ECHELON®

2.0

ShortStack FX User's Guide

078-0365-01B

Echelon Corporation
www.echelon.com

Welcome

Echelon's ShortStack® Micro Server enables any product that contains a microprocessor or microcontroller to quickly and inexpensively become a networked, Internet-accessible device. The ShortStack Micro Server provides a simple way to add LONWORKS® networking to new or existing smart devices. The ShortStack Micro Server is easy to use due to a simple host API, a simple driver, a simple hardware interface, a small host memory footprint, and comprehensive tool support.

This document describes how to develop an application for a LONWORKS device using Echelon's ShortStack FX Micro Server. It describes the architecture of a ShortStack device and how to develop a ShortStack device. Development of a ShortStack device includes interfacing the ShortStack Micro Server with your microprocessor, creating your ShortStack serial driver, creating a Neuron® C model file, running the LonTalk® Interface Developer utility, and using the LonTalk Compact API functions to program your ShortStack application.

Audience

This document assumes that the reader has a good understanding of the LONWORKS platform and microprocessor or microcontroller programming.

What's New for ShortStack FX

The ShortStack FX Developer's Kit is part of the LONWORKS 2.0 product family.

The ShortStack FX Developer's Kit includes all of the features and functions of ShortStack 2.1, and adds new features and functions.

New Hardware Support

The ShortStack FX Developer's Kit provides standard Micro Servers for the FT 5000 Smart Transceiver and the PL 3170 Smart Transceiver. You can also build a custom Micro Server for the Neuron 5000 Processor.

ISI Controlled Enrollment

A custom ISI Micro Server can provide support for controlled enrollment. With controlled enrollment support, your ISI network can include a connection controller to manage ISI enrollment.

LonTalk Compact API

The ShortStack application programming interface (API) includes new functions and callback handler functions, including a function to determine the version number of the Micro Server application and Micro Server core library, and an echo function to test communications with the link layer.

LonTalk Compact API Compatibility

The LonTalk Compact API for ShortStack FX is essentially the same as the LonTalk Compact API for ShortStack 2.1. Thus, a ShortStack 2.1 application requires no changes to the application or the link-layer driver; you need only re-run the LonTalk Interface Developer utility and recompile the application to run with a ShortStack FX Micro Server.

Although new API functionality, such as the version or echo functions, is not available to unmodified existing applications, you can add these functions to existing applications or create new applications to include this functionality. Adding such new functionality to existing applications is optional.

Important: The LonTalk Compact API for ShortStack FX is substantially different than the ShortStack 2 API; see Chapter 13, *Converting a ShortStack 2 Application to a ShortStack FX Application*, on page 257, for information about how to convert a ShortStack 2 application to use the ShortStack FX LonTalk Compact API.

ShortStack User's Guide

The appendix, *Neuron C Syntax for the Model File*, has been deleted. All of the Neuron C syntax is described in the *Neuron C Reference Guide*. Chapter 8, *Creating a Model File*, on page 115, describes how to use the Neuron C programming language to create a mode file. See the *Neuron C Reference Guide* for detailed Neuron C language syntax.

The appendix, *LonTalk Interface Developer Utility Error and Warning Messages*, has been deleted. The error codes for the LonTalk Interface Developer utility have been moved to the *Neuron Tools Errors Guide* (078-0402-01B) and have been removed from the *ShortStack FX User's Guide*.

What's New for ShortStack 2.1

ShortStack 2.1 included many new features and functions compared with ShortStack 2. This section describes some of the major new features and functions of ShortStack 2.1.

LonTalk API

The ShortStack application programming interface (API) that was used by ShortStack 2 has been replaced with a new LonTalk API. This API is 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 with support for up to 4096 network variables and a compact version with support for up to 254 network variables. The compact version is used by ShortStack 2.1, and the full version is used by the LonTalk Platform for FTXL Transceivers. Together, they provide a simple migration path and the opportunity for shared application code between ShortStack and FTXL applications. Chapter 13, *Converting a ShortStack 2 Application to a ShortStack FX Application*, on page 257, describes how to convert a ShortStack 2 application to use the ShortStack 2.1 LonTalk Compact API.

LonTalk Interface Developer

The ShortStack Wizard has been enhanced, and is now known as the LonTalk Interface Developer utility. The new tool includes several usability improvements and a documented command-line interface for use with some development platforms (such as the Eclipse IDE), or for automated, script-driven, build processes. This utility is shared with the LonTalk Platform for FTXL Transceivers, thus providing a compatible set of tools for developing ShortStack and FTXL applications.

Self-Installation Functions

Two new APIs have been added to support self-installation. One is a low-level API that supports reading and writing the Smart Transceiver's network configuration. The other is an optional high-level API that supports the Interoperable Self-Installation (ISI) protocol. These APIs build on the existing ShortStack LonTalk Compact host API and therefore have minimal impact on existing portions of the host API.

Support for 254 Network Variables and 127 Aliases

For FT 3120, FT 3150, PL 3150, or PL 3170 devices, ShortStack 2.1 supports up to 254 network variables and up to 127 aliases, rather than the 62 maximum for each that ShortStack 2 supported. There are minor changes to the host API for this feature. The primary changes are a modification to the data structure used to pass the network variable index from the host to the Micro Server and back, a change to the initialization sequence, and a change to the link-layer protocol. The device interface data is also updated to accommodate more than 62 network variables.

This feature requires the use of a Micro Server with Neuron firmware version 16 (or later). The PL 3120 Smart Transceiver uses Neuron firmware version 14, and is thus limited to 62 network variables and 62 aliases.

Changeable-Type NV Support

ShortStack 2.1 supports changeable network variable types as described in the LONMARK® Application-layer Guidelines. This addition requires minor changes to the host API. The LonTalk Interface Developer utility provides a new **LonNvDescription** type definition in the **ShortStackDev.h** file, and generates the network variable table accordingly in the **ShortStackDev.c** file. The new definition is incompatible with ShortStack 2 applications.

Direct Memory Files

ShortStack 2.1 simplifies implementation of configuration properties within configuration files by allowing a network management tool to access configuration property files without having to implement a LONWORKS file transfer protocol (LW-FTP) server on the device. ShortStack 2.1 allows a ShortStack application to implement configuration properties within configuration files, and exposes an interface that enables a network management tool to use standard LonTalk memory read and write network management messages to access the configuration properties. To support this access, a window of the Smart Transceiver's memory space is defined so that whenever a Smart Transceiver receives a memory read or write network management command that uses addresses within this window, the Micro Server routes it to the application. This approach eliminates the need to implement the LONWORKS file transfer protocol on most ShortStack devices, but requires some new code to handle the read and write requests from the Micro Server. The LonTalk Interface Developer utility generates this code automatically.

This feature requires the use of a Micro Server with Neuron firmware version 16 (or later). The PL 3120 Smart Transceiver uses Neuron firmware version 14, and cannot use the direct memory files access method.

Host SI Data Storage

In ShortStack 2 applications, the device's self-identification (SI) and self-documentation (SD) data was transferred to the Micro Server during

initialization, and was limited to the size of the Micro Server's related buffer. ShortStack 2.1 applications no longer transfer this data to the Micro Server, thus allowing for simplified initialization, and fewer restrictions to the size of this data. Code has been added to the ShortStack LonTalk Compact API to handle the SI data read and write requests forwarded by the ShortStack Micro Server.

Uplink Reset Message

The reset message sent by the ShortStack Micro Server has been extended. The new message reports link-layer protocol version 3 instead of 2, reports whether the Micro Server is configured, provides a unique key for the specific version of the Micro Server, reports the existence and state of an IO9 input, reports the last reset cause, reports the last error logged, provides information about the Micro Server's capacity, and includes a flag to indicate whether the Micro Server is initialized.

Configuration Property Arrays

ShortStack 2.1 supports implementing configuration property arrays. Configuration property arrays are multi-dimensional configuration properties that, as a unit, apply to a network variable or a functional block (or to multiple network variables or functional blocks).

Support for Non-Volatile Data

Configuration property values, and values of network variables declared with the **eeprom** modifier, must persist after resetting or power-cycling the device. ShortStack 2.1 provides an improved API and framework to assist with the implementation of persistent configuration property and network variable values.

Configuration Property and Network Variable Initializers

Configuration properties, and sometimes network variables, require well-defined initial values. ShortStack 2.1 provides an improved application framework, which includes fully and correctly initialized configuration properties and network variables, thereby further simplifying the development of interoperable ShortStack devices.

Custom Micro Servers

Users of the NodeBuilder® Development Tool or the Mini EVK Evaluation Kit can create their own Micro Server to target a Smart Transceiver with a custom hardware configuration.

Improved Diagnostics

ShortStack 2.1 simplifies debugging and diagnosing a new Micro Server, especially while in quiet mode. This includes a combination of API, driver, Micro Server, and documentation changes.

Example Ports

ShortStack 2.1 host API ports are provided as separate download packages, rather than being included with the ShortStack Developer's Kit. This change allows examples to be delivered independently of the Developer's Kit. The ShortStack Developer's Kit consists of the ShortStack firmware, ShortStack LonTalk Compact API, LonTalk Interface Developer utility, and documentation. Each example port consists of the port's example serial driver, the ported host API, one or more example applications, and documentation for the port.

Related Documentation

In addition to this manual, the ShortStack FX Developer's Kit includes the following manuals:

- *Neuron C Programmer's Guide* (078-0002-02H). 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* (078-0140-02F). This manual provides reference information for writing programs that use the Neuron C language.
- *Neuron Tools Errors Guide* (078-0402-01B). This manual describes error codes issued by the Neuron C compiler and related development tools.

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

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

In addition to the ShortStack Developer's Kit, Echelon provides example ports for selected host processors. These example ports include example implementations of the serial driver, API callback handler routines, and one or more sample applications. You can download these example ports from the Echelon ShortStack Web site (www.echelon.com/shortstack). The following manual describes the example port that is currently available:

- *ShortStack FX ARM7 Example Port User's Guide* (078-0366-01B). This manual describes the ShortStack FX ARM7 Example Port for an ARM7-family microprocessor, the Atmel® ARM® AT91SAM7S64. The manual also describes working with the example applications, which were built with the IAR Embedded Workbench®.

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

- *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® and FT 3150® Smart Transceivers.
- *Introduction to the LONWORKS Platform* (078-0183-01B). This manual provides an introduction to the ISO/IEC 14908 (ANSI/CEA-709.1 and EN14908) Control Network Protocol, and provides a high-level introduction to LONWORKS networks and the tools and components that are used for developing, installing, operating, and maintaining them.
- *ISI Programmer's Guide* (078-0299-01F). Describes how you can use the Interoperable Self-Installation (ISI) protocol to create networks of control devices that interoperate, without requiring the use of an installation tool. Also describes how to use Echelon's ISI Library to develop devices that can be used in both self-installed as well as managed networks.

- *ISI Protocol Specification* (078-0300-01F). Describes the Interoperable Self-Installation (ISI) protocol, which is a protocol used to create networks of control devices without requiring the use of an installation tool.

- *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.

- *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.

- *NodeBuilder® FX User's Guide* (078-0405-01A). This manual describes how to develop a LONWORKS device using the NodeBuilder tool.

  You can use the NodeBuilder FX Development Tool to create a model file for a ShortStack application. See Chapter 8, *Creating a Model File*, on page 115, for more information about model files. You can also use the NodeBuilder FX Development Tool to create a custom ShortStack Micro Server. See Chapter 12, *Custom Micro Servers*, on page 241, for more information about custom Micro Servers. Most ShortStack developers will not need to create a custom ShortStack Micro Server.

- *Mini FX User's Guide* (078-0398-01A). This manual describes how to use the Mini FX Evaluation Kit. You can use the Mini kit to develop a prototype or production control system that requires networking, or to evaluate the development of applications for such control networks using the LONWORKS platform.

  You can also use the Mini FX Evaluation Kit to create a custom ShortStack Micro Server. See Chapter 12, *Custom Micro Servers*, on page 241, for more information about custom Micro Servers. Most ShortStack developers will not need to create a custom ShortStack Micro Server.

- *PL 3120 / PL 3150 / PL 3170 Power Line Smart Transceiver Data Book* (005-0193-01A). This manual provides detailed technical specifications on the electrical interfaces, mechanical interfaces, and operating environment characteristics for the PL 3120, PL 3150, and PL 3170™ Smart Transceivers.

- *Series 5000 Chip Data Book* (005-0199-01A). This manual provides detailed specifications on the electrical interfaces, mechanical interfaces, and operating environment characteristics for the FT 5000 Smart Transceiver and Neuron 5000 Processor.

All of the ShortStack 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.

As you create your serial driver for communications between your host processor and the ShortStack Micro Server, you will need to be familiar with either the SCI

or SPI interface standard. You will find having an appropriate reference for the interface helpful. Likewise, you should have documentation for your host processor and development environment available.

Table of Contents

1

Introduction to ShortStack

This chapter introduces the LonTalk Platform for ShortStack Micro Servers. It describes the architecture of a ShortStack device, the requirements and restrictions of a ShortStack Micro Server, and the ShortStack 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 ISO/IEC 14908 (ANSI/CEA 709.1-B and 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 1**. The implementation of these layers in a LONWORKS device provides standardized interconnectivity for devices within a LONWORKS network.

Table 1. 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 ISO/IEC 14908 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® SmartServers, and the FTXL LonTalk protocol stack. This document refers to

the ISO/IEC 14908 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.

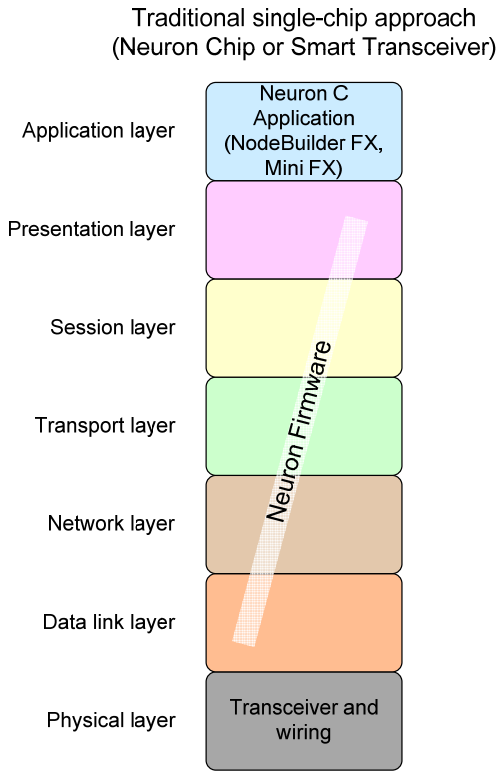

Figure 1. A Single-Chip LONWORKS Device

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 with either the NodeBuilder® FX Development Tool or the Mini FX Evaluation Kit.

A LONWORKS Device with Two Processor Chips

Some LONWORKS devices run applications that require more memory, I/O, 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 kit application, to develop the application.

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

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

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 Transceiver with firmware that implements the data link layer (layer 2) of the LonTalk protocol, as shown in **Figure 2** on page 6. 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 typically 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. 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.

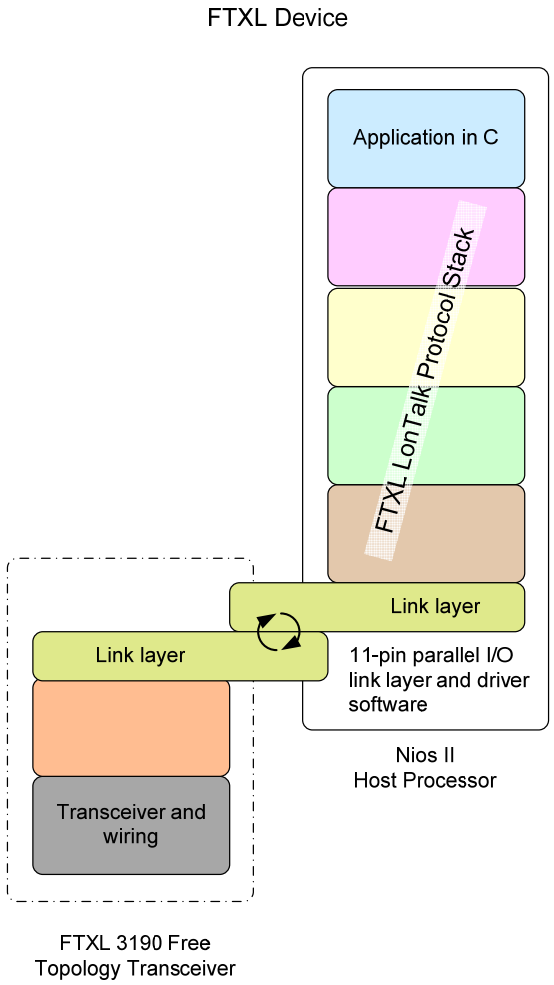

Figure 2. An FTXL Device

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 3** on page 7. 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 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 a specific host processor. Currently, an example driver is available for an Atmel ARM7 microprocessor.

For ShortStack device development, you use the C programming language[1]. As with FTXL development, 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.

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

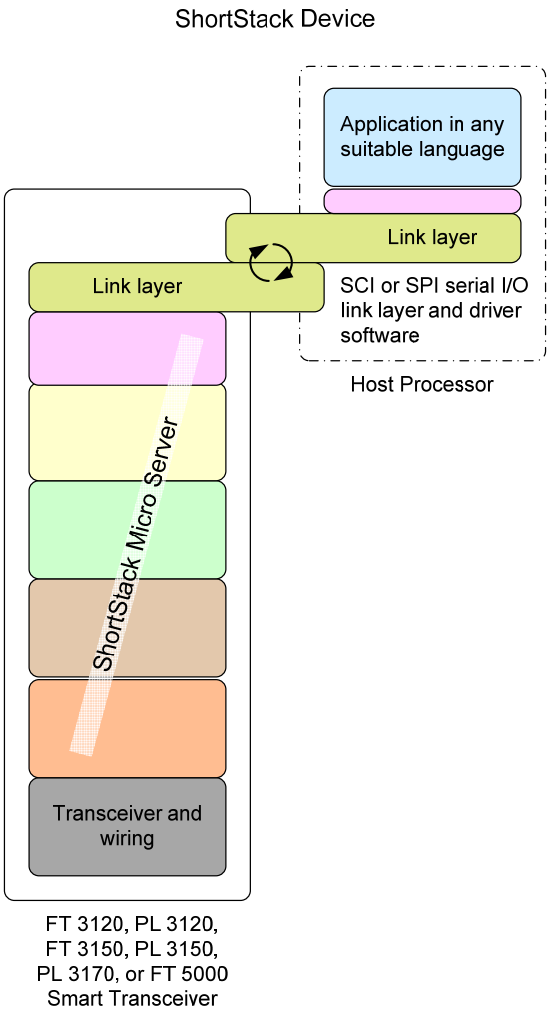

Figure 3. A ShortStack Device

[1] For ShortStack device development, you could alternatively use any standard programming language supported by the host processor if you also port the LonTalk Compact API and the application framework generated by the LonTalk Interface Developer utility to that language.

Comparing Neuron Hosted, FTXL, and ShortStack Devices

Table 2 compares some of the key characteristics of the Neuron hosted and host-based solutions for LONWORKS devices.

Table 2. Comparing Neuron Hosted and Host-Based Solutions for LONWORKS Devices

Characteristic	Neuron Hosted Solution	FTXL Solution	ShortStack Solution
Maximum number of network variables	254 or 62 [1]	4096	254, 120, or 62 [2]
Maximum number of aliases	127 or 62 [1]	8192	127, 75, or 62 [2]
Maximum number of addresses	15	4096	15
Maximum number of dynamic network variables	0	4096	0
Maximum number of receive transaction records	16	200	16
Maximum number of transmit transaction records	2	2500	2
Support for the LonTalk Extended Command Set	No	Yes [3]	No
File access methods supported	LW-FTP [4], DMF[5,6]	LW-FTP [4], DMF [5]	LW-FTP [4], DMF[5,6]
Link-layer type	N/A	11-line parallel I/O [7]	4- or 5-line SCI or 6- or 7-line SPI
Typical host API runtime footprint	N/A	540 KB (includes LonTalk protocol stack, but does not include the application or operating system)	5-6 KB code with 1 KB RAM (includes serial driver, but does not include optional API or ISI API)

Host processor type	N/A	Altera Nios II embedded processor	Most 8-, 16-, 32-, or 64-bit microprocessors or microcontrollers
Application development language	Neuron C	ANSI C or C++ for the Nios II processor	Most standard programming languages (typically ANSI C)

Notes:

1. Neuron Chips and Smart Transceivers with firmware version 16 and later support up to 254 network variables and up to 127 aliases. Neuron Chips and Smart Transceivers with firmware version 15 and earlier are limited to 62 network variables and 62 aliases. Although these limits are architectural maxima, they are subject to available resources (EEPROM, RAM).
2. ShortStack Micro Servers running on FT 3120, FT 3150, FT 5000, PL 3150, or PL 3170 Smart Transceivers support up to 254 network variables and up to 127 aliases. However, ISI-enabled ShortStack Micro Servers running on PL 3170 Smart Transceivers only support up to 120 network variables and up to 75 aliases. ShortStack Micro Servers running on PL 3120 Smart Transceivers support up to 62 network variables and up to 62 aliases.
3. See the *Control Network Protocol Specification*, ISO/IEC 14908, for more information about the extended command set (ECS) network management commands.
4. An implementation of the LONWORKS file transfer protocol (LW-FTP) is not provided with the product.
5. For more information about the direct memory file (DMF) access method, see *Using Direct Memory Files* on page 189.
6. Neuron firmware version 16 or later is required to support direct memory file (DMF) access method for either Neuron hosted or ShortStack devices. The PL 3120 Smart Transceiver uses Neuron firmware version 14, and cannot use the direct memory files access method; consider using the PL 3170 Smart Transceiver. Also, older FT 3120 Smart Transceivers use version 13 firmware; consider using a newer (RoHS-compliant) FT 3120 Smart Transceiver, which uses version 16 firmware.
7. 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 an available example for an Atmel ARM7 host processor), and supports both the TP/FT-10 and PL-20 channels.

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 ShortStack

The host processor for a ShortStack device can be an 8-, 16-, 32-, or 64-bit microprocessor, microcontroller, or embedded processor implemented in an FPGA device. The ShortStack LonTalk Compact API and serial driver typically require about 6 KB of program memory on the host processor (approximately 2 KB for the API and 3 to 4 KB for the serial driver) and less than 1 KB of RAM. There is no requirement for additional non-volatile memory unless you choose to implement non-volatile interfaces in your application.

The ShortStack FX firmware requires a Smart Transceiver with a minimum of 4 KB of application memory and 2 KB of RAM. The ShortStack Developer's Kit includes a variety of standard Micro Server images, which support FT 3120, FT 3150, FT 5000, PL 3120, PL 3150, and PL 3170 Smart Transceivers in various configurations. You can create a custom Micro Server to support other hardware configurations for these Smart Transceivers or to provide support for the Neuron 5000 Processor. ShortStack does not support the FTXL 3190 Free Topology Transceiver.

The interface between your host processor and the ShortStack Micro Server can be the asynchronous Serial Communications Interface (SCI) or the synchronous Serial Peripheral Interface (SPI). The speed of the interface depends both on the type of serial interface and the clock speed of the ShortStack Micro Server:

- The highest bit rate for the SCI interface is approximately 1.2 Mbps for a ShortStack Micro Server running on an FT 5000 Smart Transceiver with an 80 MHz system clock.
- The highest bit rates for the SPI interface are approximately 906 kbps uplink and 690 kbps downlink for a ShortStack Micro Server running on an FT 5000 Smart Transceiver with an 80 MHz system clock.

The interface rate scales with the ShortStack Micro Server system clock. See *Setting the SCI Bit Rate* on page 73 and *Setting the SPI Bit Rate* on page 77.

Note that some Micro Servers might be feature-restricted. For example, the ISI-enabled standard Micro Server for the PL 3170 Smart Transceiver supports only the SCI interface at 38400 bps.

The ShortStack Micro Server can support up to 254 network variables in your ShortStack application. You can implement configuration properties as configuration network variables or in configuration files. To access the configuration files, you can implement the LONWORKS file transfer protocol (LW-FTP), or, when possible, have the network management tool use standard memory read and write messages for minimum overhead on your device, using the direct memory file (DMF) access method. However, because DMF supports only a finite-sized memory window, you must implement LW-FTP if the total size of all files and the directory exceeds the window's size. In addition, the Micro Server Neuron firmware must be version 16 or later to use the direct memory file access method.

Development Tools for ShortStack

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

- The ShortStack LonTalk Compact API
- ShortStack firmware for creating a PL-20 power line or TP/FT-10 free topology twisted pair ShortStack Micro Server using a Smart Transceiver (you can also create a custom Micro Server for the Neuron 5000 Processor)
- The LonTalk Interface Developer utility for defining the interface for your ShortStack device and generating the application framework

In addition to the ShortStack Developer's Kit, you can download example ports for selected host processors. These example ports include example implementations of the serial driver, API callback handler routines, and one or more sample applications. Currently, an example driver is available for an Atmel ARM7 microprocessor.

You can create a ShortStack device that installs itself using the Interoperable Self-Installation (ISI) protocol, or you can create a device that is installed with a network management tool. You can also create a device that supports both installation methods, that is, you can create a device that installs itself in self-installed networks, or is installed by a network management tool in a managed network.

For installation into a managed network, you can use the LonMaker Integration Tool, or another tool that can install and monitor LONWORKS devices. See the *LonMaker User's Guide* for more information about the LonMaker tool. However, if your ShortStack device supports the Interoperable Self-Installation (ISI) protocol, you might not need a network management tool.

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

The ShortStack Developer's Kit includes pre-built Micro Server images for a variety of hardware and buffer configurations. You can use the NodeBuilder Development Tool to create a custom Micro Server image to support different hardware or buffer configurations. Most standard hardware is compatible with one of the standard Micro Server images that are supplied with ShortStack FX, and do not require either the NodeBuilder Development Tool or the Mini kit.

For diagnosing and troubleshooting, the LonScanner™ protocol analyzer is highly recommended for most LONWORKS developers. The LonScanner protocol analyzer collects and displays low-level protocol packets, and often provides important diagnostics.

Selecting a Host Processor

Although ShortStack supports almost any microprocessor for the host processor, there are certain requirements and considerations for a ShortStack host processor that apply if you are choosing a processor or development tool for a new ShortStack device, or if you are assessing the applicability of existing hardware or of previously acquired development tools. See Chapter 4, *Selecting the Host Processor*, on page 61, for a description of these requirements and considerations.

ShortStack Architecture

A ShortStack device consists of the following components:

1. The ShortStack Micro Server running the ShortStack firmware
2. An SCI or SPI serial driver for the host processor
3. The ShortStack LonTalk Compact API for the host processor
4. A ShortStack application that uses the ShortStack LonTalk Compact API

Figure 4 shows the basic software architecture of a ShortStack device.

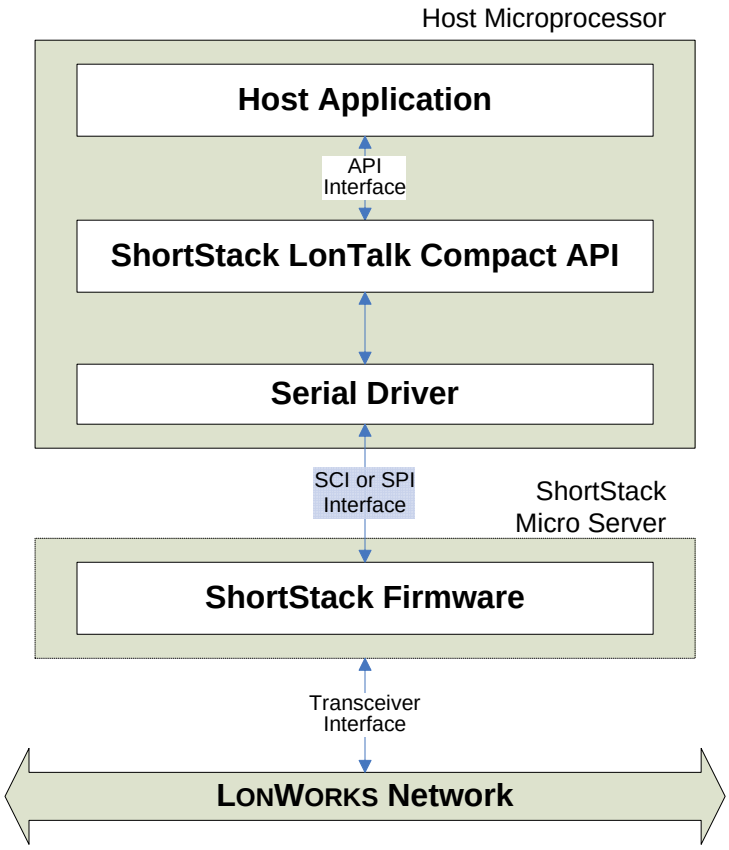

Figure 4. ShortStack Architecture

The ShortStack Micro Server

A ShortStack Micro Server consists of the ShortStack firmware running in an Echelon Smart Transceiver. The ShortStack Micro Server implements layers 1-6 of the LonTalk protocol. You can create a ShortStack Micro Server by loading a ShortStack firmware image into an Echelon Smart Transceiver. For example, **Figure 5** on page 13 shows an Echelon PL 3120 Smart Transceiver, a PL 3150 Smart Transceiver, and an FT 3120 Smart Transceiver with an FT-X1 Communication Transformer – all of which can be used to create a ShortStack Micro Server.

Figure 5. ShortStack Components

The ShortStack Micro Server communicates with the host processor using either the SCI or the SPI interface. The ShortStack SCI interface is a half-duplex asynchronous serial interface with 1 start bit, 8 data bits, and 1 stop bit (least significant bit first). The ShortStack SPI interface is a half-duplex synchronous serial interface between the ShortStack Micro Server and the host processor, where the Micro Server is the SPI master.

The ShortStack Serial Driver

The ShortStack serial driver provides the hardware-specific interface between the ShortStack LonTalk Compact API and ShortStack Micro Server. The serial driver manages data exchange between the host processor and the ShortStack Micro Server. You must create the serial driver that resides on the host microprocessor, or use one of the available example drivers. An example driver is available for an ARM7 host processor, the Atmel ARM AT91SAM7S64 microprocessor. You can use or modify this driver, or create your own driver for a different processor.

SCI Architecture

The SCI interface is an asynchronous serial interface, similar to the EIA-232 standard interface, as shown in **Figure 6** on page 14. Standard UART or USART hardware and software support is generally sufficient to implement this link.

See *SCI Interface* on page 71 for more information about the SCI interface for ShortStack devices.

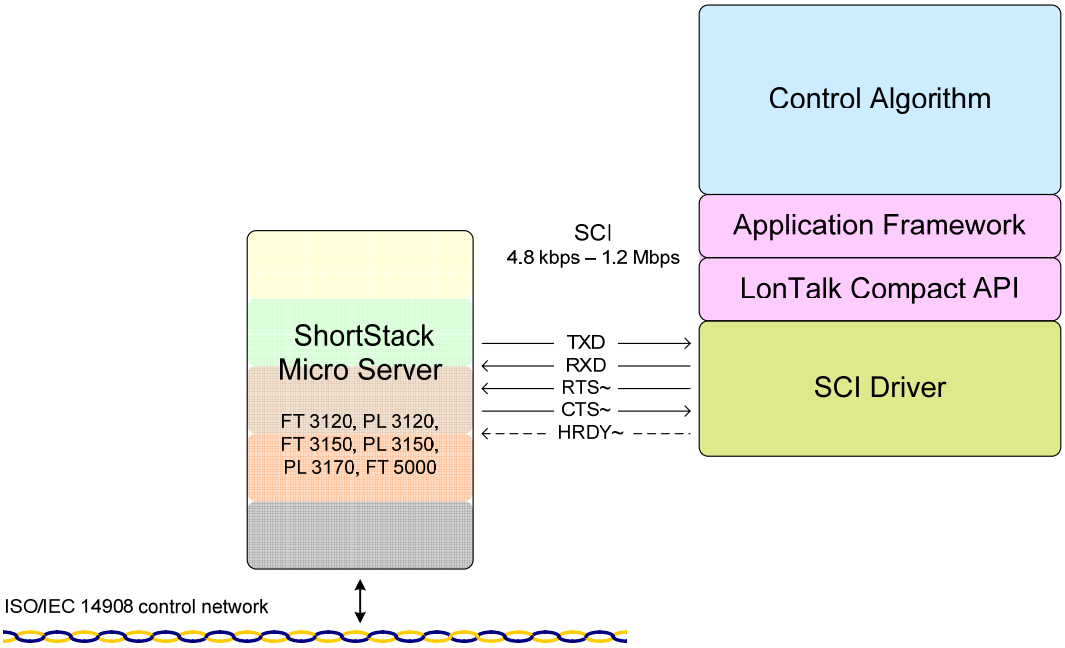

Figure 6. SCI Architecture for a ShortStack Device

SPI Architecture

The SPI interface is a synchronous serial interface, where the Micro Server acts as the master, as shown in **Figure 7** on page 15. Although most ShortStack devices use the SCI interface because of the need for fewer I/O lines for the synchronous link, and because the requirements for the SPI driver are more complex, the SPI interface can nonetheless be useful if all SCI resources on the host processor are already in use.

See *SPI Interface* on page 76 for more information about the SPI interface for ShortStack devices.

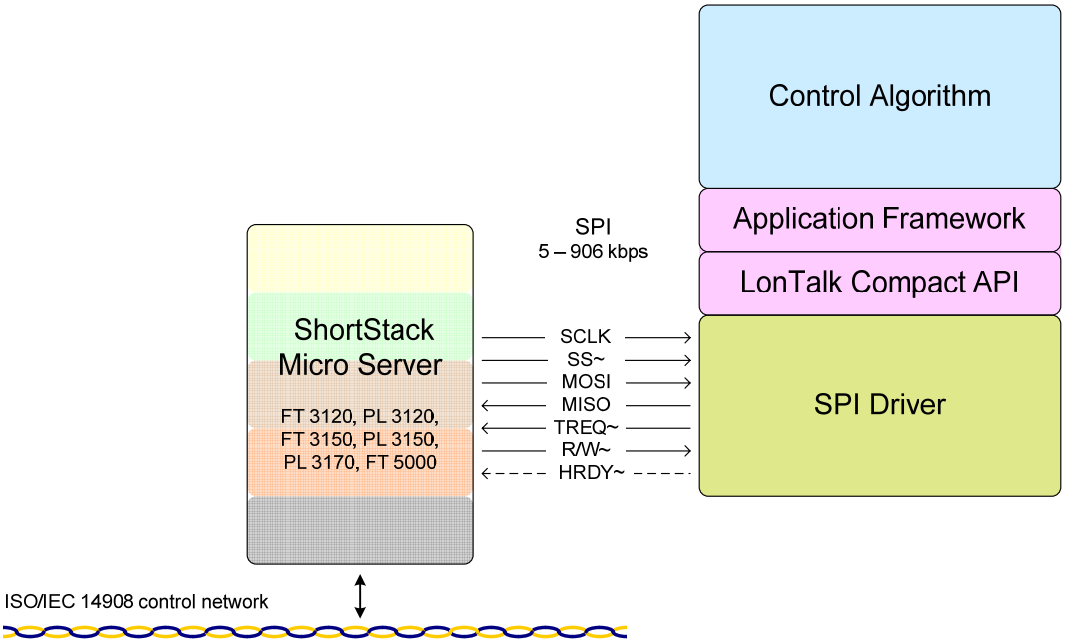

Figure 7. SPI Architecture for a ShortStack Device

The ShortStack LonTalk Compact API

The ShortStack Developer's Kit includes source code for the ShortStack LonTalk Compact API that you compile and link with your application. This API defines the functions that your application calls to communicate with other devices on a LONWORKS network. The API code is written in ANSI C. You might need to port the code for your host processor if an ANSI C compiler is not available.

