
- API source code: **Start → Programs → Echelon FTXL Developer's Kit → Source Code**

The following sections provide an overview of the functions that the FTXL HAL provides. See the *FTXL Hardware Guide* for a description of the FTXL Transceiver hardware interface.

## *Managing the FTXL Transceiver*

To manage the FTXL Transceiver, the FTXL HAL provides the functions listed in **Table 44** on page 179.

Table 44. FTXL HAL Transceiver Functions

| Function | Description |
|---|---|
| LonAssertTransceiverReset() | Asserts the FTXL Transceiver's RESET~ pin, which causes the Transceiver to reset. |
| LonDeassertTransceiverReset() | Deasserts the FTXL Transceiver's RESET~ pin, which causes the Transceiver to complete post-reset processing. |
| LonReadTransceiverDataRegister() | Reads the FTXL Transceiver's data register. |
| LonReadTransceiverReset() | Reads the state of the FTXL Transceiver's RESET~ pin. |
| LonTransceiverIsBusy() | Reads the FTXL Transceiver's status register to determine whether the Transceiver is busy. |
| LonWriteTransceiverDataRegister() | Writes to the FTXL Transceiver's data register. |

# Managing the Service Pin

To manage the service pin for an FTXL device, the FTXL HAL provides the functions listed in Table 45.

Table 45. FTXL HAL Service Functions

| Function | Description |
|---|---|
| LonGetServicePinStatus() | Gets the status of the FTXL Transceiver's SERVICE~ pin, which represents the status of the Service button. |
| LonSetServiceLed() | Sets the state of the FTXL device's Service LED, which can be on or off. |

# Managing Interrupts

To manage interrupts for the FTXL device, the FTXL HAL provides the functions and interrupt service routines (ISRs) listed in Table 46.

Table 46. FTXL HAL Interrupt Functions

| Function | Description |
|---|---|
| LonDisableInterrupt() | Disables the FTXL ISRs. |
| LonDriverServicePinIrq() | The main body of the ISR for processing input from the FTXL Transceiver's SERVICE~ pin. This function is implemented in the FTXL library, and is called by an interrupt routine defined in the FTXL HAL. |

| Function | Description |
|---|---|
| **LonDriverTransceiverIrq()** | The main body of the ISR for processing input from the FTXL Transceiver's IRQ~ pin. This function is implemented in the FTXL library, and is called by an interrupt routine defined in the FTXL HAL. |
| **LonEnableInterrupt()** | Enables the FTXL ISRs. |
| **LonRegisterIsr()** | Registers the FTXL ISRs with the Altera HAL. |

# E

# Determining Memory Usage for FTXL Applications

This Appendix describes how much volatile and non-volatile memory an FTXL application requires, and how to determine the application's memory requirements.

# Overview

The FTXL LonTalk protocol 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.

# *Total Memory Use*

After you create your FTXL application and compile it, you can determine how much flash memory and RAM the application requires.

When you compile a project, the Nios IDE displays the size of the executable file (.elf) in the IDE's Console view, for example:

```
Info: (FTXL_Simple_0.elf) 579 KBytes program size (code +
initialized data).
Info:                     15804 KBytes free for stack + heap.
```

You can also run the **nios2-elf-size** utility from a Windows command line to shows the size of sections (program and data) within the executable file, for example:

```
C:\>nios2-elf-size FTXL_Simple_0.elf
   text    data     bss     dec     hex filename
 540492    8704   44300  593496   90e58 FTXL_Simple_0.elf
```

The output from the utility shows the following information:

- Text size, which represents the size of the program code
- Data size, which represents the size of initialized data
- Bss size, which represents the size of uninitialized data
- Total size in decimal
- Total size in hexadecimal
- The file name

The **nios2-elf-size** utility is in the [*Altera*]**\nios2eds\bin\nios2-gnutools\H-i686-pc-cygwin\bin** directory. You run the utility against your executable file, which is in the **\Debug** (or **\Release** or other build configuration) directory for your project.

You can determine the flash requirements for an application directly from the text, data, and bss output of the **nios2-elf-size** utility (or the program size listed in the Console view after a project build). For the **FTXL_Simple_0.elf** file shown above, the flash requirement is 593 496 bytes (579 KB).

You can potentially reduce the code size by modifying some of the settings for the project's system library, which include the settings for the operating system.

To determine the RAM requirements for the application, you need to include the heap and stack usage for the program. Because the heap and stack are allocated at runtime, you need to gather heap and stack information while the application program is running. One way to gather this information is to use the **mallinfo()** function, and interpret the **uordblocks** field of the **mallinfo** structure as the heap size. You should call the **mallinfo()** function after your program completes the

call to the **LonInit()** function (you could also call it after the FTXL device is commissioned or during peak activity). For the **FTXL_Simple_0.elf** file shown above, the **uordblocks** field is 134 244 bytes (131 KB). Thus, the RAM requirement for the application includes the 593 496 bytes from the flash requirement plus the 134 244 bytes from the heap requirement, for a total RAM requirement of 727 740 bytes (711 KB).

Remember that the flash and RAM requirements are estimates, because a production application will likely not include the **malloc.h** file, nor call the **mallinfo()** function and associated **printf()** functions. In addition, for some applications, not all of the code and uninitialized data need to reside both in flash and RAM.

# Memory Use for Transactions

The FTXL LonTalk protocol 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 a 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 FTXL LonTalk protocol stack allocates for transactions.

**Table 47** lists the amount of memory required for each type of transaction. The number of bytes required for each type of transaction is an estimate; you should round these numbers upward when you use them in memory usage calculations.

**Table 47**. RAM Usage per Transaction Record

| 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 FTXL application should use. The values that you specify in the utility are defined in the **FtxlDev.h** file that the utility generates.

The FTXL LonTalk protocol 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.

**Table 48** on page 184 lists the amount of memory required for each type of application buffer. The number of bytes required for each type of application buffer is an estimate; you should round these numbers upward when you use them in memory usage calculations.

**Table 48**. RAM Usage per 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. Thus, the RAM usage for the default number of application buffers is approximately 15 KB.

## Memory for LONWORKS Resources

Each FTXL device uses LONWORKS resources, such as network variables defined for the device, address table entries, and aliases supported by the device.

The FTXL LonTalk protocol 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. However, when you calculate memory requirements, you should assume that all resources are in use.

**Table 49** lists the amount of memory required for each type of LONWORKS resource. The number of bytes required for each type of resource is an estimate; you should round these numbers upward when you use them in memory usage calculations. For example, as network variables can vary in their actual sizes, so the table uses an average value.

**Table 49**. RAM Usage per LonWorks Resource

| 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 |

**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. **Table 50** on page 185 lists the amount of flash memory

required for each type of LONWORKS resource. The number of bytes required for each type of resource is an estimate; you should round these numbers upward when you use them in flash memory usage calculations.

**Table 50**. Flash Usage per 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

An FTXL application typically has some non-volatile data that it must maintain across device reset (see *Providing Persistent Storage for Non-Volatile Data* on page 77). The FTXL LonTalk protocol 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. However, you should define enough free non-volatile storage to support the maximum use configuration. 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:

- The network image (**LonNvdSegNetworkImage**)
- The node definition (**LonNvdSegNodeDefinition**)
- The 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.

**Table 51** describes the amount of non-volatile memory required for the network image.

**Table 51**. 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) |

**Table 52** describes the amount of non-volatile memory required for the node definition.

**Table 52**. 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]) |
| **Notes**:<br>• *Node_SD_length* is the length of the self-documentation string for the node<br>• *NV_SD_length* is the length of the self-documentation string for all network variables (both static and dynamic) | |

**Table 53** on page 187 describes the amount of non-volatile memory required for the application data.

**Table 53**. Non-Volatile Memory Required for the Application Data

| Application Data | Bytes Required |
|---|---|
| Header | 16 |
| CPNVs | $\sum_{j}\left(CPNVlen_{j}\right)$ |
| File-based CPs | *File_length* |
| Application-specific data | *Data_length* |

**Notes**:

- CPNVs are configuration property network variables
- File-based CPs are configuration properties that are defined in configuration files
- $CPNVlen_j$ is the configuration network variable (CPNV) length of a specific CPNV value – the application data includes the sum of the CPNV lengths of all CPNV 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

**Table 54** on page 188 shows the amount of RAM and flash that are required for various example FTXL 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.

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, and does not alter the amount of flash memory required. Of course, the number of buffers can affect the application's performance.

**Table 54**. Example Memory Usage

| Number of Network Variables | Number of Transmit Transactions | Number of Receive Transactions | Number of Aliases | Number of Address Table Entries | RAM Required (in KB) | Flash Required (in KB) |
|---|---|---|---|---|---|---|
| 10 | 15 | 20 | 3 | 15 | 721 | 519 |
| 100 | 20 | 20 | 33 | 20 | 757 | 529 |
| 250 | 50 | 20 | 83 | 50 | 822 | 545 |
| 500 | 101 | 20 | 166 | 101 | 932 | 571 |
| 1000 | 203 | 25 | 333 | 203 | 1153 | 625 |
| 2000 | 407 | 50 | 666 | 407 | 1601 | 731 |
| 4000 | 814 | 100 | 1333 | 814 | 2497 | 945 |

**Figure 24** on page 189 shows the relative RAM and flash memory requirements for the data listed in **Table 54**. The figure shows that as the number of network variables for the FTXL application grows, the RAM requirement grows significantly, while the flash requirement grows modestly. These memory requirements do not include the requirements for application-specific data.