The ShortStack LonTalk Compact API consists of the following:

- A service to initialize the ShortStack device after each reset.
- A service that the application must call periodically. This service processes messages pending in any of the data queues.
- Services to initiate typical operations, such as the propagation of network variable updates.
- Callback 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.
- Optional API components to perform low-level self-installation tasks.
- Optional API components to perform high-level ISI self-installation tasks.
- Optional API components for additional utility services.

Overview of the ShortStack Development Process

This manual describes the development process for creating a ShortStack device, which includes the general tasks listed in **Table 3** on page 16.

Table 3. Tasks for Developing a ShortStack Device

Task	Additional Considerations	Reference
Install the ShortStack Developer's Kit and become familiar with it		Chapter 2, *Getting Started with ShortStack*, on page 19
Select hardware for the ShortStack Micro Server and prepare it by loading the ShortStack firmware into it	You must select the Micro Server configuration and preferences for every new device, but you can reuse a Micro Server hardware and software configuration for a different application, and thus implement a different device.	Chapter 3, *Selecting and Creating a ShortStack Micro Server*, on page 27 Chapter 4, *Selecting the Host Processor*, on page 61
Integrate the ShortStack Micro Server with your device hardware	You integrate the Micro Server with the device hardware. You can reuse many parts of a hardware design for different applications to create different ShortStack devices.	Chapter 5, *Designing the Hardware Interface*, on page 65
Create the serial driver for the host processor	You must create a serial driver (typically derived from an example driver), for each device's hardware. You can reuse the driver with the same device hardware for different applications, and thus create different ShortStack devices. You do not need to re-create a new serial driver for each application.	Chapter 6, *Creating a ShortStack Serial Driver*, on page 89
Port the ShortStack LonTalk Compact API to the host processor	You must port the ShortStack LonTalk Compact API once for each host processor and compiler, but you can reuse the ported API files with any number of applications that share the same hardware and software development environment.	Chapter 7, *Porting the ShortStack LonTalk Compact API*, on page 109 Appendix C, *ShortStack LonTalk Compact API*, on page 287

Task	Additional Considerations	Reference
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 8, *Creating a Model File*, on page 115
Structure the layout and interoperable interface of your ShortStack 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) language, for every application that you implement. You can write this code by hand, derive it from an existing Neuron C application, or use the NodeBuilder Code Wizard included with the NodeBuilder Development Tool to create the required code using a graphical user interface.	Chapter 8, *Creating a Model File*, on page 115 The *Neuron C Reference Guide*
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 compile with your application. This source code includes data that is required for initialization and for complete implementations of some aspects of your device.	Chapter 9, *Using the LonTalk Interface Developer Utility*, on page 145
Complete the ShortStack LonTalk Compact API callback handler functions to process application-specific LONWORKS events	You must complete the callback handler functions for every application that you implement, or supply the Micro Server with application-specific data. The completed callback handler functions provide input from network events to your application; they are part of your networked device's control algorithm.	Chapter 10, *Developing a ShortStack Application*, on page 163 Appendix C, *ShortStack LonTalk Compact API*, on page 287

Task	Additional Considerations	Reference
Modify your application to interface with a LONWORKS network by using the ShortStack LonTalk Compact 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 output network variable value to the network. Together with the completion of the callback handler functions, this task forms the core of your networked device's control algorithm.	Chapter 10, *Developing a ShortStack Application*, on page 163 Appendix C, *ShortStack LonTalk Compact API*, on page 287
Optionally, add Interoperable Self-Installation (ISI) functions to your ShortStack device, add low-level functions to implement self-installation, or add other optional utility functions and callbacks	This step is optional. The necessary code is typically at least partially unique for each application, and needs to be reviewed and refined for each application that you write that uses self-installation procedures.	Chapter 11, *Developing a ShortStack Application with ISI*, on page 197 Appendix D, *ShortStack ISI API*, on page 301
Optionally, create a custom Micro Server image that supports your own hardware configuration	The standard Micro Servers are pre-compiled binary images that support a variety of hardware configurations. You can create a custom Micro Server and use it in place of a standard one to provide better support for your hardware, or even to offload some of the application's control algorithm to the Micro Server.	Chapter 12, *Custom Micro Servers*, on page 241
Test, install, and integrate your ShortStack device using self-installation or a LONWORKS network tool such as the LonMaker Integration Tool		The *LonMaker User's Guide*

If you have a ShortStack 2 application for a ShortStack device, and you want to take advantage of the new features and API of ShortStack FX, see Chapter 13, *Converting a ShortStack 2 Application to a ShortStack FX Application*, on page 257.

2

Getting Started with ShortStack

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

ShortStack Developer's Kit Overview

The ShortStack FX Developer's Kit is a software toolkit that contains software tools, the LonTalk Compact API, ShortStack firmware, and documentation needed for developing applications for any microcontroller or microprocessor that uses a ShortStack Micro Server to communicate with a LONWORKS network. You can use the software with ShortStack Micro Servers that use an Echelon Series 3100 or Series 5000 Smart Transceiver or an Echelon Neuron 5000 Processor.

The kit includes the following components:

1. Portable ANSI C source code for the ShortStack LonTalk Compact API and ShortStack ISI API.
2. ShortStack firmware images for free topology twisted-pair and power line configurations. Firmware images are provided for both TP/FT-10 and PL-20 channel types, including 3120, 3150, 3170, and 5000 Smart Transceiver devices.
3. ANSI C source code and pre-compiled library files that you can use to create custom Micro Servers to provide support for different hardware configurations.
4. The LonTalk Interface Developer utility. The LonTalk Interface Developer utility translates a model file into device interface data and device interface files that simplify the implementation of your ShortStack application, and it creates a skeleton application framework that provides much of the code required by your application to interface with the ShortStack Micro Server.
5. Documentation. This *ShortStack User's Guide* describes how to use the components of the ShortStack Developer's Kit to create a ShortStack device. The kit also includes detailed HTML documentation for the ShortStack LonTalk Compact API and ShortStack ISI API.

In addition to the ShortStack Developer's Kit, Echelon provides example ports for selected host processors. Each example port is separately installable, and includes its own documentation. For the ShortStack FX release, an example port is available for an ARM7 host processor (the Atmel ARM AT91SAM7S64 microprocessor). Other example ports may become available after release, from Echelon or from other vendors.

The ShortStack Developer's Kit and the example ports are available as free downloads from www.echelon.com/shortstack.

Installing the ShortStack Developer's Kit

You can install the ShortStack FX Developer's Kit on any computer that runs Microsoft® Windows® XP or Windows Vista®.

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

1. Download the ShortStack FX Developer's Kit from www.echelon.com/shortstack. Although the download is free, you must agree to the license terms for the ShortStack Developer's Kit when you download it. By agreeing to the license terms, you will receive a license key.

2. Double click the **ShortStack400.exe** file that you downloaded. The Echelon ShortStack FX Developer's Kit main installer window opens.
3. Follow the installation dialogs to install the ShortStack FX Developer's Kit onto your computer. During installation, you are prompted for the license key that you received after agreeing to the license terms.

In addition to the ShortStack FX Developer's Kit, the installation program also installs current versions of:

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

Important: You can have only one version of the ShortStack Developer's Kit installed at a time on a single PC. That is, you cannot install both ShortStack FX and a prior version of the ShortStack Developer's Kit and switch between versions during device development. The version of the kit that you install last is the one that is usable. Echelon recommends that if you should need to revert to an earlier version, uninstall ShortStack FX before installing the older version.

If you are installing the ShortStack FX Developer's Kit onto a computer that already has the ShortStack 2 Developer's Kit installed, the ShortStack FX Developer's Kit installer performs the following actions:

1. Copies all of the files from the **C:\LonWorks\ShortStack** directory to a new directory, **C:\LonWorks\ShortStack 2.0**, as a backup of your ShortStack 2 data; if you maintain a separate backup of your ShortStack 2 data, you can delete this directory.
2. Uninstalls ShortStack 2.
3. Deletes the **C:\LonWorks\ShortStack** directory to remove any service pack data or user files.
4. Installs ShortStack FX.

If you are installing the ShortStack FX Developer's Kit onto a computer that already has the ShortStack 2.1 Developer's Kit installed, the ShortStack FX Developer's Kit installer overwrites the files in the default installation path. However, any user-created files in this directory (or its subdirectories) are not changed.

ShortStack LonTalk Compact API Files

The ShortStack LonTalk Compact API is provided as a set of portable ANSI C files, which are listed in **Table 4** on page 22. These files are contained in the [*ShortStack*]**\API** directory (where [*ShortStack*] is the directory in which you installed ShortStack FX, usually **C:\Program Files\LonWorks\ShortStack** or **C:\LonWorks\ShortStack**). The LonTalk Interface Developer utility automatically copies these files into your project folder, but does not overwrite existing files with the same names.

You need to port the API to your host processor; for more information about porting the API, see Chapter 7, *Porting the ShortStack LonTalk Compact API*, on page 109.

Table 4. ShortStack LonTalk Compact API Files

File Name	Description
LonBegin.h LonEnd.h	Optional definitions for implementing data packing and alignment preferences
LonPlatform.h	Definitions for adjusting your compiler and environment to the requirements of the ShortStack LonTalk Compact API
ShortStackApi.c ShortStackApi.h	Function definitions for the ShortStack LonTalk Compact API
ShortStackHandlers.c	Function definitions for the ShortStack callback handler functions
ShortStackInternal.c	Internal functions and utilities that are used by the ShortStack LonTalk Compact API, but not exported to the host application
ShortStackIsiApi.c ShortStackIsiApi.h	Function definitions for the ShortStack ISI API
ShortStackIsiHandlers.c	Function definitions for the ShortStack ISI callback handler functions
ShortStackIsiInternal.c	Internal functions and utilities that are used by the ShortStack ISI API, but not exported to the host application
ShortStackIsiTypes.h	Definitions of the data structures that are typically used by ShortStack ISI applications
ShortStackTypes.h	Definitions of the data structures that are typically used by ShortStack applications

Standard ShortStack Micro Server Firmware Images

Several standard ShortStack Micro Server firmware images are provided as pre-compiled image files that you can program into onchip memory for FT or PL 3120 Smart Transceivers or PL 3170 Smart Transceivers, into flash memory chips to be used with FT or PL 3150 Smart Transceivers, or into EEPROM memory chips for FT 5000 Smart Transceivers.

Important: You can use the ShortStack Micro Server only with an Echelon Smart Transceiver or an Echelon Neuron 5000 Processor. If you run the ShortStack Micro Server on a different Neuron Chip, the Micro Server exits quiet mode and enters the applicationless state.

Each set of pre-compiled images includes the following files:

- An APB and an NDL file for downloading the images over a LONWORKS network
- An XIF and a SYM file for use by the LonTalk Interface Developer utility
- For 3120 and 3170 devices, an NFI file for ex-circuit programming of the Smart Transceiver
- For 3150 devices, an NEI file for ex-circuit programming of the flash memory chip
- For Series 5000 devices, an NME file for ex-circuit programming of the EEPROM memory chip
- For standard Micro Servers that support ISI, a *.h file that you use with your application when writing code to use the ShortStack ISI API; see Chapter 11, *Developing a ShortStack Application with ISI*, on page 197, for more information.

When you use the LonTalk Interface Developer utility, it selects the appropriate set of Micro Server image files based on your preferences, and copies them to the project's output folder. These image files have the project's base name (rather than the image's base name) and the appropriate file extension (APB, NDL, NFI, NEI, NME, XIF, SYM, or H).

Table 5 describes the standard firmware image files for a ShortStack Micro Server, along with other information about each image. See *Firmware Image File Names* on page 33 for a description of the firmware file naming convention.

Table 5. Standard ShortStack Firmware Image Files

Smart Transceiver Type	Channel Type	Supported Clock Rates (MHz) [1]	Neuron Firmware Version [2]	Support for ISI	Supported CP Access Methods [3]
FT 3120-E4 V16	TP/FT-10	10 20 40	16	No	DMF, LW-FTP, CPNV
FT 3150 2K [4]	TP/FT-10	10	17.1	Yes	DMF, LW-FTP, CPNV
FT 5000 ES	TP/FT-10	20	18	Yes	DMF, LW-FTP, CPNV
FT 5000	TP/FT-10	20	19	Yes	DMF, LW-FTP, CPNV
PL 3120-E4	PL-20C, PL-20N	10	14	No	LW-FTP, CPNV
PL 3150 [4]	PL-20C, PL-20N	10	17.1	Yes	DMF, LW-FTP, CPNV

PL 3170	PL-20C, PL-20N	10	17	Yes	DMF, LW-FTP, CPNV

Notes:

1. The supported clock rates refer to external crystal or oscillator frequency for Series 3100 devices, but refer to internal system clock rate for Series 5000 devices.
2. The Neuron firmware versions listed refer to the versions used to create the standard Micro Server images.
3. The configuration property access methods listed are:
   - Direct memory file (DMF); see *Using Direct Memory Files* on page 189
   - The LONWORKS file transfer protocol (LW-FTP); see the File Transfer engineering bulletin at www.echelon.com
   - Configuration network variables (CPNVs); see *Declaring a Configuration Property* on page 124 and *CPNV and EEPROM NV* on page 193
4. The standard Micro Servers for FT 3150 and PL 3150 devices support a standard hardware design with external flash memory of 32 KB or more, and 128 bytes per sector.

Because not all combinations of hardware, channel type, and ISI features are supported by the pre-compiled Micro Server images, you might need to create your own custom Micro Server image. Specifically, you need to create a custom Micro Server image:

- If your device uses a different Smart Transceiver than the ones listed in **Table 5** (such as a Neuron 5000 Processor).
- If your device uses a different Neuron firmware version than the ones used for the standard Micro Server images.
- If your device uses a clock speed or system clock setting that is supported by the chosen hardware and transceiver, but is not listed in **Table 5**.
- If your device uses a memory map that is different from the one described in *Micro Server Memory Map* on page 29.
- If your Micro Server device requires ISI-DAS support, or a different level of ISI support.
- If you want to create application-specific custom Micro Servers that support ISI. Such a Micro Server can execute part of the ISI API local to the Micro Server for optimum performance and minimum host memory footprint.
- If your application requires a DMF window different from the default size or location; see *Using Direct Memory Files* on page 189 for more information.
- If your device requires a Micro Server with different properties than those used for the standard Micro Server images.

See *Custom Micro Servers* on page 241 for more information about creating a custom Micro Server.

Important: A ShortStack FX Micro Server cannot run on an FTXL 3190 Free Topology Transceiver.

LonTalk Interface Developer

The LonTalk Interface Developer utility generates the device interface data and the device interface file required to implement the interoperable interface for your ShortStack 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 device 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:\Program Files\LonWorks\InterfaceDeveloper** or **C:\LonWorks\InterfaceDeveloper**).

The [*ShortStack*]**\MicroServers** directory includes Micro Server XIF files. The LonTalk Interface Developer utility uses these files to extract Micro Server-specific details (such as the hardware description or buffer configuration), which the utility merges with application-specific details (such as the network variable configuration) to generate the device's XIF (and XFB) files in your project folder.

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 275.

For more information about the LonTalk Interface Developer utility, see Chapter 9, *Using the LonTalk Interface Developer Utility*, on page 145.

3

Selecting and Creating a ShortStack Micro Server

This chapter describes how to create a ShortStack Micro Server using one of the standard ShortStack Micro Server images that are included with the ShortStack Developer's Kit, and how to load them into a Smart Transceiver.

Overview

A ShortStack device uses a ShortStack Micro Server to interface with a LONWORKS network. The ShortStack Micro Server provides the physical interface to the network, and also implements layers 1-6 of the LONWORKS network protocol. A Micro Server uses an Echelon Smart Transceiver that connects to the power line (PL) or twisted-pair free topology (FT) network, runs the ShortStack Micro Server firmware, and includes some mandatory peripheral components. A Micro Server based on the FT 3150 or PL 3150 Smart Transceiver can contain off-chip flash, ROM, or RAM memory for enhanced capabilities, a Micro Server based on the FT 5000 Smart Transceiver contains off-chip EEPROM or flash memory for enhanced capabilities, whereas a Micro Server based on the FT 3120, PL 3120, or PL 3170 Smart Transceiver implements a single-chip solution.

You can embed the Micro Server hardware within your device's hardware, or you can use off-the-shelf hardware for the Micro Server and connect it to your device. For example, you can use the Echelon Mini FX Evaluation Boards for rapid prototyping of a Micro Server. You can also use the Pyxos™ FT EV Pilot Evaluation Board (part of the Pyxos FT EVK Evaluation Kit) for development of ShortStack devices that use an FT 3150 Smart Transceiver with an ARM7 host processor.

You can load the ShortStack Micro Server firmware image into an FT 3120, PL 3120, or PL 3170 Smart Transceiver, or into a flash memory device, such as an Atmel AT29C512 or AT29C010A flash memory device, for an FT 3150 or PL 3150 Smart Transceiver, or into an SPI or I^2C EEPROM or SPI flash memory device for an FT 5000 Smart Transceiver. You can load these images over the LONWORKS network, or you can use standard ex-circuit programming tools.

This chapter describes how to select a Micro Server, how to load the ShortStack firmware image into the Micro Server, how to initialize the Micro Server, and how to work with non-volatile memory within the Micro Server. For information about creating a custom Micro Server, see *Custom Micro Servers* on page 241.

Selecting the Micro Server Hardware

The ShortStack Micro Server supports two transceiver types at various clock rates for either power line (PL) or free-topology (FT) networks. For a ShortStack device, you can use an Echelon 3120 Smart Transceiver, a 3150 Smart Transceiver, a 3170 Smart Transceiver, an FT 5000 Smart Transceiver, or a Neuron 5000 Processor. In addition, for the Neuron 5000 Processor, the ShortStack Micro Server supports all of the transceiver types supported by that chip.

For device evaluation and development with the Smart Transceivers, you can use the Echelon Mini FX Evaluation Kit (with an FT 5000 Smart Transceiver for FT networks, or a 3150 or 3170 Smart Transceiver for PL networks), which includes optional Electronic Industries Alliance (EIA) standard RS-232-C level shifters, jumpers, I/O connectors, and (for most of these boards) a small prototyping area, to configure the Smart Transceiver for use as a ShortStack Micro Server.

More information about Echelon's evaluation boards is available from the Echelon Web site, www.echelon.com. In addition, other companies offer similar products, for which you can create a custom Micro Server.

When considering whether to use a 3120, 3150, 3170, or 5000 Smart Transceiver for a ShortStack device, the following factors are the most important: the Micro Server clock rate, memory map, and development device type.

Micro Server Clock Rate

The Micro Server clock rate determines the available bit rate for the link-layer transfer and the overall performance of the Micro Server. For Series 3100 devices, the clock rate is determined by the external crystal or oscillator; for Series 5000 devices, the clock rate is determined by the internal system clock rate. You can specify a Series 5000 device's internal system clock rate within the device's hardware template when you create a custom Micro Server. For the standard Micro Servers, the internal system clock rate is fixed. Each device type has its own clock rate maximum:

- For PL 3120, PL 3150, and PL 3170 Smart Transceivers, the highest possible external clock rate is 10 MHz. Typical PL 3120, PL 3150, or PL 3170 ShortStack devices use a 10 MHz crystal.
- For FT 3120 Smart Transceivers, the highest possible external clock rate is 40 MHz. Typical FT 3120 ShortStack devices use a 20 MHz crystal.
- For FT 3150 Smart Transceivers, the highest possible external clock rate is 20 MHz. However, using a flash memory device for off-chip storage limits the Micro Server's clock rate to 10 MHz. Thus, typical FT 3150 ShortStack devices use a 10 MHz crystal.
- For FT 5000 Smart Transceivers, the external clock rate is always 10 MHz, from which the chips generate an on-chip system clock rate (the clock multiplier is configurable). The highest possible system clock rate is 80 MHz. For this highest system clock rate, the link-layer transfer speed is very high, and generally non-standard for most UARTs and USARTs. That is, not all host processors will support all possible bit rates for the highest system clock rates. The standard Micro Server uses a 20 MHz system clock rate, which allows standard bit rates to be used.

See *Selecting the Link-Layer Bit Rate* on page 68 for more information about requirements for the bit rate.

Micro Server Memory Map

The Micro Server needs its own data storage, which it maintains in mapped non-volatile memory. For an FT 3120, PL 3120, or PL 3170 Smart Transceiver the memory map is fixed, but Micro Servers that are based on FT 3150, PL 3150, or FT 5000 Smart Transceivers can use a variety of memory maps. The memory map for all standard Micro Servers is fixed, but you can create a custom Micro Server to provide a different memory map.

For example, additional RAM can be used for creating 3150 Micro Servers that support ISI-DAS devices or other advanced Micro Server configurations.

A Micro Server with large off-chip flash memory can store additional Micro Server code, which allows the device to embed feature-rich versions of the ISI

self-installation protocol, or to implement a feature-rich custom Micro Server. A Micro Server with smaller off-chip flash memory areas leave larger areas of unused memory in the Micro Server's physical memory map, which allows the application to use direct memory files (DMF). Larger areas of such unused memory allow the application to store configuration property files in the direct memory files.

A 3120 or 3170 Smart Transceiver provides up to 4 KB of on-chip non-volatile memory, whereas a 3150 Smart Transceiver uses off-chip flash memory which can provide 32 KB or more of non-volatile memory. For many applications, the memory provided with the FT 3120, PL 3120, or PL 3170 Smart Transceivers is sufficient, but more complex ShortStack applications that implement a large number of network variables, include a feature-rich self-installation library, or require an increased buffer configuration, could require a Micro Server based on an FT 3150, PL 3150, or FT 5000 Smart Transceiver.

For FT 3150 and PL 3150 devices, the standard ShortStack Micro Server images require a flash device that supports a 128-byte sector size, such as the Atmel AT29C512 (64 KB) or AT29C010A (128 KB) flash device. The memory map used in the Micro Server images is declared for a 32 KB flash device, with a 128-byte sector size (which yields a memory map of 0x0000 to 0x7FFF). This memory map leaves significant memory available for applications to use the direct memory file access method; see *Using Direct Memory Files* on page 189 for more information.

For FT 5000 devices, the standard ShortStack Micro Server images require an SPI or I²C EEPROM memory device; see the *Series 5000 Chip Data Book* for additional information about the external memory requirements for an FT 5000 Smart Transceiver. The memory map used in the Micro Server images is declared for a 32 KB EEPROM device. This memory map leaves significant memory available for applications to use the direct memory file access method; see *Using Direct Memory Files* on page 189 for more information.

Recommendation: For a free-topology channel, use an FT 5000 Micro Server (with at least 32 KB of EEPROM, or with 2 KB of EEPROM and at least 32 KB of flash memory). For a power-line channel, use a PL 3170 Micro Server if the 2 KB of onchip RAM is sufficient for the required buffer configuration, or a PL 3150 Micro Server if offchip RAM is required. For all other supported channel types, use a Neuron 5000 Processor with a custom Micro Server.

Development Device Type

Many ShortStack devices use compact, low-cost Micro Server hardware based on FT 3120, PL 3120, or PL 3170 Smart Transceivers. Other more generic or more powerful devices use advanced Micro Server hardware based on FT 3150, PL 3150, or FT 5000 Smart Transceivers with additional off-chip memory.

Recommendation:

Use a Micro Server that is based on a FT 3150, PL 3150, or FT 5000 Smart Transceiver for your initial development of a ShortStack device.

Using a 3150 or 5000 Smart Transceiver with off-chip flash memory rather than a 3120 or 3170 Smart Transceiver allows easier and more rapid recovery in case of device programming errors.

After you complete the critical early steps of development, you can use the Micro Server hardware that your ShortStack device requires.

Preparing the ShortStack Micro Server

You can load an application image into a LONWORKS device, such as an Echelon Smart Transceiver. For ShortStack Micro Servers, the application image is the ShortStack Micro Server executable image.

You must load the Micro Server executable image into the Micro Server hardware before you can use it as a ShortStack device. After you complete hardware development, you can load the Micro Server image into your ShortStack device as part of your manufacturing process.

You typically load the Micro Server infrequently. After you have correctly loaded a particular Micro Server, you do not normally need to reload it. However, if you load a new version of the Neuron firmware into a Smart Transceiver, be sure to load an updated Micro Server image into the Smart Transceiver at the same time.

Important:

- After you load a new Micro Server image, the initial initialization of the Micro Server, together with the initialization of the host application, can take up to one minute to complete. The Micro Server is unresponsive to the network until this initialization is complete. After the initial initialization is complete, resetting or power-cycling the Micro Server with the same host application completes much more quickly.
- Reloading a Micro Server with an updated version of the Micro Server firmware could require changes in the serial driver or the API that resides in your host processor. For example, migrating an application from ShortStack 2 to ShortStack FX requires some changes to the serial driver because you use an updated ShortStack Micro Server. Loading a Micro Server with a version that is incompatible with the current host application can sever link-layer communications.

Table 6 summarizes the processor and memory combinations that you can use with the standard, pre-compiled, Micro Server images, along with the files and tools that you use to program each. See *Firmware Image File Names* on page 33 for a description of the Micro Server image file extensions and file naming convention.

Table 6. Loading the Micro Server Executable Image

Smart Transceiver	Memory Type	Micro Server Image File Extension	Micro Server Image Programming Tool	Example Programming Tools
FT 3120, PL 3120, or PL 3170 Smart Transceiver	On-chip EEPROM	APB, NDL, or NEI	Network management tool	NodeLoad utility LonMaker Integration Tool
		NFI	PROM programmer	A universal programmer, such as one from BPM Microsystems or HiLo Systems

Smart Transceiver	Memory Type	Micro Server Image File Extension	Micro Server Image Programming Tool	Example Programming Tools
FT 3150 or PL 3150 Smart Transceiver	Off-chip flash	APB or NDL	Network management tool	NodeLoad utility LonMaker Integration Tool
		NEI	Ex-circuit flash programmer	A universal programmer, such as one from BPM Microsystems or HiLo Systems
FT 5000 Smart Transceiver	Off-chip EEPROM or flash	APB or NDL	Network management tool	NodeLoad utility LonMaker Integration Tool
		NME or NMF	EEPROM or flash programmer	A universal programmer, such as one from BPM Microsystems or HiLo Systems In-circuit programmer, such as Total Phase™ Aardvark™ I2C/SPI Host Adapter

Notes:

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

For device production, you typically use ex-circuit programming (where the chip is programmed prior to soldering it to the device circuit board); for development, you typically use in-circuit programming (where the chip is part of the device during programming) for simplicity rather than programming speed.

Firmware Image File Names

The base file names for the standard Micro Server firmware images use the following naming convention:

SS400_ + *image base file name* + _ + *5-digit clock speed* + kHz + *file extension*

For a Series 3100 device, the clock speed figure contained in the file name refers to the external clock speed (for example, "10000kHz" for a 10 MHz crystal). For Series 5000 devices, because the external clock speed is fixed (a 10 MHz crystal), the clock speed figure embedded in the image file name refers to the internal system clock frequency. The system clock rate is prefixed with "SYS" to highlight this difference. Micro Servers created for pre-production parts include "ES" (to signify Engineering Sample) in the name.

Examples:

- The universal chip programmer standard image for the PL 3120-E4 Smart Transceiver has the following name: **SS400_PL3120E4_10000kHz.nfi**.
- The NodeLoad standard image for the ISI-enabled FT 5000 Smart Transceiver has the following name: **SS400_FT5000Isi_SYS20000kHz.ndl**.

The firmware images with these names are located in the [*ShortStack*]**\MicroServers** directory, and are intended as backup copies of the images.

When you use the LonTalk Interface Developer utility, it selects the appropriate set of Micro Server image files based on your preferences, and copies them to the project's output folder. These image files have the project's base name (rather than the image's base name) and the appropriate file extension.

Table 7 lists the valid *file extension* values for the firmware image files.

Table 7. Micro Server Image File Extensions

Extension	Description
APB	Micro Server firmware image file for network management tools, such as the LonMaker Integration tool. Applies to all Smart Transceivers.
NDL	Micro Server firmware image file for the Nodeload utility. Applies to all Smart Transceivers.
NEI, NXE	Micro Server firmware image file for a universal chip programmer (for 3150 or 5000 Smart Tranceivers) or for image download tools (for 3120 or 3170 Smart Transceivers).
NFI	Micro Server programmable firmware image file for a universal chip programmer. Applies only to 3120 and 3170 Smart Transceivers.

Extension	Description
NME, NMF	Micro Server programmable firmware image file for a universal chip programmer. Applies only to FT 5000 Smart Transceivers and Neuron 5000 Processors.

In addition, the [*ShortStack*]**MicroServers** directory includes files with the following file extensions for each Micro Server type:

- XIF – The Micro Server's device interface (XIF) file (used only by the LonTalk Interface Developer utility)
- SYM – The Micro Server's device symbol file (used only by the LonTalk Interface Developer utility)
- H – A C header file that is shared between the Micro Server and the host application to define the location of ISI callbacks and other implementation details for an ISI application (present only for Micro Servers that support the ISI protocol)

Loading an FT 3120, PL 3120, or PL 3170 Smart Transceiver

Because a 3120 or 3170 Smart Transceiver does not support external memory, the only memory to program is on-chip EEPROM, which you program over the network or with a PROM programmer that supports the 3120 or 3170 Smart Transceiver.

To load the ShortStack Micro Server firmware using ex-circuit programming, you can use:

- A 3120 chip programmer to load a ShortStack Micro Server's NEI file into the 3120 or 3170 Smart Transceiver's non-volatile memory.
- A general-purpose programmer that supports the 3120 or 3170 Smart Transceiver, such as a BPM Microsystems or Hi-Lo Systems universal programmer, to load a ShortStack Micro Server's NFI file into the 3120 or 3170 Smart Transceiver's non-volatile memory.

To load the ShortStack Micro Server firmware using in-circuit programming, use the NodeLoad utility or the LonMaker Integration tool. See *Using a Network Management Tool for In-Circuit Programming* on page 36 for information about using these tools to load a ShortStack Micro Server.

Recommendation: Do not use the LonMaker Integration tool for the *initial* load of a ShortStack Micro Server into a power line Smart Transceiver. You can use the LonMaker Integration tool for any subsequent loads as long as the channel type does not change (for example by adding or removing support for the CENELEC protocol). See *Using the LonMaker Integration Tool with ShortStack* on page 38.

Loading an FT 3150 or PL 3150 Smart Transceiver

A device based on a 3150 Smart Transceiver always has non-volatile off-chip memory (PROM, EEPROM, or flash memory), and might also have off-chip RAM.

The ShortStack firmware must reside in the non-volatile memory. The standard Micro Servers for FT 3150 and PL 3150 Smart Transceivers support offchip flash memory with at least 32 KB and 128 bytes per sector.

You can load the ShortStack Micro Server firmware into a flash memory device, such as an Atmel AT29C512 or AT29C010A flash memory device, for an Echelon FT 3150 Smart Transceiver or PL 3150 Smart Transceiver.

To load the ShortStack firmware using ex-circuit programming, use an appropriate flash programmer to load a ShortStack Micro Server's NEI file into the 3150 Smart Transceiver's off-chip memory.

Important: Although you can reload the FT 3150 or PL 3150 Micro Server using in-circuit programming, you must perform an *initial* load for the Micro Server firmware using ex-circuit programming. This initial load is required because the 3150 Smart Transceiver does not contain boot loader code on chip.

After the off-chip non-volatile memory part has been initially programmed and inserted into the device, you can reload the Micro Server image using in-circuit programming using network management tools such as the NodeLoad utility or the LonMaker Integration tool. See *Using a Network Management Tool for In-Circuit Programming* on page 36 for information about using these tools to load a ShortStack Micro Server.

Loading a Blank Application

ShortStack device development does not require the loading of an initially blank application into the Smart Transceiver. However, for FT 3150 or PL 3150 Smart Transceivers, you can load a blank application into off-chip memory to clear the off-chip memory.

Although a device normally performs initialization once for a given firmware image, it is possible to force this process to occur again with the same firmware image by resetting the 3150 Smart Transceiver to the blank state (the initial state of the EEPROM on a newly manufactured Smart Transceiver) using the EEBLANK utility.

This utility is available as a free download from the LonWorks Downloads page, www.echelon.com/downloads, in the Development Tools category. To reset a 3150 chip's state, program the appropriate EEBLANK image (there is an image for each Smart Transceiver clock rate) into a 3150 flash memory chip and power up the device. For a short period, the service LED flashes, then it changes to full on to indicate that the chip has been returned to the blank state. The next time that any image is loaded into the flash memory for this device, the on-chip EEPROM is re-initialized.

Loading an FT 5000 Smart Transceiver

A device based on a Series 5000 Chip always has non-volatile off-chip memory (EEPROM or flash memory). The ShortStack firmware must reside in the non-volatile memory. The standard Micro Server for an FT 5000 Smart Transceiver supports a 32 KB EEPROM memory part. Note that there is no standard Micro Server image for a Neuron 5000 Processor.

To load the ShortStack firmware using ex-circuit programming, use an appropriate EEPROM or flash programmer (such as the Total Phase Aardvark

I2C/SPI Host Adapter) to load a ShortStack Micro Server's NME or NMF file into the FT 5000 Smart Transceiver's off-chip memory. For the FT 5000 EVB, connect the programmer to the **JP23** header, as described in the *FT 5000 EVB Hardware Guide*.

To load the Micro Server image using in-circuit programming, use network management tool such as the NodeLoad utility or the LonMaker Integration tool. See *Using a Network Management Tool for In-Circuit Programming* on page 36 for information about using these tools to load a ShortStack Micro Server.

Important: Do not use the LonMaker Integration tool for the *initial* load of a ShortStack Micro Server into an FT 5000 Smart Transceiver or Neuron 5000 Processor. You can use the LonMaker Integration tool for any subsequent loads as long as the Micro Server's system clock multiplier does not change. See *Using the LonMaker Integration Tool with ShortStack* on page 38.

Using a Network Management Tool for In-Circuit Programming

To load the ShortStack firmware images using in-circuit programming, you can use network management tools such as Echelon's NodeLoad utility or LonMaker Integration tool.

Network management tools load Smart Transceiver application images (for a ShortStack device, this image is the Micro Server firmware) and normally complete the load process by resetting the device, waiting for the device to complete its boot sequence, and confirming a healthy device state.

However, for a ShortStack Micro Server, this health check is likely to fail for typical load scenarios. Following the device reset, the Micro Server enters quiet mode, in which all network interaction is suspended, and it waits for the host processor to complete the ShortStack initialization sequence. The Micro Server must enter quiet mode in this case to prevent an incomplete implementation of the LonTalk protocol stack from attaching to the network, but in this state it also prevents the loader from confirming the successful load completion.

The NodeLoad utility provides a parameter that suppresses the final reset and health check (the **-M** parameter) that allows an automated load process to complete without error.

For the LonMaker Integration tool, you might see an error during the load process; if you reset the physical device and re-commission the device from the drawing, the error should resolve itself. However, you should not use the LonMaker Integration tool for the *initial* load of a ShortStack Micro Server into an FT 5000 Smart Transceiver, Neuron 5000 Processor, or a power line Smart Transceiver. You can use the LonMaker Integration tool for any subsequent loads as long as the Micro Server's system clock multiplier does not change.

After the Micro Server image has been loaded, and while the Micro Server is in quiet mode, the Micro Server performs an extensive one-time initialization. This initialization period can take as long as one minute. The tasks performed during initialization depend on the chosen Micro Server hardware and clock settings, as well as the features and limits supported by the chosen Micro Server.

Using the NodeLoad Utility with ShortStack

For in-circuit programming of a Series 3100 or Series 5000 Smart Transceiver, you can use the NodeLoad utility to load an NDL file into the Smart Transceiver's non-volatile memory over a LONWORKS network. To use the NodeLoad utility, you need a LONWORKS network interface, such as the U10 or U20 USB Network Interface or the PCLTA-21 PCI Network Interface.

Important:

- The NodeLoad utility is designed for loading known and tested application images. If you use the utility to load a custom Micro Server image, or an incorrect Micro Server image for the hardware, the NodeLoad utility might not prevent you from loading an incompatible image into the Smart Transceiver. For a 3120 Smart Transceiver, it can be difficult to recover from such an incompatibility. For example, if you load an FT Micro Server image into a PL Smart Transceiver, you might not be able to recover without desoldering the 3120 chip and reprogramming it with a device programmer.
- Be sure to specify the **-M** switch for the **nodeload** command when you load a Micro Server image into a Series 3100 or Series 5000 Smart Transceiver. This switch specifies that a Micro Server image is to be loaded.
- For loading application images during development or manufacture, use the **-X** switch for the **nodeload** command, combined with the **-L** switch, to ensure that the correct communication parameter and clock multiplier settings are loaded.

  However, you should generally not use the **-X** switch for devices in field (after device deployment) because uploading incompatible communication parameters or clock multiplier settings can render the device inoperable or unresponsive to network communication.
- Use the NodeLoad utility only for NDL files. Do not use the utility to load other files into a Smart Transceiver.

Example: To load an NDL file called **ss400_ft5000isi_sys20000khz.ndl** over a LONWORKS network interface named **LON1**, allowing 20 seconds to press the service pin on the destination device, specifying that the utility load a Micro Server image file, and specifying that the load use the communication parameters included in the NDL file, use the following command:

```
nodeload –DLON1 –W20 –M –X –Lss400_ft5000isi_sys20000khz.ndl
```

The result of running the NodeLoad utility should look similar to the following:

```
nodeload -DLON1 -W20 -M –X –Lss400_ft5000isi_sys20000khz.ndl
Echelon NodeLoad Release 1.20
Received uplink local reset
Received an ID message from device 1.
Program ID is 9FFFFF0000000400
Received uplink local reset
Resetting node
Successfully loaded SS400_FT5000ISI_SYS20000KHZ.NDL
NodeLoad Result: Success; NID=04c5c5e20100.
```

The Nodeload utility is available as a free download from www.echelon.com/downloads, in the Development Tools category.

See the *NodeLoad Utility User's Guide* for more information about the NodeLoad utility.

Using the LonMaker Integration Tool with ShortStack

You can use the LonMaker Integration Tool to load the ShortStack firmware into a Smart Transceiver, or upgrade it. A blank FT 3120 Smart Transceiver has a TP/FT-10 twisted-pair compatible communications interface initialized for a 10 MHz input clock, and its Neuron firmware state is applicationless. Likewise, a blank PL 3120 Smart Transceiver has a PL-20 power line compatible communications interface initialized for a 10 MHz input clock, and its Neuron firmware state is applicationless. If your device uses the appropriate communications parameters with a 10 MHz clock, you can load the Micro Server and network configuration over the network, using a network management tool, such as the LonMaker Integration tool. Otherwise, you must load the Smart Transceiver using an ex-circuit programmer.

Important: You cannot use the LonMaker Integration tool for the *initial* load of an FT 5000 Smart Transceiver, Neuron 5000 Processor, or power line Smart Transceiver (either for the load of a blank device or after the device's hardware and clock settings have changed). Because LonMaker cannot adjust the Series 5000 device's on-chip system clock multiplier (just as it would not adjust a Series 3100 device's external crystal speed) or a power line Smart Transceiver's channel characteristics (such as addition or removal of support for the CENELEC protocol), a blank or recently changed device could become inoperative after loading. After you load the device with the correct properties (either by using a PROM programmer or the Nodeload utility), you can use the LonMaker Integration tool for subsequent loading as long as the system clock multiplier remains unchanged.

Recommendation: Use an ex-circuit programmer to perform the initial load for the Micro Server (either Series 3100 or Series 5000). You can use either an ex-circuit programmer or in-circuit network management tool for subsequent loads. For the initial load for the Micro Server, an in-circuit network management tool can report a failed load because the Micro Server enters quiet mode after an initial load. In this mode, the network management tool cannot communicate with the Micro Server. However, for subsequent loads, the Micro Server does not enter quiet mode.

To load the ShortStack firmware using in-circuit programming using LonMaker Integration Tool:

1. Add a Device shape to your LonMaker drawing. See the *LonMaker User's Guide* for more information about working with LonMaker drawings.
2. Optional: Ensure that the host processor is loaded with the ShortStack LonTalk Compact API and the appropriate application program and serial driver. This step ensures that the host application, serial driver, and Micro Server synchronize after the load.
3. Ensure that the Smart Transceiver and the host processor are connected and able to communicate with one another.
4. Ensure that the device is connected to the LONWORKS network.

5. Complete the information required by the LonMaker New Device Wizard.

   Do not select the **Commission device** checkbox (or use the Commission Device Wizard).

After you add the device to the LonMaker drawing, load the Micro Server firmware into the device. When prompted for the device application image name, specify the ShortStack Micro Server image in the **Image name** field, and specify the device's interface file that was generated by the LonTalk Interface Developer utility in the **XIF name** field. Do not use the Micro Server's XIF file.

Important: In the **Image name** field, be sure to select the correct Micro Server image for your Smart Transceiver. The LonMaker Integration Tool can prevent some incompatibilities between the hardware, firmware, and Micro Server image, but some incorrect configurations are still possible.

Recommendation: To verify that the entire device is operational, do not import the device's XIF prior to commissioning, but instead specify **Upload from device** for the External Interface Definition in either the New Device Wizard or the Commission Device Wizard. Because the SI data is located on the host, reading the SI data requires communications with the Micro Server through the link layer. If the device can perform such communication successfully, the device is likely to be fully operational. See *Performing an Initial Micro Server Health Check* on page 82 for additional information about verifying the operational status of the Micro Server.

To test the device within LonMaker, right-click the device's shape in the LonMaker drawing and select **Manage**. From the LonMaker Device Manager window, select **Test**.

See the *LonMaker User's Guide* for more information about using the LonMaker Integration Tool.

Working with FT 5000 EVB Evaluation Boards

You can use an Echelon FT 5000 EVB evaluation board to develop your ShortStack application. However, you must set the jumpers to configure the Smart Transceiver for the ShortStack Micro Server and to set the appropriate link-layer bit rate.

You can connect the host processor board to an FT 5000 EVB through either of the following connectors:

- The evaluation board's general-purpose peripheral I/O connector **P201** (the Gizmo and MiniGizmo connector). This connection allows the ShortStack Micro Server and the host processor to use a common power supply with either a 3.3 V or a 5 V signal level. If the ShortStack Micro Server and the host processor use separate power supplies, you must ensure that they share a common ground for the link-layer; use the **P201** connector to provide the ground connection. This connection supports either an SCI or an SPI serial driver connection. See *Using the Gizmo Interface (SCI or SPI)* on page 41.
- The on-board EIA-232 connector **J201**. This connection includes a Maxim® Integrated Products MAX3387E AutoShutdown Plus™ RS-232 Transceiver that allows ShortStack link-layer drivers to use standard EIA-232 communications levels, with handshake signals, and maintain

separate power supplies. This connection supports only SCI serial driver connections. See *Using the EIA-232 Interface (SCI)* on page 45.

To enable the FT 5000 EVB to support a ShortStack application, you must mount or dismount jumpers on the following headers: **JP31**, **JP32**, **JP201**, and **JP203**. In addition, you should verify the settings for the **JP1**, **JP33**, **JP202**, **JP204**, and **JP205** jumpers. See the *FT 5000 EVB Hardware Guide* for more information about these jumpers.

General Jumper Settings for the FT 5000 EVB

Verify and set the following jumpers to run a ShortStack Micro Server on an FT 5000 EVB.

Although the jumper settings for headers **JP1**, **JP33**, and **JP202** are not specific to running a ShortStack Micro Server on the FT 5000 EVB, they are included so that you can verify the settings for all of the headers on the board.

JP1

Leave the jumpers for the JP1 header mounted as shown in **Figure 8**. This header connects the FT 5000 Smart Transceiver to the onboard serial flash and serial EEPROM memory.

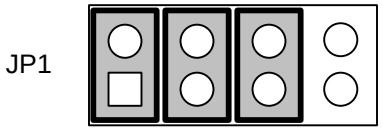

Figure 8. FT 5000 EVB Serial Memory Connections Header (JP1)

JP31

Dismount all of the jumpers from the **JP31** header, as shown in **Figure 9**. The settings shown in the figure disconnect the FT 5000 Smart Transceiver's I/O lines from the onboard I/O.

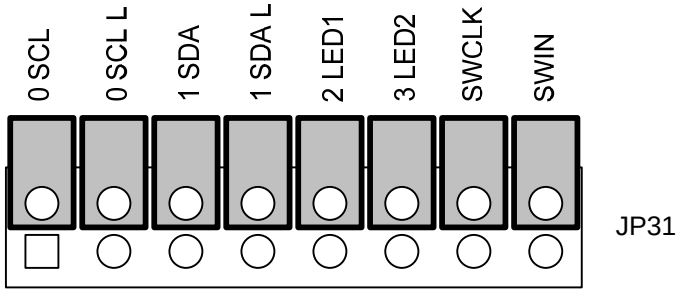

Figure 9. FT 5000 EVB I/O Connections Header (JP31)

JP33

The ShortStack Micro Server does not use the onboard LCD display, so you can dismount jumper on the **JP33** header to remove power to the LCD display, as shown in **Figure 10** on page 41. This jumper setting is optional.

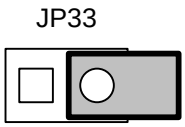

Figure 10. FT 5000 EVB LCD Display Power Header (JP33)

Using the Gizmo Interface (SCI or SPI)

To use the **P201** Gizmo interface on an FT 5000 EVB for a ShortStack application, set the following jumpers as described below.

JP32

Dismount all of the jumpers from the **JP32** header, as shown in **Figure 11** and **Figure 12** on page 42. The settings for pins 1-10 of the header shown in the figure disconnect the FT 5000 Smart Transceiver's I/O lines from the onboard I/O.

The **3 PD** jumper setting in **Figure 11** specifies the SCI interface for the ShortStack Micro Server. The **3 PD** jumper setting in **Figure 12** specifies the SPI interface for the ShortStack Micro Server.

For SCI, if your ShortStack serial driver does not use the HRDY~ signal, mount the jumper for **1 PD** to tie the HRDY~ signal low. For SPI, leave the **1 PD** jumper unmounted, as shown in the figures.

If you use a standard Micro Server or a custom ShortStack Micro Server that does not use the IO9 pin, you can dismount the **9 PD** jumper to engage the R226 pull-up. If you use a custom ShortStack Micro Server that uses the IO9 pin, you can mount or dismount the **9 PD** jumper as needed.

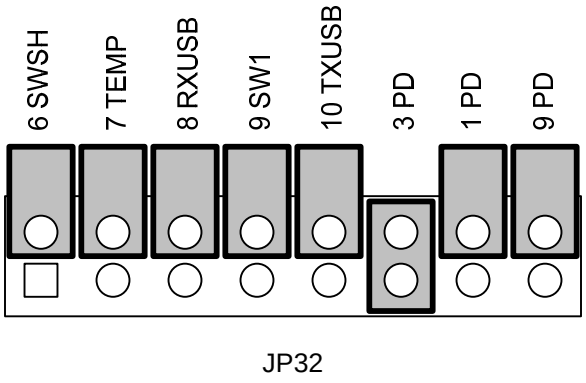

Figure 11. FT 5000 EVB I/O Connections Header (JP32) – SCI

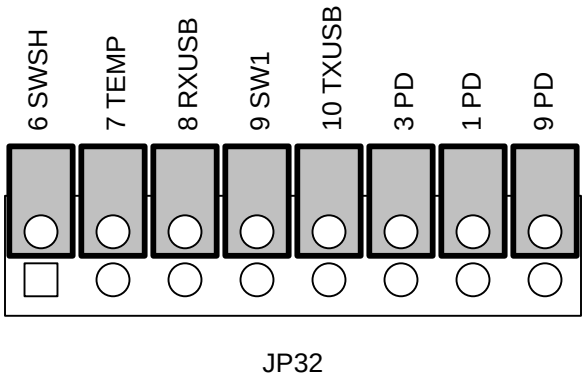

Figure 12. FT 5000 EVB I/O Connections Header (JP32) – SPI

JP201

Dismount all of the jumpers on the **JP201** header, except the **10T1IN** jumper, as shown in **Figure 13**. Although this header enables the EIA-232 interface, and is not needed for the Gizmo interface, the **10T1IN** jumper connects the R213 pull-up resistor for the Micro Server's IO10 pin (TXD for SCI or HRDY~ for SPI).

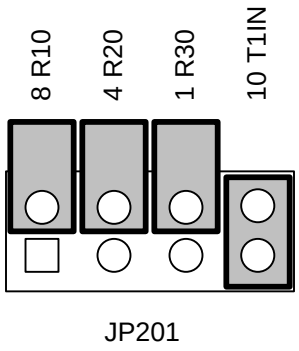

Figure 13. FT 5000 EVB EIA-232 Communications Header (JP201)

JP202

Mount the jumper for the **JP202** header to determine the external power source for the FT 5000 EVB, as shown in **Figure 14**.

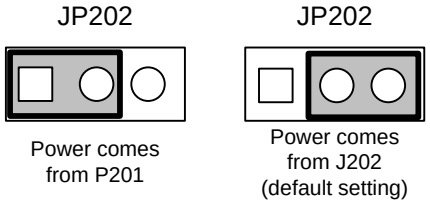

Figure 14. FT 5000 EVB Power Selector Header (JP202)

To use the Echelon power supply that ships with the FT 5000 EVB, mount the jumper so that power comes from the **J202** connector. This is the factory-default setting.

To allow the FT 5000 EVB to share a common 5 V power supply with your host board, mount the jumper so that power comes from pin 25 of the **P201** Gizmo header.

Recommendation: When possible, use a single power domain for both the host processor board and the FT 5000 EVB:

1. **Important**: Do not connect the external power supply to **J202** connector of the FT 5000 EVB.
2. Set the FT 5000 EVB's **JP202** jumper to use power from the **P201** Gizmo header (the left-hand image of **Figure 14**).
3. Connect power from the host board to pin 25 of the **P201** Gimzo header.
4. Connect the two boards to a common ground (you can use pin 20 or 23 of the **P201** Gizmo header to provide ground to the FT 5000 EVB).
5. Supply power to the host processor board.

If your host processor board is the Pyxos FT EV Pilot EVB, dismount jumper **JP61** (located on the left-hand side of the EVB, near the Smart Transceiver's flash memory socket), and use pin 1 (the right-hand pin) of the header for the 5 V power in step 3. Use pin 43 or 44 of header **JP505** for the ground connection in step 4.

JP203

Dismount the **0 T2IN** and **FON PD** jumpers, as shown in **Figure 15**. These jumpers apply to the EIA-232 interface only.

The figure also shows the **5 PD** and **6 PD** jumpers configured to specify the serial bit rate for the standard 20 MHz Micro Server (76800 bps for SCI; 76700 bps uplink and 38600 bps downlink for SPI).

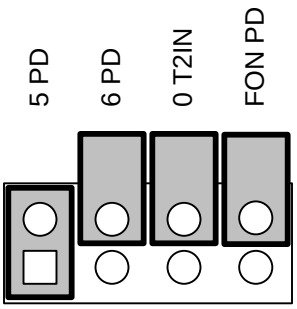

Figure 15. FT 5000 EVB ShortStack Header (JP203)

To set the link-layer bit rate for the Micro Server, determine the correct bit rate for your device according to *Selecting the Link-Layer Bit Rate* on page 68, and then mount the FT 5000 EVB's **5 PD** and **6 PD** jumpers on the **JP203** header appropriately to match the correct settings for the IO5 and IO6 pins on the Smart Transceiver. Depending on which serial driver you use, see either *Setting the SCI Bit Rate* on page 73 or *Setting the SPI Bit Rate* on page 77 for the correct settings for the IO5 and IO6 pins.

JP204

Mount the jumper for the **JP204** header, as shown in **Figure 16** on page 44, to determine whether power is supplied to pin 19 of the **P201** Gizmo header.

The default setting is to provide no power to pin 19, but you can supply either +5 V or +3.3 V.

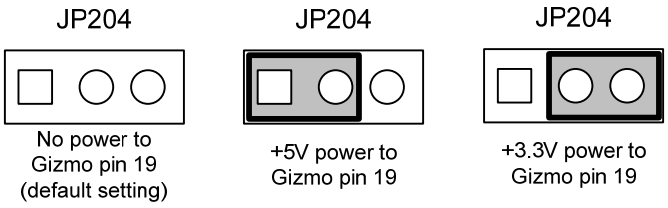

Figure 16. FT 5000 EVB Gizmo Pin 19 Power Selector Header (JP204)

JP205

Mount the jumper for the **JP205** header to determine whether power is supplied to pin 17 of the **P201** Gizmo header, as shown in **Figure 17**. The default setting is to provide no power to pin 17, but you can supply +3.3 V.

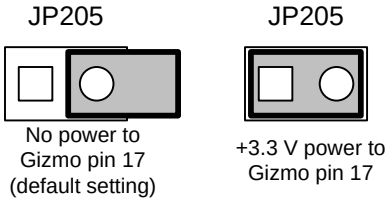

Figure 17. FT 5000 EVB Gizmo Pin 17 Power Selector Header (JP205)

P201

To connect your host evaluation board or Micro Server custom board to the **P201** Gizmo header, you must create a custom connection cable. For rapid prototyping, you might consider using short 0.25" (0.635 mm) square socket test leads for these connections. **Figure 18** on page 45 shows the Gizmo header (**P201**) on the FT 5000 EVB. The figure shows the signal names as used by the FT 5000 EVB, and also shows the signal names for the first 12 pins as used by the SCI and SPI interfaces for a ShortStack Micro Server (signal names are from the Micro Server's point of view).

When connecting an FT 5000 EVB to a host processor board, be sure to provide a solid ground connection between the two boards. You can use pin 20 or 23 of the **P201** Gizmo header for this ground connection.

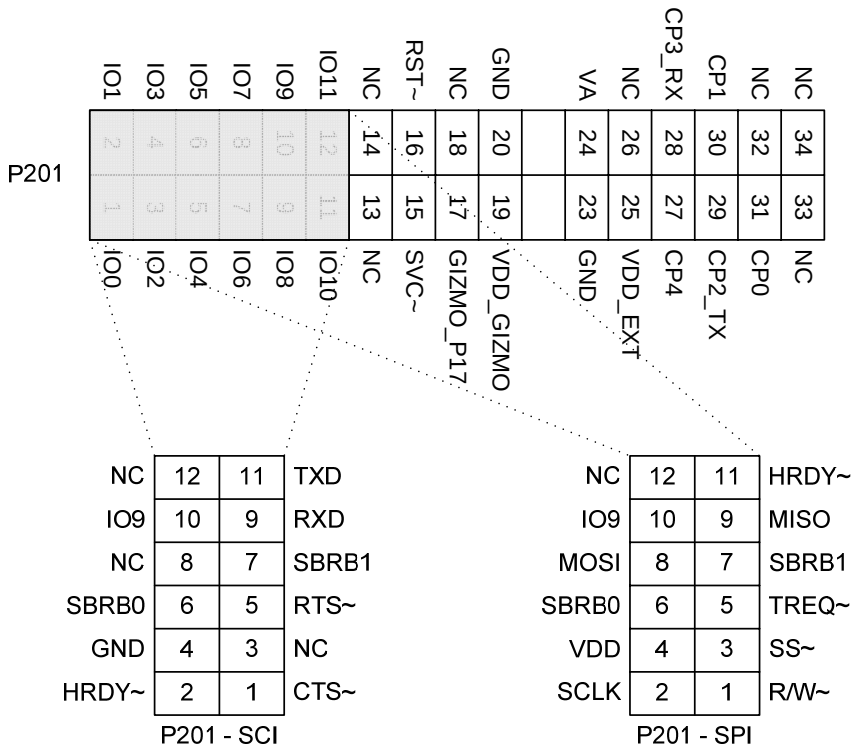

Figure 18. The Gizmo Header (P201) with the SCI and SPI Micro Server Signals

Using the EIA-232 Interface (SCI)

To use the EIA-232 interface on the FT 5000 EVB for a ShortStack application, set the following jumpers as described below.

JP32

Dismount all of the jumpers from the **JP32** header, as shown in **Figure 19** on page 46. The settings for pins 1-10 of the header shown in the figure disconnect the FT 5000 Smart Transceiver's I/O lines from the onboard I/O.

The **3 PD** jumper setting specifies the SCI interface for the ShortStack Micro Server. If your ShortStack serial driver does not use the HRDY~ signal, mount the jumper for **1 PD** to tie the HRDY~ signal low.

If you use a standard Micro Server or a custom ShortStack Micro Server that does not use the IO9 pin, you can dismount the **9 PD** jumper to connect the R226 pull-up. If you use a custom ShortStack Micro Server that uses the IO9 pin, you can mount or dismount the **9 PD** jumper as needed.

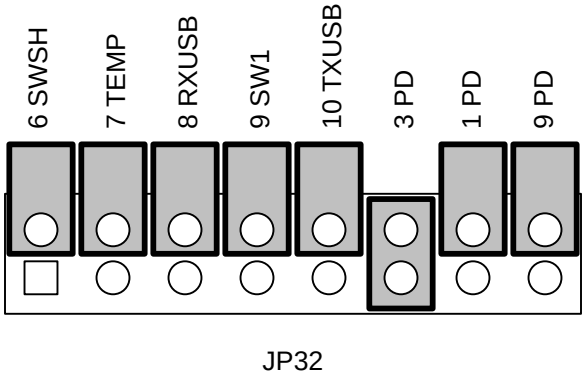

Figure 19. FT 5000 EVB I/O Connections Header (JP32)

JP201

Mount all of the jumpers on the **JP201** header, as shown in **Figure 20**. These jumper settings connect the Micro Server's IO1 (HRDY~), IO4 (RTS~), IO8 (RXD), and IO10 (TXD) signals to the EIA-232 connector. If your ShortStack serial driver does not use the HRDY~ signal, you can dismount the jumper for **1 R30**.

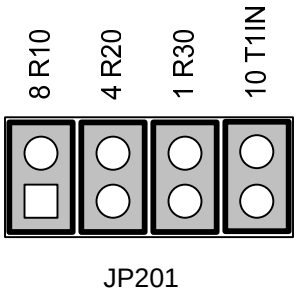

Figure 20. FT 5000 EVB EIA-232 Communications Header (JP201)

JP203

Mount the **0 T2IN** and **FON PD** jumpers on the **JP203** header, as shown in **Figure 21**. The figure also shows the **5 PD** and **6 PD** jumpers configured to specify a 76800 bps serial bit rate for the standard 20 MHz Micro Server.

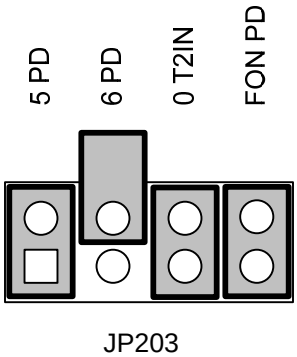

Figure 21. FT 5000 EVB ShortStack Header (JP203)

The MAX3387E RS-232 transceiver that is used on the FT 5000 EVB is configured to enter autoshutdown mode after inactivity of approximately 30 seconds. For applications that use high link-layer bit rates, the time required for the transceiver to become fully active (approximately 100 µs) might be long enough to cause a framing error on the serial link-layer signals.

Recommendation: To prevent the MAX3387E RS-232 transceiver from entering autoshutdown mode, you can mount the **FON PD** jumper on the **JP203** header connect the chip's FORCEON pin (pin 11) to GND, as shown in **Figure 21**. Alternatively, your SCI serial driver should briefly toggle the ShortStack Micro Server's HRDY~ signal every 10 to 20 seconds during periods of idleness. This toggle causes the MAX3387E transceiver to detect transmission activity and not enter autoshutdown mode.

To set the link-layer bit rate for the Micro Server, determine the correct bit rate for your device according to *Selecting the Link-Layer Bit Rate* on page 68, and then mount the FT 5000 EVB's **5 PD** and **6 PD** jumpers on the **JP203** header appropriately to match the correct settings for the IO5 and IO6 pins on the Smart Transceiver. See *Setting the SCI Bit Rate* on page 73 for the correct settings for the IO5 and IO6 pins.

The EIA-232 interface requires a null-modem cable for the D-SUB9 EIA-232 connector (**J201**) on the FT 5000 EVB. To define the null-modem EIA-232 interface, use the pin connections listed in **Table 8**. Keep the total cable length to a maximum of 24 inches (0.6 meters).

Table 8. EIA-232 Header to D-SUB9 Connector Pin Connections

D-SUB9 Connector Pin	Micro Server SCI Signal
1	N/A
2	TXD
3	RXD
4	HRDY~
5	GND
6	N/A
7	RTS~
8	CTS~
9	N/A

Clearing the Non-Volatile Memory

In general, if you have a working device, you should not need to clear the onboard non-volatile memory (the I²C serial EEPROM memory **U2**) on the FT 5000 EVB. For a working device, you can receive a service-pin message and reload the non-

volatile memory as needed. However, if it should become necessary to clear the non-volatile memory, perform the following tasks:

1. Press and hold the EVB's **RESET** button.
2. Temporarily connect pin 8 of header **JP1** to pin 7 of header **JP23**. This step connects the EEPROM's SCL pin to GND.
3. Release the EVB's **RESET** button.
4. Wait a few seconds until the EVB's Service Pin LED is illuminated (on solid, not flashing).
5. Disconnect pin 8 of header **JP1** from pin 7 of header **JP23**. Ensure that the jumpers for header JP1 are as shown in **Figure 8** on page 40.
6. Use NodeUtil utility to set the memory configuration and set the state for the device:
    a. Connect the PC that will run the NodeUtil utility to the same network interface that connects to the FT 5000 EVB. For example, if you connect to the FT 5000 EVB using **LON1**, connect the NodeUtil utility to **LON1**.
    b. Start the NodeUtil utility.
    c. Press the **SVC** button on the FT 5000 EVB to send a service-pin message to the NodeUtil utility. If you cannot receive a service-pin message from the device, repeat steps 1 to 5.
    d. Within the NodeUtil utility, select the **L** option to see all connected devices.
    e. Select the **G** option to manage the device that just sent a service-pin message (the FT 5000 EVB). Typically, this is device 1.
    f. Select **W** to write to a memory location. When prompted, do not update the application checksum and do not update the configuration checksum.
    g. Enter FDE8 for the starting address. Enter a value of 2 for address FDE8. This value specifies the memory type as I^2C serial EEPROM memory.
    h. Enter a period (.) to exit the memory write session.
    i. Select **W** to write to a different memory location. When prompted, do not update the application checksum and do not update the configuration checksum.
    j. Enter F037 for the starting address. Enter a value of 0 (zero) for address 0xF037. This value triggers device re-initialization.
    k. Enter a period (.) to exit the memory write session.
    l. Select **E** to exit device management mode.
    m. Select **E** to exit the NodeUtil utility.

At this point, you can reload the board with whatever application is required (for example, a ShortStack Micro Server or a Neuron C application). Because the device has returned to its default (empty) state and default settings, use the NodeLoad utility with the **-X** switch when loading an application or Micro Server

image (see *Using the NodeLoad Utility with ShortStack* on page 37). Do not use the LonMaker Integration Tool to load an image following this procedure.

Using a Logic Analyzer

During device development, it is recommended that you use a logic analyzer, such as the TechTools DigiView™ Logic Analyzer, to verify the link-layer signals. For an example, see *Performing an Initial Micro Server Health Check* on page 82. You can use the **JP24** header (see **Figure 22**) on the FT 5000 EVB to connect a logic analyzer to the EVB.

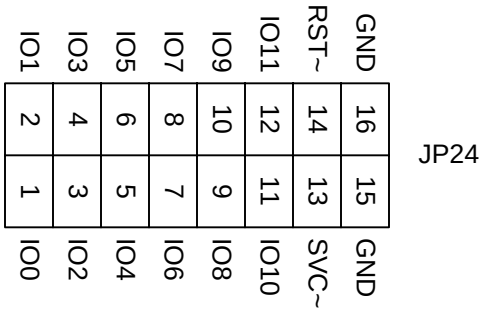

Figure 22. FT 5000 EVB Logic Analyzer Header (JP24)

Working with Mini EVB Evaluation Boards

You can use an Echelon Mini FX or Mini EVK evaluation board to develop your ShortStack application. However, you must set the jumpers to configure the Smart Transceiver for the ShortStack Micro Server and to set the appropriate bit rate.

You can connect the host processor to a Mini EVB through either of the following connectors:

- The evaluation board's general-purpose peripheral I/O connector **P201** (the Gizmo and MiniGizmo connector). This connection allows the ShortStack Micro Server and the host processor to use a common power supply with a 5 V signal level. By default, this connection supports only SCI serial driver connections. If you want to use the SPI interface, you must drive the IO3 (SPI/SCI~) pin high with a 10 kΩ pull-up resistor through the Gizmo (**P201**) header. See *Using the Gizmo Interface (SCI)* on page 50.
- The on-board EIA-232 connector **J201.** This connection includes a Maxim® Integrated Products MAX3387E AutoShutdown Plus™ RS-232 Transceiver, which allows ShortStack link-layer drivers to use standard EIA-232 communications levels and maintain separate power supplies. This connection supports only SCI serial driver connections. See *Using the EIA-232 Interface (SCI)* on page 52.

When connecting a Mini EVB to a host processor board, be sure to provide a solid ground connection between the two boards.

To enable the Mini EVB to support a ShortStack application, you must mount or dismount jumpers on the following headers: **JP201** and **JP203**. See the *Mini FX PL Hardware Guide* for more information about these jumpers.

Using the Gizmo Interface (SCI)

To use the **P201** Gizmo interface on a Mini EVB for a ShortStack application, set the following jumpers as described below.

JP201

Dismount all of the jumpers on the **JP201** header, as shown in **Figure 23**. This header enables the EIA-232 interface, which is not needed for the Gizmo interface. In the figure, the jumpers for the FT 3120 and 3150 boards are on the left, and the jumpers for the PL 3120, 3150, and 3170 boards are on the right.

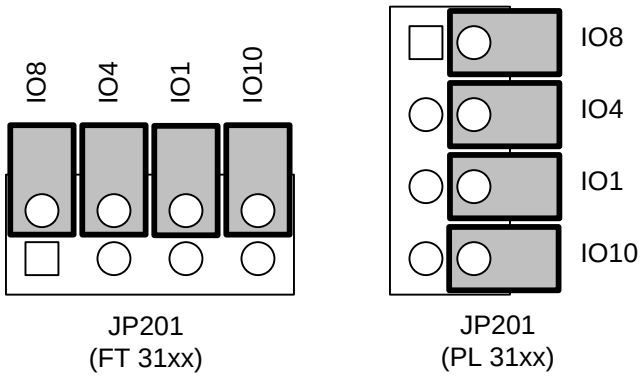

Figure 23. Mini EVB EIA-232 Enable Jumpers (JP201)

JP203

Dismount the IO0 jumper as shown in **Figure 24**; this jumper applies to the EIA-232 interface only. The figure also shows the IO5 and IO6 jumpers configured to specify a 38 400 bit rate on a Mini FX PL 3150 Evaluation Board for a 10 MHz Micro Server.

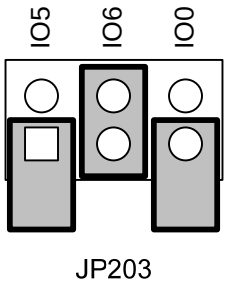

Figure 24. PL 3150 Mini FX ShortStack Enable Jumper (JP203)

To set the link-layer bit rate for the Micro Server, determine the correct bit rate for your device according to *Selecting the Link-Layer Bit Rate* on page 68, and then mount the Mini EVB's **JP203** jumpers appropriately to match the correct settings for the IO5 and IO6 pins on the Smart Transceiver. See *Setting the SCI Bit Rate* on page 73 for the correct settings for the IO5 and IO6 pins.

Important: The PL 3170 Smart Transceiver supports the 38400 bit rate only. Therefore, the **JP203** jumper settings for the IO5 and IO6 pins do not apply to the Mini FX PL 3170 Evaluation Board.

Recommendation: When possible, use a single power domain for both the host processor board and the Mini EVB. If you use the Pyxos FT EV Pilot EVB as your host processor board, you can allow the Mini EVB to provide 5 V power:

1. **Important**: Do not connect the external power supply to either the **JP201** connector or the **J31** connector of the Pyxos FT EV Pilot EVB.
2. Connect pin 26 (VDD5) of the **P201** Gimzo header on the Mini EVB to pin 2 of the **JP33** header on the Pyxos FT EV Pilot EVB. The **JP33** header is near the center of the EVB, to the right of the **JP512** and **JP510** headers. By default, there is a jumper that connects pins 1-2 of the **JP33** header; remove this jumper to connect to pin 2 of the header.
3. Connect the two boards to a common ground: Use pin 20 or 23 of the **P201** Gizmo header to provide ground from the Mini EVB, and use pin 43 or 44 of the **JP505** header to provide ground to the Pyxos FT EV Pilot EVB.
4. Supply power to the Mini EVB.

If you use a host processor board other than the Pyxos FT EV Pilot EVB, you should still use a common power domain. In this case, you should use a common power supply that meets the input power requirements of both the host processor board and the Mini EVB (note that the power line EVBs have different power requirements from the FT EVBs).

To connect your host evaluation board or Micro Server custom board to the **P201** Gizmo header, you must create a custom connection cable. For rapid prototyping, you might consider using short 0.25" (0.635 mm) square socket test leads for these connections. **Figure 25** on page 52 shows the Gizmo header (**P201**) on the PL 3170 EVB. The figure shows the signal names as used by the PL 3170 EVB, and also shows the signal names for the first 12 pins as used by the SCI and SPI interfaces for a ShortStack Micro Server (signal names are from the Micro Server's point of view).

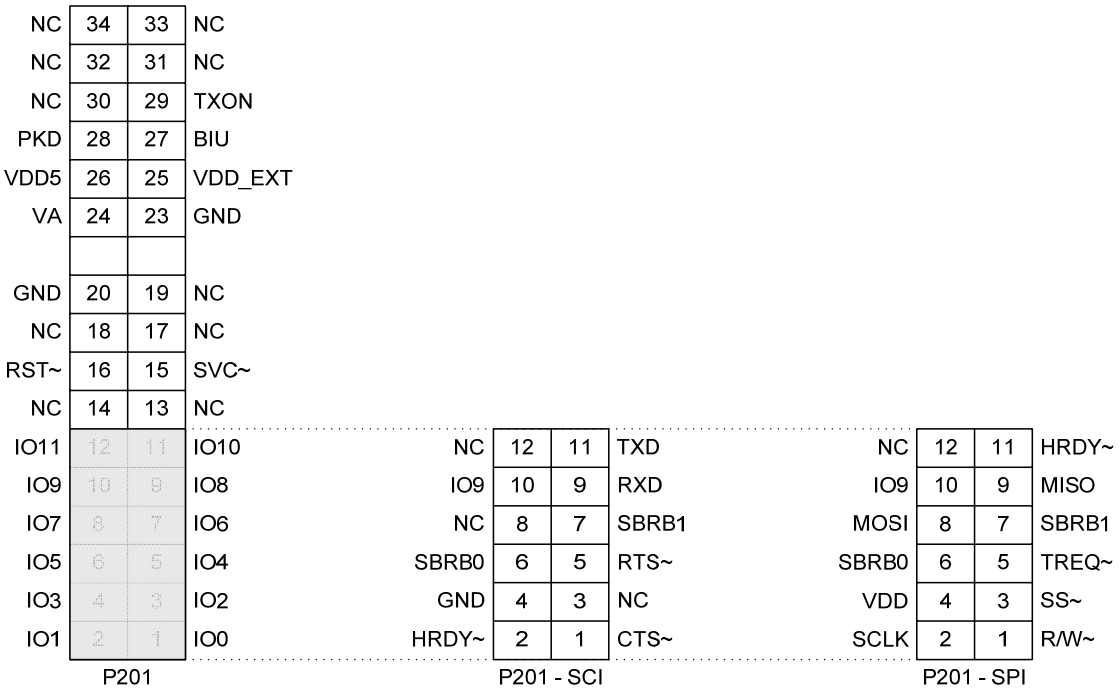

Figure 25. The Gizmo Header (P201) with the SCI and SPI Micro Server Signals

Using the EIA-232 Interface (SCI)

To use the EIA-232 interface on a Mini EVB for a ShortStack application, set the following jumpers as described below.