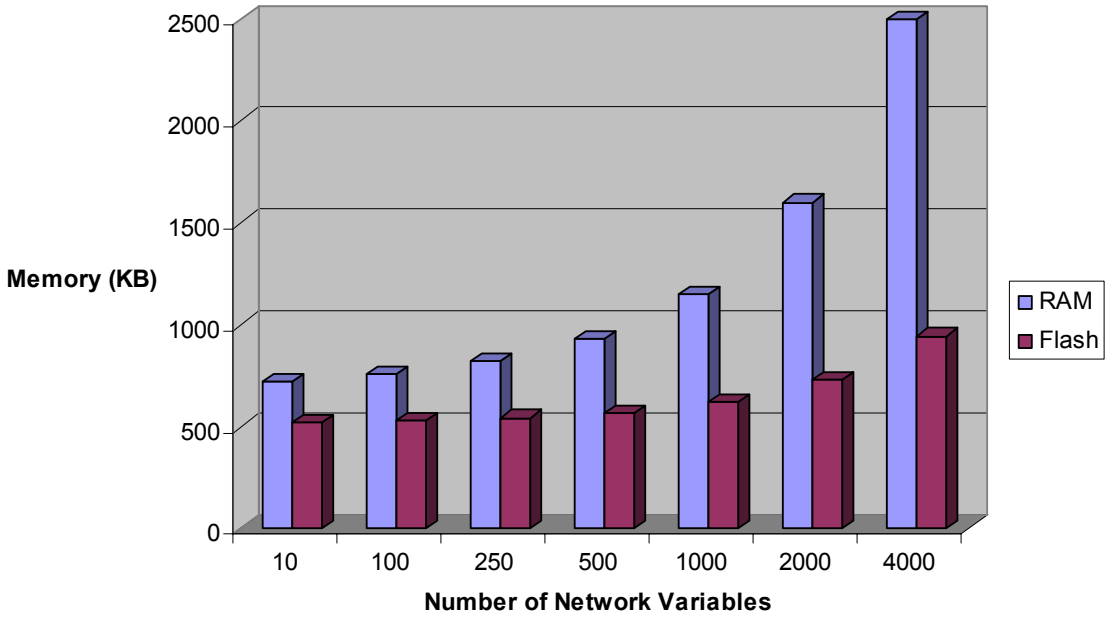



**Figure 24**. Example Memory Usage

# F

# Downloading an FTXL Application Over the Network

This Appendix describes considerations for designing an FTXL application that allows application updates over the network.

# Overview

For a Neuron-hosted device, you can update the application image over the network using LNS or another network management tool. However, you cannot use the same tools or technique to update an FTXL application image over the network. Many FTXL 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 an FTXL 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 FTXL LonTalk API nor the FTXL LonTalk protocol stack directly supports updating the FTXL 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 FTXL 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 an FTXL application download function. For additional considerations, see the Altera application note AN 429: Remote Configuration Over Ethernet with the Nios II Processor.

**Important**: This appendix does not describe how to download updates to the firmware image into the FTXL 3190 Free Topology Smart Transceiver. It only describes updates to the application image running on the host processor.

# Custom Application Download Protocol

The custom FTXL application protocol that you define for downloading an FTXL application over the network should support the following steps:

1. Prepare for application download.

   When the application download utility informs the current FTXL 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 FTXL 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 FTXL application should update the flash in the background (see *Download Capability within the Application* on page 193), however, it might be necessary for the protocol to define additional flow control to allow the FTXL 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 FTXL 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 Nios II processor can load the new application and restart the processor with the new image. See the Altera application note AN 458: Alternative Nios II Boot Methods for information about creating a custom boot loader.

**Important**: For the duration of the first three steps, the application must be running, the FTXL LonTalk protocol stack must be started, and the FTXL 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* on page 192. You can write the utility as an LNS 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 FTXL 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 FTXL 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 Nios II 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 FTXL 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).

The Altera tools provide a number of methods to control how memory is organized and how the system is booted. For information about these methods, see the *Nios II Software Developer's Handbook*. In particular, refer to the section on "Memory Usage" within the "Developing Programs using the HAL" chapter, and the descriptions of the **alt_load_section()** function within the "HAL API Reference" appendix.

By using the **alt_load_section()** function, your application can control how program data is copied from flash to RAM. Thus, you can, at boot time, decide whether to boot the old application image or the new one. In preparation for the boot, your application must direct how the boot should be performed. For example, your application could include a table at a fixed location that includes boot parameters, such as the flash locations of the **main()** function and your reset vectors. Switching to the new application is accomplished by patching these boot parameters and then resetting the processor. However, there should be a fallback if the boot parameters become corrupted, for example, because the device reset while writing to the sector that contains the boot parameters. This can be accomplished by maintaining two sets of boot parameters, a primary and backup, in different flash sectors. The backup is only updated after successfully booting from the primary.

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 FTXL LonTalk protocol 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 FTXL 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 FTXL LonTalk protocol stack library and operating system.

# G

# Example FTXL Applications

This Appendix describes the example applications that are included in the FTXL Developer's Kit. 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 FTXL Developer's Kit includes two example applications: the simple example and a dynamic interface example. The simple example application is a very simple application that simulates a voltage amplifier device. This device receives an input voltage value, multiplies the value by 2, and returns the new output value. The dynamic interface example application includes the same functionality as the simple example application, but adds the ability to change the SNVT types for two of the network variables and the ability to add (and modify and delete) dynamic network variables.

The following sections describe the two example applications, including their design, how to build them in the Nios II IDE, how to load them into the Nios II processor in the FPGA device on the DBC2C20 development board, and how to run them.

## *Example Application Files*

The two FTXL example applications are provided as project templates for the Nios IDE: the **FTXL Simple** project and the **FTXL Dynamic Interface** project. Each project template includes only a few files and cannot be compiled or run as is; you must run the LonTalk Interface Developer utility to generate and copy the required files for the project.

The **FTXL Simple** project includes the files listed in **Table 55**. They are installed to the [*Altera*]**\nios2eds\examples\software\FTXL_Simple** directory.

**Table 55**. FTXL Simple Example Files

| File Name | Description |
|---|---|
| FtxlHandlers.c | C file for the implementations of the callback handler functions for the application |
| main.c | Main application file |
| readme.txt | Readme file for the project, including a brief description of the project, and hardware and operating system requirements |
| Simple Example.lidprj | LonTalk Interface Developer project file |
| Simple Example.nc | Model file for the application |
| template.xml | Nios IDE project template file |

The **FTXL Dynamic Interface** project includes the files listed in **Table 56** on page 197. They are installed to the [*Altera*]**\nios2eds\examples\software\ FTXL_DynamicInterface** directory.

**Table 56**. FTXL Dynamic Interface Example Files

| File Name | Description |
|---|---|
| Dynamic Interface Example.lidprj | LonTalk Interface Developer project file |
| Dynamic Interface Example.nc | Model file for the application |
| FtxlHandlers.c | C file for the callback handler functions for the application |
| main.c | Main application file |
| readme.txt | Readme file for the project, including a brief description of the project, and hardware and operating system requirements |
| SNVT_RQ.H | C header file for the enumeration used with the **SNVT_obj_request** structure |
| template.xml | Nios IDE project template file |

See *Using the LonTalk Interface Developer Files* on page 61 for information about the files that the LonTalk Interface Developer utility generates and copies for an FTXL project.

# The Simple Example Application

The simple voltage amplifier example application is a very simple 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 design of the example application is very simple. It includes a single C source file (**main.c**), along with the FTXL API files generated by the LonTalk Interface Developer utility and a version of **FtxlHandlers.c** that has been customized for this example application.

The following sections describe the application's **main()** function, the application task (**appTask()**) function, event handler functions, callback handler functions, and model file.

# Main Function

The **main()** function is in the **main.c** file. The **main()** function creates an operating system task by calling the **OSTaskCreateExt()** µC/OS-II operating system function with the following arguments:

- Task entry point, which is a pointer to the **appTask()** function
- A NULL pointer for the arguments to the task
- A pointer to the top of the application stack
- The base priority of the task
- The ID of the task
- A pointer to the bottom of the application stack
- The size of the stack, in OS_STK units
- A NULL pointer for the extended task control block pointer (OSTCBExtPtr)
- A value of 0 for the options

The **main()** function then calls the **OSStart()** µC/OS-II operating system function to start the operating system. An application should not use the FTXL OSAL functions for creating a task; if the application needs extra tasks, it should call operating system functions directly. The OSAL functions for creating a task are designed for creating FTXL LonTalk protocol stack tasks only.

The **main()** function is shown below.

```
/* The main function simply creates the application task
 * and then starts multi-tasking
 */
int main(void) {
  /* Create an application task to implement the main
     control loop. */
  OSTaskCreateExt(appTask,
                  NULL,
                  (void *)&appStack[APP_STACKSIZE],
                  OS_APPLICATION_PRIORITY_BASE,
                  OS_APPLICATION_PRIORITY_BASE,
                  appStack,
                  APP_STACKSIZE,
                  NULL,
                  0);

  /* Start the operating system. The rest of the
   * application executes under appTask.
   */
  OSStart();
  return 0;
}
```

# Application Task Function

The application task function, **appTask()**, is in the **main.c** file. The **appTask()** function performs the following tasks:

- Creates the "event ready" event using the FTXL OSAL **OsalCreateEvent()** function. The **LonEventReady()** callback handler function uses this event to wake up the application task to process FTXL network events.
- Calls the **LonInit()** function to initialize the FTXL LonTalk protocol stack and FTXL Transceiver. If this function fails, the **appTask()** function calls the FTXL OSAL **OsalDeleteEvent()** function to delete the "event ready" event.
- If the **LonInit()** function is successful, the **appTask()** function begins an infinite loop to wait for FTXL network events. When an event occurs, it calls the **LonEventPump()** function to process the event.

Although the **main()** and **appTask()** functions for this application are part of an example, you can use the same basic algorithmic approach for a production-level application.

The the **appTask()** function is shown below.

```
/* The application task initializes the FTXL LonTalk
 * protocol stack and implements the main control loop.
 * The bulk of the application processing is performed in
 * the myNvUpdateOccurred event handler.
 */
void appTask(void* pData) {
  /* Create the "event ready" event, which is signaled by
   * the myEventReady callback to wake this task up to
   * process FTXL events.
   */
  if (OsalCreateEvent(&eventReadyHandle) ==
        OSALSTS_SUCCESS) {
    /* Initialize the FTXL LonTalk API and FTXL Transceiver
     */
    if (LonInit() == LonApiNoError) {
      /* This is the main control loop, which runs
       * forever. */
      while (TRUE) {
        /* 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();
        }
      }
      OsalDeleteEvent(&eventReadyHandle);
  }
}
```

# Event Handler Function

To signal to the main application the occurrence of certain types of events, the FTXL 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* on page 201.

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 FTXL 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 FTXL LonTalk protocol 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 FTXL 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 file, **Simple Example.nc,** defines the LONWORKS interface for the example FTXL device.

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* on page 199.

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* on page 23.

To change the LONWORKS interface and functionality of the example application, perform the following steps:

1. Define the interface in the **Simple Example.nc** model file.
2. Run the LonTalk Interface Developer utility to generate an updated application framework.
3. Make appropriate changes to the callback handler functions in the **FtxlHandlers.c** file or the **main.c** file.
4. Rebuild the project.
5. Optional: Load the generated XIF file into the FTXL Transceiver.
6. Load the new executable file into the Nios II processor.

# The Dynamic Interface Example Application

The dynamic interface example application demonstrates the basics of using both dynamic network variables and the LONMARK changeable-type protocol in an FTXL application. The example is not a complete implementation of the protocol, but is meant to serve as a starting point for writing your own application.

To demonstrate dynamic network variables, the example application allows you to add, modify, or delete network variables of type **SNVT_amp**. Each output dynamic network variable represents the aggregated current consumption of one or more input dynamic network variables. For this example, the output dynamic network variables represent logical circuits. You can define as many circuits as needed, but the total number of dynamically added network variables can be no more than 50.

The dynamic network variables must use the following naming convention:

- For output network variables, use the name **nvoAmp** plus a single-character suffix that represents a particular logical circuit.
- For input network variables, use the name **nviAmp**, plus the suffix for the corresponding output network variable, with an additional suffix that identifies the particular input. The additional suffix can be one or more characters.

**Example**: For logical circuit A, you could name the output network variable **nvoAmpA**, and you could name three of its input network variables **nviAmpA01**, **nviAmpA02**, and **nviAmpA03**. The value of the **nvoAmpA** network variable would be the aggregrate sum of the values of the three input network variables.

The example application maintains an array for a map of the circuits defined for the device. The application uses the functions described in *Application-Specific Utility Functions* on page 213 to maintain the circuits of dynamic network variables and to propagate them to the network.

To demonstrate changeable-type network variables, the example application has a configuration network variable (CPNV) named **nciNvType**, which maintains the current type and last-known good value of the two network variables, **nviVolt** and **nvoVoltFb**. The application supports changing the network variable type for these two network variable between the default type (**SNVT_volt**) and the

**SNVT_volt_mil** type. Any attempt to change the NV to an unsupported type causes the device to reject the change and to revert the **nciNvType** CPNV to its last-known good value.

The design of the example application is relatively simple. It includes a single C source file (**main.c**), along with the FTXL LonTalk API files generated by the LonTalk Interface Developer utility and a version of **FtxlHandlers.c** that has been customized for this example application.

The following sections describe the application's **main()** function, the application task (**appTask()**) function, event handler functions, callback handler functions, application-specific utility functions, and model file.

## Main Function

The **main()** function is in the **main.c** file. The **main()** function initializes the status for the application's functional blocks, and then initializes the circuit map for the logical circuit defined for tracking the device's aggregated current usage.

The **main()** function creates an operating system task by calling the **OSTaskCreateExt()** µC/OS-II operating system function with the following arguments:

- Task entry point, which is a pointer to the **appTask()** function
- A NULL pointer for the arguments to the task
- A pointer to the top of the application stack
- The base priority of the task
- The ID of the task
- A pointer to the bottom of the application stack
- The size of the stack, in OS_STK units
- A NULL pointer for the extended task control block pointer (OSTCBExtPtr)
- A value of 0 for the options

The **main()** function then calls the **OSStart()** µC/OS-II operating system function to start the operating system. An application should not use the FTXL OSAL functions for creating a task; if the application needs extra tasks, it should call operating system functions directly. The OSAL functions for creating a task are designed for creating FTXL LonTalk protocol stack tasks only.

The **main()** function is shown below.

```
/* The main function initializes some global variables,
 * creates the application task, and then starts
 * multi-tasking.
 */
int main(void) {

  unsigned fbIndex;

  /* Initialize the FbStatus array. */
  memset(FbStatus, 0, sizeof(FbStatus));
  for (fbIndex = 0; fbIndex < FBIDX_count; fbIndex++) {
```

```
    LON_SET_UNSIGNED_WORD(FbStatus[fbIndex].object_id,
          fbIndex);
  }

  /* Initialize the circuitMap.  Initially there are no
   * members.  If any dynamic NVs have been defined, the
   * FTXL LonTalk API will call the LonNvAdded event
   * handler during LonInit, which will in turn call
   * myNvAdded.  The myNvAdded function will add the NV to
   * circuitMap assuming that it follows the naming
   * convention and its type is SNVT_amp.
   */
  memset(circuitMap, 0, sizeof(circuitMap));

  /* Create an application task to implement the main
   * control loop. */
  OSTaskCreateExt(appTask,
                  NULL,
                  (void *)&appStack[APP_STACKSIZE],
                  OS_APPLICATION_PRIORITY_BASE,
                  OS_APPLICATION_PRIORITY_BASE,
                  appStack,
                  APP_STACKSIZE,
                  NULL,
                  0);

  /* Start the operating system. The rest of the
   * application executes under appTask.
   */
  OSStart();
  return 0;
}
```

# Application Task Function

The application task function, **appTask()**, is in the **main.c** file. The **appTask()** function performs the following tasks:

- Creates the "event ready" event using the FTXL OSAL **OsalCreateEvent()** function. The **LonEventReady()** callback handler function uses this event to wake up the application task to process FTXL network events.
- Calls the **LonInit()** function to initialize the FTXL LonTalk protocol stack and FTXL Transceiver. If this function fails, the **appTask()** function calls the FTXL OSAL **OsalDeleteEvent()** function to delete the "event ready" event.
- If the **LonInit()** function is successful, the **appTask()** function:
    - Reads the **nciNvType** configuration property network variable in non-volatile data to set its type to the last known good value.
    - Begins an infinite loop to wait for FTXL network events. When an event occurs, it calls the **LonEventPump()** function to process the event.

Although the **main()** and **appTask()** functions for this application are part of an example, you can use the same basic algorithmic approach for a production-level application.

The the **appTask()** function is shown below.

```
/* The application task initializes the FTXL LonTalk
 * protocol stack and implements the main control loop.
 * The bulk of the application processing is performed in
 * the myNvUpdateOccurred event handler.
 */
void appTask(void* pData) {
  /* Create the "event ready" event, which is signaled by
   * the myEventReady callback to wake this task up to
   * process FTXL events.
   */
  if (OsalCreateEvent(&eventReadyHandle) ==
        OSALSTS_SUCCESS) {
    /* Initialize the FTXL LonTalk API and FTXL Transceiver
     */
    if (LonInit() == LonApiNoError) {
      /* The CP may have been updated by reading
       * non-volatile data.  If it looks good, update
       * nciNvTypeLastKnownGoodValue.
       */
      if (LON_GET_UNSIGNED_WORD(nciNvType.type_index)
          == INDEX_SNVT_VOLT ||
          LON_GET_UNSIGNED_WORD(nciNvType.type_index)
          == INDEX_SNVT_VOLT_MIL) {
        memcpy((void*)&nciNvTypeLastKnownGoodValue,
          (void*)&nciNvType, sizeof(SCPTnvType));
      }
      /* This is the main control loop, which runs
         forever. */
      while (TRUE) {
        /* 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();
      }
    }
    OsalDeleteEvent(&eventReadyHandle);
  }
}
```

# *Event Handler Functions*

To signal to the main application the occurrence of certain types of events, the FTXL LonTalk API calls specific event handler functions. For the dynamic interface example application, six of the API's event handler functions have been implemented to provide application-specific behavior.

The **FtxlHandlers.c** file contains the modified functions, each of which simply calls a 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.

For the dynamic interface example application, the following API event handler functions have been implemented to provide application-specific behavior:

- **LonReset()**: This function is called when the device is reset. This function calls **myReset()** in **main.c**. The **myReset()** function calls the **ProcessTypeChange()** utility function.
- **LonOnline()**: This function is called when the device comes online. This function calls **myOnline()** in **main.c**. The **myOnline()** function calls the **ProcessTypeChange()** utility function.
- **LonNvUpdateOccurred()**: This function is called when the host processor receives a network-variable update. This function calls the **myNvUpdateOccurred()** function in **main.c**.
- **LonNvAdded()**: This function is called when a dynamic network variable is added to the FTXL device. It is also called during device startup to retrieve dynamic network variables that were created prior to device reset. This function calls the **myNvAdded()** function in **main.c**.
- **LonNvTypeChanged()**: This function is called when the any of the attributes of a dynamic network variable change. This function calls the **myNvTypeChanged()** function in **main.c**.
- **LonNvDeleted()**: This function is called when a dynamic network variable is deleted. This function calls the **myNvDeleted()** function in **main.c**.

# myNvUpdateOccurred()

The **myNvUpdateOccurred()** function contains a C **switch** statement, which contains three **case** statements and a **default** statement to process the following types of updates:

- A change to the **nciNvType** configuration network variable (CPNV), which controls the type of the **nviVolt** and **nvoVoltFb** network variables.
- A change to the node object's **nviRequest** network variable, which controls the status of the FTXL device's functional blocks.
- A change to the voltage amplifier's **nviVolt** network variable, which controls the application's behavior as a voltage amplifier.
- Any other change, which is processed as a dynamic network variable update. The change is ignored if it does not affect the logical circuit defined for tracking the device's aggregated current usage (that is, the dynamic network variable is not of type **SNVT_amp** and named according the required naming convention).

The **case** statement for the **nciNvType** CPNV (specified by the **LonNvIndexNciNvType** network variable index) calls the **ProcessTypeChange()** utility function.

The **case** statement for the **nviRequest** CPNV (specified by the **LonNvIndexNviRequest** network variable index) performs the following tasks:

- Checks whether the object index represents a supported object:
    - For non-supported objects, sets the object status to invalid.
    - For supported objects:
        - Checks whether the command applies to all objects or to a specified functional block
        - Performs the specified command (implemented in a **switch** statement based on the **nviRequest.object_request** variable)
        - Checks whether it should report the status for the object
- Propagates the change to the network

The **case** statement for the **nviVolt** network variable (specified by the **LonNvIndexNviVolt** network variable index) performs the following tasks:

- Checks whether the functional block is disabled. If it is disabled, it does nothing. If is it not disabled:
    - Performs range checking for the network variable
    - Sets the output network variable to double the value of the input network variable
    - Propagates the output network variable to the network

The **default** statement checks whether the network variable index is within the total allowed for the device and whether the change is for an input network variable. In this case, it calls the **UpdateCircuitOutput()** utility function to update the logical circuit defined for tracking the device's aggregated current usage. Otherwise, it ignores the change.

The functional blocks and network variables are defined in the model file, which is described in *Model File* on page 214.