JP201

To enable the EIA-232 communications on a Mini EVB, mount all of the jumpers on the **JP201** header, as shown in **Figure 26**. In the figure, the jumpers for the FT 3120 and 3150 boards are on the left, and the jumpers for the PL 3120, 3150, and 3170 boards are on the right.

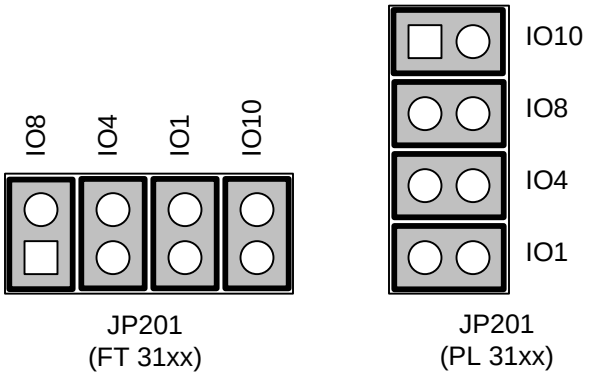

Figure 26. Mini EVB EIA-232 Enable Jumpers (JP201)

The MAX3387E RS-232 transceiver that is used on the Mini EVBs is configured to enter autoshutdown mode after inactivity of approximately 30

seconds. For applications that use high link-layer bit rates, the time required for the transceiver to become fully active (approximately 100 µs) might be long enough to cause a framing error on the serial link-layer signals.

Recommendation: To prevent the MAX3387E RS-232 transceiver from entering autoshutdown mode, your serial driver should briefly toggle the ShortStack Micro Server's HRDY~ signal every 10 to 20 seconds during periods of idleness. This toggle causes the MAX3387E transceiver to detect transmission activity and not enter autoshutdown mode. Alternatively, you can connect the FORCEON pin (pin 11) either to V_{DD5} or to the V_L pin (pin 15).

JP203

Mount the IO0 jumper as shown in **Figure 27** to connect the CTS~ signal to the MAX3387E RS-232 transceiver. The figure also shows the IO5 and IO6 jumpers configured to specify a 19 200 bit rate on a Mini FX PL 3170 Evaluation Board for a 10 MHz Micro Server.

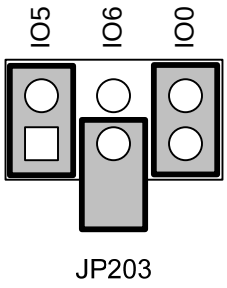

Figure 27. PL 3170 Mini FX ShortStack Enable Jumper (JP203)

To set the link-layer bit rate for the Micro Server, determine the correct bit rate for your device according to *Selecting the Link-Layer Bit Rate* on page 68, and then mount the Mini EVB's **JP203** jumpers appropriately to match the correct settings for the IO5 and IO6 pins on the Smart Transceiver. See either *Setting the SCI Bit Rate* on page 73 for the correct settings for the IO5 and IO6 pins.

The EIA-232 interface requires a null-modem cable for the D-SUB9 EIA-232 connector (**J201**) on the Mini EVB. To define the null-modem EIA-232 interface, use the pin connections listed in **Table 9**. Keep the total cable length to a maximum of 24 inches (0.6 meters).

Table 9. EIA-232 Header to D-SUB9 Connector Pin Connections

D-SUB9 Connector Pin	Micro Server SCI Signal
1	N/A
2	TXD
3	RXD
4	HRDY~
5	GND

6	N/A
7	RTS~
8	CTS~
9	N/A

Working with Pyxos FT EV Pilot Evaluation Boards

For SCI link-layer connections, you can use an Echelon Pyxos FT EV Pilot EVB as an Atmel ARM7 host processor board, either with the on-board FT 3150 Smart Transceiver for the Micro Server, or with an FT 5000 EVB or Mini EVB (as described above).

For SPI link-layer connections, you cannot use the Pyxos FT EV Pilot EVB, but you can use another ARM7 host processor board, such as an Atmel AT91SAM7S-EK evaluation board.

Regardless of which ARM7 host EVB you use (the Pyxos Pilot or the Atmel evaluation board), you must create a custom connection cable for either the **P201** Gizmo header or the **J201** EIA-232 connector. For rapid prototyping, you might consider using short 0.25" (0.635 mm) square socket test leads for these connections.

To use the Pyxos FT EV Pilot EVB with an external Micro Server board (such as an FT 5000 EVB or Mini EVB), rather than using the on-board FT 3150 Smart Transceiver, you must disconnect the ARM7 processor's GPIO signals from the FT 3150 Smart Transceiver's SCI link-layer signals. To disconnect these signals, dismount all jumpers from header **JP512** (near the flash memory socket on the left side of the board) on the Pyxos FT EV Pilot EVB, as shown in **Figure 28** on page 55. The labels for the header represent the ARM7 pin names.

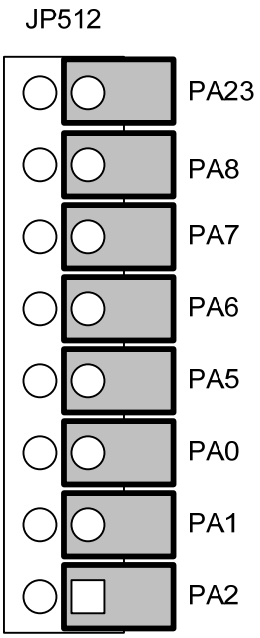

Figure 28. Pyxos FT EV Pilot EVB ShortStack Header (JP512)

For your custom connection cable, you can use either the rightmost side (ARM7-facing side) of the **JP512** header or you can use the relevant pins from the **JP505** header to connect the Micro Server SCI signals to the ARM7 host. **Table 10** lists the pin correspondences between the ARM7 host processor and the ShortStack Micro Server (for hardware pin numbers, see the specific chip's data sheet or schematic). **Figure 29** on page 56 shows the **JP505** header.

Table 10. ARM7 to Micro Server Pin Connections for the SCI Interface

ARM7 Pin Name	Micro Server Pin Name
PA8	IO0 (CTS~)
PA2	IO1 (HRDY~)
N/A	IO3 (SPI/SCI~) → Tie to GND for SCI
PA7	IO4 (RTS~)
PA0	IO5 (SBRB0)
PA1	IO6 (SBRB1)
PA6	IO8 (RXD)
PA5	IO10 (TXD)
PA23	Reset (RST~)

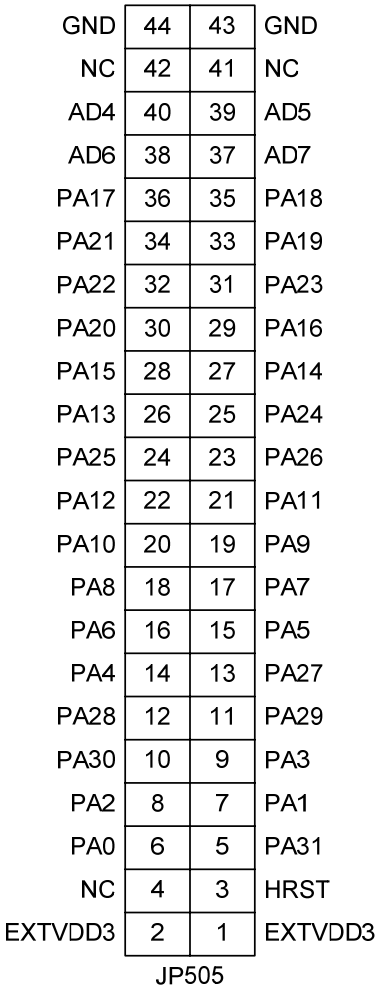

Figure 29. Pyxos FT EV Pilot Host Power and I/O Connector Header (JP505)

Table 11 summarizes the connections between the FT 5000 EVB and the Pyxos FT EV Pilot EVB for the SCI link-layer interface using the Gizmo interface. The notation "P201:4" represents pin 4 of header **P201**.

Table 11. SCI Connections for the Pyxos Pilot EVB and the FT 5000 EVB

Signal Name	FT 5000 EVB Pins	Pyxos Pilot EVB Pins
CTS~	P201:0	JP512:13 (PA8)
RTS~	P201:4	JP512:11 (PA7)
RXD	P201:8	JP512:9 (PA6)
TXD	P201:10	JP512:7 (PA5)
GND	P201:20	JP505:44 (GND)

If you use an external EVB for the ShortStack Micro Server (such as the FT 5000 EVB), do not connect the Micro Server's IO5 and IO6 lines (SBRB0 and SBRB1) to the ARM7 processor's PA0 and PA1 pins. By default, the ShortStack FX

ARM7 example port's serial driver sets the SCI serial bit rate to 76800 bps for a 10 MHz FT 3150 Smart Transceiver; connecting these pins can create a mismatch in the expected bit rate for the Micro Server. That is, set the Micro Server bit rate through the jumpers on the FT 5000 EVB, rather than through the serial driver.

Important: The SCI link layer requires pull-up resistors for the communications lines (see *Serial Communication Lines* on page 66):

- If you use the Pyxos FT EV Pilot EVB's onboard FT 3150 Smart Transceiver with the ARM7 host processor, the jumper settings for the **JP512** header not only connect the ARM7's GPIO lines to the Smart Transceiver, but also provide the necessary pull-ups.
- If you connect the Pyxos FT EV Pilot EVB to an FT 5000 EVB, the needed pull-up resistors are already present if you connect the boards as described in *Working with FT 5000 EVB Evaluation Boards* on page 39.
- If you connect the Pyxos FT EV Pilot EVB to other hardware (a Mini EVB or your custom hardware), be sure to supply the pull-up and pull-down resistors as necessary (see your hardware's schematics or other documentation).

See the *ShortStack FX ARM7 Example Port User's Guide* for more information about the Pyxos FT EV Pilot EVB and the ARM7 host processor.

ShortStack Device Initialization

A ShortStack device performs the following tasks during initialization:

1. Upon power-up or return from reset, the Micro Server performs initial health checks, and initializes itself.

   Depending on the chosen hardware and the Micro Server's properties, this step can take several tens of seconds the first time the Micro Server initializes; however, this step completes almost instantly for all subsequent resets.

   The Micro Server also enters quiet mode at the end of this step, unless an application has previously been registered with this Micro Server.
2. While the Micro Server performs initialization step 1, the host application runs its own local initialization code.
3. When the host application's initialization is complete, and its serial driver is ready to receive messages from the Micro Server, it must assert the HRDY~ signal. This assertion must occur before the Micro Server's watchdog timer expires (840 ms after reset for a Series 5000 device; 210 to 840 ms after reset for a Series 3100 device, depending on the external clock rate). For fast host processors, you can tie the HRDY~ signal low, so that the Micro Server assumes that the host is always ready to receive messages. However, your host-side circuitry must ensure that the HRDY~ signal is reliably high (deasserted) during power-up and host initialization.
4. When the Micro Server's initialization is complete and the host signals its readiness to receive packets (by asserting the HRDY~ signal), the Micro Server sends an uplink reset message. This message includes

information about the Micro Server, including its current state, last known error condition, and its initialization state.

The ShortStack host application must register with the Micro Server to complete the initialization of the ShortStack device (the Micro Server together with the host processor) before it can communicate as a LONWORKS device on a LONWORKS network. Before the application is correctly registered with the Micro Server, the Micro Server is in quiet mode and does not respond to network events and appears inoperative to the network. In addition, after you load a new Micro Server image, the first initialization of the Micro Server, together with the initialization of the host application and its registration with the Micro Server, can take up to one minute to complete. Subsequent initializations complete much more quickly.

The ShortStack host application sends registration information to the ShortStack Micro Server on startup. The registration information includes the device's program ID, communication parameters, network variable configuration data, and miscellaneous preferences.

The application must send this registration data whenever the Micro Server reports a reset and indicates that no application is registered. That is, the host application should re-run its **LonInit()** function. See *Performing an Initial Micro Server Health Check* on page 82 for more information.

After the registration data has been accepted and successfully processed by the Micro Server, the Micro Server leaves quiet mode, and thus allows the device to communicate as a LONWORKS device on a LONWORKS network.

See *Initializing the ShortStack Device* on page 169 for more information about the initialization ShortStack LonTalk Compact API function, and see *Running the LonTalk Interface Developer* on page 146 for more information about generating the self-identification, self-documentation, and initialization data.

Using the ShortStack Micro Server Key

Each ShortStack Micro Server firmware image has a version number and a key value that identifies it. The key value identifies the Micro Server in terms of its Smart Transceiver chip type (FT or PL, 3120, 3150, 3170, or 5000), its clock rate, whether it supports ISI, and its channel type (FT or PL). The key value is a 16-bit number that is reported to the host whenever the Micro Server sends a reset notification; **Table 12** defines the bit values that comprise the key for standard Micro Servers.

Table 12. Micro Server Key Bit Values

Micro Server Type	Bit Values						Key Value
	Custom	Revision	Chip Type	Clock Speed	ISI Support	Channel Type	
FT 3120 @ 10 MHz	0	0001	0000	001	0	000	0x0010
FT 3120 @ 20 MHz	0	0001	0000	010	0	000	0x0020

Micro Server Type	Bit Values						Key Value
	Custom	Revision	Chip Type	Clock Speed	ISI Support	Channel Type	
FT 3120 @ 40 MHz	0	0001	0000	011	0	000	0x0030
FT 3150 @ 10 MHz	0	0001	0001	001	0	000	0x0090
FT 3150 @ 10 MHz	0	0001	0001	001	1	000	0x0098
PL 3120 @ 10 MHz	0	0001	0010	001	0	001	0x0111
PL 3150 @ 10 MHz	0	0001	0011	001	0	001	0x0191
PL 3150 @ 10 MHz	0	0001	0011	001	1	001	0x0199
PL 3170 @ 10 MHz	0	0001	0100	001	1	001	0x0A19
FT 5000 ES	0	0000	0101	011	1	000	0x02B8
FT 5000	0	0001	0101	011	1	000	0x0AB8

In the table:

- *Custom* is a one-bit field that identifies whether the Micro Server is a standard Echelon-supplied Micro Server or a custom Micro Server. 0b0[2] indicates standard; 0b1 indicates custom.
- *Revision* is a four-bit field that can distinguish otherwise-identical Micro Servers:
    - 0b0000 indicates the initial version.
    - 0b0001 indicates the first revision level.
- *Chip type* is a four-bit field that identifies the chip type:
    - 0b0000 indicates an FT 3120 Smart Transceiver
    - 0b0001 indicates an FT 3150 Smart Transceiver
    - 0b0010 indicates a PL 3120 Smart Transceiver
    - 0b0011 indicates a PL 3150 Smart Transceiver
    - 0b0100 indicates a PL 3170 Smart Transceiver

[2] "0b0" represents a binary literal or constant value of 0 (zero).

    - 0b0101 indicates an FT 5000 Smart Transceiver
    - 0b0110 indicates a Neuron 5000 Processor[3]
- *Clock speed* is a three-bit field that identifies the clock speed for the Smart Transceiver or Neuron Processor[4]:
    - 0b000 indicates 5 MHz
    - 0b001 indicates 10 MHz
    - 0b010 indicates 20 MHz
    - 0b011 indicates 40 MHz
    - 0b100 indicates 80 MHz
    - 0b101 indicates 160 MHz
- *ISI support* is a one-bit field that identifies whether the Micro Server supports Interoperable Self-Installation (ISI):
    - 0b0 indicates no ISI support
    - 0b1 indicates ISI support.
- *Channel type* is a three-bit field that identifies the LONWORKS network type:
    - 0b000 indicates a TP/FT-10 channel
    - 0b001 indicates a PL-20C channel
    - 0b010 indicates a PL-20N channel
    - 0b111 indicates all other channel types

A ShortStack host application could use this key value to determine whether its Micro Server is running with an FT or PL transceiver, and perform an appropriate initialization for that tranceiver type. Alternatively, a host application could use this key to bypass initialization for ISI for a Micro Server that does not support ISI.

If you develop a custom Micro Server, you can set the key to any value that has meaning for your application, however, you must set the most-significant bit to 1 to signify that the key applies to a custom Micro Server. The key is defined in the [*ShortStack*]**\Custom MicroServer\MicroServer.h** header file:

```
#define MICRO_SERVER_KEY 0x8000ul
```

Thus, the key is a 16-bit number as defined in the context of Neuron C's unsigned long type.

---

[3] The Neuron 5000 Processor is not supported by the standard Micro Servers that are included with the ShortStack FX Developer's Kit. You must create a custom Micro Server to support a Neuron 5000 Processor.

[4] For a Series 3100 Smart Transceiver, this value is the external crystal or oscillator frequency value. For an FT 5000 Smart Transceiver or Neuron 5000 Processor, this value is twice its system clock value (from the device's hardware template), to represent an equivalent Series 3100 clock rate.

4

Selecting the Host Processor

This chapter describes considerations for selecting a new host processor for a ShortStack device, and for evaluating an existing host processor. It also describes considerations for selecting the host programming environment.

Selecting a Host Processor

For most applications, the choice of the host processor is determined by the overall needs of the application, rather than the needs of the ShortStack Micro Server. Other considerations for choosing the host processor include prior experience with the processor or architecture, cost, performance, memory support, power requirements, I/O support, and availability of development tools.

The Micro Server has few requirements for the host processor. The following sections describe considerations that can help you choose a host processor or determine the suitability of your current host processor.

Serial Communications

The host processor must be able to connect to the ShortStack Micro Server through either the four (or five) line Serial Communications Interface (SCI) or the six (or seven) line Serial Peripheral Interface (SPI). In addition, the host processor's implementation of the serial interface must support at least one of the bit rates listed in *Setting the SCI Bit Rate* on page 73 or *Setting the SPI Bit Rate* on page 77.

An existing serial driver, which might be available as part of an embedded operating system's services, must allow for flow control that complies with the ShortStack link layer protocol. Alternatively, you must be able to supply your own serial driver that implements the required protocol. See *SCI Interface* on page 71 or *SPI Interface* on page 76 for information about the required protocol.

If your application uses SPI, the host processor must support SPI Slave mode, because the Micro Server always operates as the SPI Master.

Both the SCI and SPI interfaces provide a host ready (HRDY~) signal. Your application can use this signal to prevent new link layer uplink transfers to the host processor, but because Micro Server has limited buffering capabilities, the application should only assert the HRDY~ signal briefly. A typical driver implementation deasserts this signal only briefly while it enqueues a received packet, to protect the temporarily busy receiver routine from an input data buffer overflow. The host must ensure that this signal this deasserted reliably through the entire power-up and initialization phase, until the host asserts it after the host application and serial driver are fully initialized and ready to exchange link-layer data.

If your ShortStack application makes no requirements for which interface to use, you should consider using the SCI interface. The SCI interface requires fewer I/O lines, and is more standardized, which allows for easier possible future transition to a different host platform. In addition, the ShortStack SCI driver is easier to port because of its simpler link-layer protocol.

Byte Orientation

Unless your application requires a processor with a little-endian (least significant byte at low address) architecture, you should consider using a processor with a big-endian (most significant byte at low address) architecture for a ShortStack device. Network data in a LONWORKS network uses big-endian byte orientation.

A big-endian host processor does not need to change byte orientation, and thus requires fewer processing instructions and machine cycles to access network data. If you use a little-endian host processor, you might need to implement code for byte re-ordering on the uplink and downlink. Some processor architectures, such as that used in the ARM processor family, are bi-endian, and feature switchable "endianness".

The ShortStack LonTalk Compact API and application framework provide utilities to handle the byte orientation correctly.

Processing Power

The processing power required by the ShortStack host processor is generally determined by the application's control algorithm. ShortStack has minimal processing requirements.

However, the ShortStack LonTalk Compact API requires frequent periodic servicing through the **LonEventHandler()** API function (see *Periodically Calling the Event Handler* on page 170). Different host processors take different amounts of time to run this function. The time required to run this function also depends on the incoming and outgoing network traffic.

Most modern microprocessors can run this function without impacting the application's control algorithm. However, a device with a very demanding control algorithm, or a device with a performance-limited host processor might need additional RAM to buffer link-layer packets to avoid loss of data.

Volatile Memory

Although every application is different, a general ShortStack device requires about 800 bytes of RAM (as well as approximately 4 to 6 KB of memory for the application program plus application framework [serial driver, ShortStack LonTalk Compact API, and so on]). See *API Memory Requirements* on page 290 for a description of the memory requirements for the ShortStack LonTalk Compact API and optional APIs.

If your application uses non-interoperable messages, which can include larger payload data and can require larger buffers or additional buffers in the host application, the RAM requirement could increase significantly.

Modifiable Non-Volatile Memory

Although the ShortStack LonTalk Compact API does not require modifiable non-volatile memory, most interoperable ShortStack devices require a small amount of modifiable non-volatile data storage. This data includes configuration property values, which control and configure the interoperability and networking aspects of the ShortStack device.

The total amount of such data depends on your application, and can range from zero bytes to several kilobytes. Many simple interoperable devices require no more than a few hundred bytes of modifiable non-volatile memory. Devices typically use flash or EEPROM memory to store such data, but ShortStack makes no requirement for the type of memory.

How the application accesses this memory depends on the application's requirements. The ShortStack LonTalk Compact API provides tools and code that can help manage non-volatile memory. See *Providing Persistent Storage for Non-Volatile Data* on page 192 for more information, including recommendations and considerations for handling non-volatile data.

Compiler and Application Programming Language

The ShortStack Developer's Kit provides the ShortStack LonTalk Compact API and application framework as portable ANSI C source code. Thus, a standard ANSI C (or C++) compiler for application development is appropriate. Other development tools and languages are possible, but you must then port the driver, API, and application framework to the other language.

The ShortStack LonTalk Compact API and application framework can be used with most ANSI C compilers with little or no changes. The **LonPlatform.h** file provides a set of common definitions for various compilers.

The ShortStack LonTalk Compact API and application framework use many data structures and unions, some of which are deeply nested types. All of these structures are based on byte-sized entities (and combinations of multiple single-byte entities, rather than multi-byte entities), so the application compiler must be able to generate the exact memory image of these structures and unions without inserting any padding bytes. By exclusively using single-byte entities, the ShortStack LonTalk Compact API allows most compilers to be used with a ShortStack FX application.

See *Porting the ShortStack LonTalk Compact API* on page 109 for more information, including considerations for porting a ShortStack application to a host development environment and embedded operating system.

Selecting the Application Development Environment

The ShortStack LonTalk Compact API and framework have no requirement for an embedded operating system, and use only a few basic routines from the standard ANSI C toolkit, such as the **memcpy()** or **memset()** functions.

Many simple ShortStack devices do not include an embedded operating system. These devices typically call the ShortStack LonTalk Compact API from the application's main loop.

Devices that use an embedded operating system can use dedicated threads, tasks, or processes to call and process data from the ShortStack LonTalk Compact API. Other solutions can call and process data from the API from a timer-based interrupt service handler routine.

Although the ShortStack LonTalk Compact API and application framework support all of these approaches, the ShortStack model is single-threaded and not re-entrant. An application that uses a multi-tasking (or multi-threaded) or interrupt-driven ShortStack LonTalk Compact API must ensure that all ShortStack LonTalk Compact API access is within a single thread (or task or interrupt context).

See Appendix C, *ShortStack LonTalk Compact API*, on page 287, for additional considerations and recommendations regarding threading and execution context.

5

Designing the Hardware Interface

This chapter describes what you need to design the hardware interface between your ShortStack host processor and the ShortStack Micro Server.

Overview of the Hardware Interface

The hardware interface for a ShortStack Micro Server consists of the 11 or 12 I/O-pin interface of an Echelon Smart Transceiver. However, a ShortStack Micro Server does not use all 11 or 12 pins. The ShortStack Micro Server supports two serial interfaces for communications with the host processor: the Serial Communications Interface (SCI) and the Serial Peripheral Interface (SPI). One I/O pin selects the serial interface, two pins set the interface bit rate, and five to seven I/O pins comprise the interface. One pin (IO9) is optionally available to the host processor, and the remaining I/O pins are not used.

This chapter describes the hardware interface, including the requirement for pull-up resistors, checking the status of the optional IO9 pin, selecting a minimum communications interface bit rate, considerations for host latency, specifying the SCI interface, specifying the SPI interface, and how to perform an initial health check of the Micro Server.

Reliability

A ShortStack Micro Server considers the serial link reliable, similar to other serial interfaces that are commonly used within computing equipment and embedded devices, such as an inter-integrated circuit (I^2C) bus connection to a serial EEPROM device.

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

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

The example applications contain example implementations of the link layer driver, including examples and recommendations for time-out guards within the various states of that driver. See the *ShortStack FX ARM7 Example Port User's Guide* for more information about the ARM7 example applications. The optional local utility API functions also include health-check features, such as the facility to 'ping' the Micro Server or to echo data across the serial link layer, to help your application to prevent and detect unrecoverable link-layer errors.

Serial Communication Lines

For both serial interfaces (SCI and SPI), you must add 10 kΩ pull-up resistors to all communication lines between the host processor and the ShortStack Micro Server (including those marked as N/A in **Table 13** on page 72 and **Table 15** on page 77, and not connected to the host processor). These pull-up resistors

prevent invalid transactions on start-up and reset of the host processor or the Micro Server. Without a pull-up resistor, certain I/O pins can revert to a floating state, which can cause unpredictable results.

If your link-layer driver does not use the HRDY~ signal, you can tie it to GND. However, it is recommended that the host drive the HRDY~ signal, even if the host processor is fast and always ready to receive uplink data, to assist with a synchronized start-up after power-up or reset.

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

The RESET~ Pin

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

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

For most ShortStack devices, you should not connect the two reset pins to each other. It is usually better for the Micro Server and the host application to be able to reset independently. For example, when the Micro Server encounters an error that causes a reset, it logs the reset cause (see *Querying the Error Log* on page 187); if the host processor resets the Micro Server directly, possibly before the Micro Server can detect and log the error, your application cannot query the Micro Server's error log after the reset to identify the problem that caused the reset. The Micro Server also resets as part of the normal process of integrating the device within a network; there is normally no need for the host application to reset at the same time.

In addition, the host processor should not reset the Micro Server while the Micro Server is starting up (that is, before the Micro Server sends the uplink reset message, **LonResetNotification**, to the host processor).

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

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

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

Configuring the GPIO pin as either open drain or tri-state ensures that the GPIO pin is in a high-impedance state until it is driven low. Using a Schottky diode is preferable to using a regular diode because a Schottky diode has a low forward voltage drop (typically, 0.15 to 0.45 V), whereas a regular diode has a much higher voltage drop (typically, 0.7 V), that is, the Schottky diode ensures that the voltage drop is low enough to ensure a logic-low signal.

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

Using the IO9 Pin

Neither of the standard serial interfaces for a ShortStack Micro Server uses the IO9 pin of the Smart Transceiver chip. However, an application can read the static input signal that is available to the IO9 pin.

To make this signal available to the application, the Micro Server includes the following information in each uplink reset notification:

- Whether the IO9 input signal is available for application use (always TRUE for a ShortStack FX Micro Server)
- The logic state of the IO9 static input

Applications can use this information for automatic configuration of the Micro Server. For example, your ShortStack device could use a jumper or configuration switch to select, or deselect, the comité européen de normalisation electrotechnique[5] (CENELEC) media access protocol for power line use, thus potentially allowing the device to use a single application image for use in CENELEC member states as well as in countries that are not governed by the CENELEC committee.

Selecting the Link-Layer Bit Rate

The minimum bit rate for the serial link between the ShortStack Micro Server and the host processor is most directly determined by the expected number of packets per second, the type of packets, and the size of the packets. Another factor that can significantly influence the required bit rate is support for explicit

[5] European Committee for Electrotechnical Standardization

addressing, an optional feature that the ShortStack application can enable and disable.

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

- ShortStack Micro Server external clock frequency
  - 10 MHz or higher for TP/FT-10 devices (for Series 5000 devices, specify a minimum 5 MHz system clock rate)
  - 5 MHz or higher for power-line devices
- Bit rate
  - 38 400 bps or higher for TP/FT-10 devices
  - 9600 bps or higher for power-line devices

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

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

where:

- The constant 5 represents general communications overhead
- P_{type} is the packet-type overhead, and has one of the following values:
  - 3 for network-variable messages
  - 1 for application messages
- EA is the explicit-addressing overhead, and has one of the following values:
  - 0 for no explicit-addressing support
  - 11 for explicit-addressing support enabled
- P_{size} is the packet size of the payload, and has one of the following values:
  - sizeof(network_variable)
  - sizeof(message_length)
- $BPT_{Interface}$ represents data transfer overhead for the serial interface, and has one of the following values:
  - 1 bit per transfer for SPI
  - 10 bits per transfer for SCI
- $PPS_{\exp}$ is the expected packet-per-second throughput value

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

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

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

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

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

FT 3150 and PL 3150-based Micro Servers using off-chip flash memory are limited to 10 MHz operation, but faster operation might be possible with FT 3120 or FT 3150-based Smart Transceivers. FT 5000 Smart Transceivers can operate with up to an 80 MHz system clock rate, but the standard Micro Server for the FT 5000 uses a 20 MHz system clock, making its performance equivalent to that of an FT 3120 Smart Transceiver with an external 40 MHz crystal. The selection of the 20 MHz clock rate is a compromise between processing performance and power consumption.

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

Host Latency Considerations

The processing time required by the host processor for a ShortStack Micro Server can have a significant impact on link-layer transit time for network communications and on the total duration of network transactions. This impact is the host latency for the ShortStack application.

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

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

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

where:

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

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

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

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

SCI Interface

The ShortStack Serial Communications Interface (SCI) is a half-duplex asynchronous serial interface between the ShortStack Micro Server and the host processor. The communications format is:

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

The SCI link-layer interface uses two serial data lines: RXD (receive data) and TXD (transmit data). The signal directions are from the point of view of the Micro Server. An *uplink* transaction describes data exchange from the Micro Server to the host processor, and uses the TXD line. A *downlink* transaction refers to data exchange from host processor to the Micro Server, and uses the RXD line.

The SCI interface includes three flow-control lines: the RTS~ (request to send) signal that informs the Micro Server of a pending downlink, the CTS~ (clear to send) signal that allows a downlink transfer to begin, and an optional HRDY~

(host ready) signal that can be used to temporarily prevent uplink transfers. These three signals are all active low.

The interface also includes two bit-rate selection signals and an interface type selection signal. These signals can be connected to the host processor, but do not need to be. However, if the host processor does not control the bit-rate selection signals, you must ensure that the host processor and the Micro Server run at the same SCI bit rate.

ShortStack Micro Server I/O Pin Assignments for SCI

A ShortStack Micro Server has 11 or 12 I/O pins that control the configuration of the Micro Server and provide the interface to the host processor. The IO3 input pin selects the serial interface: SCI or SPI. The serial interface also determines the usage of the other I/O pins. **Table 13** summarizes these pin assignments for the SCI interface.

Recommendation: If your host processor can support both the SCI and SPI interfaces, use the SCI interface because it is typically faster and easier to implement, both in hardware and software.

Table 13. ShortStack Micro Server Pin Assignments for the SCI Interface

Smart Transceiver Pin	Signal Name	Direction
IO0	CTS~	Output
IO1	HRDY~	Input
IO2	N/A	No connection
IO3	SPI/SCI~	Input (tie to GND for SCI)
IO4	RTS~	Input
IO5	Serial Bit Rate Bit 0 (SBRB0; LSB)	Input
IO6	Serial Bit Rate Bit 1 (SBRB1; MSB)	Input
IO7	N/A	No connection
IO8	RXD	Input
IO9	N/A	No connection (but see *Using the IO9 Pin* on page 68)
IO10	TXD	Output
IO11	N/A	No connection

Notes:

- Signal direction is from the point of view of the Smart Transceiver (Micro Server).
- N/A = Not applicable.

Setting the SCI Bit Rate

You select the SCI interface by setting the ShortStack Micro Server's IO3 input pin to logic 0 (ground). The settings for pins IO5 and IO6 determine the SCI serial bit rate, as listed in **Table 14**. The rates are listed as bits per second; the values are also approximate and rounded to the nearest 100 bits per second.

Table 14. SCI Serial Bit Rates

Series 3100 External Clock	Series 5000 System Clock	SBR1 (IO6)	SBR0 (IO5)	SBR1 (IO6)	SBR0 (IO5)	SBR1 (IO6)	SBR0 (IO5)	SBR1 (IO6)	SBR0 (IO5)
		GND	GND	GND	V_{DD}	V_{DD}	GND	V_{DD}	V_{DD}
5 MHz	—	38400		19200		9600		4800	
10 MHz	5 MHz	76800		38400		19200		9600	
20 MHz	10 MHz	153600		76800		38400		19200	
40 MHz	20 MHz	302100		153600		76800		38400	
—	40 MHz	604200		302100		153600		76800	
—	80 MHz	1208400		604200		302100		153600	

Note: Specify the Series 5000 system clock rate in the hardware template for a custom Micro Server. The standard Series 5000 Micro Server images use a 20 MHz system clock. The external crystal clock frequency for a Series 5000 chip is 10 MHz.

The standard Series 3100 ShortStack Micro Server images support only the 10 MHz, 20 MHz, and 40 MHz clock rates; you need to create a custom Micro Server image to use the 5 MHz clock rates listed in **Table 14**. The standard Series 5000 ShortStack Micro Server images support only the 20 MHz system clock rate; you need to create a custom Micro Server image to use one of the other system clock rates. See *Custom Micro Servers* on page 241 for more information about creating a custom Micro Server image.

Important: The PL 3170 Smart Transceiver supports the 38400 bit rate only.

Note that some of the higher bit rates listed in **Table 14** are not standard SCI bit rates, therefore, some host processors or UART/USART implementations might not be able to communicate at the specific rate listed in the table. In this case, modify the UART/USART setting to the closest bit rate to the desired value in the table, or modify the Micro Server's bit rate setting.

Important: For implementations with higher bit rates, be sure that the link-layer hardware provides low impedance and correct termination. Also consider adding extra ground connections between the data signals. If a high-bit rate application presents link-layer problems, be sure to analyze the waveform with an oscilloscope to be sure it has the correct shape before proceding to other debugging procedures.

SCI Communications Interface

The SCI communications interface shown in **Figure 30** on page 75 is implemented with the following inputs and outputs:

- Interface Selector (SPI/SCI~): Tied to GND to specify the SCI interface.
- Request to Send (RTS~): When asserted, indicates that the host processor has data to send. The serial driver asserts this signal low if the CTS~ signal is deasserted (high), and waits for the Micro Server to assert CTS~.
- Clear to Send (CTS~): When asserted, informs the host processor that Micro Server is ready to receive data from the serial driver. Set by the Micro Server after the host has asserted RTS~. The Micro Server keeps CTS~ asserted until it receives the expected number of bytes. The host must deassert RTS~ after the CTS~ acknowledgement has been received, and must start transmitting the related data with minimal delay (under 400 ms for a 10 MHz Series 3100 Micro Server; under 100 ms for a 40 MHz Series 3100 Micro Server; under 25 ms for an 80 MHz Series 5000 Micro Server).
- Host Ready (HRDY~): When deasserted, indicates that the host processor is temporarily not able to accept data transfers from the Micro Server. This signal is optional; if your application does not use this signal, you must tie it low so that it is continually asserted (to specify that the host is always ready to accept data transfers). See *Serial Communications* on page 62 for additional considerations for the HRDY~ signal. Typical host applications deassert the HRDY~ signal in the following situations:
    - During power-up and initialization following a reset (until the serial driver is ready to receive data from the Micro Server)
    - When enqueuing received data, following a completed uplink transfer
- Receive Data (RXD): Transfers data from the host processor to the Micro Server.
- Transmit Data (TXD): Transfers data from the Micro Server to the host processor.
- Serial Bit Rate Bit 0 (SBRB0) and Serial Bit Rate Bit 1 (SBRB1): Together set the communications bit rate (see **Table 14** on page 73).