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 LonNvIndexNciNvType:
    {
      /* The nciNvType has been updated, which controls the
       * type of nviVolt and nvoVoltFb. Validate the change
       * to determine whether to accept the update or not.
       */
      ProcessTypeChange();
      break;
    }

    case LonNvIndexNviRequest:
    {
      /* Node object request has been received. */
```

```
LonBool processActuatorInputs = FALSE;
unsigned index =
    LON_GET_UNSIGNED_WORD(nviRequest.object_id);
memset((void *)&nvoStatus, 0, sizeof(nvoStatus));
LON_SET_UNSIGNED_WORD(nvoStatus.object_id, index);

if (index >= FBIDX_count) {
  /* We don't support this object - flag it as an
   * invalid ID. */
  LON_SET_ATTRIBUTE(nvoStatus, LON_INVALIDID, 1);
}
else {
  /* If reportStatus is TRUE, set the objectStatus to
   * the status of the specified functional block.
   */
  LonBool reportStatus = TRUE;
  int i;

  /* start and limit define which functional block or
   * blocks will be effected.
   */
  int start;
  int limit;
  if (index == FBIDX_NodeObject) {
    /* Command applies to all functional blocks. */
    start = 0;
    limit = FBIDX_count-1;
  }
  else {
    /* Command only applies to the specified
     * functional block. */
    start = index;
    limit = index;
  }

  switch (nviRequest.object_request) {
    case RQ_NORMAL:
      /* Set the object (or all objects) to normal by
       * clearing the disabled and in_override flags.
       */
      for (i = start; i <= limit; i++) {
        if (i == FBIDX_VoltActuator &&
        LON_GET_ATTRIBUTE(FbStatus[i], LON_DISABLED))
        {
          /* Actuator was disabled, but is now
           * enabled. Process current input values.
           */
          processActuatorInputs = TRUE;
        }
        LON_SET_ATTRIBUTE(FbStatus[i], LON_DISABLED,
            0);
        LON_SET_ATTRIBUTE(FbStatus[i],
            LON_INOVERRIDE, 0);
      }
      break;

    case RQ_UPDATE_STATUS:
```

```
  /* Update the status.  If the object is not the
   * node object, just return the current status
   * of the object.  Special processing below for
   * node object only.
   */
  if (index == FBIDX_NodeObject) {
    /* When requesting the status of the node
     * object, return a status that represents
     * the OR of the statuses of all functional
     * blocks.
     * Don't report the status of the node object
     * - use the summary below.
     */
    reportStatus = FALSE;

    for (i = start; i <= limit; i++) {
      nvoStatus.Flags_1 |= FbStatus[i].Flags_1;
      nvoStatus.Flags_2 |= FbStatus[i].Flags_2;
      nvoStatus.Flags_3 |= FbStatus[i].Flags_3;
      nvoStatus.Flags_4 |= FbStatus[i].Flags_4;
    }
  }
  break;

case RQ_REPORT_MASK:
  /* All bits are zero unless set explicitly.
   * Don't report the status of the object.  The
   * nvoStatus is filled in below.  All fields
   * that are untouched are left as 0, indicating
   * that the function block does not support the
   * associated operation.
   */
  reportStatus = FALSE;

  /* Mark this as the result of a RQ_REPORT_MASK
   */
  LON_SET_ATTRIBUTE(nvoStatus, LON_REPORTMASK,
      1);

  /* All objects support disable */
  LON_SET_ATTRIBUTE(nvoStatus, LON_DISABLED, 1);

  break;

case RQ_DISABLED:
  /* Disable the object or all objects */
  for (i = start; i <= limit; i++) {
    LON_SET_ATTRIBUTE(FbStatus[i], LON_DISABLED,
        1);
  }
  break;

case RQ_ENABLE:
  /* Enable the object or all objects */
  for (i = start; i <= limit; i++) {
    if (i == FBIDX_VoltActuator &&
    LON_GET_ATTRIBUTE(FbStatus[i], LON_DISABLED))
```

```
            {
              /* Actuator was disabled, but is now
               * enabled. Process current input values.
               */
              processActuatorInputs = TRUE;
            }
            LON_SET_ATTRIBUTE(FbStatus[i], LON_DISABLED,
                0);
          }
          break;

        default:
          /* Mark all other requests as invalid */
          LON_SET_ATTRIBUTE(nvoStatus,
              LON_INVALIDREQUEST, 0);
            reportStatus = FALSE;
      }
      if (reportStatus) {
        /* Report the current status of the functional
         * block */
        nvoStatus = FbStatus[index];
      }
    }

    /* Propagate the value of nvoStatus */
    if (LonPropagateNv(LonNvIndexNvoStatus) !=
        LonApiNoError) {
      /* Handle error here, if desired. */
    }

    /* The actuator was disabled, but has been enabled.
     * Process its input values.
     */
    if (processActuatorInputs) {
      ProcessNviVoltUpdate();
    }
    break;
  }

  case LonNvIndexNviVolt:
  {
    /* nviVolt has been updated.  Process it unless the
     * FB is disabled.
     */
    ProcessNviVoltUpdate();
    break;
  }
  /* Add more input NVs here, if any */

  default:
    if (nvIndex >= LON_STATIC_NV_COUNT &&
        circuitMap[nvIndex].isInput) {
      /* This may be a circuit input.  If so, update the
       * circuit output.
       */
      UpdateCircuitOutput(circuitMap[nvIndex].circuitId);
    }
```

```
        break;
    }
}
```

# myNvAdded()

The **myNvAdded()** function calls the **UpdateCircuitMap()** utility function to update the logical circuit for the added network variable.

The **myNvAdded()** function is shown below.

```
/* This function is called by the FTXL LonNvAdded event
 * handler, indicating that a dynamic network variable has
 * been added.  It is also called during LonInit for each
 * dynamic NV that is found in the FTXL persistent storage.
 */
void myNvAdded(const unsigned index,
         const LonNvDefinition* const pNvDef) {
  /* Based on the NV name and type, update the circuit
   * map. */
  UpdateCircuitMap(index, pNvDef);
}
```

# myNvTypeChanged()

The **myNvTypeChanged()** function calls the **UpdateCircuitMap()** utility function to update the logical circuit for the changed network variable.

The **myNvTypeChanged()** function is shown below.

```
/* This function is called by the FTXL LonNvTypeChanged
 * event handler, indicating that a dynamic network
 * variable LonNvDefinition has been updated.  Since either
 * the SVNT_id or the name may have changed, the NV may
 * need to be added to a circuit, removed from a circuit,
 * or moved to a different circuit.
 */
void myNvTypeChanged(const unsigned index,
         const LonNvDefinition* const pNvDef) {
  /* Based on the NV name and type, update the circuit
   * map. */
  UpdateCircuitMap(index, pNvDef);
}
```

# myNvDeleted()

The **myNvDeleted()** function checks to see if there are any dynamic network variables defined. If there are, this function removes the specified network variable from the logical circuit defined for the application, and calls the **UpdateCircuitOutput()** utility function to update the logical circuit and propagate the changes to the network.

The **myNvDeleted()** function is shown below.

```
/* This function is called by the FTXL LonNvDeleted event
 * handler, indicating that a dynamic network variable has
 * been deleted.
 */
```

```
void myNvDeleted(const unsigned index) {
  LonByte oldCircuitId = circuitMap[index].circuitId;
  if (oldCircuitId != NO_CIRCUIT) {
    /* Remove the NV from the circuit. */
    circuitMap[index].circuitId = NO_CIRCUIT;
    /* Update the output NV of the circuit to which this NV
     * used to belong.
     */
    UpdateCircuitOutput(oldCircuitId);
  }
}
```

# myReset()

The **myReset()** function calls the **ProcessOnlineEvent()** utility function to process changes to input network variables and configuration property values.

The **myReset()** function is shown below.

```
/*
 *  This function is called by the FTXL LonReset event
 * handler, indicating that the FTXL LonTalk protocol stack
 * has been reset.
 */
void myReset(void) {
  LonStatus status;
  /* Check to see if the device is online. */
  if (LonQueryStatus(&status) == LonApiNoError &&
        status.NodeState == LonConfigOnLine)
  {
    /* Process inputs that may have changed while the
     * device was offline. */
    ProcessOnlineEvent();
  }
}
```

# myOnline()

The **myOnline()** function calls the the **ProcessOnlineEvent()** utility function to process changes to input network variables and configuration property values.

The **myOnline()** function is shown below.

```
/*
 *  This function is called by the FTXL LonOnline event
 * handler, indicating that the FTXL LonTalk protocol stack
 * has been set online.
 */
void myOnline(void) {
  /* Process inputs that may have changed while the device
   * was offline. */
  ProcessOnlineEvent();
}
```

## Application-Specific Utility Functions

The dynamic interface 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.
- **ProcessTypeChange()**: For each type change to the **nviVolt** or **nvoVoltFb** network variables, this function processes the type change. It checks that the type change is valid (from **SNVT_volt** to **SNVT_volt_mil** or from **SNVT_volt_mil** to **SNVT_volt**), converts the network variable's value to match the new type, stores the type and value in the **nciNvType** configuration property network variable, and stores the value in non-volatile memory.
- **UpdateCircuitMap()**: As dynamic network variables are added, modified, or deleted from the logical circuits defined for the device, this function updates the circuit map table. This function verifies that the update is for a vaid circuit, that the network variables use the proscribed naming convention, and calls the **UpdateCircuitOutput()** function.
- **UpdateCircuitOutput()**: This function reads the circuit map table to calculate the sum of the input dynamic network variables for each output dynamic network variable, and then propagates the output dynamic network variable to the network.

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 FTXL LonTalk API calls specific callback handler functions. For the dynamic interface 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 FTXL LonTalk protocol 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 FTXL 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 file, **Dynamic Interface Example.nc,** defines the LONWORKS interface for the example FTXL device.

The model file defines two functional blocks: **NodeObject** and **VoltActuator**. The **NodeObject** functional block allows a network management tool to enable or disable the functional blocks for the FTXL device. The **VoltActuator** functional block defines the interface for the application.

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 *Callback Handler Function* on page 213.

The two network variables for the **VoltActuator** functional block include references to a configuration network variable (CPNV), **nciNvType**. This reference allows the **nviVolt** and **nvoVoltFb** network variables to maintain type changes in non-volatile memory, and thus be preserved across device resets.

The model file does not include definitions for any dynamic network variables. However, the application supports the addition, modification, and deletion of dynamic network variables. You use the LonTalk Interface Developer utility to specify the number of dynamic network variables supported by the application.

The model file is shown below.

```
#pragma enable_sd_nv_names

network input cp SCPTnvType nciNvType;
network input SNVT_obj_request nviRequest;
network output sync SNVT_obj_status nvoStatus;

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

network input changeable_type SNVT_volt nviVolt
nv_properties
{
   global nciNvType
};

network output changeable_type SNVT_volt bind_info(unackd)
   nvoVoltFb
nv_properties
{
   global nciNvType
};
```

```
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* on page 23.

To change the LONWORKS interface and functionality of the example application, perform the following steps:

1. Define the interface in the **Dynamic Interface Example.nc** model file.
2. Run the LonTalk Interface Developer utility to generate an updated application framework.
3. Make appropriate changes to the callback handler functions in the **FtxlHandlers.c** file or the **main.c** file.
4. Rebuild the project.
5. Optional: Load the generated XIF file into the FTXL Transceiver.
6. Load the new executable file into the Nios II processor.

# Setting up the Nios II IDE for the Example Applications

To set up the Nios II IDE to use the example FTXL applications, perform the following general steps:

1. Optional: Create a new workspace for each example application project.
2. Create a new application project based on one of the two FTXL project templates.
3. Run the LonTalk Interface Developer utility to generate and copy the necessary files for the project.
4. Build the project.

The following sections describe these steps. After you build the project, you can load it into the Nios II processor and run it.

## *Creating a New FTXL Application Project*

You can create each example project in a new workspace or use an existing workspace. To work in a new workspace, select **File → Switch Workplace** to open the Workspace Launcher window, from which you can select a new or existing workspace.

To create a new application project for the FTXL simple example application:

1. Select **File → New → Nios II C/C++ Application** to open the New Project window.
2. From the New Project window's **Select Project Template** selection box, select the **FTXL Simple** project.

3. Optional: Enter a project name in the **Name** field. The default name is **FTXL_Simple_0**.
4. Specify a location for this project (such as **C:\MyFtxl**) by selecting the **Specify Location** checkbox and specifying the location in the **Location** field. The directory name must not contain spaces. If you use the default location, your source files will be placed in the project workspace directory.
5. Specify the target hardware. Click **Browse** in the Select Target Hardware area to open the Select Target Hardware dialog.
    a. In the Select Target Hardware dialog, browse to the [*Altera*]**\nios2eds\examples\vhdl\DBC2C20_FTXL\Standard** directory and select the SOPC Builder system file for the project (**nios_cpu.ptf**).
    b. Click **Open** to select the file and close the Select Target Hardware dialog.
6. Do not modify the **CPU** field in the Select Target Hardware area; the name of the CPU is contained in the **nios_cpu.ptf** file. However, if this file specified more than one Nios II processor, you would need to select which one the application project should use. The **nios_cpu.ptf** file specifies only one Nios II processor.
7. Click **Finish** to create the project and generate the project's system library. The New Project window should look similar to **Figure 25** on page 217.

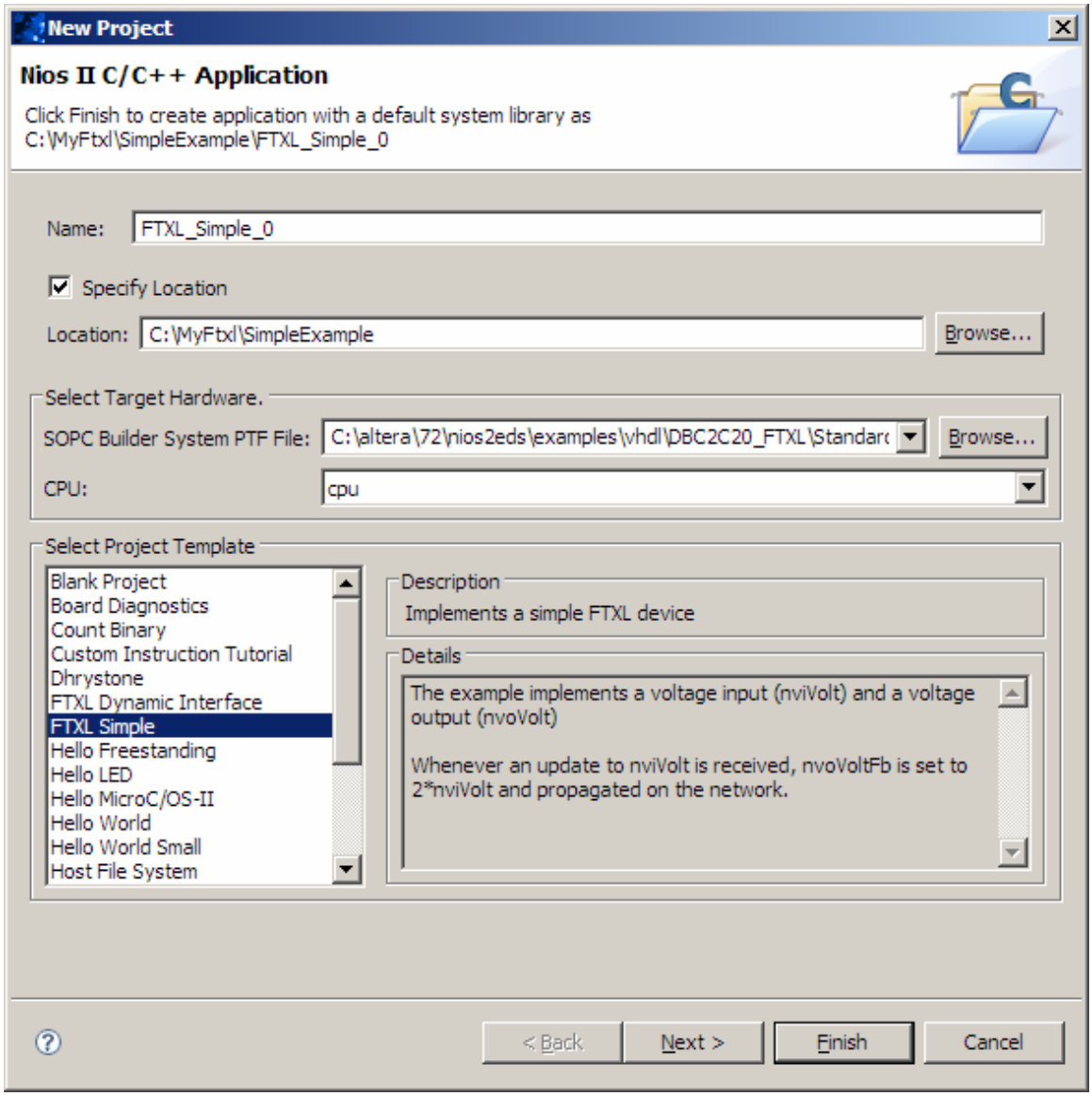



**Figure 25**. New Project Window for the FTXL Simple Project

Perform the same steps to create a new project for the dynamic interface example application, but in step 2, select the **FTXL Dynamic Interface** project. Both example applications use the same target hardware.

# *Running the LonTalk Interface Developer Utility*

Before you can compile the newly created project, you must run the LonTalk Interface Developer utility to generate application-specific files and copy other required files to the project directory.

To run the LonTalk Interface Developer utility for the project:

1. Start the LonTalk Interface Developer utility from the Windows Start menu: **Start → Programs → Echelon FTXL Developer's Kit → LonTalk Interface Developer**.

2. From the Welcome to LonTalk Interface Developer page of the utility, click **Browse** to open the appropriate project file:
   - [*MyFTXL*]**\FTXL_Simple_0\Simple Example.lidprj**
   - [*MyFTXL*]**\FTXL_DynamicInterface_0\Dynamic Interface Example.lidprj**
     where [*MyFTXL*] is the location that you specified for the project when you created it within the Nios IDE. The subdirectory name is the same as the project name that you specified when you created the project.
3. Because you opened an existing project file, you can accept the predefined values on all of the subsequent pages of the LonTalk Interface Developer utility by clicking **Next** on each page.
4. Click **Finish** on the Build Progress and Summary page to close the LonTalk Interface Developer utility.

Within the Nios IDE, right-click within the Nios II C/C++ Projects pane and select **Refresh** to see the newly generated and copied files for the project. You can now compile and build the project.

# Building the Example Application Image

The examples that are included with the FTXL Developer's Kit include the hardware design for the Nios II processor, as well as the software for the FTXL LonTalk API, and two example applications. You must separately build the hardware image and the software image.

The example applications define a single configuration, **Debug**. If you want to define other configurations (for example, **Release**) for the example applications, define them within the Nios IDE as described in *Specifying the Properties for the Application* on page 106.

For more information about the hardware image, see the *FTXL Hardware Guide*.

## *Building the Reference Design Hardware Image*

The FTXL Developer's Kit includes a pre-built hardware design image for the Nios II processor running on the Cyclone II FPGA device on the devboards.de DBC2C20 development board. To use the pre-built image, see *Loading the Example Application Image into Flash* on page 219.

You might need to rebuild the hardware image, for example, if you want to modify the design, run the Nios II processor on a different device than a Cyclone II FPGA on the DBC2C20 development board, or if your Altera tools license requires you to rebuild the image.

See the *FTXL Hardware Guide* for information about building the reference design hardware image.

## *Building the Example Software Image*

To build the software image for either of the example applications that are included with the FTXL Developer's Kit:

1. Start the Nios II EDS IDE.
2. Ensure that the workspace includes the example application project that you want to build.
3. Select **Project → Build Project** or **Project → Build All.** You can also right-click the project folder from the Nios II C/C++ Projects pane and select **Build Project**.

The first build for a new project can take a few minutes.

To build the application automatically, see *Building the Application Image* on page 107.

After you build the project, you can run it, as described in *Running the Example Application* on page 220, or you can load the software image into flash, as described in *Loading the Example Application Image into Flash*.

# Loading the Example Application Image into Flash

You can choose to load the hardware and software images into the FPGA device at the same time, or you can choose to load them separately. To load the hardware design separately into RAM for the FPGA device, see the *FTXL Hardware Guide* for more information. To load both images at the same time, use the Nios IDE, as described below.

You can load the example application software image into the FPGA's flash memory if you have a development license from Altera Corporation for the Nios II processor. If you do not have a license, you can still run the example application, but you must run it from the Nios II EDS IDE while the USB-Blaster download cable remains connected to the DBC2C20 development board.

To load the software image into flash:

1. Ensure that the DBC2C20 development board is powered on and that the USB-Blaster download cable is connected to the JTAG header connector (**P1**).
2. Start the Nios II EDS IDE.
3. Ensure that the workspace includes the example application project that you want to load.
4. Select the appropriate project (either **FTXL_Simple_0** or **FTXL_Dynamic_Interface_0**) from the Nios II C/C++ Projects pane.
5. Select **Tools → Flash Programmer** to open the Flash Programmer window.
    a. In the Flash Programmer window, right-click **Flash Programmer** and select **New** to create a configuration for the selected project.
    b. Provide a name for the configuration in the **Name** field.
    c. Select **Program software project into flash memory**. Ensure that the **Project** field displays the correct project name, or click **Browse** to select the project.
    d. Do not modify the information displayed in the Target Hardware area.

e. Select **Program FPGA configuration data into hardware-image region of flash memory** to load the hardware design into flash along with the software design.