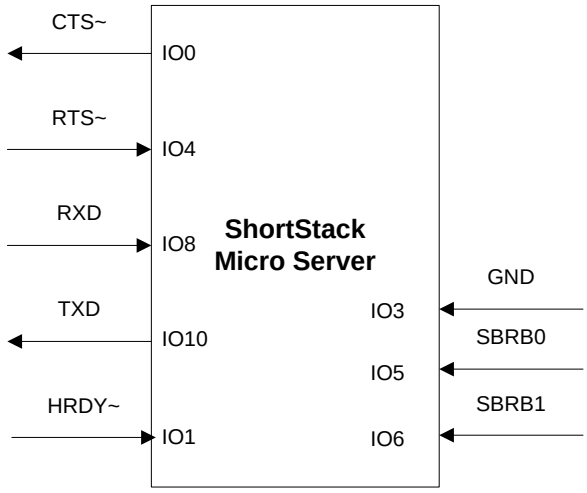

Figure 30. ShortStack SCI Communications Interface

SCI Micro Server to Host (Uplink) Control Flow

The host must assert the HRDY~ pin low to indicate that it is ready to receive data. Because the Micro Server has a limited set of buffers, the host processor should deassert the HRDY~ pin for only a short duration. A typical application deasserts the HRDY~ signal during its power-up and initial initialization following a reset, and after an uplink data packet has been completely received, while the packet data is enqueued for further processing, then reasserts the signal.

If your host processor is always able to receive data, you can hardwire the HRDY~ input low.

Figure 31 shows an example for the Micro Server to host SCI control flow, including the states of the various I/O pins.

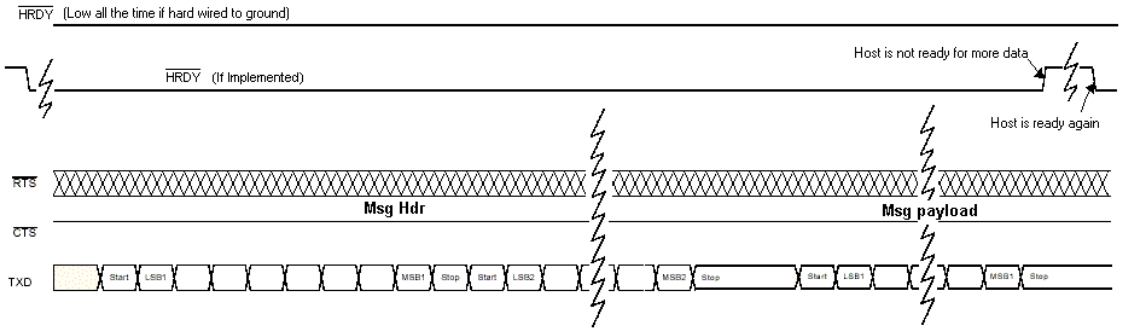

Figure 31. SCI Micro Server to Host Transfer Control Flow Diagram

SCI Host to Micro Server (Downlink) Control Flow

The Micro Server uses the CTS~ pin to enforce a half-duplex interface. Every downlink transfer must be guarded with a complete RTS~ / CTS~ handshake between the host processor and the Micro Server, by implementing the following simple protocol:

1. The serial link-layer driver awaits the completion of the previous transaction. That is, it monitors the CTS~ line and waits until the Micro Server has deasserted this signal.
2. The serial link-layer driver asserts the RTS~ line to indicate the availability of downlink data.
3. The driver awaits confirmation from the Micro Server, which it indicates by asserting the CTS~ line. Depending on the type of operation and the current availability of buffers within the Micro Server, the driver could wait for a significant amount of time. The driver should include a timeout guard that can accommodate this wait period, for example, a 60 second timeout guard should suffice for most applications, even though the CTS~ assertion will usually occur much sooner.
4. After the driver detects that the CTS~ line is asserted (low), it releases (deasserts) the RTS~ line.
5. The driver transmits the data.
6. After the Micro Server receives the number of bytes of data (indicated in the message header), it releases (deasserts) the CTS~ line.

See Chapter 6, *Creating a ShortStack Serial Driver*, on page 89, for more information about the serial driver.

Figure 32 shows an example for the host to Micro Server SCI control flow. The figure also shows the transfer of the two-byte header, followed by the payload.

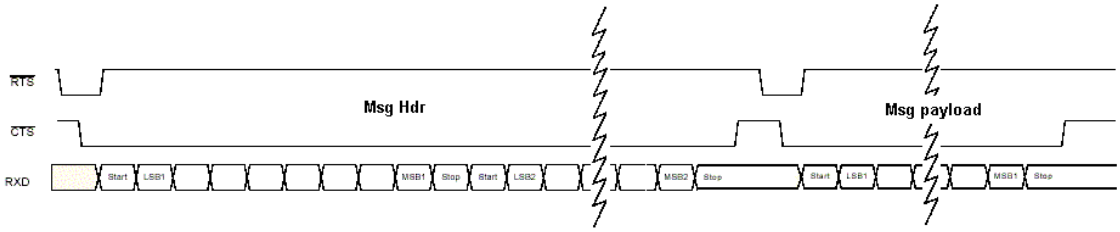

Figure 32. SCI Host to Micro Server Transfer Control Flow Diagram

SPI Interface

The ShortStack Serial Peripheral Interface (SPI) is a half-duplex synchronous serial interface between the ShortStack Micro Server and the host processor. The Micro Server is configured as the SPI master. The host processor is configured as the SPI slave.

If the host processor does not control the bit-rate selection signals, you must ensure that the host processor and the Micro Server run at the same SPI bit rate.

ShortStack Micro Server I/O Pin Assignments for SPI

A ShortStack Micro Server has 11 or 12 I/O pins that control the configuration of the Micro Server and provide the interface to the host processor. The IO3 input pin selects the serial interface: SCI or SPI. The serial interface also determines the usage of the other I/O pins. **Table 15** on page 77 summarizes these pin assignments for the SPI interface.

Recommendation: If your host processor can support both the SCI and SPI interfaces, use the SCI interface because it is typically faster and easier to implement, both in hardware and software.

Table 15. ShortStack Micro Server Pin Assignments for an SPI Interface

Smart Transceiver Pin	Signal Name	Direction
IO0	R/W~	Output
IO1	SCLK	Output
IO2	SS~	Output
IO3	SPI/SCI~	Input (tie to V_{DD} for SPI)
IO4	TREQ~	Input
IO5	Serial Bit Rate Bit 0 (SBRB0; LSB)	Input
IO6	Serial Bit Rate Bit 1 (SBRB1; MSB)	Input
IO7	MOSI	Output
IO8	MISO	Input
IO9	N/A	No connection (but see *Using the IO9 Pin* on page 68)
IO10	HRDY~	Input
IO11	N/A	No connection

Notes:

- Signal direction is from the point of view of the Smart Transceiver (Micro Server).
- N/A = Not applicable.

Setting the SPI Bit Rate

You select the SPI interface by setting the ShortStack Micro Server's IO3 input pin to logic 1 (V_{DD}) with a 10 kΩ pull-up resistor. The effective SPI bit rate is controlled by the SCLK output from the ShortStack Micro Server, but the desired bit rate can be preselected using the input signals SBRB0 and SBRB1 (IO5 and IO6). For the SPI interface, there are different bit rates for uplink transfers and downlink transfers. The settings for pins IO5 and IO6, and the resulting link layer bit rates, are listed in **Table 16** and **Table 17** on page 78. The rates in the tables are listed as bits per second; the values are also approximate and rounded to the nearest 100 bits per second.

Table 16. SPI Serial Bit Rates for Uplink

Series 3100 External Clock	Series 5000 System Clock	SBR1 (IO6)	SBR0 (IO5)	SBR1 (IO6)	SBR0 (IO5)	SBR1 (IO6)	SBR0 (IO5)	SBR1 (IO6)	SBR0 (IO5)
		GND	GND	GND	V_{DD}	V_{DD}	GND	V_{DD}	V_{DD}
5 MHz	—	29200		16600		10200		5100	
10 MHz	5 MHz	58300		33200		20300		10300	
20 MHz	10 MHz	116700		66300		40600		20500	
40 MHz	20 MHz	226600		129500		76700		40900	
—	40 MHz	453100		258900		153300		81900	
—	80 MHz	906200		517900		306600		163700	

Note: Specify the Series 5000 system clock rate in the hardware template for a custom Micro Server. The standard Series 5000 Micro Server images use a 20 MHz system clock. The external crystal clock frequency for a Series 5000 chip is 10 MHz.

Table 17. SPI Serial Bit Rates for Downlink

Series 3100 External Clock	Series 5000 System Clock	SBR1 (IO6)	SBR0 (IO5)	SBR1 (IO6)	SBR0 (IO5)	SBR1 (IO6)	SBR0 (IO5)	SBR1 (IO6)	SBR0 (IO5)
		GND	GND	GND	V_{DD}	V_{DD}	GND	V_{DD}	V_{DD}
5 MHz	—	21700		9200		4800		2900	
10 MHz	5 MHz	43400		18400		9700		5700	
20 MHz	10 MHz	86800		36800		19300		11500	
40 MHz	20 MHz	172600		73300		38600		22800	
—	40 MHz	345200		146700		77100		45600	
—	80 MHz	690500		293400		154300		91300	

The standard Series 3100 ShortStack Micro Server images support only the 10 MHz, 20 MHz, and 40 MHz clock rates; you need to create a custom Micro Server image to use the 5 MHz clock rates listed in **Table 16** and **Table 17**. The standard Series 5000 ShortStack Micro Server images support only the 20 MHz system clock rate; you need to create a custom Micro Server image to use the other clock rates listed in **Table 16** and **Table 17**. See *Custom Micro Servers* on page 241 for more information about creating a custom Micro Server image.

Note that some host processors or UART/USART implementations might not be able to process data at some of the higher bit rates listed in **Table 16** and **Table 17**. In this case, modify the UART/USART setting to the closest bit rate to the desired value in the table, or modify the Micro Server's bit rate setting. Most host processors should be able to process uplink data at up to 129500 bps and downlink data at up to 73300 bps.

Important: For implementations with higher bit rates, be sure that the link-layer hardware provides low impedance and correct termination. Also consider adding extra ground connections between the data signals. If a high-bit rate application presents link-layer problems, be sure to analyze the waveform with an oscilloscope to be sure it has the correct shape before proceding to other debugging procedures.

SPI Communications Interface

The SPI communications interface shown in **Figure 33** on page 80 is implemented with the following inputs and outputs:

- Interface Selector (SPI/SCI~): Tied to V_{DD} to specify the SPI interface.
- Host Ready (HRDY~): When deasserted, indicates that the host processor is temporarily not able to accept any data transfers from the Micro Server. This signal is optional; if your application does not use this signal, you must tie it low so that it is continually asserted (to specify that the host is always ready to accept data transfers). Typical host applications deassert the HRDY~ signal in the following situations:
    - During power-up and initialization following a reset (until the serial driver is ready to receive data from the Micro Server)
    - When enqueuing received data, following a completed uplink transfer
- Master Input Slave Output (MISO): Transmits control and data bytes from the host to the Micro Server. Data is presented at the falling clock edge, and sampled at the rising edge, MSB first, 8 bit.
- Master Output Slave Input (MOSI): Transmits control and data bytes from the Micro Server to the host. Data is presented at the falling clock edge, and sampled at the rising edge, MSB first, 8 bit.
- Serial Clock (SCLK): Provides a clock signal for all data transfers. Data is presented at the falling clock edge, and sampled at the rising edge.
- Slave Select (SS~): When asserted, selects the host SPI interface for SPI communication. This signal can be used to drive a (low-active) Enable signal on the host's SPI interface, when necessary.
- Transmit Request (TREQ~): When asserted, indicates that the host processor is ready to send data. The host asserts this signal low and waits for the Micro Server to assert the R/W~ pin.
- Read/Write (R/W~): Indicates which direction is active during a byte transfer (low indicates write). The R/W~ pin is low during a transfer from the Micro Server to the host (MOSI); the R/W~ pin is high during a transfer from the host to the Micro Server (MISO). See *SPI Host to Micro*

Server Control Flow (MISO) on page 81 for more information about the MISO flow.

- Serial Bit Rate Bit 0 (SBRB0) and Serial Bit Rate Bit 1 (SBRB1): Together set the communications bit rate.

The ShortStack SPI interface supports only one host processor on the bus; it does not support any other devices or microprocessors on the bus.

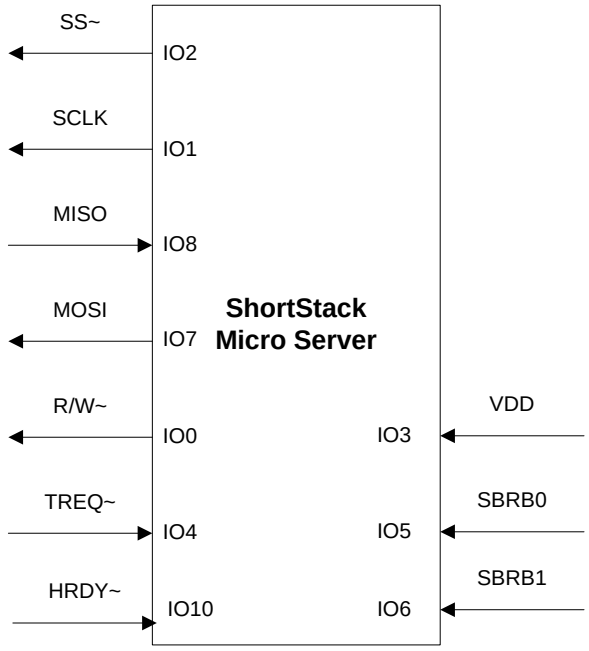

Figure 33. ShortStack SPI Communications Interface

SPI Micro Server to Host Control Flow (MOSI)

The host must assert the HRDY~ pin low to indicate that it is ready to receive data. Because the Micro Server has a limited set of buffers, the host processor should deassert the HRDY~ pin for only a short duration. A typical application deasserts the HRDY~ signal during its power-up and initial initialization following a reset, and after an uplink data packet has been completely received, while the packet data is enqueued for further processing, then reasserts the signal.

If your processor is always able to receive data, you can hardwire the HRDY~ input low.

Before sending a byte to the host, the Micro Server waits for the HRDY~ pin to be asserted low, then it sets the R/W~ pin low to indicate the direction of the data transfer. The Micro Server presents data on each falling edge of the SCLK pin; the host samples the data on each rising edge.

During MOSI transmissions, the MISO pin is ignored, and any data transferred to the Micro Server during this time is discarded. The SCLK period and duty cycle can vary during MISO and MOSI transmissions; the SCLK signal should not be used for any other purpose than ShortStack SPI interface data transfers.

Figure 34 on page 81 shows an example for the Micro Server to host SPI control flow.

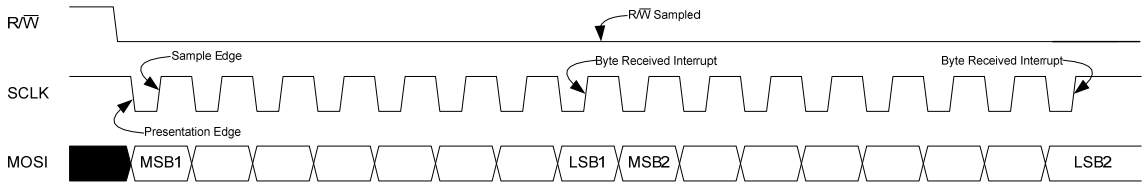

Figure 34. SPI Micro Server to Host (MOSI) Transfer Control Flow Diagram

SPI Host to Micro Server Control Flow (MISO)

Because the Micro Server is the SPI master, the host processor loads the first byte to be transmitted and asserts the TREQ~ pin. Asserting the TREQ~ pin causes the Micro Server to start the data transfer by driving the SCLK signal. Loading the data byte before asserting the TREQ~ pin ensures that:

- The data is transmitted as soon as the Micro Server begins sending a clock signal (the SCLK signal)
- The data is sampled on the rising edge of the SCLK signal

After the byte-received interrupt in the host's SPI status register is set, the host tests the R/W~ signal to determine if the transmission was successful. If the R/W~ pin is low (indicating a write operation by the Micro Server), the host must save the incoming byte as part of an uplink transfer and retry transmission until the R/W~ pin is high. When the host attempts to write data while the Micro Server is already writing data, this condition is known as a *write collision*.

After the host samples the R/W~ line and it is still high after the transfer of the first byte, it immediately de-asserts the TREQ~ pin before it loads the second byte of the burst transfer into its SPI transmission data register.

Because the host samples the R/W~ signal between the transmission of the first and second byte, the minimum length for a transfer in either direction is two bytes. This requirement is inherently met by the ShortStack SPI interface message structure because each link layer packet is two or more bytes in length. For some packets with only one byte of payload, an extra padding byte (zero) is added. In addition, the Micro Server keeps the R/W~ signal high for the duration of one byte; this extra time allows the host to confirm transfer direction.

The Micro Server samples data on the rising edge of the SCLK signal. The host must ensure that it presents data on the falling edge of the SCLK signal, because the SCLK signal is high between bytes (idle line). For most SPI implementations, this idle state is achieved by setting the Clock Polarity Bit (CPOL) to one and the Clock Phase Bit (CPHA) to one.

Figure 35 on page 82 shows an example for the host to Micro Server SPI control flow, without a write collision. The figure also shows the transfer of the two-byte header.

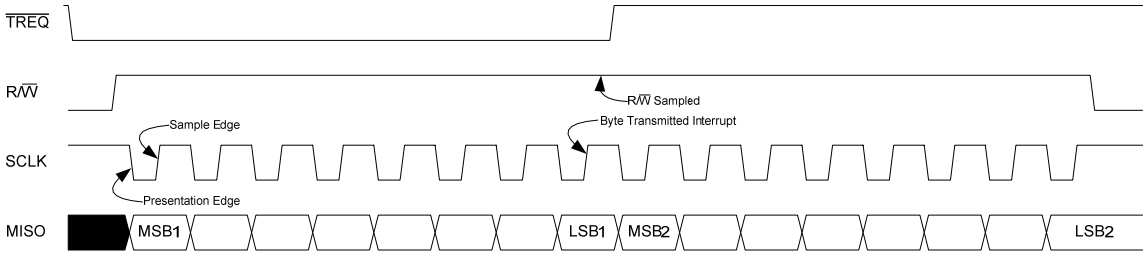

Figure 35. SPI Host to Micro Server (MISO) Transfer Control Flow Diagram without Write Collision

Figure 36 shows the sequence for a MISO transaction when there is a write collision with a MOSI transmission. The host tests the R/W~ signal after loading the first byte to be transmitted to determine if the transmission was successful. Because the R/W~ pin is low, indicating that the ShortStack Micro Server is currently performing a MOSI transfer, the host saves the incoming byte and retries transmission until the R/W~ pin is high after the attempted transfer of the first byte. The figure shows that the host successfully transmits the data on the second attempt.

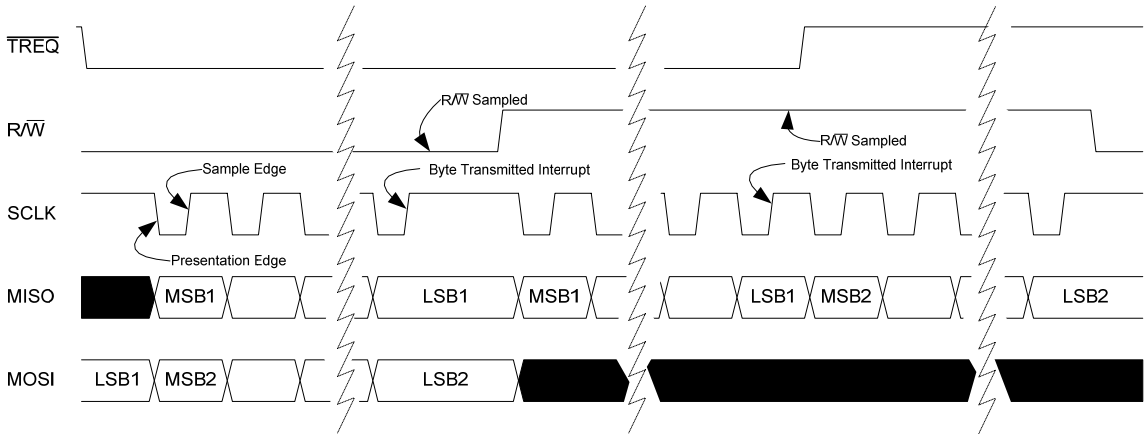

Figure 36. SPI Host to Micro Server (MISO) Transfer Control Flow Diagram with Write Collision

SPI Resynchronization

The Micro Server resynchronizes the ShortStack SPI interface by de-asserting the SS~ pin during a byte transfer, or by de-asserting the SS~ pin and issuing several SCLK pulses. This resynchronization occurs during Micro Server start-up and when the Micro Server resets.

Performing an Initial Micro Server Health Check

After you load the ShortStack Micro Server image into a Smart Transceiver, the Micro Server enters quiet mode (also known as flush mode). While the Smart Transceiver is in quiet mode, all network communication is paused.

The Smart Transceiver enters quiet mode to ensure that only complete implementations of the LonTalk protocol stack attach to a LONWORKS network. In a functioning ShortStack device, the application initializes the Micro Server.

After that initialization is complete, the Micro Server leaves quiet mode and enables regular network communication.

To check that the Micro Server is functioning correctly before the host processor has initialized it, you can use an oscilloscope or a logic analyzer to observe the activity on the TXD (IO10) pin that reflects the uplink **LonNiReset** message transfer that follows a Micro Server reset, as shown in **Figure 37**.

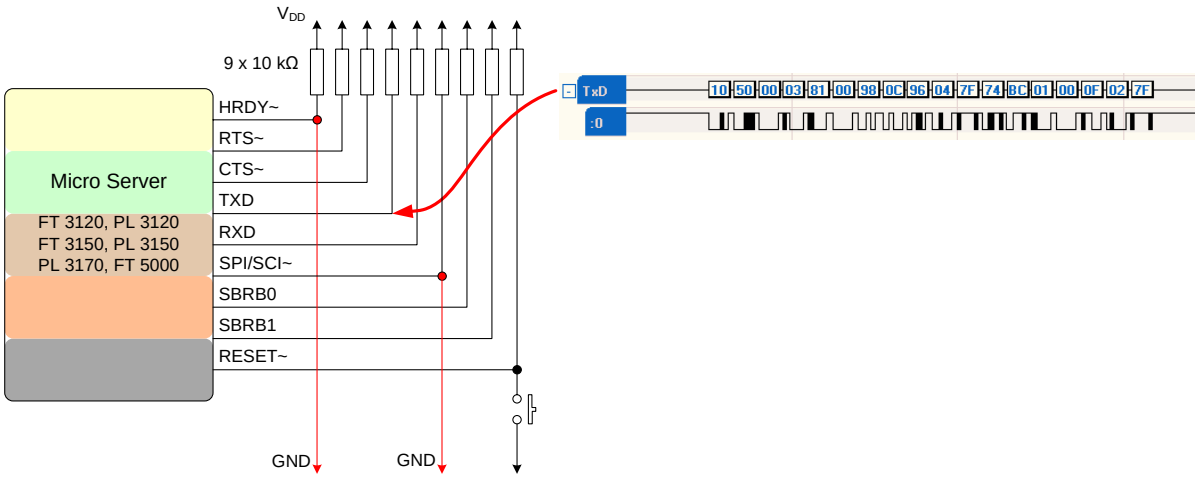

Figure 37. Uplink LonNiReset Message Transfer

For a Mini EVB, the Micro Server's service pin LED flashes slowly (which indicates that the Smart Transceiver is in the unconfigured state), and all network communications are disabled while it is in quiet mode.

In general, you should ensure that all communication and handshake lines are connected to V_{DD} with 10 kΩ pull-up resistors. But for the initial hardware test, the HRDY~ and SPI/SCI~ input signals should be grounded (asserted). Your hardware design should include a switch that connects the **RESET~** pin to ground; you press this switch to reset the Micro Server.

When you press the reset switch for a ShortStack device, the Smart Transceiver firmware performs reset processing, as described in the data books for the Smart Transceiver chips. Then, the Micro Server performs reset processing that is generally independent of the host processor. See *ShortStack Device Initialization* on page 57 for more information about the Micro Server's reset processing.

After the Micro Server is fully initialized, it transmits the uplink **LonResetNotification** message to the host. The host normally registers (or re-registers) its application with the Micro Server; the host application (through the ShortStack LonTalk Compact API) begins application registration with the Micro Server, in which the driver sends the following messages to the Micro Server (in the **LonInit()** function and interrupt service routine for the CTS~ line):

- The **LonNiAppInit** message
- One or more **LonNiNvInit** messages (how many depends on the number of network variables that are defined for the device)
- The **LonNiReset** message

After the Micro Server completes processing for the **LonNiReset** message, it sends the uplink reset message (**LonResetNotification**) to the host processor.

After the host application processes this message, the host application can begin processing. If the message (in the **Flags** field) indicates that the Micro Server is not initialized, the host application should re-run the **LonInit()** function.

Example: **Figure 38** through **Figure 42** on page 88 show sample logic analyzer traces[6] for the communications activity between the host processor and the Micro Server during the initialization sequence after device reset. This example assumes an SCI setup for a 10 MHz Series 3100 Micro Server, with both the SBRB0 and SBRB1 pins connected to GND to set the bit rate at 76800 bps. The data transmission signals (RXD and TXD) in the figures are labeled from the host's point of view. This example shows the reset behavior of the serial driver from the ARM7 example port that is available from www.echelon.com/shortstack.

Figure 38 shows a high-level logic analyzer trace for this initialization sequence:

- The boxed area labeled A represents sending the **LonNiAppInit** message
- The boxed area labeled B represents sending the **LonNiNvInit** message
- The boxed area labeled C represents sending the **LonNiReset** message

The trace also shows the handshake protocol (the RTS~ and CTS~ lines) that the serial driver and the Micro Server use to negotiate communications. The handshake interaction is described in the subsequent figures.

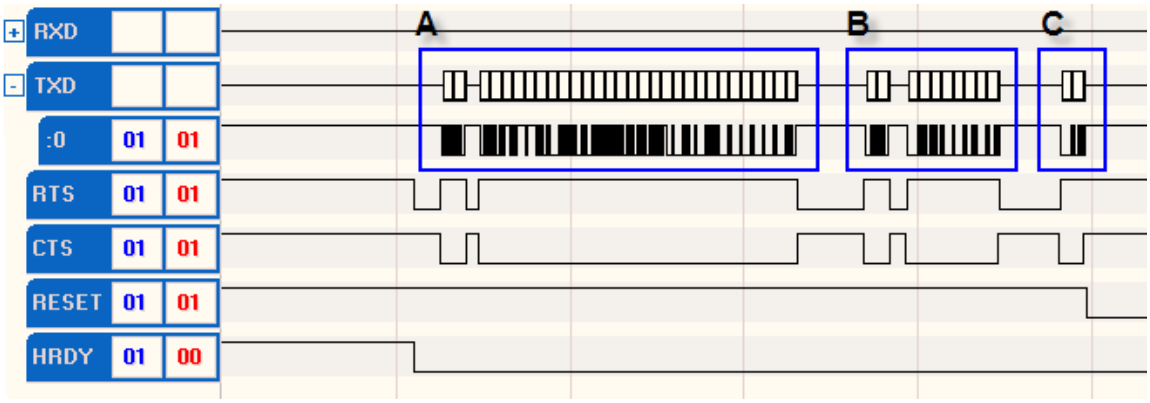

Figure 38. High-Level Logic Analyzer Trace for ShortStack Device Reset

Figure 39 shows the detailed trace for the serial driver and Micro Server interactions for sending the **LonNiAppInit** message.

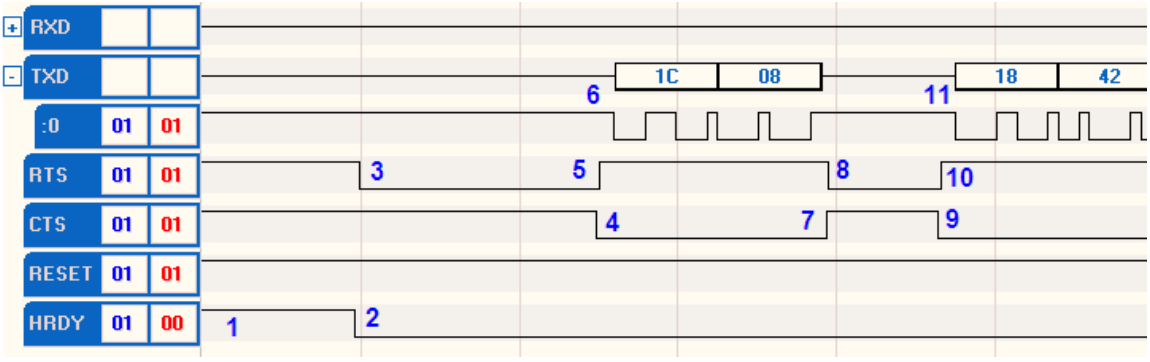

Figure 39. Detailed Logic Analyzer Trace for Sending the LonNiAppInit Message

[6] The logic analyzer traces were captured using the TechTools DigiView™ Logic Analyzer.

The figure shows the following actions by the host processor and the Micro Server:

1. After a device reset, the driver sleeps for a specified amount of time. For the ARM7 serial driver, it sleeps for 255 ms (specified by the **LDV_DRVWAKEUPTIME** macro in the **LdvSci.h** file).
2. When the driver wakes up, it asserts the HRDY~ line to inform the Micro Server that it is ready to receive data (if any).
3. Because the driver needs to send the initialization messages, it confirms that the CTS~ line is not asserted, and then it asserts the RTS~ line to inform the Micro Server that the driver has data to send to the Micro Server (in this case, the header packet for the **LonNiAppInit** message).
4. The Micro Server asserts the CTS~ line to inform the driver that the Micro Server is ready to receive data.
5. The driver deasserts the RTS~ line. The handshake between the driver and the Micro Server is complete, so the driver deasserts the RTS~ line so that the line can be asserted when the driver needs to send more data to the Micro Server. It is important that the driver deassert the RTS~ line before the last byte of data is transmitted, and it is recommended that the driver deassert the RTS~ line as soon as the CTS~ line is asserted.
6. The driver sends the two-byte header packet to the Micro Server. In this case, the length byte is 0x1C (decimal 28) and the command byte is 0x08, which specifies the **LonNiAppInit** message.
7. After the Micro Server receives the header packet, it deasserts the CTS~ line to inform the driver that the Micro Server is no longer ready to receive data. The Micro Server is always aware of the number of bytes that it expects to receive from the driver. In this case, because the packet is the header, the Micro Server knows that the driver will send only 2 bytes, so it deasserts the CTS~ line after it has received the 2 bytes.
8. The driver confirms that CTS~ is deasserted, and again asserts the RTS~ line to inform the Micro Server that the driver has data to send to the Micro Server (in this case, the payload packet for the **LonNiAppInit** message).
9. After the Micro Server has processed the header information for the **LonNiAppInit** message, it asserts the CTS~ line to inform the driver that the Micro Server is ready to receive the payload data.
10. The driver deasserts the RTS~ line. The handshake between the driver and the Micro Server is complete.
11. The driver sends the 28-byte payload packet for the **LonNiAppInit** message to the Micro Server. The size of this message depends on the specific device interface.

Although the figure does not show it, after the Micro Server receives the last byte of the payload data for the **LonNiAppInit** message, it deasserts the CTS~ line to inform the driver that the Micro Server is no longer ready to receive data. There might be a significant delay between the last downlink data byte and the deassertion of the CTS~ signal, during which the Micro Server processes the data received, and prepares for another link-layer exchange. Because it parses the

data in the link-layer header to read the length byte, the Micro Server is always aware of the number of bytes that it expects to receive from the driver.

Figure 40 shows the detailed trace for the serial driver and Micro Server interactions for sending the **LonNiNvInit** message. The figure also includes the end of the transaction for the **LonNiAppInit** message.

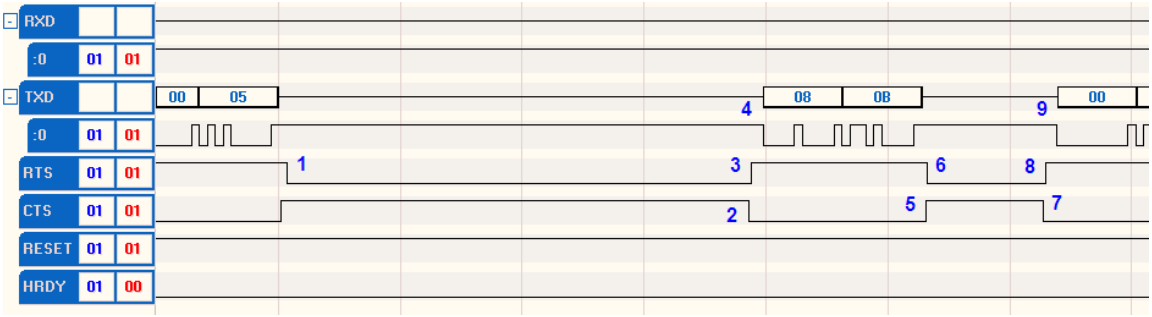

Figure 40. Detailed Logic Analyzer Trace for Sending the LonNiNvInit Message

The figure shows the following actions by the host processor and the Micro Server:

1. The driver confirms that the CTS~ line is not asserted, and then asserts the RTS~ line to inform the Micro Server that the driver has more data to send to the Micro Server (in this case, the header packet for the **LonNiNvInit** message).
2. The Micro Server asserts the CTS~ line to inform the driver that the Micro Server is ready to receive data. During the long delay between the driver's asserting RTS~ and the Micro Server's asserting CTS~, the Micro Server processes the **LonNiAppInit** message.
3. The driver deasserts the RTS~ line. The handshake between the driver and the Micro Server is complete.
4. The driver sends the two-byte header packet to the Micro Server. In this case, the length byte is 0x08 and the command byte is 0x0B, which specifies the **LonNiNvInit** message.
5. After the Micro Server receives the header packet, it deasserts the CTS~ line to inform the driver that the Micro Server is no longer ready to receive data. The Micro Server is always aware of the number of bytes that it expects to receive from the driver. In this case, because the packet is the header, the Micro Server knows that the driver will send only 2 bytes, so it deasserts the CTS~ line after it has received the 2 bytes.
6. After confirming that CTS~ is deasserted, the driver again asserts the RTS~ line to inform the Micro Server that the driver has data to send to the Micro Server (in this case, the payload packet for the **LonNiNvInit** message).
7. After the Micro Server has processed the header information for the **LonNiNvInit** message, it asserts the CTS~ line to inform the driver that the Micro Server is ready to receive the payload data.
8. The driver deasserts the RTS~ line. The handshake between the driver and the Micro Server is complete.

9. The driver sends the eight-byte payload packet for the **LonNiNvInit** message to the Micro Server. The size of this message depends on the number of network variables defined for the device.

When necessary (depending on the application's set of network variables), steps 1 to 9 can be repeated several times to transfer additional **LonNiNvInit** data to the Micro Server.

The last **LonNiNvInit** packet signals the end of the registration sequence. The Micro Server completes the final registration steps, and leaves quiet mode. Quiet mode ensures that only a complete and fully functioning protocol stack attaches to the network. While in quiet mode, the host processor can use local commands to communicate with the Micro Server, such as query status or ping, but cannot communicate with other devices on the network.

Although the figure does not show it, after the Micro Server receives the last byte of the payload data for the **LonNiNvInit** message, it deasserts the CTS~ line to inform the driver that the Micro Server is no longer ready to receive data. Because it parses the data in the link-layer header to read the length byte, the Micro Server is always aware of the number of bytes that it expects to receive from the driver.