   If necessary, specify the following information for the FPGA configuration:

   i. Click **Browse** next to the **FPGA Configuration (SOF)** dropdown list box to select the **DBC2C20_FTXL.sof** hardware design file from the [*Altera*]**\nios2eds\examples\vhdl\DBC2C20_FTXL\Standard** directory.

   ii. Select **Nios II EP2C35 epcs** from the **Hardware Image** dropdown list box.

   iii. Select **epcs_controller** from the **Memory** dropdown list box.

   iv. Enter **0x0** in the Offset field.

f. Select **Validate Nios II system ID before software download**.

g. Click **Apply** to save the named configuration.

h. Click **Program Flash** to load the software image into the Nios II processor. Loading the software image can take a few moments.

i. If the Program Flash Now? dialog appears, click **Yes**.

6. After the software is loaded, perform a reset by disconnecting power from the DBC2C20 development board and reconnecting power to the board.
7. Close the Quartus II Programmer window. You can also close the Nios II IDE window.

The Nios II processor runs the loaded software as soon as the processor completes restart processing.

# Running the Example Applications

If you loaded the application image into the Nios II processor, the application runs automatically as soon as the Nios II processor is properly programmed and reset.

You can also run the application from the Nios II EDS IDE:

1. Ensure that the DBC2C20 development board is powered on and that the USB-Blaster download cable is connected to the JTAG header connector (**P1**).
2. Start the Nios II EDS IDE.
3. Ensure that the workspace includes the example application project that you want to run.
4. Right-click the appropriate project (either **FTXL_Simple_0** or **FTXL_Dynamic_Interface_0**) from the Nios II C/C++ Projects pane and select **Run As → Nios II Hardware**. The Nios II EDS IDE recompiles the project.

5. If you have a valid Nios II development license, and have already loaded the configuration data into the Cyclone II FPGA, proceed to step 7.
6. If you do not have a valid Nios II development license, or have not loaded the configuration data into the Cyclone II FPGA:
    a. The Nios II EDS IDE displays the following text in the Console window:

    ```
    There are no Nios II CPUs with debug modules
    available which match the values specified.  Please
    check that your PLD is correctly configured,
    downloading a new SOF file if necessary.
    ```

    b. The Quartus II Programmer window opens.
    c. In the Quartus II Programmer window, click **Add File** to open the Select Programming File dialog.
    d. In the Select Programming File dialog, select the **DBC2C20_FTXL.sof** file and click **Open**.
    e. Ensure that the USB-Blaster download cable is defined in the Chain Description File for the project. See *Using a Device Programmer for the FPGA Device* on page 103 for information about configuring the Chain Description File.
    f. Select the **Program/Configure** checkbox for the **DBC2C20_FTXL.sof** file.
    g. Click **Start** to load the selected SRAM Object File (**DBC2C20_FTXL.sof**) into the Nios II processor.
    h. Do not close the Quartus II Programmer window. You must leave this window open while you are running the example application.
    i. Return to the Nios II EDS IDE, right-click the **Application** project from the Nios II C/C++ Projects pane, and select **Run As → Nios II Hardware**. The Nios II EDS IDE recompiles the project.
7. You should see the Service Pin LED (LED 4) on the DBC2C20 development boad flash slowly, indicating that the FTXL device is not configured. Use a network management tool, such as the LonMaker Integration tool, to configure and commission the device (you do not need to load the application into the FTXL Transceiver). The Service Pin LED should be off when the device is properly configured.

When you commission the FTXL device, use one of the following XIF files, depending on which example application you want to run:

- [*MyFTXL*]**\FTXL_Simple_0\Simple Example.xif**
- [*MyFTXL*]**\FTXL_DynamicInterface_0\Dynamic Inteface Example.xif**
  where [*MyFTXL*] is the location that you specified for the project when you created it within the Nios IDE. The subdirectory name is the same as the project name that you specified when you created the project.

To send a Service Pin message to the network management tool, press the Service Pin Button (button 0, labeled **P25**) on the DBC2C20 development board. Of the set of four pushbuttons, Button 0 is the closest to the center of the board.

## Running the Simple Example

To verify that the application runs as expected, connect the FTXL Transceiver Board to a network management tool, such as the LonMaker tool. From the tool, modify the value for the **nviVolt** network variable and confirm that the value for the **nvoVoltFb** network variable is double that value:

1. Open the LonMaker drawing for the FTXL device.
2. Ensure that the FTXL device is properly configured, commissioned, and online.
3. Right-click the FTXL device on the LonMaker drawing and select **Browse** to open the LonMaker Browser window.
4. Within the LonMaker Browser window, select the row for the **nviVolt** network variable.
5. Enter a value for the network variable in the **Value** field at the top of the window. Click the **Set value** button to set the network variable's value.
6. Select the row for the **nvoVoltFb** network variable, and click the **Get value** button to see its current value. That value should be twice the **nviVolt** value.

## Running the Dynamic Interface Example

To verify that the application runs as expected, connect the FTXL Transceiver Board to a network management tool, such as the LonMaker Integration tool. From the tool, modify the value for the **nviVolt** network variable and confirm that the value for the **nvoVoltFb** network variable is double that value, as described for the simple example.

You can also change the type for the **nviVolt** or **nvoVoltFb** network variables, or add dynamic variables for the logical circuits described in *The Dynamic Interface Example Application* on page 202.

# Changing Network Variable Types

To change the type of the two network variables that are defined for the **VoltAmplifier** functional block:

1. Open the LonMaker drawing for the FTXL device.
2. Ensure that the FTXL device is properly configured, commissioned, and online.
3. Right-click the FTXL device on the LonMaker drawing and select **Browse** to open the LonMaker Browser window.
4. Select the row for either the **nviVolt** or **nvoVoltFb** network variable. Be sure *not* to select the row for the corresponding configuration property (displayed in green, with the value **SCPTnvType** in the Config Prop column).
5. Right-click the **nviVolt** or **nvoVoltFb** network variable and select **Change Type** to open the Select Network Variable Type dialog.

6. In the Select Network Variable Type dialog, expand **C:\LonWorks\Types\STANDARD.FMT** and select either **SNVT_volt** or **SNVT_volt_mil**. Click **OK** to change the type and close the Select Network Variable Type dialog.

   **Important**: You can only change the type for a network variable if it is not bound to the network. LonMaker implicitly binds network variables when you enable monitoring from the LonMaker drawing (from the drawing itself or from any connections to the functional blocks) or the LonMaker Browser window. Thus, you must disable monitoring for both the **nviVolt** and **nvoVoltFb** network variables.

To verify that the type changed successfully:

1. Within the LonMaker Browser window, select the row for the **nviVolt** network variable.
2. Right-click the **nviVolt** network variable and select **Properties** to open the Network Variable Properties dialog.
3. On the Description page of the Network Variable Properties dialog, the **Type name** field displays the current type for the network variable. The current type should be either **SNVT_volt** or **SNVT_volt_mil**.

The valid range for the value of the **nviVolt** and **nvoVoltFb** network variables depends on its current type:

- For **nviVolt** as **SNVT_volt**: ± 1.6 V
- For **nvoVoltFb** as **SNVT_volt**: ± 3.2 V
- For **nviVolt** as **SNVT_volt_mil**: -1638.4 mV to +1638.3 mV
- For **nviVolt** as **SNVT_volt_mil**: -3276.8 mV to +3276.6 mV

**Important**: Be sure to change the type to either **SNVT_volt** or **SNVT_volt_mil**, not to **SNVT_vol**, **SNVT_vol_mil**, or any other type. If you change the type to an invalid type, the dynamic interface example application rejects the change and disables the **NodeObject** and **VoltAmplifier** functional blocks. In this case, you must re-enable the functional blocks:

1. Change the type for either the **nviVolt** or **nvoVoltFb** network variable to a valid type, **SNVT_volt** or **SNVT_volt_mil**.
2. Within the LonMaker drawing, right-click the **NodeObject** functional block and select **Manage** to open the LonMaker Device Manager dialog.
3. From the Functional Blocks tab of the LonMaker Device Manager dialog, click **Enable** to re-enable the functional block. Click **Test** to verify that the output for the functional block displays "Diabled: 0", which signifies that the functional block is not disabled.

# Adding Dynamic Network Variables

To add a set of dynamic network variables to the FTXL device to demonstrate a logical circuit as described in *The Dynamic Interface Example Application* on page 202:

1. Open the LonMaker drawing for the FTXL device.
2. Ensure that the FTXL device is properly configured and commissioned.

3. Drag a **Functional Block** shape from the NodeBuilder Basic Shapes 3.0 pane of the Shapes window to the drawing. The Functional Block Wizard dialog opens.
4. In the Functional Block Wizard dialog:
    a. Select the FTXL device from the **Name** dropdown list box in the Device area.
    b. Select **Virtual Functional Block** from the **Name** dropdown list box in the Functional Block area.
    c. Select **Create all network variable shapes**.
    d. In the **Number of FBs to create** box, select 1.
    e. Give the new functional block a name in the **New FB name** field, or allow LonMaker to give a default name.
    f. Click **Finish** to add the virtual functional block to the drawing and close the Functional Block Wizard dialog.
5. Drag an **Output Network Variable** shape onto the virtual functional block. The Choose a Network Variable dialog opens.
6. In the Choose a Network Variable dialog:
    a. Click **Create NV** to open the Create Network Variable dialog.
    b. In the Create Network Variable dialog:
        i. Type **nvoAmpA** (you can specify any letter for the "A", but the name must include "nvoAmp") in the **New NV name** field.
        ii. Type **1** in the **How many?** field.
        iii. In the New NV Type area, select **Specify**.
        iv. Click **Select** to open the Select Network Variable Type dialog.
        v. In the the Select Network Variable Type dialog, expand **C:\LonWorks\Types\STANDARD.FMT** and select **SNVT_amp**. Click **OK** to close the the Select Network Variable Type dialog.

vi. The Create Network Variable dialog should look similar to **Figure 26**.

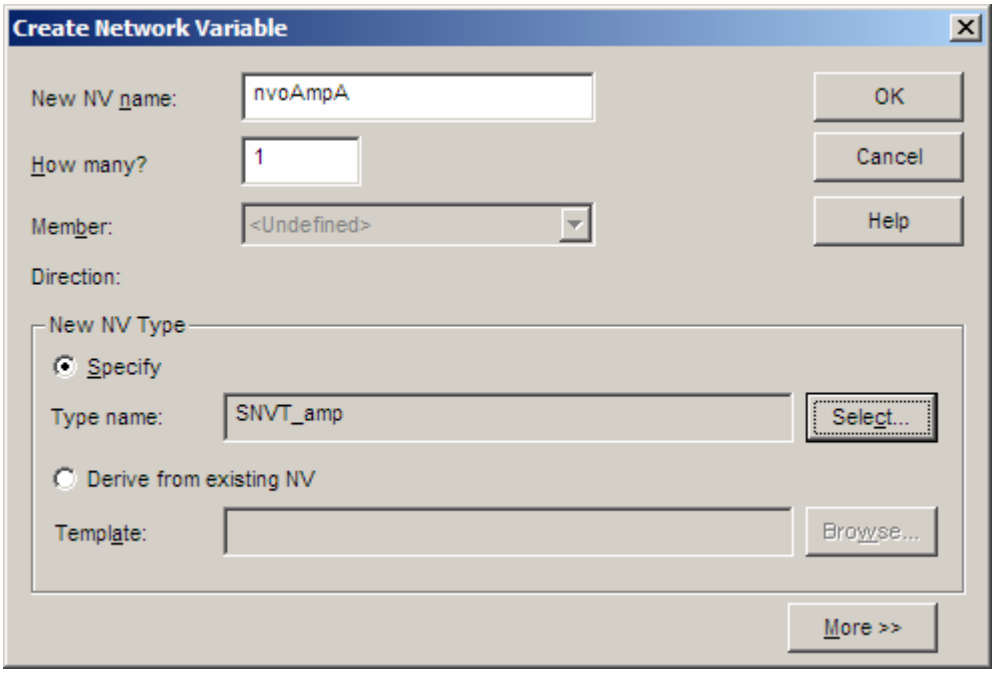



**Figure 26**. The LonMaker Create Network Variable Dialog for nvoAmpA

vii. Click **OK** to close the Create Network Variable dialog.

c. Click **OK** to add the nvoAmpA network variable to the virtual functional block and close the Choose a Network Variable dialog.

7. Drag an **Input Network Variable** shape onto the virtual functional block. The Choose a Network Variable dialog opens.

8. In the Choose a Network Variable dialog:

   a. Click **Create NV** to open the Create Network Variable dialog.

   b. In the Create Network Variable dialog:

      i. Type **nviAmpA01** (specify the same letter that you specified for the corresponding **nvoAmp** network variable; you can specify any characters for the "01", but the name must include "nviAmp") in the **New NV name** field.

      ii. Type **3** in the **How many?** field. You can specify any number, but the dynamic interface example application supports a maximum of 50 dynamic network variables.

      iii. In the New NV Type area, select **Specify**.

      iv. Click **Select** to open the Select Network Variable Type dialog.

      v. In the the Select Network Variable Type dialog, expand **C:\LonWorks\Types\STANDARD.FMT** and select **SNVT_amp**. Click **OK** to close the the Select Network Variable Type dialog.

vi. The Create Network Variable dialog should look similar to **Figure 27**.

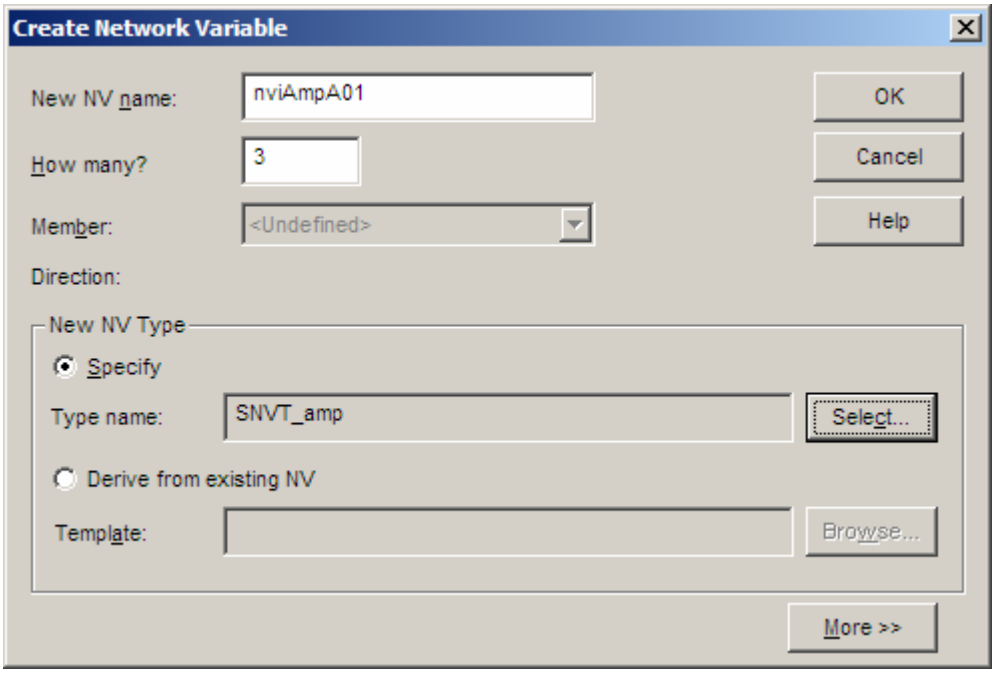



**Figure 27**. The LonMaker Create Network Variable Dialog for nviAmpA01

vii. Click **OK** to close the Create Network Variable dialog.

c. Click **OK** to add the three input network variables (named **nviAmpA01**, **nviAmpA02**, and **nviAmpA03**) to the virtual functional block and close the Choose a Network Variable dialog.

After you add the four dynamic network variables to the virtual functional block, it should look similar to **Figure 28**.

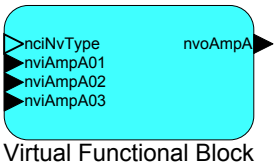


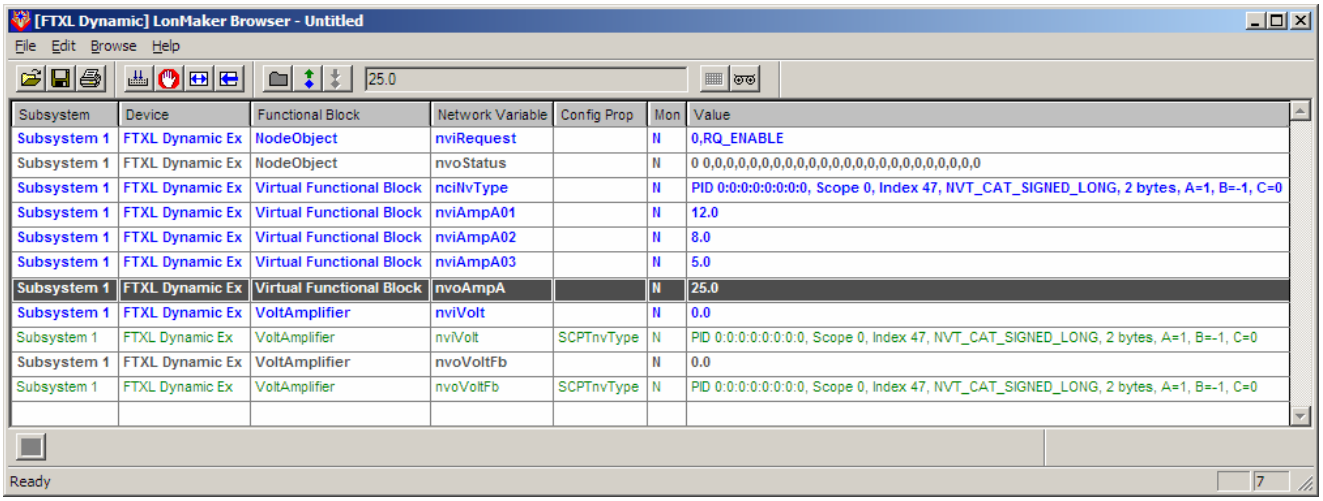

**Figure 28**. The Virtual Functional Block for the Dynamic Interface Example

To demonstrate that the four dynamic network variables act as a logical circuit, open the LonMaker Browser for the FTXL device and set the values for the three input network variables and observe the value of the output network variable, which should be the sum of the three inputs:

1. Right-click the FTXL device on the LonMaker drawing and select **Browse** to open the LonMaker Browser window.
2. Select the row for the **nviAmpA01** network variable.
3. Enter a value for the network variable in the **Value** field at the top of the window. Click the **Set value** button to set the network variable's value.
4. Select the row for the **nvoAmpA** network variable, and click the **Get value** button to see its current value.

5. Repeat steps 2 - 4 to set values for the **nviAmpA02** and **nviAmp03** network variables and observe the change to the **nvoAmpA** network variable.

After you add update the values for the three input dynamic network variables, the LonMaker Browser window should look similar to **Figure 29**.




| Subsystem | Device | Functional Block | Network Variable | Config Prop | Mon | Value |
|---|---|---|---|---|---|---|
| Subsystem 1 | FTXL Dynamic Ex | NodeObject | nviRequest | | N | 0,RQ_ENABLE |
| Subsystem 1 | FTXL Dynamic Ex | NodeObject | nvoStatus | | N | 0 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 |
| Subsystem 1 | FTXL Dynamic Ex | Virtual Functional Block | nciNvType | | N | PID 0:0:0:0:0:0:0:0, Scope 0, Index 47, NVT_CAT_SIGNED_LONG, 2 bytes, A=1, B=-1, C=0 |
| Subsystem 1 | FTXL Dynamic Ex | Virtual Functional Block | nviAmpA01 | | N | 12.0 |
| Subsystem 1 | FTXL Dynamic Ex | Virtual Functional Block | nviAmpA02 | | N | 8.0 |
| Subsystem 1 | FTXL Dynamic Ex | Virtual Functional Block | nviAmpA03 | | N | 5.0 |
| Subsystem 1 | FTXL Dynamic Ex | Virtual Functional Block | nvoAmpA | | N | 25.0 |
| Subsystem 1 | FTXL Dynamic Ex | VoltAmplifier | nviVolt | | N | 0.0 |
| Subsystem 1 | FTXL Dynamic Ex | VoltAmplifier | nviVolt | SCPTnvType | N | PID 0:0:0:0:0:0:0:0, Scope 0, Index 47, NVT_CAT_SIGNED_LONG, 2 bytes, A=1, B=-1, C=0 |
| Subsystem 1 | FTXL Dynamic Ex | VoltAmplifier | nvoVoltFb | | N | 0.0 |
| Subsystem 1 | FTXL Dynamic Ex | VoltAmplifier | nvoVoltFb | SCPTnvType | N | PID 0:0:0:0:0:0:0:0, Scope 0, Index 47, NVT_CAT_SIGNED_LONG, 2 bytes, A=1, B=-1, C=0 |

**Figure 29**. The LonMaker Browser Window for the Dynamic Interface Example

If you enable monitoring for the LonMaker drawing, and add connector shapes to the virtual functional block, it should look similar to **Figure 30**.