Figure 41 shows the detailed trace for the serial driver and Micro Server interactions for sending the **LonNiReset** message. The figure also includes the end of the transaction for the **LonNiNvInit** message.

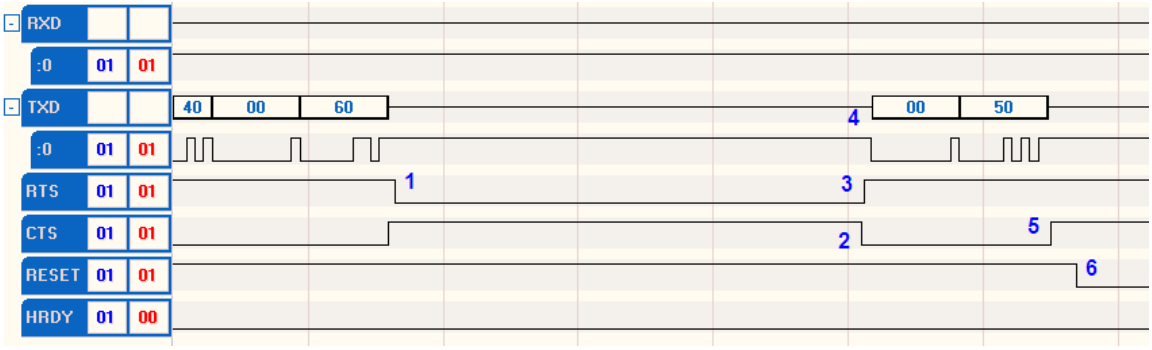

Figure 41. Detailed Logic Analyzer Trace for Sending the LonNiReset Message

The figure shows the following actions by the host processor and the Micro Server:

1. The driver confirms that the CTS~ line is not asserted, and then asserts the RTS~ line to inform the Micro Server that the driver has more data to send to the Micro Server (in this case, the header packet for the **LonNiReset** message).
2. The Micro Server asserts the CTS~ line to inform the driver that the Micro Server is ready to receive data. During the long delay between the driver's asserting RTS~ and the Micro Server's asserting CTS~, the Micro Server processes the **LonNiNvInit** message.
3. The driver deasserts the RTS~ line. The handshake between the driver and the Micro Server is complete.
4. The driver sends the two-byte header packet to the Micro Server. In this case, the length byte is 0x00 (there is no payload for this message) and the command byte is 0x50, which specifies the **LonNiReset** message.

5. After the Micro Server receives the header packet, it deasserts the CTS~ line to inform the driver that the Micro Server is no longer ready to receive data.
6. Because the Micro Server received the **LonNiReset** message, it resets.

In **Figure 41**, note that the driver does not re-assert the RTS~ line. For this example, the host processor has no more data to send to the Micro Server because there is no payload for the **LonNiReset** message. The Micro Server deasserts the **RESET~** line as it completes reset processing.

Approximately 1 second (for a Series 3100 Smart Transceiver running at 10 MHz) after the Micro Server receives the **LonNiReset** message, the Micro Server sends the uplink reset message (**LonResetNotification**) to the host processor, as shown in **Figure 42**. The **LonNiReset** message is shown on the RXD line because the signals are labeled from the host's point of view.

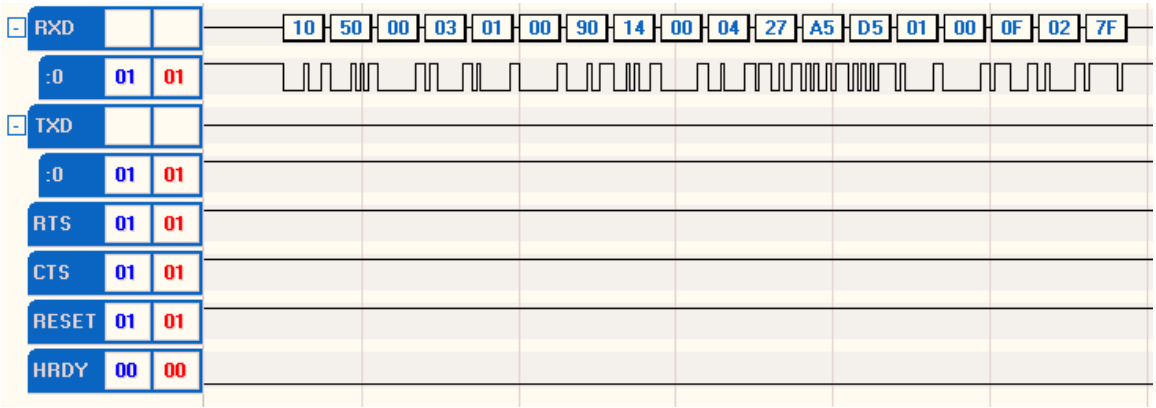

Figure 42. Detailed Logic Analyzer Trace for Receiving the Uplink Reset Message

There is no handshake through the RTS~ and CTS~ control lines for an uplink message, and the message includes both the two-byte header and the message payload in a single message transfer. In this case, length byte is 0x10 (decimal 16) and the command byte is 0x50, which specifies the **LonNiReset** message. This message is always the first message a Micro Server should send to the host processor after a reset. The actual content of this message depends on the characteristics of the Micro Server.

Although it is not likely during Micro Server initialization, an uplink transfer can interrupt the downlink transmission between the sending of the header and the sending of the related payload. If the header has been transmitted and an uplink occurs before the payload can be delivered, the driver must accept the uplink data before it continues with handshake negotiations for the downlink payload transfer.

The example described in this section showed the Micro Server initialization sequence, which consists of two separate message transfers: a two-byte header and the related payload, both of which require a complete handshake. However, a link-layer downlink operation for polling or propagating output network variables with indices larger than 62 consists of three message transfers: a two-byte header, a second two-byte extended header, and the related payload, all of which require a complete handshake. See *Overview of the ShortStack Serial Driver* on page 90 for more information about the link-layer header.

6

Creating a ShortStack Serial Driver

This chapter describes the link-layer serial driver and how to develop a ShortStack serial driver for your host processor. This driver manages the handshaking and data transfers between the host and the ShortStack Micro Server. The driver also manages the buffers in the host for communication with the ShortStack Micro Server.

If a ShortStack driver is available for your host processor that matches your buffer memory and I/O configuration, you can skip this chapter.

Overview of the ShortStack Serial Driver

Each data exchange on the serial link layer consists of one or more segments. For downlink messages, the serial driver and Micro Server perform a handshake for each segment. For uplink messages, there is no handshake.

The link-layer message consists of the following segments:

- A two-byte link-layer header
- A two-byte link-layer extended header (applies only to downlink messages for network variable updates or polls where the network variable index is greater than 62)
- The message payload, if any

The link-layer header consists of two parts:

- The length byte. This value describes the length of the message payload. This value is 0x00 if there is no message payload, and is at least 0x02 if there is a message payload.
- The command byte. This value determines the command being sent to the Micro Server or being received from the Micro Server.

The link-layer extended header consists of two parts:

- The info byte. This value is the actual network variable index for the update or poll request. The command byte of the link-layer header contains a network variable index of 0x3F (decimal 63) to inform the Micro Server and the serial driver that an extended header is required to process the command.
- A reserved byte. For a ShortStack FX Micro Server, the value of this byte is 0x00.

Figure 43 on page 91 shows the structure of the link-layer message.

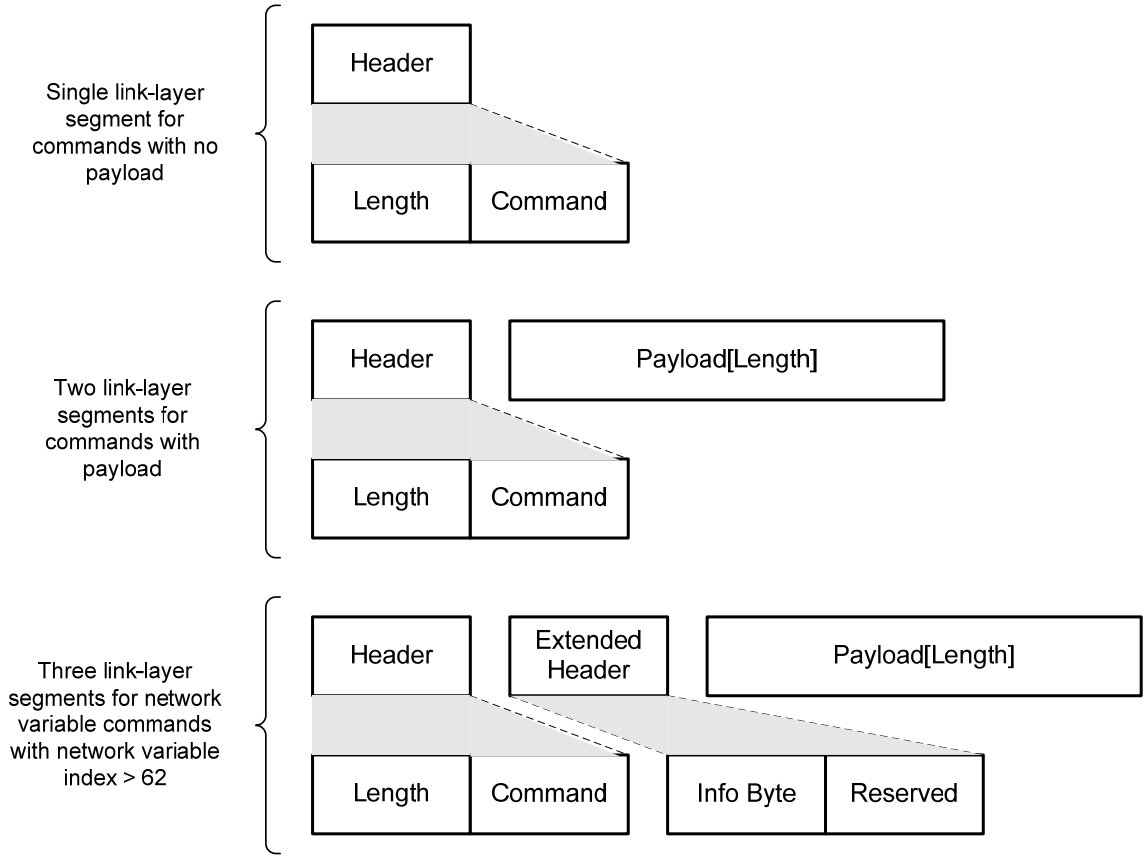

Figure 43. Link-Layer Message Structure

Thus, for a typical link-layer message, the link-layer message includes the link-layer header and the data payload. Not all link-layer messages include payload, but all use the same two-byte header. For network variable polls or updates, the link-layer message can include three segments: the link-layer header, the link-layer extended header, and the data payload.

For both the SCI and SPI interfaces, each link-layer downlink transmission consists of the link-layer header transmission, followed by the link-layer extended header transmission (if applicable), followed by the optional payload transmission. For downlink messages, all segments are individually verified with the handshake procedure between the host and Micro Server that is described in Chapter 5, *Designing the Hardware Interface*, on page 65.

However, there is no handshake process for an uplink transfer. If uplink data is ready in the Micro Server, and the host processor signals its readiness by asserting the HRDY~ line (or has its HRDY~ line permanently tied low), the Micro Server transfers the link layer header, immediately followed by the payload data (if any). In addition, for uplink transfers, the link-layer extended header is not required.

After each downlink transfer, an uplink transfer can occur. If an uplink transfer occurs after sending one segment, but prior to sending the next segment, the subsequent segement transmission must wait for the uplink to complete.

After the uplink is complete, it should be enqueued within the serial driver, and the pending downlink should be completed before processing the newly arrived packet.

Important: The actual payload length must match the specified length in the header byte of the link-layer message. If the actual length exceeds the specified length, extra bits are ignored, but could cause problems for subsequent transactions. Transmitting fewer bits than specified in the link-layer header's length byte causes the Micro Server to wait for the missing bits, and then reset when its watchdog timer expires.

Role of the ShortStack LonTalk Compact API

One of the most important tasks performed by the ShortStack LonTalk Compact API is the processing of uplink link-layer packets into pre-parsed data packets that it passes to the appropriate callback handler function defined by your application.

The application periodically calls the **LonEventHandler()** API function, which queries the serial driver's uplink queue and, upon availability of an uplink packet, dequeues and processes this packet.

For any downlink operation, typically initiated by your application's calling one of the ShortStack LonTalk Compact API functions, such as **LonPropagateNv()**, the API translates the application-friendly data used with the API call into the corresponding link-layer packet, and enqueues this packet for downlink transfer.

Some link-layer transfers can occur without any interaction of your application; for example, a network variable poll or fetch request can typically be satisfied by the API alone, without intervention by your application.

Role of the ShortStack Serial Driver

The ShortStack serial driver provides a hardware-specific interface between the ShortStack LonTalk Compact API and the ShortStack Micro Server. The driver exchanges link-layer messages with ShortStack Micro Server, and implements the host-side of the link-layer protocol.

The serial driver includes buffer management for incoming and outgoing messages, and typically allows for non-blocking operation.

Interface to the ShortStack LonTalk Compact API

Typically, the ShortStack serial driver implements a set of interrupt handlers that respond to USART events such as *transmit buffer empty* or *receive buffer full*. The ShortStack LonTalk Compact API uses eight functions, listed in **Table 18** on page 93, that communicate between the API and the driver, including handling all uplink and downlink data transfers. Your ShortStack serial driver must support these functions. These functions are declared in the **ShortStackApi.h** file (in the serial driver API functions section).

For more information about these interface functions, see an example port's implementation of the functions; for example, see the *ShortStack FX ARM7 Example Port User's Guide*.

Table 18. Interface Functions for the ShortStack LonTalk Compact API

Function	Description
LdvInit()	Initializes the ShortStack serial driver and the underlying communication interface.
LdvFlushMsgs()	Completes pending transmissions and flushes the transmit buffer.
LdvAllocateMsg()	Allocates a transmit buffer in the ShortStack serial driver.
LdvPutMsg()	Sends a downlink message by putting a message in an allocated transmit buffer. This is a non-blocking function.
LdvPutMsgBlocking()	Sends a downlink message without first allocating a transmit buffer in the driver. This is a blocking function, and is used only during the device's initialization phase.
LdvGetMsg()	Gets an incoming message (if any) from the ShortStack serial driver's receive buffer.
LdvReleaseMsg()	Releases a message buffer back to the ShortStack serial driver after receiving and processing a message.
LdvReset()	Resets the serial driver when it receives an uplink reset message from the Micro Server.

Creating an SCI ShortStack Driver

This section describes how to implement an SCI ShortStack driver. The SCI hardware interface is described in *SCI Interface* on page 71.

A ShortStack Micro Server considers the serial link reliable. An inter-byte time-out (or any other time-out condition) is considered a serious error, and recovery generally requires resetting the Micro Server and the host driver state. To minimize the effects of such a time out, set a large time-out interval based on the communications bit rate or use another appropriate large value (such as 3 or 5 seconds).

SCI Uplink Operation

In an SCI uplink operation, data is transferred from the ShortStack Micro Server to the host processor. **Figure 44** on page 94 and **Figure 45** on page 95 show the activity that the driver must manage for an uplink operation. The figures also show how the Micro Server, serial driver, LonTalk Compact API, and the application interact to process an uplink message.

The host processor uses the HRDY~ handshake signal to inform the Micro Server when it is ready to receive uplink data. The Micro Server does not send uplink data unless the HRDY~ pin is asserted. While an uplink transfer is in progress, the Micro Server does not re-sample the HRDY~ pin. To prevent loss of uplink data, the host must assert this handshake signal whenever possible, and de-assert it for the shortest time possible.

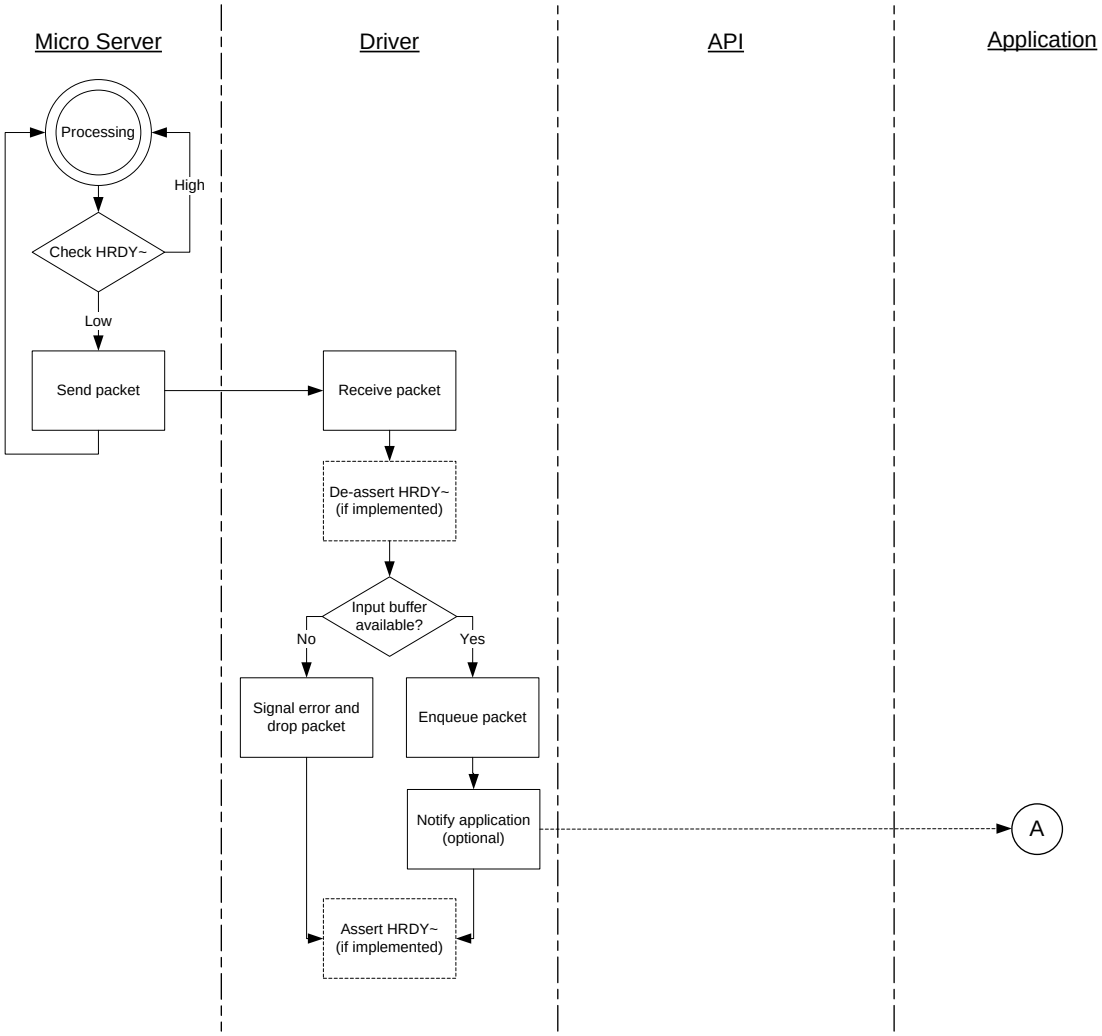

Figure 44. SCI Uplink Operation (Part 1)

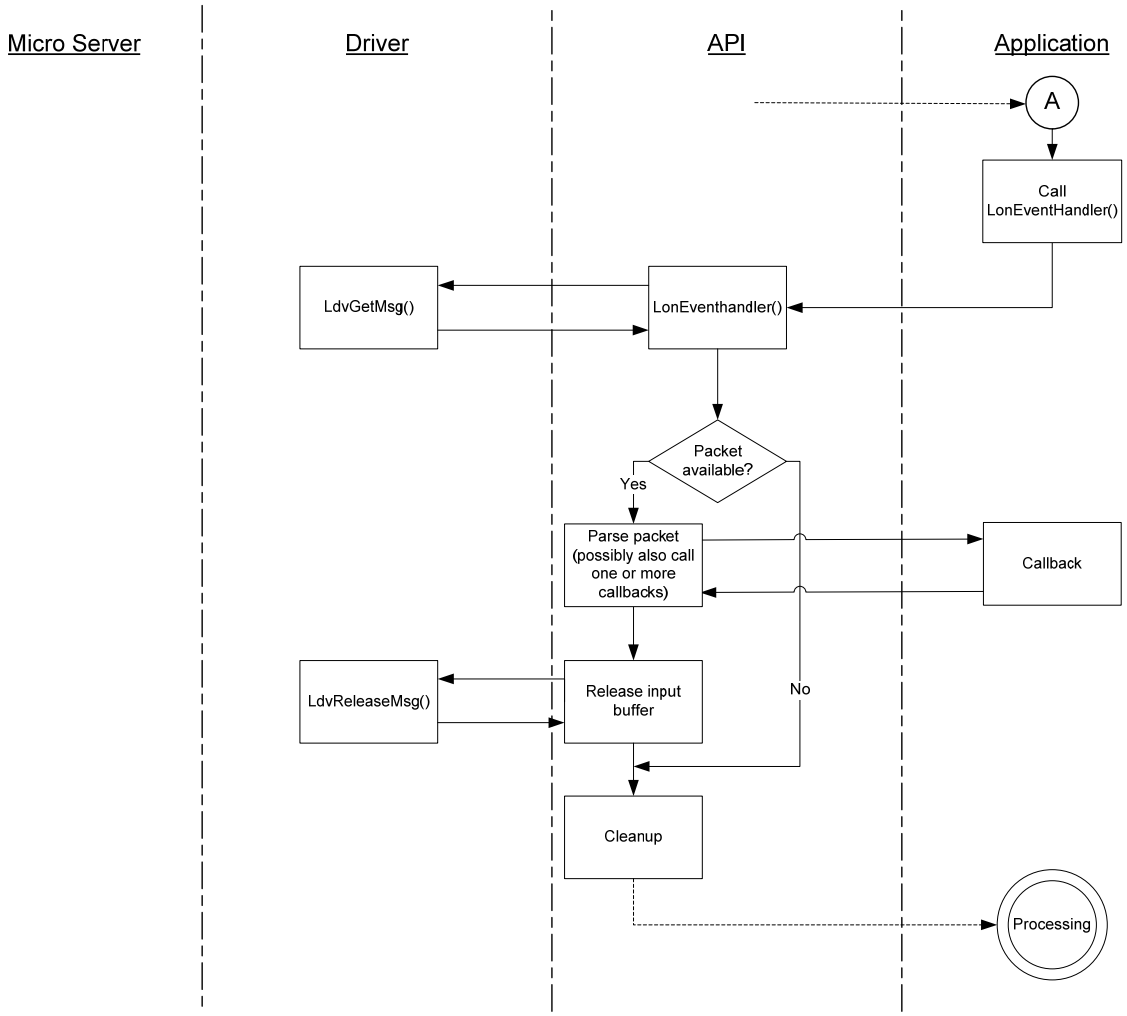

Figure 45. SCI Uplink Operation (Part 2)

SCI Downlink Operation

In an SCI downlink operation, data is transferred from the host processor to the ShortStack Micro Server. **Figure 46** on page 97 shows the activity that the driver must manage for a downlink operation. **Figure 47** on page 98 shows the SCI handshake and data transfer for the header, extended header, or payload.

To send a message downlink, the driver needs to initiate a downlink operation for each link-layer message segment: one for the link-layer message header, one for the extended header (if applicable), and one for the message payload (if any):

1. The driver first initiates the transfer of the link-layer message header, then, if allowed, transfers the header.
2. If the message applies to a network variable with index greater than 62, the driver then initiates the transfer of the link-layer extended header, then, if allowed, transfers the extended header.
3. Then, if payload data exists (indicated by the non-zero length byte in the header), the driver initiates the transfer of the message payload, and, if allowed, transfers the message payload.

When the host asserts the RTS~ signal for the first time, the Micro Server assumes that the assertion is for the 2-byte header. It asserts the CTS~ line until it has read the two bytes. It then extracts the length of the payload from the header and parses the command byte to determine if an extended header is needed. When the host asserts the RTS~ signal a second time, the Micro Server asserts the CTS~ line until it receives either the extended header or the entire payload (based on its length and command byte, as indicated in the header), depending on which is expected. Some messages have no payload (for example, the reset message), thus the payload length for these messages is zero.

Before beginning a transfer, or after having transferred the entire transaction payload, the host must wait for the CTS~ signal to become inactive (high) again. The Micro Server deasserts this signal after it receives all bytes of the current transaction, and after it has completed any immediate processing that might be required. If the application does not query this signal state, error states can occur. For example, the host might attempt to transfer a new transaction because it would assume that the CTS~ signal's being asserted is the acknowledgment of the new transfer request rather than the acknowledgment from the previous transfer.

It is possible for an uplink transfer to occur after the Micro Server receives the downlink header, but before it is ready to receive the downlink payload. No uplink can occur while the CTS~ signal is asserted.

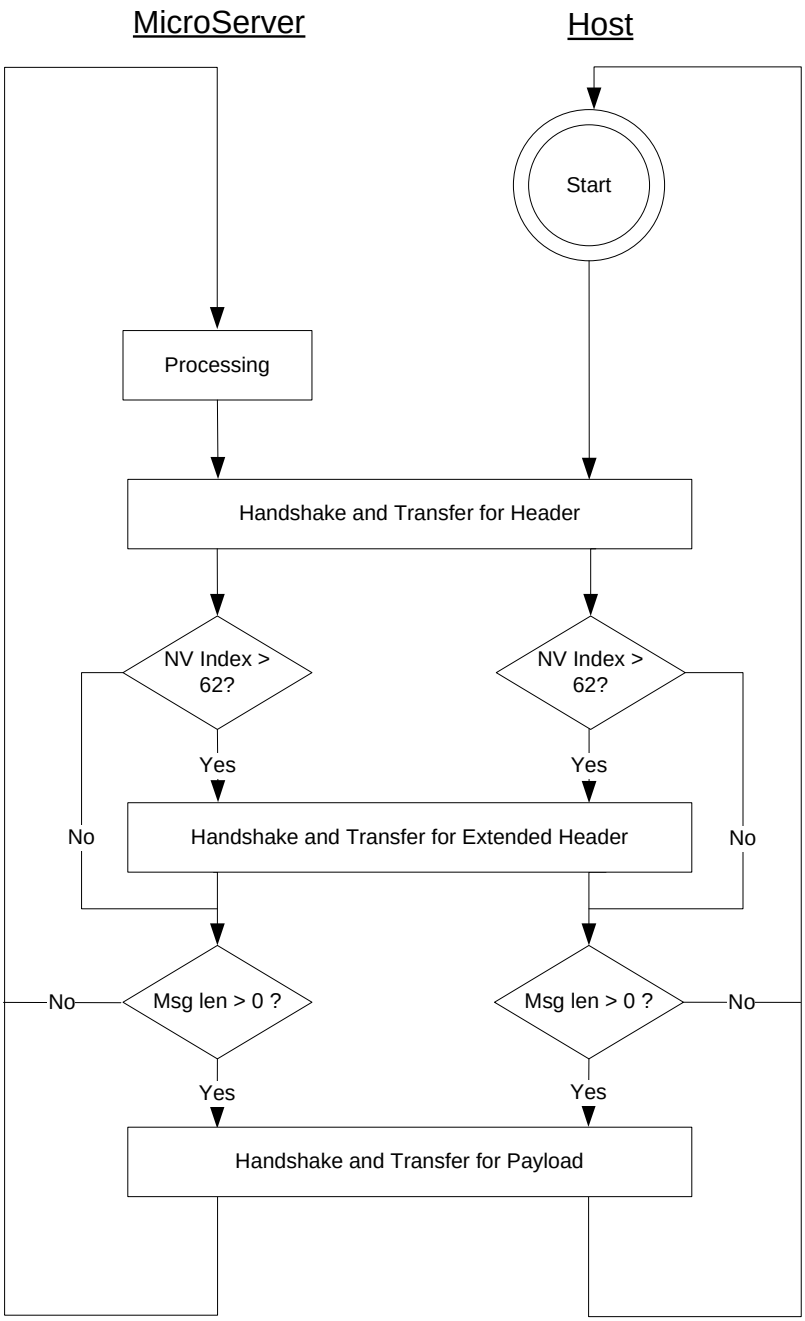

Figure 46. Downlink Operation

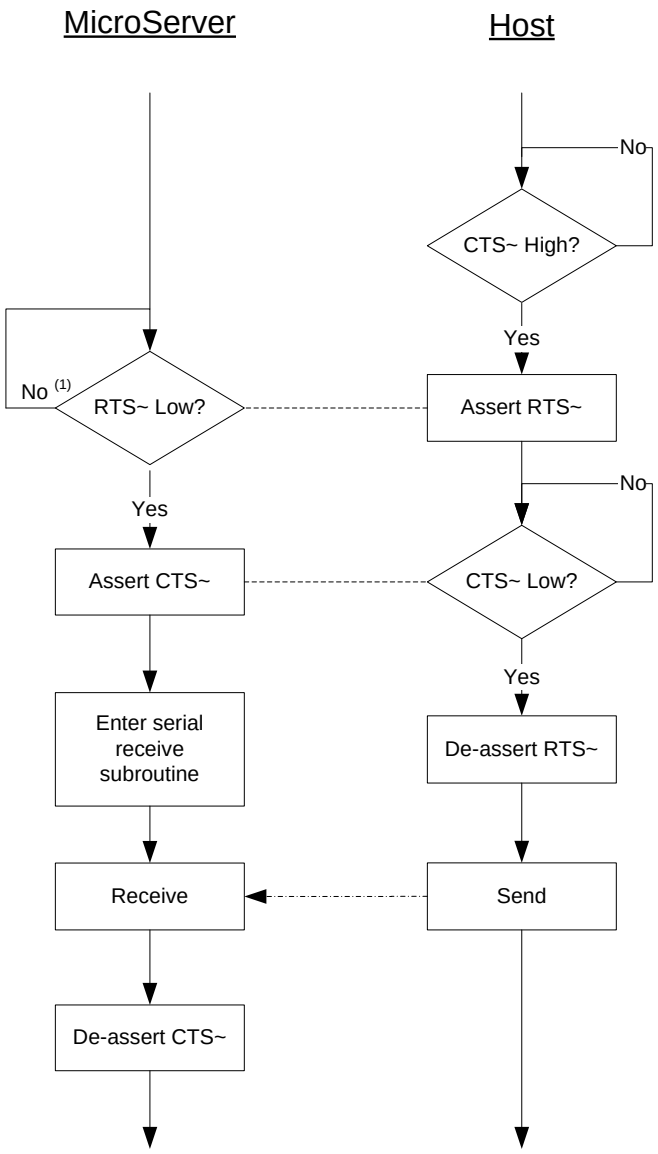

Figure 47. SCI Handshake and Data Transfer

Note (1): When the Micro Server checks the RTS~ signal for most commands (in the "RTS~ Low?" decision box), if the signal remains high without data transfer for longer than the watchdog timer setting for the Smart Transceiver (approximately 840 ms for a Series 3100 Smart Transceiver at 10 MHz or for a Series 5000 Smart Transceiver), the Micro Server performs a watchdog reset.

Prior to receiving the payload (if any), the Micro Server prepares to receive the payload data. For most downlink operations, this preparation includes allocating an output buffer. If no buffers are available, acknowledgement for the RTS~ signal with CTS~ assertion could take a significant amount of time, depending on the local channel type, channel usage, the types of transactions that are holding the buffers, and transport and transaction control properties. Your driver must be able to handle such delays.

Example: Network Variable Fetch

You can use a logic analyzer or oscilloscope to observe the interactions between the host and Micro Server during network operations, such as a fetch of a network variable. A logic analyzer trace can be a helpful tool to verify that the serial driver works as expected.

Figure 48 shows an example logic analyzer trace after the Micro Server receives a network variable fetch request from the network. The timing for the logic analyzer trace is 5 ms per division. The example used an FT 3150 Micro Server running at 10 MHz with an ARM7 host running at 20 MHz.

Notice in the figure that the host waits for the CTS~ signal to become inactive before it starts a new transfer by asserting the RTS~ signal.

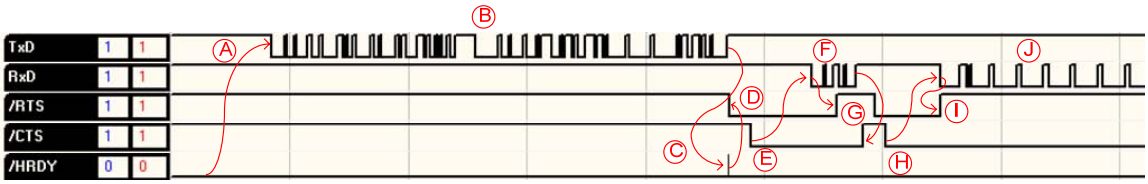

Figure 48. Logic Analyzer Trace for an NV Fetch

The figure shows the following events:

A. The Micro Server samples the HRDY~ signal. If it is asserted, which it is in this example, the Micro Server begins to transfer the uplink data.

B. The TXD signal shows the uplink data transfer.

C. The host briefly de-asserts the HRDY~ signal while it stores the packet in an incoming queue (if the host has buffers available, it need not de-assert the HRDY~ signal). The host can optionally notify the application of the available data for asynchronous processing.

D. The host prepares its response, waits for the CTS~ signal to be inactive, asserts the RTS~ signal, then waits for the CTS~ signal to be asserted.

E. The Micro Server asserts the CTS~ signal.

F. The host de-asserts the RTS~ signal and transmits the message header (shown on the RXD signal).

G. The host waits for the CTS~ signal to become inactive, re-asserts the RTS~ signal, and waits for the CTS~ signal to be asserted again.

H. The Micro Server is ready for the payload, and asserts the CTS~ signal.

I. The host de-asserts (releases) the RTS~ signal and begins the payload transfer.

J. The RXD signal shows the payload transfer (the downlink response containing the requested NV value).

Creating an SPI ShortStack Driver

This section describes how to implement an SPI ShortStack driver. The SPI hardware interface is described in *SPI Interface* on page 76.

SPI Uplink Operation

In an SPI uplink operation, data is transferred from the ShortStack Micro Server to the host processor. **Figure 49** and **Figure 50** on page 101 show the activity that the driver must manage for an uplink operation. The figures also show how the Micro Server, serial driver, ShortStack LonTalk Compact API, and the application interact to process an uplink message. The driver must see the R/W~ signal low between the arrivals of the first and second bytes in the burst when it is receiving a packet.

The host processor uses the HRDY~ handshake signal to inform the Micro Server when it is ready to receive uplink data. The Micro Server does not send uplink data unless the HRDY~ pin is asserted. To prevent loss of uplink data, the host must assert this handshake signal whenever possible, and de-assert it for the shortest time possible.

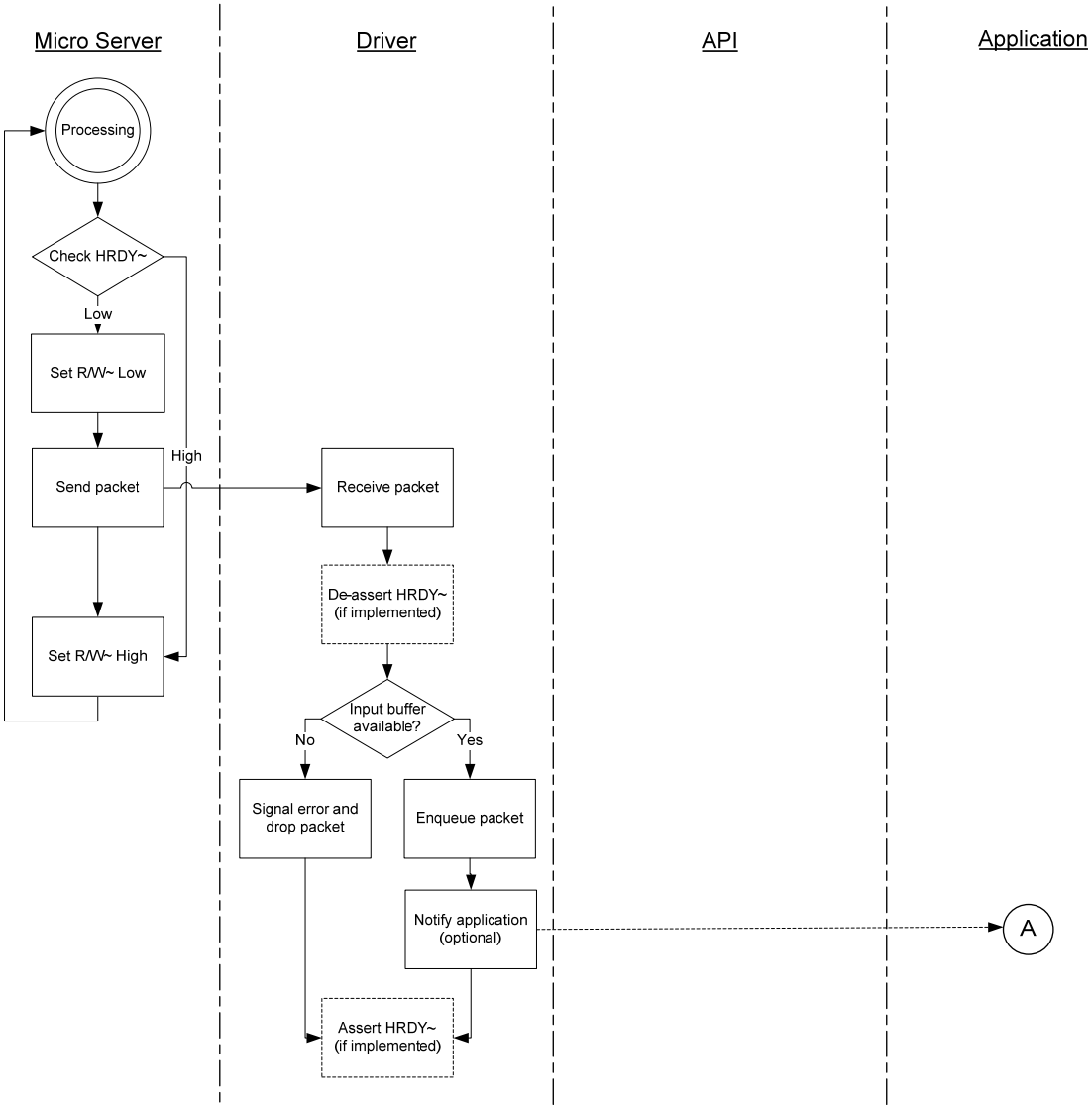

Figure 49. SPI Uplink Operation (Part 1)

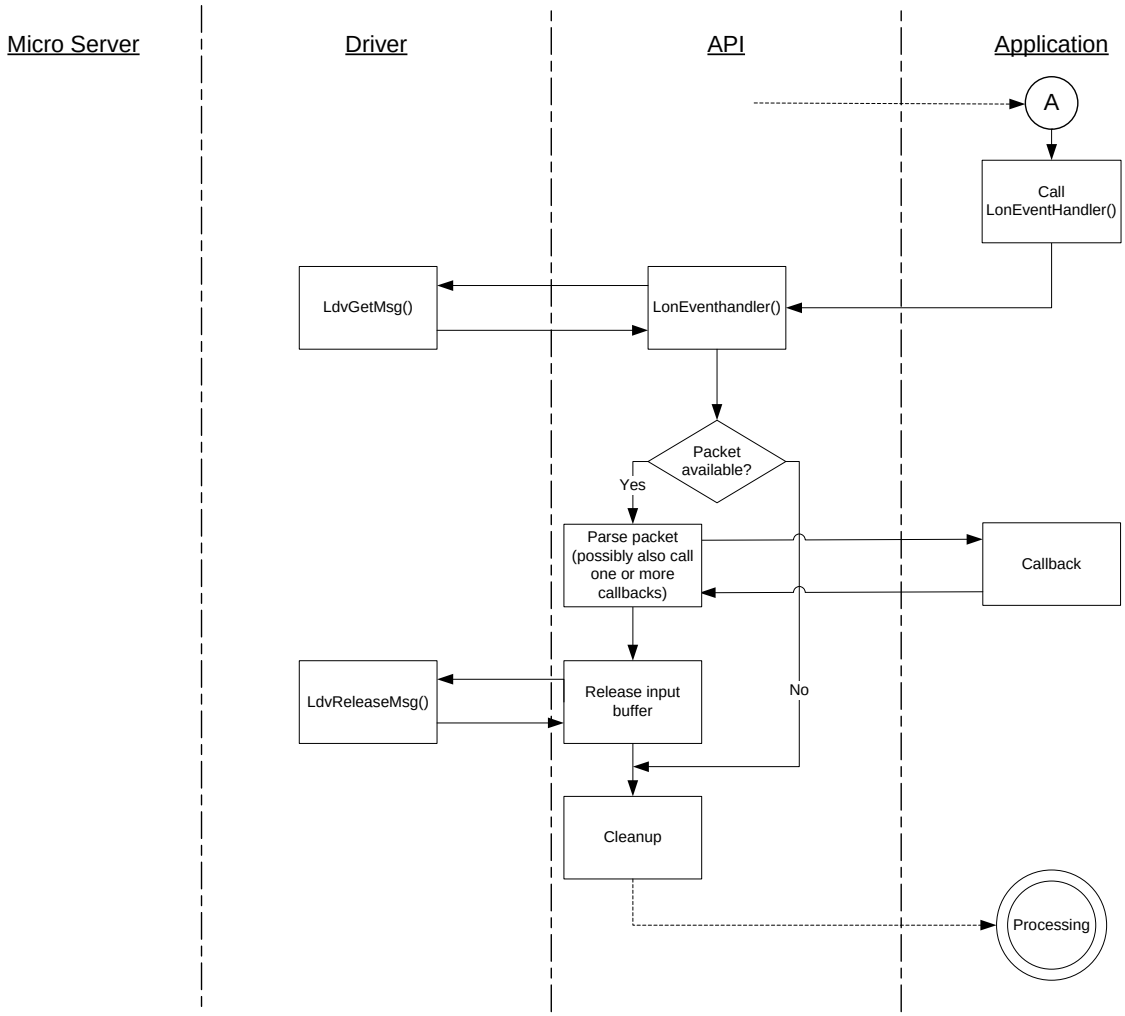

Figure 50. SPI Uplink Operation (Part 2)

SPI Downlink Operation

In an SPI downlink operation, data is transferred from the host processor to the ShortStack Micro Server. To send a link-layer message downlink, the driver initiates two downlink operations: one for the link-layer message header, and the other for the message payload. **Figure 51** on page 102 shows the activity that the driver must manage for a downlink operation (this figure is the same as **Figure 46** on page 97). **Figure 52** on page 103 shows the SPI handshake and data transfer for the header, extended header, or payload. The driver must see the R/W~ signal high between transmissions of the first and second bytes in the burst when it is transmitting a packet. In addition, the Micro Server keeps the R/W~ signal high for an additional byte time; this extra time allows the host to confirm transfer direction.

As described in *SPI Host to Micro Server Control Flow (MISO)* on page 81, the host must detect possible write collisions during data transfer.

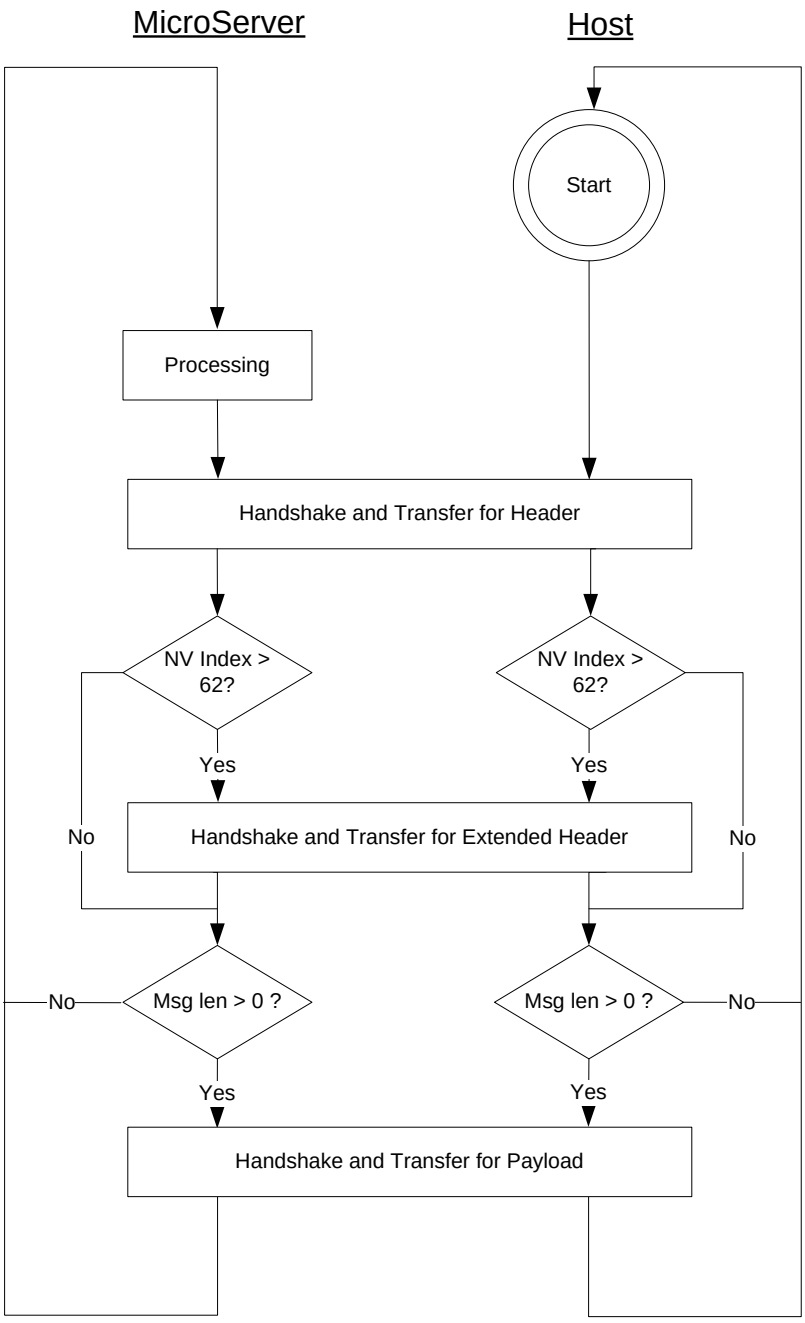

Figure 51. Downlink Operation

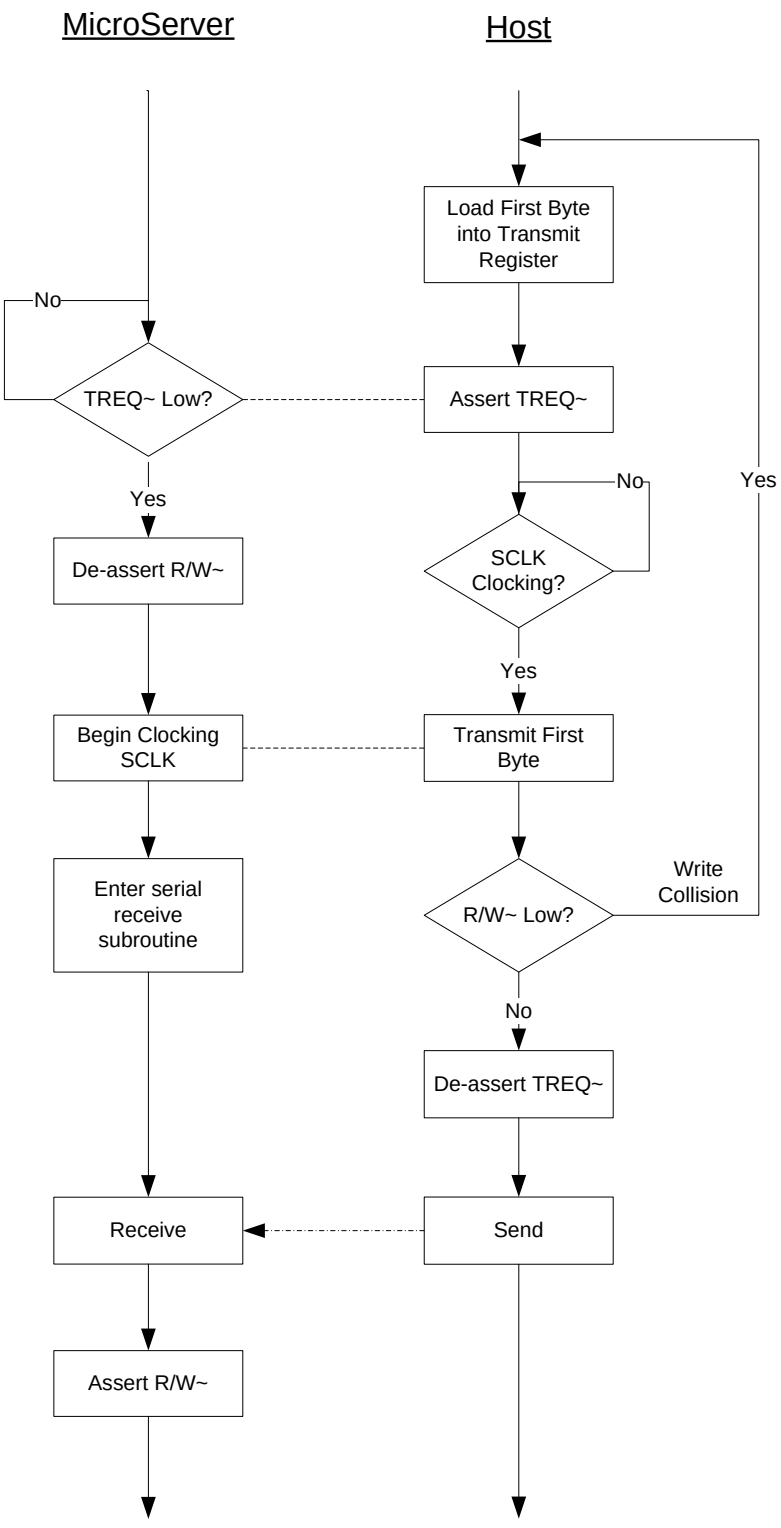

Figure 52. SPI Handshake and Data Transfer

Prior to receiving the payload (if any), the Micro Server prepares to receive the payload data. For most downlink operations, this preparation includes allocating an output buffer. If no buffers are available, the Micro Server could take a

significant amount of time to de-assert the R/W~ signal after the host asserts the TREQ~ signal, depending on the local channel type, channel usage, the types of transactions that are holding the buffers, and transport and transaction control properties. Your driver must be able to handle such delays.

Transmit and Receive Buffers

The ShortStack serial driver needs to define the number and size of the transmit and receive buffers in the host processor. More buffers require more memory, but can also increase performance and minimize the potential for lost messages.

Recommendation: Set the serial driver's buffer count for both transmit and receive buffers to the number of application buffers defined for the Micro Server, and adjust upward as necessary for the application. For example:

```
#define LDV_TXBUFCOUNT 5
#define LDV_RXBUFCOUNT 5
```

Important: The transmit and receive buffers within the host must not be smaller than those defined in the Micro Server.

Link-Layer Error Detection and Recovery

The ShortStack Micro Server and the ShortStack LonTalk Compact API both assume that the serial communication between the host microprocessor and the ShortStack Micro Server is a reliable link. To maximize performance, the ShortStack Micro Server uses a simple link layer protocol with minimal error detection. Your hardware design for the interface between your host and the ShortStack Micro Server must provide this reliable link.

When either the Micro Server or the host processor resets, your serial driver must synchronize with the ShortStack Micro Server. Your serial driver must also implement the following timing characteristics to maintain synchronization with the ShortStack Micro Server:

- An inter-byte timeout for both the serial receiver and transmitter. If the receiver timer expires, the current message should be discarded. If the transmitter timer expires, the current message should be resent later.
- A sleep period of 250 ms during driver startup. This delay allows synchronization with the ShortStack Micro Server during startup.

Your serial driver should implement appropriate timeout guards. For example, when your driver waits for an SCI CTS~ assertion by the Micro Server, or for the byte-transmitted interrupt after asserting the SPI TREQ~ signal, a timeout period of 5 seconds can help to detect serious malfunction.

Likewise, when the driver expects a predetermined number of bytes to arrive from the Micro Server, an inter-byte timeout of 1 second, or a total packet timeout that is a function of the expected byte count, is recommended.

If the link-layer is idle for a period of time, the serial driver or host application can issue a ping command (the **LonSendPing()** function with the **LonPingReceived()** callback handler function) to verify that the Micro Server is still running properly and has an operational link layer. The ping command is a short link-layer message that is echoed by the Micro Server; no other action is triggered by this command.

You can also use the echo command (the **LonRequestEcho()** function with the **LonEchoReceived()** callback handler function) to test the link layer. The echo command provides more functionality than the ping command, but at the cost of additional bytes and transfer time. Using the echo command, the application can send six arbitrary bytes to the Micro Server. The Micro Server receives the data, increments each of the six bytes (using unsigned 8-bit arithmetic, ignoring any overflow conditions), and returns the entire data packet to the host.

You can use the echo command when the device is idle to verify that the link layer and the Micro Server are operational. You can also use the echo command during device stress testing to verify robust link-layer operations under high traffic conditions. For such a stress test, an application would repeatedly send echo requests with different data and confirm that the data received meets expectations. Data errors detected during such a test could indicate poor link-layer line termination, excessive crosstalk on the link-layer lines, out-of-sync bit rates (for SCI), or excessive bit rates (for SPI).

Because the echo command can be processed before the application registers with the Micro Server, it can be a good early indicator for correct implementation of both the serial driver and the link-layer protocol.

See *Local Utility Functions* on page 294, *Local Utility Callback Handler Functions* on page 299, or the HTML API documentation for more information about the ping command and the echo command.

When a serious error condition is detected, your application should log an error and, if possible, signal the event to the user. You can also optionally assert the Micro Server's reset line in an attempt to recovery from the error condition, but such a reset is not normally necessary.

Loading the ShortStack Application into the Host Processor

Before you can test and debug your ShortStack device, you need to load the ShortStack application into the host processor. For an FPGA-based embedded processor, you might have to load the hardware design into the FPGA, as well load the ShortStack software application into the FPGA.

How you load the ShortStack application into the host processor depends on the host processor that your ShortStack device uses. Typically, you use a device programmer for in-circuit flash programming through a JTAG connection to the host processor.

For a description of a method for loading a ShortStack application into an ARM7 host processor, see the *ShortStack FX ARM7 Example Port User's Guide*.

Performing an Initial Host Processor Health Check

To check that the host processor and the serial driver implementation are working properly, you need to connect the host to a ShortStack Micro Server. To ensure that an initial health check of the host tests only the host, you should use a Micro Server that is already known to work properly.

For an initial health check of the host, you can use an Echelon Mini EVB evaluation boards, available with PL 3120, FT 3120, PL 3150, FT 3150, and PL 3170 Smart Transceivers, or an FT 5000 EVB evaluation board. These boards

are ideally suited for an initial host check, because they include EIA-232 level shifters and a set of jumpers to run the transceiver as a ShortStack Micro Server. You could also use a Micro Server that you tested according to the test described in *Performing an Initial Micro Server Health Check* on page 82.

A basic health check for the host includes the following steps:

1. Connect the host to the Micro Server, and supply power to both
2. Issue a downlink reset command (command code 0x50)
3. Observe that the Micro Server resets
4. Observe the uplink reset notification

The reset pulse on the Micro Server is typically very short, and often not noticeable when visually monitoring the Reset LED. Boards with external flash memory include pulse-stretching devices that enforce a longer Reset pulse, which could provide a more visible state change on the Reset LED. However, using an oscilloscope or logic analyzer is recommended.

During this and similar tests in the early stages of development, you should also monitor the Reset line carefully, because errors in the host-side driver implementation can cause the Micro Server to reset. For example, if the host asserts the RTS~ pin, but fails to deliver data in time, or if the host fails to deliver the entire packet, or if the host fails to assert the HRDY~ pin in a timely fashion, the Micro Server could reset due to a watchdog timer timeout. A Smart Transceiver Chip's watchdog timer expires in approximately 840 ms (for a Series 3100 Smart Transceiver at 10 MHz or for a Series 5000 Smart Transciever).

Prior to initialization, the Micro Server is in quiet mode, which prevents all network communication, until the downlink initialization is complete. However, the basic host health check described in this section works while the Micro Server is in quiet mode, and can thus be used for an initial health check before the application framework (which includes the initialization data structure) is complete.

When you power-up the Micro Server for the first time, allow up to a minute for it to complete its first-time boot sequence. The duration for the first-time boot varies with the Micro Server hardware and software configuration, but subsequent boots require much less time. See *ShortStack Device Initialization* on page 57 for more information about the Micro Server's reset processing.

Then, use a simple test application and your serial driver to issue a downlink reset command. This is a simple command without a payload; it consists only of two header bytes: 0x00 for the payload length, and 0x50 for the command (**LonNiReset**). The **LonNiReset** command instructs the Micro Server to reset. You should be able to observe the Smart Transceiver's reset line's being asserted for a brief moment.

When the Micro Server completes the reset sequence, it notifies the host processor of the event. The uplink reset message also uses the **LonNiReset** (0x50) command in the link-layer header, but includes 16 payload bytes.

The uplink reset message contains information about the state, version, and type of the Micro Server, its capacity for various system resources, and whether it is initialized. The message can be helpful to diagnose problems (or success) during early stages of development.

Before your application attempts to register with the Micro Server for the first time, it should execute an echo command (the **LonRequestEcho()** function with the **LonEchoReceived()** callback handler function). Repeated use of this command provides an early link-layer stress test, and can provide early indication of errors in the physical design of the link layer.

7

Porting the ShortStack LonTalk Compact API

If you are using a host processor and development environment that does not have an available ShortStack FX example port, you must port the ShortStack LonTalk Compact API files to work with your chosen host processor and development environment. A minimal port requires you to provide definitions that control the portable code, but a more substantial port might be required. A completed port applies to all applications that use the same hardware and software configuration.

This chapter describes the steps and considerations for porting the ShortStack LonTalk Compact API.

Portability Overview

The ShortStack LonTalk Compact API is implemented in ANSI C. Although ANSI C is a standard programming language, different implementations are required to meet the requirements of different target processors. To support the largest possible number of target processors and compilers, the ShortStack LonTalk Compact API implementation is based on the following portability concepts:

- Host-side types and interfaces use standard ANSI C types and style. For example, the **LonPropagateNv()** function, which takes a network variable's index as an argument, expects this argument to be of the standard C type *unsigned*.
- All data types that interface with the Micro Server or the LONWORKS network are based on streams of bytes, and do not use multi-byte scalar types such as 16 or 32-bit integers. Using streams of bytes helps to control byte padding and packing issues within structures.

  All types are based on the **LonByte** type. Multibyte scalars are composed of multiple **LonByte** members in big-endian byte order, such as the **LonWord** type.

  Optionally, you can use macros such as **LON_GET_UNSIGNED_WORD** or **LON_SET_UNSIGNED_WORD** to assist in transforming those types into the host processor's native types. Native types can be more efficient in numeric algorithms.
- Structures and unions are declared using macros because some compilers allow you to control packing and alignment of aggregates for each type definition individually through non-standard keyword extensions. These macros are **LON_BEGIN_STRUCT**, **LON_END_STRUCT**, **LON_BEGIN_UNION**, and **LON_END_UNION**.

  Example: For the GNU C Compiler, the macros controlling structure declarations could be:

  ```
  #define LON_STRUCT_BEGIN(n) struct
  #define LON_STRUCT_END(n)   attribute((__packed__)) n
  ```
- Structures and unions that are embedded in other structures or unions use another set of macros to provide further support for non-standard keywords that control packing and alignment of aggregates. These macros are **LON_BEGIN_NESTED_STRUCT**, **LON_END_NESTED_STRUCT**, **LON_BEGIN_NESTED_UNION**, and **LON_END_NESTED_UNION**.
- Because some compilers might not allow control over packing and alignment though non-standard keyword extensions, but do support compiler directives (pragmas) for this purpose, the ShortStack Developer's Kit includes two optional include files: **LonBegin.h** and **LonEnd.h**. The **LonBegin.h** file can be optionally (and automatically) inserted prior to any type definition made by the ShortStack LonTalk Compact API files, and the **LonEnd.h** file can be optionally (and automatically) included following the last type definition made by the

ShortStack LonTalk Compact API. This method allows you to use one set of packing and alignment preferences for the ShortStack LonTalk Compact API, and another set of preferences for the remainder of your application.

Example: The **LonBegin.h** file could contain the following directive:

```
#pragma pack(push,1)
```

And the **LonEnd.h** file could contain the following directive:

```
#pragma pack(pop)
```

Refer to your compiler's documentation to determine which directives or other methods for packing and alignment control are supported. Compiler directives (pragmas) are implementation-specific for each ANSI C compiler.

- Enumerations are used to provide literals for many types. Although ANSI C enumerations are derived from a signed integer type, enumerations for a ShortStack application (or a LONWORKS network) must be based on a signed character type (or a signed eight-bit integer). The ShortStack LonTalk Compact API provides a set of macros that allows you to define enumerated types with the possible use of non-standard keyword extensions. It also provides another macro that references an enumerated type so that the reference consumes only a single byte.

  Example: For a compiler that supports a non-standard syntax extension to force an enumeration to fit into a user-defined compound (other than "int"), these macros might be defined as:

  ```
  #define LON_ENUM_BEGIN(n)   enum : LonByte
  #define LON_ENUM_END(n)     n
  #define LON_ENUM(n)         n
  ```

- The ShortStack LonTalk Compact API does not use bit fields. For ANSI C, the standard compound for bit fields is the native word size of the target processor (equivalent to "int"). However, for a ShortStack application (or a LONWORKS network), bit fields must be packed into byte-sized entities. This packing requires non-standard keywords, and another set of implementation-specific controls to determine the placement of the individual bits within each byte. Not all compilers for embedded development support bit fields, or standard ways to control bit fields (for example, anonymous bit fields and zero-length bit fields).

See *Using Types* on page 154 for information about how the LonTalk Interface Developer utility handles data types.

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 & LON_ALPHA_MASK) >>
                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 for the alpha flag:

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

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

Enumerations

The LonTalk Interface Developer utility does not produce enumerations. The ShortStack LonTalk Compact API 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.

A ShortStack 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 19** shows an example enumeration using the ShortStack **LON_ENUM_*** macros, and the equivalent ANSI C enumeration.

Table 19. Enumerations in ShortStack

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

LonPlatform.h

The file within the ShortStack LonTalk Compact API that helps implement the portability concepts described in *Portability Overview* on page 110 is the **LonPlatform.h** include file. The ShortStack LonTalk Compact API and application framework automatically include this file before any other ShortStack LonTalk Compact API-specific definition or file inclusion.

The **LonPlatform.h** file uses conditional compilation to detect the specific compiler and to set various preferences and definitions for portability.

Before you begin porting the ShortStack LonTalk Compact API, you should ensure that the **LonPlatform.h** file includes support for your compiler. The LonTalk Interface Developer utility copies the **LonPlatform.h** file into your project directory so that you can modify the file if it does not include support for your compiler. However, if this file already exists in your project directory, the utility does not overwrite it.

Recommendation: Make any necessary modifications to the copy of the **LonPlatform.h** file in your project directory, rather than modifying the version of the file in the **[*ShortStack*]\api** directory. The master copy of this file might be overwritten when you install service updates or new versions of the ShortStack Developer's Kit.

After you make the appropriate modifications to the **LonPlatform.h** file, you should be able to successfully compile the ShortStack LonTalk Compact API files and the skeleton application framework files generated by the LonTalk Interface Developer utility.

Testing the Ported API Files

After the ShortStack LonTalk Compact API files and the LonTalk Interface Developer utility-generated files can be compiled without errors or significant warnings, you might want to perform a simple test to ensure that the port works correctly.

For this simple test, compile the following source code:

```
#include "LonPlatform.h"
#ifdef  INCLUDE_LON_BEGIN_END
#   include "LonBegin.h"
#endif  /* INCLUDE_LON_BEGIN_END */

LON_ENUM_BEGIN(Color) {
   red, green, blue
} LON_ENUM_END(Color);

LON_STRUCT_BEGIN(Test) {
   LON_ENUM(Color) color;       // offset 0
   LonByte         a;           // offset 1
   LonWord         b;           // offset 2+3

   LON_UNION_NESTED_BEGIN(x) {
      LON_STRUCT_NESTED(r) {
         LonByte   r1;          // offset 4
         LonWord   r2;          // offset 5+6
      } LON_STRUCT_NESTED(r);
      LonWord   w;              // offset 4+5
   } LON_UNION_NESTED_END(x);
} LON_STRUCT_END(Test);

#ifdef  INCLUDE_LON_BEGIN_END
#   include "LonEnd.h"
#endif  /* INCLUDE_LON_BEGIN_END */
```

Link (or include) this code with a test application. The test application can be a simple one, and the ShortStack serial driver is not required. Within the test application, instantiate a variable of type *Test*, using an appropriate set of initial values, as shown in the following example:

```
int main(void) {
    Test test = {
      (LON_ENUM(Color))green, 12, {2, 100}, { 4, {50, 60}}
    };

    return 0;
}
```

Within your development environment, load this test application into your hardware, start a debug session, and use the debugger to inspect the memory image that contains the *test* variable. Verify that the values provided with the initializer can be read at the correct offset locations. For example, the most significant bit of *test.x.w* should evaluate to 4, the least significant bit of *test.x.w* should evaluate to 50, *test.x.r.r1* should be found at offset 4, and so on.

See your development environment documentation for information about using the debugger and inspecting memory at the location of a given variable.

8

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 the *Neuron C Reference Guide*.

Model File Overview

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

The model file describes the functional blocks, network variables, configuration properties, and their relationships, that make up the interoperable interface for a ShortStack 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 a ShortStack application because the model file does not include executable code. All of the tools required to process model files are included with the ShortStack Developer's Kit; you do not need to license another Neuron C development tool to work with a ShortStack model file. The model file uses Neuron C Version 2 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. The *Neuron C Reference Guide* 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 ShortStack Developer's Kit, and must be licensed separately. See the *NodeBuilder FX User's Guide* for details about using the NodeBuilder Code Wizard.

See the *Neuron C Reference Guide* 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 a ShortStack Application

Within a 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_lux" with the name "nviLux".

```
network input SNVT_lux nviLux;
```

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_lux" with the name "nvoLux".

```
network output SNVT_lux nvoLux;
```

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

See the *Neuron C Reference Guide* 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 *nviLux* and *nvoLux* pair of network variables above, the utility generates a standard ANSI C type definition for the *SNVT_lux* network variable type and implements two global C-language variables:

```
typedef ncuLong  SNVT_lux;
…
volatile SNVT_lux nviLux;
SNVT_lux nvoLux;
```

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

Your ShortStack application can simply read the *nviLux* 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 *nvoLux* global C variable, and call the appropriate ShortStack LonTalk Compact 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_lux* 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 specify network variable members that are generic so that the type can be selected by each implementation of the profile. The *SFPTopenLoopSensor* functional block (described in the *Defining a Functional Block* on page 119) is an example for such a functional profile with generic members. This profile defines the *nvoValue* member to be of type *SNVT_xxx*, which means "any standard network variable type."

Implementing a profile with generic members 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 122 for more information about implementing changeable-type network variables for ShortStack applications.

You can also define your own nonstandard data types. The NodeBuilder Resource Editor utility, which is included with the ShortStack 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 132. You can find detailed documentation for each of the standard functional profiles at types.lonmark.org[7].

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 132.

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

```
network output SNVT_lux nvoLux;

fblock SFPTopenLoopSensor {
   nvoLux implements nvoValue;
} fbLightMeter;
```

This functional block:

- Is named *fbLightMeter* (network management tools use this name unless you include the **external_name** keyword to define a more human-readable name)

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

- Implements the **SFPTopenLoopSensor** standard profile
- Includes a single network variable, named *nvoLux*, 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 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_lux nvoLux[10];

fblock SFPTopenLoopSensor {
   nvoLux[0] implements nvoValue;
} fbLightingDevice[10];
```

This functional block array:

- Contains ten functional blocks, *fbLightingDevice[0]* to *fbLightingDevice[9]*, each implementing the **SFPTopenLoopSensor** profile.
- Distributes the ten *nvoLux* 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 132, 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 53**.

Figure 53. 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 54** shows the same switch being used to control three lights.

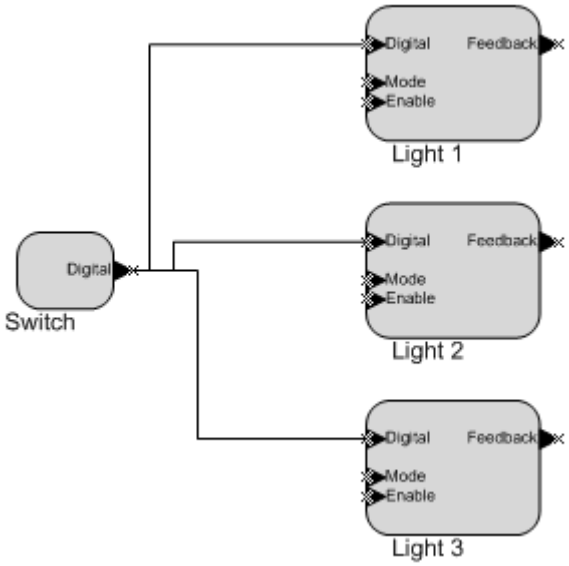

Figure 54. Simple Switch Controlling Three Lights

The ShortStack 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 ShortStack application updates a value for an output network variable, it passes the new value to the ShortStack Micro Server by using a simple API function call.

Through a process called *binding* that takes place during network design and installation, the ShortStack Micro Server 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 ShortStack Micro Server assembles and sends the appropriate packets to these devices. Similarly, when the ShortStack Micro Server 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, tools 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 ShortStack application does not initiate type changes.

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

1. Declare the network variable (in the model file) 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 (part of the LonTalk Interface Developer utility) to set the changeable-interface bit in the program ID when you create the device interface.
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 ShortStack 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 (and later), supports changeable-type network variables. However, if you use LonMaker 3 or earlier to manage a ShortStack device with changeable-type network variables, you must explicitly set the **SCPTnvType** CP value in the LonMaker browser (or in a device plug-in) to inform the ShortStack Micro Server of the type changes in addition to using the "Change Network Variable Type" facility that is provided with LonMaker 3 or earlier to change the type of a network variable in the LNS database.

See *Handling Changes to Changeable-Type Network Variables* on page 176 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 set point used during daytime control, or the limit value 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 132, 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 network variable
- A configuration file

A *configuration network variable* (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 a ShortStack application is limited to a maximum of 254 network variables, including CPNVs. Use the **network … config_prop** syntax described in the *Neuron C Reference Guide* 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 file access method, the total size of the directory, template file, and value files cannot exceed 32 KB. The maximum depends on the specified Micro Server, and is typically several kilobytes. The standard Micro Servers that are included with the ShortStack Developer's Kit support over 11 KB. When you use LW-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) exceeds the total number of available network variables (a maximum of 254 for a ShortStack device, but potentially fewer than 254 depending on the resources available on a particular Micro Server).
- 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 file (DMF) access method 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, LW-FTP can be faster than DMF if there are many configuration properties to be updated.

Use the **cp_family** syntax described in the *Neuron C Reference Guide* 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 (LW-FTP)
- Support for the direct memory files access method

The LonTalk Interface Developer provides most of the required code to support the direct memory file access mthod. However, if you use LW-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, or if your application implements additional files that require LW-FTP.

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 189 for more information about the direct memory file access method.

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 LW-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**. 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 file access method 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 LW-FTP server or direct memory file 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 {
  // Declare 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 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 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 20 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 20. 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 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-channel (red-green-blue, RGB) light sensor, implemented with an array of three **SFPTopenLoopSensor** functional blocks. Each channel has individual illumination set points, but shares one property to specify the sample rate for all three channels.