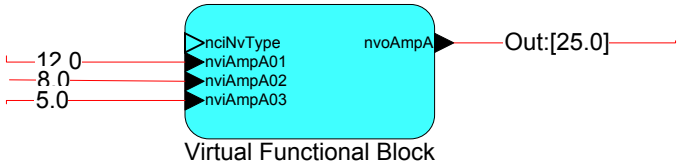



**Figure 30**. The Virtual Functional Block for the Dynamic Interface Example

You can add more dynamic network variables to the virtual functional block to represent other logical circuits, as shown in **Figure 31**.

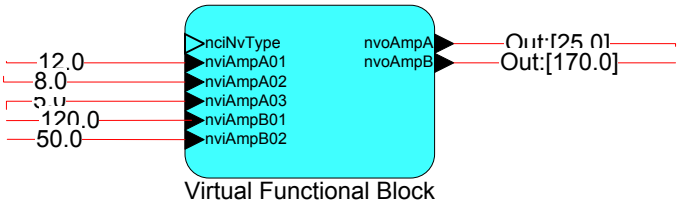



**Figure 31**. The Virtual Functional Block for the Dynamic Interface Example

# The Micrium Software License

Although the FTXL example applications use the Micrum µC/OS-II operating system that is included with the Altera development environment for the DBC2C20 development board, you must acquire a valid license for the operating system if you plan to use it for a production FTXL device.

Each time you build either of the FTXL example applications, you see the following reminder of the Micrium software license:

```
============== Software License Reminder ===============

uC/OS-II is provided in source form for FREE evaluation,
for educational use, or for peaceful research. If you
plan on using uC/OS-II in a commercial product you need
to contact Micrium to properly license its use in your
product. Micrium provides ALL the source code on the
Altera distribution for your convenience and to help you
experience uC/OS-II. The fact that the source is provided
does NOT mean that you can use it without paying a
licensing fee. Please help us continue to provide the
Embedded community with the finest software available.
Your honesty is greatly appreciated.

Please contact:

M I C R I U M
949 Crestview Circle
Weston, FL 33327-1848
U.S.A.

Phone : +1 954 217 2036
FAX : +1 954 217 2037
WEB : www.micrium.com
E-mail: Sales@Micrium.com
```

# G

# 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 *NodeBuilder 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<br>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<br>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<br>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<br>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<br><br>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<br><br>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.<br><br>(This error is similar to LID#5, but refers to a different internal component recognizing the error.) |
| 7 | A device interface file appears malformed<br><br>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<br><br>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<br><br>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<br>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. |
| 11 | Error code not in use. |
| 12 | Error code not in use. |
| 13 | Error code not in use. |
| 14 | Error code not in use. |
| 15 | Error code not in use. |
| 16 | Error code not in use. |
| 17 | Error code not in use. |
| 18 | An error occurred when composing the application XIF file: the data merge target is ill-chosen (must be the BIF file)<br>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<br>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<br>Refer to the error message for details about the failure cause. The error message contains details such as "disk full," or "file access denied". |

| LID# | Description |
|---|---|
| 21 | The xif32bin.exe utility returned an error, indicating failure when converting XIF to XFB<br><br>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)<br><br>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. |
| 23 | An error occurred when reading a type info file (.NCT)<br><br>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<br><br>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)<br><br>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. |
| 26 | Error code not in use. |
| 27 | Unexpected file I/O error when reading a file<br><br>Refer to the error message for details of the failure cause. |

| LID# | Description |
|---|---|
| 28 | Unexpected error (not a file I/O error) when reading a file<br>Refer to the error message for details of the failure cause. |
| 29 | Unexpected file I/O error when writing a file<br>Refer to the error message for details of the failure cause. |
| 30 | Unexpected error (not a file I/O error) when writing a file<br>Refer to the error message for details of the failure cause. The error message contains details such as “disk full” or “file access denied”. |
| 31 | A type definition cannot be generated: the type is referenced but not defined<br>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<br>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<br>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<br>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. |

| LID# | Description |
|---|---|
| 35 | Configuration parameters are in use, but no template file has been found<br><br>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.<br><br>If the problem persists, contact Echelon technical support, submitting all files produced by the LonTalk Interface Developer utility when running in **Trace** verbosity level. |
| 36 | The program ID found in the XIF file seems malformed and cannot be used to produce the niAppinit data<br><br>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. |
| 37 | Error code not in use. |
| 38 | Error code not in use. |
| 39 | Error code not in use. |
| 40 | Error code not in use. |
| 41 | Error code not in use. |
| 42 | A type definition cannot be generated -- the type definition has more elements than expected<br><br>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. |
| 43 | Error code not in use. |
| 44 | Error code not in use. |
| 45 | Error code not in use. |

| LID# | Description |
|---|---|
| 46 | One or more configuration parameters implemented within a file are present, FTP or DMF must be implemented<br>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 |
| 48 | Error code not in use. |
| 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]*<br>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 |
| 55 | The specified framework is not yet supported<br>This is an internal error. |
| 56 | Error code not in use. |
| 57 | Required source file missing |
| 58 | Error code not in use. |
| 59 | Too many network variables. The sum of static and dynamic variables cannot exceed 4096. |
| 60 | Insufficient number of addresses<br>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 |

| LID# | Description |
|---|---|
| 62 | Insufficient buffer space<br>The message includes the total number of bytes available for transceiver buffers and how many additional bytes your selected configuration requires. |

# Warning codes

(warning codes start at 4000)

| LID# | Description |
|---|---|
| 4001 | An XIF file contains more fields than expected<br>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 FTXL 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<br>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. |
| 4003 | Warning code not in use. |
| 4004 | Warning code not in use. |
| 4005 | Warning code not in use. |
| 4006 | A file cannot be copied<br>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. |
| 4007 | Warning code not in use. |
| 4008 | Warning code not in use. |

| LID# | Description |
|---|---|
| 4009 | Warning code not in use. |
| 4010 | Warning code not in use. |
| 4011 | The .NCT file references a built-in type with no host equivalent known to LonTalk Interface Developer utility<br><br>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. Carefully 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 FTXL device. |
| 4012 | Warning code not in use. |
| 4013 | Warning code not in use. |
| 4014 | Explicit addressing specified but not required<br><br>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<br><br>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<br><br>The implementation of the file transfer protocol is suspect, as the FTP-related network variables are present but no message tag has been declared. |
| 4017 | Files cannot be made writable<br><br>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. |
| 4018 | Warning code not in use. |

| LID# | Description |
|---|---|
| 4019 | Warning code not in use. |
| 4020 | Warning code not in use. |
| 4021 | Warning code not in use. |
| 4022 | Warning code not in use. |
| 4023 | Insufficient addresses are implemented for the specified number of network variables<br>For more robust device behavior, increase the number of addresses. |
| 4024 | Warning code not in use. |
| 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

(hint codes start at 8000)

| LID# | Description |
|---|---|
| 8001 | Your device supports the file transfer protocol, but no configuration property files are available |
| 8002 | Hint code not in use. |
| 8003 | Hint code not in use. |
| 8004 | Hint code not in use. |
| 8005 | Your transceiver buffer configuration leaves a number of bytes unused |

# H

# Glossary

This appendix defines many of the common terms used for FTXL device development.

## D

**downlink**

Link-layer data transfer from the host to the FTXL Transceiver.

## E

**Eclipse**

An open-source software framework written primarily in Java. In its default form, it is an Integrated Development Environment (IDE) for Java developers, consisting of the Java Development Tools (JDT) and the Eclipse Compiler for Java (ECJ). The Nios II IDE is an Eclipse-based development environment.

**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

**FTXL application**

An application for a LONWORKS device based on the LonTalk API and FTXL Transceiver.

**FTXL Developer's Kit**

Software required to develop high-performance LonTalk applications for an Altera Nios II host processor with an FTXL Transceiver. The kit includes software tools, examples, documentation, plus the LonTalk API and the FTXL LonTalk protocol stack.

**FTXL device**

A LONWORKS device based on the LonTalk API and an FTXL Transceiver.

**FTXL Firmware**

The firmware embedded within an FTXL Transceiver.

**FTXL 3190 Free Topology Smart Transceiver**

A chip that is used as a transceiver to attach an FTXL host processor to a LONWORKS network; the FTXL Transceiver runs the FTXL Firmware and implements layers 1 and 2 of the ANSI/CEA-709.1 (EN14908-1) Control Network Protocol.

**FTXL host processor**

An Altera FPGA (or similar device) with a Nios II processor that is integrated with the LonTalk API and an FTXL Transceiver to create a LONWORKS device.

**FTXL link layer**

The physical connection and protocol used to attach an FTXL host processor to an FTXL Transceiver; the hardware interface is an 11-pin parallel interface plus interrupt.

**FTXL LonTalk protocol stack**

A high-performance implementation of layers 3 through 6 of the ANSI/CEA-709.1 (EN14908-1) Control Network Protocol that runs on an Altera Nios II processor.

**FTXL Transceiver**

Short name for the FTXL 3190 Free Topology Smart Transceiver.

## *H*

**host processor**

A microcontroller, microprocessor, or FPGA with an embedded processor that is attached to an FTXL Transceiver or ShortStack Micro Server and runs a LonTalk application.

## *L*

**link layer**

A protocol and interface definition for communication between a host processor and either an FTXL Transceiver or ShortStack Micro Server; see FTXL link layer.

**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 FTXL devices and a compact version for ShortStack devices.

**LonTalk application**

An application for a LONWORKS device that communicates with other devices using the ANSI/CEA-709.1 (EN14908-1) 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 Interface Developer**

A utility that generates an application framework for a LonTalk application; the LonTalk Interface Developer is part of the LonTalk Platform and is

included with both the FTXL 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: the LonTalk Platform for FTXL Transceivers, and the LonTalk Platform for ShortStack Micro Servers.

**LonTalk Platform for FTXL Transceivers**

Development tools, APIs, firmware, and chips for developing LONWORKS devices that use the LonTalk API and a FTXL Transceiver; included with the FTXL Developer's Kit.

## *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.

**Nios II IDE**

An integrated development environment based on the popular Eclipse IDE framework and the Eclipse C/C++ development toolkit (CDT) plug-ins. The Nios II IDE runs other tools behind the scenes, shields you from the details of low-level tools, and presents a unified development environment.

## *P*

**perspective**

An Eclipse term for a collection of windows together in the Eclipse framework. Used within the Altera Nios II Embedded Design Suite integrated development environment (IDE).

## *U*

**uplink**

Link-layer data transfer from the FTXL Transceiver to the host.

# Index

## F



## G



## H



## I

## K



## L



## M



## N

## O



## P



## Q



## R



## S



## T



## U



## W



## X

www.echelon.com