```
#define  NUM_CHANNELS   3

SCPTluxSetPoint  cp_family cpLuxSetPoint[NUM_CHANNELS];
SCPTupdateRate   cp_family cpUpdateRate;

network output SNVT_lux nvoLux[NUM_CHANNELS];

fblock SFPTopenLoopSensor {
   nvoLux[0] implements nvoValue;
} fbLightSensor[NUM_CHANNELS] external_name("Light Sensor")
   fb_properties {
      cpLuxSetPointp[0],
      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_lux** in this case.

Example 1:

```
SCPTdefOutput cp_family cpDefaultOutput;

network output SNVT_lux nvoLux nv_properties {
   cpDefaultOutput = 450
};

fblock SFPTopenLoopSensor {
   nvoLux implements nvoValue;
} fbLightSensor;
```

The initial value (450) 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 changeable-type network variables. 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_lux nvoValue
   nv_properties {
      cpDefaultOutput = 450,
      cpNvType
   };

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

The **nvoValue** principal network variable, although it is of changeable type, must still implement a default type (**SNVT_lux** 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_lev_percent**, 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 (LW-FTP) and Interoperable Self-Installation (ISI) protocol, and are also used to implement proprietary interfaces to LONWORKS devices as described in Chapter 10, *Developing a ShortStack Application*, on page 163.

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 entry 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 entries.

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 language-specific extensions).

Each set defines functional profiles, network variable types, and configuration property type for a particular type of device or set of device types. The applicable device types are specified by a range of program IDs, where the program ID range is determined by a *program ID template*, and a *scope* value in 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 – Reserved
Reserved for future use.

2 – Reserved
Reserved for future use.

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 3 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 ShortStack 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 Turbo automatically sets the scope to the highest (most specific) applicable scope level. However, if you use LNS 3 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 FX 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.
4. Add implementation-specific members to the functional block.

Writing Acceptable Neuron C Code

When processing a 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 281, lists the acceptable and ignored compiler directives for model files. All other compiler directives are not accepted by the LonTalk Interface Developer utility and cause an error if included in a model file. A statement can be unacceptable because it controls features that are meaningless in a ShortStack device, or because it refers to attributes that are determined by the ShortStack Micro Server 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 **_SHORTSTACK** and **_MODEL_FILE** macros, so that you can use **#ifdef** or **#ifndef** directives to control conditional compilation of source code that is used for regular Neuron C compilation and as a ShortStack 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 syntax described in this manual and the *Neuron C Reference Guide*. 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

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 an authorized device, not from a person or device attempting to break into the system.

Authentication doubles the number of messages per transaction. An unauthenticated acknowledged message normally requires two messages: an update and an acknowledgment. An authenticated message requires four messages, as shown in **Figure 55** on page 138. These extra messages can affect system response time and channel 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.

If you have a NodeBuilder license, you can also create a custom Micro Server with a pre-set authentication key.

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 137. 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 **ShortStackTypes.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.

To produce highly secured ShortStack devices, consider creating a custom Micro Server using the NodeBuilder Development tool, exporting the generated image with the authentication keys pre-set. See the *NodeBuilder FX User's Guide* for more information.

How Authentication Works

Figure 55 on page 138 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 (built in to the Smart Transceiver firmware) 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.

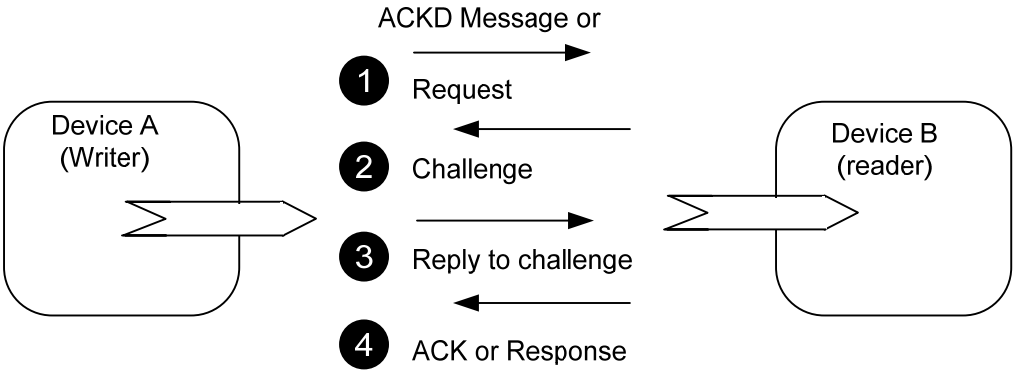

Figure 55. Authentication Process

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, a ShortStack 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.

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 156 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 **ShortStackDev.c** file:

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

When an update occurs for the input network variable (**nviCount**), the Micro Server 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 Micro Server 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 154.

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 lighting device, where all input data is retrieved from the network through the **nviLux** and **nviColor** input network variables. The result is posted to the **nvoRed**, **nvoGreen**, and **nvoBlue** output network variables. An additional **nvoUsage** output network variable is polled and uses non-volatile storage to count the device's total lifetime.

```
network input                 SNVT_lux          nviLux;
network input                 SNVT_color        nviColor;
network output                SNVT_lev_percent  nvoRed;
network output                SNVT_lev_percent  nvoGreen;
network output                SNVT_lev_percent  nvoBlue;
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_lux nviLux;
```

```
volatile SNVT_color nviColor;
SNVT_lev_percent nvoRed;
SNVT_lev_percent nvoGreen;
SNVT_lev_percent nvoBlue;
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 **ShortStackDev.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 lighting device 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 lighting devices, each based on the same user-defined *UFPTlightingDevice* profile, implementing three identical devices.

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");

// UFPTlightingDevice
// Implements the device from the previous example.
network input           SNVT_lux          nviLux[3];
network input           SNVT_color        nviColor[3];
network output          SNVT_lev_percent  nvoRed[3];
network output          SNVT_lev_percent  nvoGreen[3];
network output          SNVT_lev_percent  nvoBlue[3];
network output polled eeprom  SNVT_elapsed_tm
   nvoUsage[3];

fblock UFPTlightingDevice {
   nviLux[0]      implements nviLux;
   nviColor[0]    implements nviColor;
   nvoRed[0]      implements nvoRed;
   nvoGreen[0]    implements nvoGreen;
   nvoBlue[0]     implements nvoBlue;

   nvoUsage[0]    implements nvoUsage;
```

```
} LightingDevice[3] external_name("LightingDevice");
```

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 Lighting Device
network input cp SCPTluxSetpoint nciLuxSetpoint[3];
network input cp SCPTupdateRate nciUpdateRate;

// network variables for the Lighting Device
network input           SNVT_lux          nviLux[3];
network input           SNVT_color        nviColor[3];
network output          SNVT_lev_percent  nvoRed[3];
network output          SNVT_lev_percent  nvoGreen[3];
network output          SNVT_lev_percent  nvoBlue[3];
network output polled eeprom  SNVT_elapsed_tm
   nvoUsage[3];

fblock UFPTlightingDevice {
   nviLux[0]      implements nviLux;
   nviColor[0]    implements nviColor;
   nvoRed[0]      implements nvoRed;
   nvoGreen[0]    implements nvoGreen;
   nvoBlue[0]     implements nvoBlue;
   nvoUsage[0]    implements nvoUsage;
} LightingDevice[3] external_name("LightingDevice")
fb_properties {
   nciLuxSetPoint[0],
   static nciUpdateRate
```

```
};
```

This example implements an array of configuration network variables, *nciLuxSetPoint*. Each element of this array applies to one element of the *LightingDevice* functional block array, starting with *nciLuxSetPoint[0]*.

The **SCPTupdateRate** configuration property *nciUpdateRate* is shared among all three lighting devices. There is only a single *nciUpdateRate* configuration property, and it applies to every element of the array of three *UFPTlightingDevice* functional blocks.

The LonTalk Interface Developer utility creates a network variable table for the configuration network variables and the persistent *nvoUsage* network variable. You must provide persistent storage for such data. See *Providing Persistent Storage for Non-Volatile Data* on page 192 for more information about support for non-volatile data.

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 file access method
3. The *SFPTnodeObject* node object has been updated 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 Lighting Device
SCPTluxSetpoint cp_family cpLuxSetpoint[3];
SCPTupdateRate cp_family cpUpdateRate;

// network variables for the Lighting Device
network input           SNVT_lux          nviLux[3];
network input           SNVT_color        nviColor[3];
network output          SNVT_lev_percent  nvoRed[3];
```

```
network output          SNVT_lev_percent  nvoGreen[3];
network output          SNVT_lev_percent  nvoBlue[3];
network output polled eeprom  SNVT_elapsed_tm
   nvoUsage[3];

fblock UFPTlightingDevice {
   nviLux[0]      implements nviLux;
   nviColor[0]    implements nviColor;
   nvoRed[0]      implements nvoRed;
   nvoGreen[0]    implements nvoGreen;
   nvoBlue[0]     implements nvoBlue;
   nvoUsage[0]    implements nvoUsage;
} LightingDevice[3] external_name("LightingDevice")
fb_properties {
   cpLuxSetPoint[0],
   static cpUpdateRate
};
```

The addition of the *SNVT_address* typed network variable *nvoFileDirectory* is important for enabling the direct memory file access method 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 189 for more information.

Alternatively, you can use the LONWORKS File Transfer Protocol (LW-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.

9

Using the LonTalk Interface Developer Utility

You use the model file, described in Chapter 8, 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 can use this skeleton application framework as the basis for your own 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 can use the LonTalk Interface Developer utility to create the device interface data required for your ShortStack application. The LonTalk Interface Developer utility also generates the device interface (XIF) file that is required by network management tools to design a network that uses your device.

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

1. Create a model file as described in Chapter 8, *Creating a Model File*, on page 115.
2. Start the LonTalk Interface Developer utility: from the Windows **Start** menu, select **Programs → Echelon ShortStack FX Developer's Kit → LonTalk Interface Developer**.
3. In the LonTalk Interface Developer utility, specify the Micro Server, 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 150.
4. Add the **ShortStackDev.h** ANSI C header file to your ShortStack application with an include statement:

   ```
   #include "ShortStackDev.h"
   ```

5. Add the **ShortStackDev.c** file, and the executable ShortStack LonTalk Compact API source files (**ShortStackApi.c**, **ShortStackInternal.c**, and **ShortStackHandlers.c**) to your project.

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 ShortStack LonTalk Compact API files.

After you have the LonTalk Interface Developer utility-generated files, you need to modify and add code to your application, using the ShortStack LonTalk Compact API, to implement desired LONWORKS functionality into your ShortStack application. See Chapter 10, *Developing a ShortStack Application*, on page 163, for information about how to use the ShortStack LonTalk Compact 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 ShortStack 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 Micro Server

From the ShortStack Micro Server Selection page of the utility, you can specify the following information about the ShortStack Micro Server:

- The Micro Server type
- The Smart Transceiver type
- The Smart Transceiver external clock speed
- The Smart Transceiver clock multiplier value

The LonTalk Interface Developer utility reads the ShortStack Micro Server Database file (**StdServers.xml**) and the User Database file (**UserServer.xml**) to display the values for each of the standard and custom Micro Servers. For a standard installation, the **StdServers.xml** file is in the **\LonWorks\ShortStack\MicroServers** directory.

To select a Micro Server that is not in the database file, click **Browse** to specify the Micro Server's XIF file. In this case, the LonTalk Interface Developer utility presents the chosen Micro Server's properties as indicated in the XIF file, but for a custom Micro Server that is based on a Series 5000 Smart Transceiver or Neuron 5000 Processor, you must select the correct system clock multiplier.

Specifying System Preferences

From the ShortStack System Preferences page of the utility, you can specify the following general preferences:

- Whether explicit addresses should be enabled
- Whether application addresses should be enabled
- Whether the notification of service-pin-held events is enabled
- If service-pin-held events are enabled, how long the service pin must be pressed and held before the Micro Server receives the event

These preferences are all optional.

Recommendation: Enable explicit and application addressing only if they are needed, because they increase the size of buffers and require additional memory on the ShortStack host processor.

When you press the local service pin on the device, the Micro Server 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 Micro Server sends a service-pin-held event depends on how you configure it.

Although your ShortStack device is not required to handle service-pin-held events, you can include support in your application to receive them, even if it does not process them. By including support for receiving the service-pin-held events, you have the flexibility to add support for processing them in the future without modifying the ShortStack Micro Server image. 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.

Specifying the Device Program ID

In the Program ID Selection window, you 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 version of this list is included with the ShortStack 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. There is no charge for a temporary manufacturer ID, and you do not have to join LONMARK International to obtain one.

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. You can, however, produce a device with a temporary manufacturer ID.

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 can 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 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 8, *Creating a Model File*, on page 115, 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 **_SHORTSTACK** symbol is always predefined by the LonTalk Interface Developer utility, and does not need to be specified explicitly. You can use this symbol to control conditional compilation for ShortStack applications. In addition, the utility predefines the **_MODEL_FILE** symbol for model file definitions and the **_LID4** symbol 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, whether to include the optional Query and Update functions, and whether to include the optional interoperable self-installation (ISI) API functions.

If you use the direct memory files (DMF) access method, you can optionally specify the size and starting address of the memory window from the Interface Developer Code Generator Preferences page. If you do not specify values on this

page, the utility assigns appropriate values. See *Using Direct Memory Files* on page 189 for more information.

Compiling and Generating the Files

In the Summary and Confirmation page of the utility, you can view all of the information that you specified for the project. When you click **Next**, the LonTalk Interface Developer utility compiles the model file and generates a number of C source files and header files, as described in *Using the LonTalk Interface Developer Files*.

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

Any warning or error messages have the following format:

Message-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 semicolon 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 semicolon), and the compiler message number is NCC#21. To fix this error, add a semicolon to the end of the line.

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

Using the LonTalk Interface Developer Files

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

- **LonNvTypes.h**
- **LonCpTypes.h**
- **ShortStackDev.h**
- **ShortStackDev.c**
- ***project*.xif**
- ***project*.xfb**

The utility also copies a number of files to your project directory, as described in *Copied Files* on page 151. Together, the generated files and the copied files form the ShortStack application framework, which defines the ShortStack Micro Server initialization data and self-identification data for use in the 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 **ShortStackDev.h** file in your ShortStack application using an ANSI C **#include** statement, and add the **ShortStackDev.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:

- **LonBegin.h**
- **LonEnd.h**
- **LonPlatform.h**
- **ShortStackApi.c**
- **ShortStackApi.h**
- **ShortStackHandlers.c**
- **ShortStackInternal.c**
- **ShortStackTypes.h**

For ShortStack ISI applications, the LonTalk Interface Developer utility also copies the following files into your project directory:

- **ShortStackIsiApi.c**
- **ShortStackIsiApi.h**
- **ShortStackIsiHandlers.c**
- **ShortStackIsiInternal.c**
- **ShortStackIsiTypes.h**

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

Because your application includes the **ShortStackDev.h** file (and **ShortStackIsiApi.h** for ShortStack ISI applications), you do not normally have to explicitly include any of the header files with your application source.

You must add the **ShortStackInternal.c**, **ShortStackApi.c**, and **ShortStackHandlers.c** files to your project so that they will be compiled and linked with your application. For ShortStack ISI applications, you must add the **ShortStackIsiInternal.c**, **ShortStackIsiApi.c** and **ShortStackIsiHandlers.c** files to your project so that they will be compiled and linked with your application.

The LonTalk Interface Developer utility also copies a number Micro Server image files into your project directory if no file with the same file already exists. These image files are based on the Micro Server preferences specified in the utility, and are renamed to share the project's base name. The available file extensions depend on the selected Micro Server, but typically include files with APB, NDL, NEI, NXE, NME, or NMF file extensions.

Important: The LonTalk Interface Developer utility does not copy implementations of the serial driver into your project folder; you must supply this code.

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 154 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 network variables or configuration properties.

ShortStackDev.h

The **ShortStackDev.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 ShortStack LonTalk Compact 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 make 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. The file contains comments about how you can override some of the preferences and assumptions made by the utility.

ShortStackDev.c

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

It defines variables and constants, including the network variable table or the device's initialization data block, and a number of utility functions.

The **ShortStackDev.c** file also defines the **appInitData** structure, which contains data that is sent to the Micro Server during initialization (in the **LonNiAppInit** and **LonNiNvInit** messages). **Table 21** on page 153 describes the fields of this data structure.

Important: Although you can modify this data structure, you should not need to unless you are developing an application that supports multiple device interfaces. If you do modify this data, you must ensure that other control data remains consistent with your changes, including the **siData** array and the **nvTable** (both in **ShortStackDev.c**), and the device interface files (XIF and XFB file extensions). Other data that also must remain consistent with your preferences are definitions contained in the **ShortStackDev.h** file, including those that configure the API options.

Table 21. Fields of the appInitData Structure

Field	Description
appInitData.signature	A 16-bit number that identifies the current application. The LonTalk Interface Developer utility generates a new number whenever you regenerate the application framework. The Micro Server uses this number to distinguish repeated initialization of the same application from initialization of a new application.
appInitData.programId	The 48-bit program ID in binary form.
appInitData.communication	The 96-bit communication parameter record that is used to correctly initialize communications with the LONWORKS network.
appInitData.preferences	An 8-bit vector of flags. Includes 0x20 to enable explicit addressing, and a 5-bit value for the service-pin-held delay in seconds (mask 0x1F), where zero disables the feature. The remaining flags 0x80 and 0x40 are reserved for future use, and must be kept cleared (zero). These flags are optional.
appInitData.nvCount	One byte for the total number of network variables in the application. This number must not exceed the Micro Server's maximum network variable count (also known as the Micro Server's network variable capacity).
appInitData.nvData[]	One byte for each network variable. Each byte comprises the following flags: priority (0x80), output (0x40), service type (acknowledged [0x00], repeated [0x10], unacknowledged [0x20]), and authenticated (0x08).

You must compile and link the **ShortStackDev.c** 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 about 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 ShortStack project that you specified on the Welcome to LonTalk Interface Developer page of the LonTalk Interface Developer utility. Thus, these files have the same name as the ShortStack project file (**.lidprj** extension).

These files comply with the LONMARK device interface revision 4.402 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, and should conform to the portability rules described in *Porting the ShortStack LonTalk Compact API* on page 109. 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 NodeBuilder Resource Editor to define your own UNVTs.

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. See in *Porting the ShortStack LonTalk Compact API* on page 109 for more information.

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 variable and configuration property declarations 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 **ShortStackDev.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 **ShortStackDev.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 application framework generated by the LonTalk Interface Developer utility also includes the network variable table, which is a table that allows the ShortStack LonTalk Compact API to locate the network variable's value in memory and access other attributes of each network variable.

The ShortStack LonTalk Compact API, upon receipt of an incoming network variable update, automatically moves data into the corresponding input network variable and signals this event by calling a callback 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 a ShortStack Application* on page 163 for information about the development of a ShortStack application using the LonTalk Interface Developer utility-generated code.

The utility generates a configuration file in **ShortStackDev.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 **ShortStackDev.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 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. See *Providing Persistent Storage for Non-Volatile Data* on page 192 for more information about persistent storage considerations.

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 ShortStack 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 one that must always be read from the device by an external tool or application, rather than relying upon the value in the device interface file or upon a value stored in a network database. For example, you can use device-specific configuration property for a setpoint that is updated by a local operator interface on the device, or for a minor version number that varies from device to device.

A device-specific configuration property can be set by the device that implements the configuration property, by another device, or by a configuration tool. Network management tools must never change a device-specific configuration property value, except as a side effect of a new program download, device re-commissioning, or device replacement.

For a ShortStack application, you can specify a device-specific configuration property by specifying the **device_specific** modifier for the configuration property. You can specify the **device_specific** modifier independently of the **const** modifier. For example, specify **device_specific**, but not **const**, for a configuration property that contains a setpoint that is updated by a local operator interface on the device, and allow the setpoint to be modified by both the host application and qualifying network tools.

In some cases, you might want to set up a configuration property that is modifiable by the host application, but not by any other entities on the network. In this case, perform the following steps:

1. Declare the configuration property as **const** and, if applicable, **device_specific**.
2. At the top of the **ShortStackDev.c** file, before you include the **ShortStackDev.h** header file, define the **LON_READONLY_FILE_IS_WRITEABLE** macro with 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 **ShortStackDev.h** header file to define the read-only file's storage type with the **LON_READONLY_FILE_STORAGE_TYPE** macro, which in turn is used in the declaration and specification of the **lonReadOnlyValueFile** variable.

If you define the **LON_READONLY_FILE_IS_WRITEABLE** macro to 1, the read-only value file is writable by the local application. Because the read-only value file is now allocated in volatile memory, your driver for non-volatile data must also read and write the read-only value file.

For the network management tool, however, the read-only file remains non-writable. If your application uses the direct memory file access method 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 all write operations to the read-only value file.

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 ShortStack LonTalk Compact API. The network variable table exists only if the model file contains one or more network variables. The **LonGetNvTable()** function, used by the ShortStack LonTalk Compact API, returns the base of the network variable table or NULL if the table does not exist.

Example: A model file contains the following Neuron C declaration:

```
network input SNVT_count nviCount;
```

The LonTalk Interface Developer utility generates code to define the network variable as follows:

```
volatile SNVT_count nviCount;
```

The utility generates a pointer to the **nviCount** variable in the network variable table. The ShortStack LonTalk Compact API uses the **pData** pointer provided by the network variable table to update the **nviCount** network variable.

A ShortStack application typically accesses a network variable value through the C global variable that implements the network variable. However, the ShortStack LonTalk Compact 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);
```

The **LonGetNvValue()** function returns NULL for an invalid index, or a pointer to the value.

Applications that are designed to share application code between ShortStack and FTXL, and that are designed to support dynamic network variables, should use the **LonGetNvValue()** function because the FTXL LonTalk API requires use of the **LonGetNvValue()** function for dynamic network variables. ShortStack does not support dynamic network variables.

Network Variable Attributes

The network variable table contains an attribute byte that contains the following flags for each network variable:

- IsOutput
- IsPersist
- IsPolled
- IsSync
- IsChangeable

All network variable flags are implemented as structures, as described in *Bit Field Members* on page 111, to minimize host memory usage. The type of the network variable table, **LonNvDescription**, and the various macros to access these attributes, are defined in the **ShortStackDev.h** file.

The **IsOutput** flag identifies an output network variable. It is true for output network variables and false for input network variables. This flag is set for all network variables declared with the **output** keyword in the model file.

The ShortStack LonTalk Compact API uses the **IsOutput** flag to prevent propagating outputs to input network variables, and to prevent a poll of an output network variable on the ShortStack device.

The **IsPersist** flag indicates that a network variable must be kept in persistent storage. This flag is set for all network variables declared with the **eeprom**, **config_prop**, or **cp** keywords in the model file. See *Providing Persistent Storage for Non-Volatile Data* on page 192 for more information about persistent data.

The **IsPolled** flag indicates that a network variable is a polled network variable. The flag is set for all network variables declared with the **polled** modifier in the model file.

The **IsSync** flag indicates that a network variable is a synchronous network variable. This flag is set for all network variables declared with the **sync** modifier in the model file. This modifier specifies that all values assigned to this network variable must be propagated, in their original order. It is mutually exclusive with the **polled** modifier.

The ShortStack LonTalk Compact API does not enforce processing for critical sections. Therefore, your application must implement any required processing to ensure synchronous outputs when the **IsSync** flag is set. However, a typical ShortStack application does not require special design for synchronous outputs, because a typical ShortStack application treats all output network variables as synchronous (that is, the application calls **LonPropagateNv()** whenever it computes a new value for the network variable, which causes immediate propagation of the network variable to the network). More advanced applications that implement critical sections, during which only the last of several possible assignments to a particular network variable is preserved and propagated, must honor the **IsSync** flag to allow for the exceptional case where all value assignments must be propagated.

The **IsChangeable** flag indicates that a network variable has a changeable type. See *Defining a Changeable-Type Network Variable* on page 122 for more information about changeable-type network variables.

See *Developing a ShortStack Application* on page 163 for more information about propagation of network variable updates.

The Message Tag Table

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 messages. This flag is set for all message tags declared with the **bind_info(nonbind)** modifier in the model file.

The LonTalk Interface Developer utility declares the message tag table in **ShortStackDev.c** if you declare one or more message tags in the model file. The **LonGetMtTable()** function, used by the ShortStack LonTalk Compact API, returns the base of the message tag table or NULL if the table does not exist.

The message tag table is only used by the ShortStack LonTalk Compact API and is not used by your application. The ShortStack LonTalk Compact API uses the table to determine if an implicitly addressed message can be sent.

10

Developing a ShortStack Application

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

Overview of a ShortStack Application

This chapter describes how to use the ShortStack LonTalk Compact API and the device interface data produced by the LonTalk Interface Developer utility to perform the following tasks:

- Use the ShortStack LonTalk Compact API
- Understand how to use the API with a multitasking operating system
- Initialize the ShortStack LonTalk Compact API
- Periodically call the ShortStack event handler
- Send information to other devices using network variables
- Handle network variable poll requests from other devices
- Handle updates to changeable-type network variables
- Receive information from other devices using network variables
- Communicate with other devices using application messages
- Handle network management commands
- Handle Micro Server reset events
- Query the error log
- Reinitialize the Micro Server
- Provide persistent storage for non-volatile data and use the direct memory files feature

Most ShortStack applications need to perform only the tasks that relate to persistent storage, initialization, periodically calling the **LonEventhandler()** function, sending and receiving network variables, and handling network management commands.

This chapter assumes that you have completed the device development described in the preceding chapters. This chapter shows 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 ShortStack LonTalk Compact API

Within the seven-layer OSI Model protocol, the ShortStack LonTalk Compact API forms the majority of the Presentation layer, and provides the interface between the serial driver in the Session layer and the host application in the Application layer, as shown in **Figure 56** on page 165.

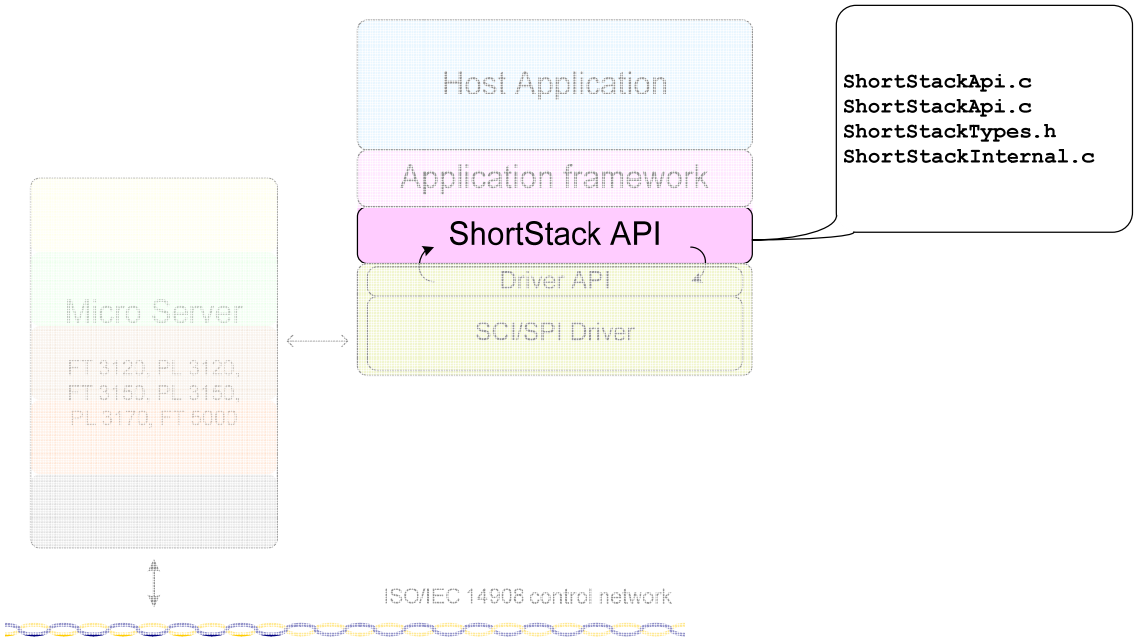

Figure 56. The ShortStack LonTalk Compact API within the OSI Model

The ShortStack LonTalk Compact API is comprised primarily of the following two ANSI C source files:

- [*ShortStack*]**API****ShortStackApi.c**
- [*ShortStack*]**API****ShortStackHandlers.c**

The **ShortStackApi.c** source file contains the core of the ShortStack LonTalk Compact API, which provides functions for handling network events, propagating network variables, responding to network variable poll requests, and so on.

A ShortStack application must call the **LonEventHandler()** API function periodically to process any pending uplink messages. This function calls specific API functions based on the type of event, and then calls callback functions to notify the application layer of these network events.

Generally, you do not need to change the ShortStack API files for each of your applications, but you might have to make some changes when porting the API source code to your target platform and environment.

The ShortStack application framework connects the ShortStack API with your application, as shown in **Figure 57** on page 166.

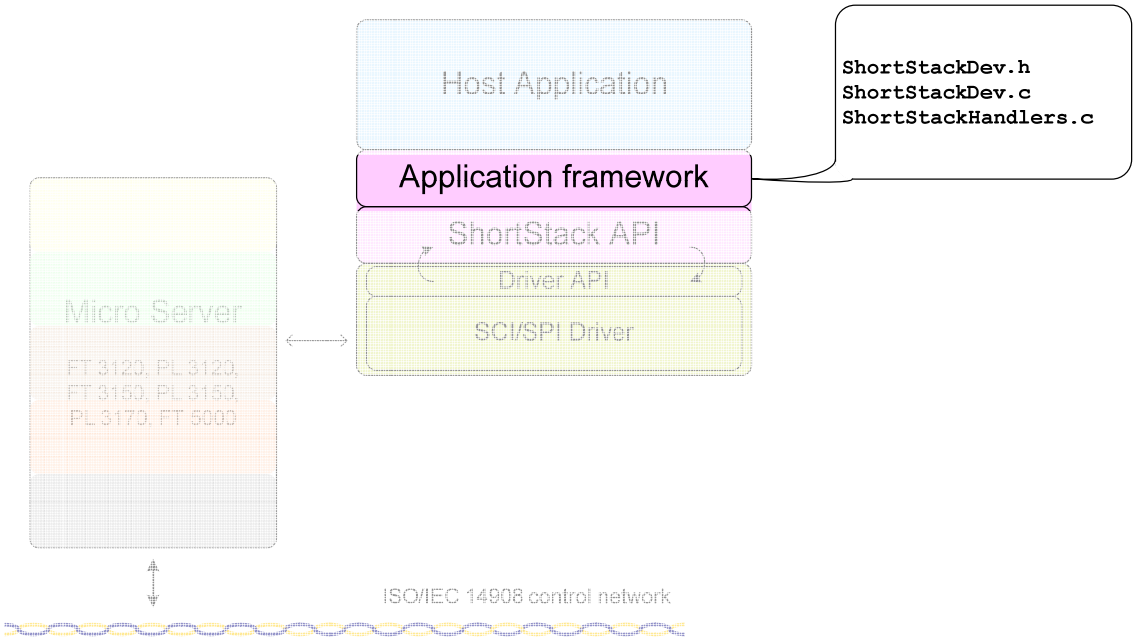

Figure 57. The ShortStack Application Framework

Note that neither **Figure 56** nor **Figure 57** shows the API or framework files that are required for ShortStack ISI applications; see Chapter 11, *Developing a ShortStack Application with ISI*, on page 197, for information about supporting ISI in your ShortStack application.

The **ShortStackHandlers.c** source file contains stubs for the callback handler functions that the ShortStack LonTalk Compact API calls. You must add code to these callback stubs to respond to specific network events. For example, the **LonNvUpdateOccurred()** callback could inform the application of the arrival of new data for a set-point value, and the callback code could re-calculate the controller's response, assign output values to peripheral I/O devices, and so on.

The following recommendations can help you manage your ShortStack application project:

- Keep edits to LonTalk Interface Developer utility-generated files to a minimum, that is, do not edit the **LonNvTypes.h**, **LonCpTypes.h**, **ShortStackDev.h**, or **ShortStackDev.c** files unless necessary.
- Add **#include "ShortStackDev.h"** to your application source files to provide access to network variable types and instantiations.
- Keep changes to the **ShortStackHandlers.c** and **ShortStackHandlers.h** files to a minimum:
    - Add calls to your own functions in files that you create and maintain.
    - Future versions or fixes to the ShortStack product might affect these API files.
- Consider using an event-driven (signaled) model, in addition to using the idle-loop calls to the **LonEventHandler()** function, to provide enhanced device responsiveness.

- The ShortStack LonTalk Compact API is a non-reentrant, single-threaded API, as described in *Using the ShortStack LonTalk Compact API in Multiple Contexts*.

Using the ShortStack LonTalk Compact API in Multiple Contexts

Although a ShortStack application does not require an operating system, you can use the ShortStack LonTalk Compact API with an operating system that supports multiple system execution contexts. A context could be a process, thread, task, interrupt service routine, or the operating system's main thread of execution, as defined by the operating system.

A typical ShortStack application would use one or more execution contexts for the link-layer driver, and use a different execution context for both the ShortStack LonTalk Compact API functions and callback handler functions.

The ShortStack LonTalk Compact API is a non-reentrant, single-threaded API. If your application uses a multi-tasking (or multi-threading) environment or interrupt service routines to access the ShortStack LonTalk Compact API, you must ensure that only one task (or thread or interrupt) accesses the ShortStack LonTalk Compact API. The same task that calls the **LonInit()** and **LonEventHandler()** functions should also be the only task that calls the ShortStack LonTalk Compact API.

In a multi-tasking environment, the link-layer driver would typically consist of USART transmit and receive interrupts, with interrupts that respond to changes on the link-layer handshake lines.

The example applications that are available from www.echelon.com/shortstack (such as the ARM7 Example Port) use a single execution context for the link-layer driver, all ShortStack LonTalk Compact API functions, and the callback handler functions.

If your application requires the use of multiple contexts, one possible approach would be to provide two execution contexts (in addition to those used by the link-layer driver): one to call all ShortStack LonTalk Compact API functions (such as **LonInit()** and **LonPropagateNv()**), and another to call the **LonEventHandler()** function. The execution context that calls the **LonEventHandler()** function also defines the context for the ShortStack callback handler functions. For such an approach, you must supply appropriate inter-context communication and synchronization, and implement and test any related API changes.

Tasks Performed by a ShortStack Application

The general ShortStack application life cycle includes two phases:

- Initialization
- Normal processing

The initialization phase of a ShortStack application typically occurs during each power-up or reset of the host application, but can also be repeated as necessary. The initialization phase defines basic parameters for the LONWORKS network communication, such as the communication parameters for the physical transceiver in use, and defines the application's device interface: its network

variables, configuration properties, and self-documentation data. Successful completion of the initialization phase causes the Micro Server to leave quiet mode, after which it can send and receive messages over the network.

Your application does not always need to run its initialization code when the Micro Server is reset. For example, the Micro Server can be reset by the network management tool to change the device's state. Your application can use the **LonResetNotification** message provided to the **LonReset()** callback handler function to determine the Micro Server's state and last reset cause, and determine whether re-initialization is required.

The Micro Server might also reset during normal operation when a configuration property (declared with the **reset_required** modifier) value changes. This changes acts as a notification that the application, but not necessarily the Micro Server and the ShortStack device as a whole, should reinitialize.

Recommendations:

- When the host processor powers-up or resets, reinitialize the ShortStack device.
- When the Micro Server reports that it is not initialized after a reset (check the **Initialized** flag of the **LonResetNotification** message), reinitialize the ShortStack device.

During normal processing, the application periodically calls the **LonEventHandler()** API function, which calls the serial driver API and might call callback functions (such as the **LonNvUpdateOccurred()** callback). Other API functions allow the ShortStack application to initiate transactions. Such a transaction might in turn lead to calling other callback functions (such as the **LonNvUpdateCompleted()** callback).

Figure 58 on page 169 shows how the ShortStack application, ShortStack LonTalk Compact API, and callback functions work together during the two phases of the application's life cycle.

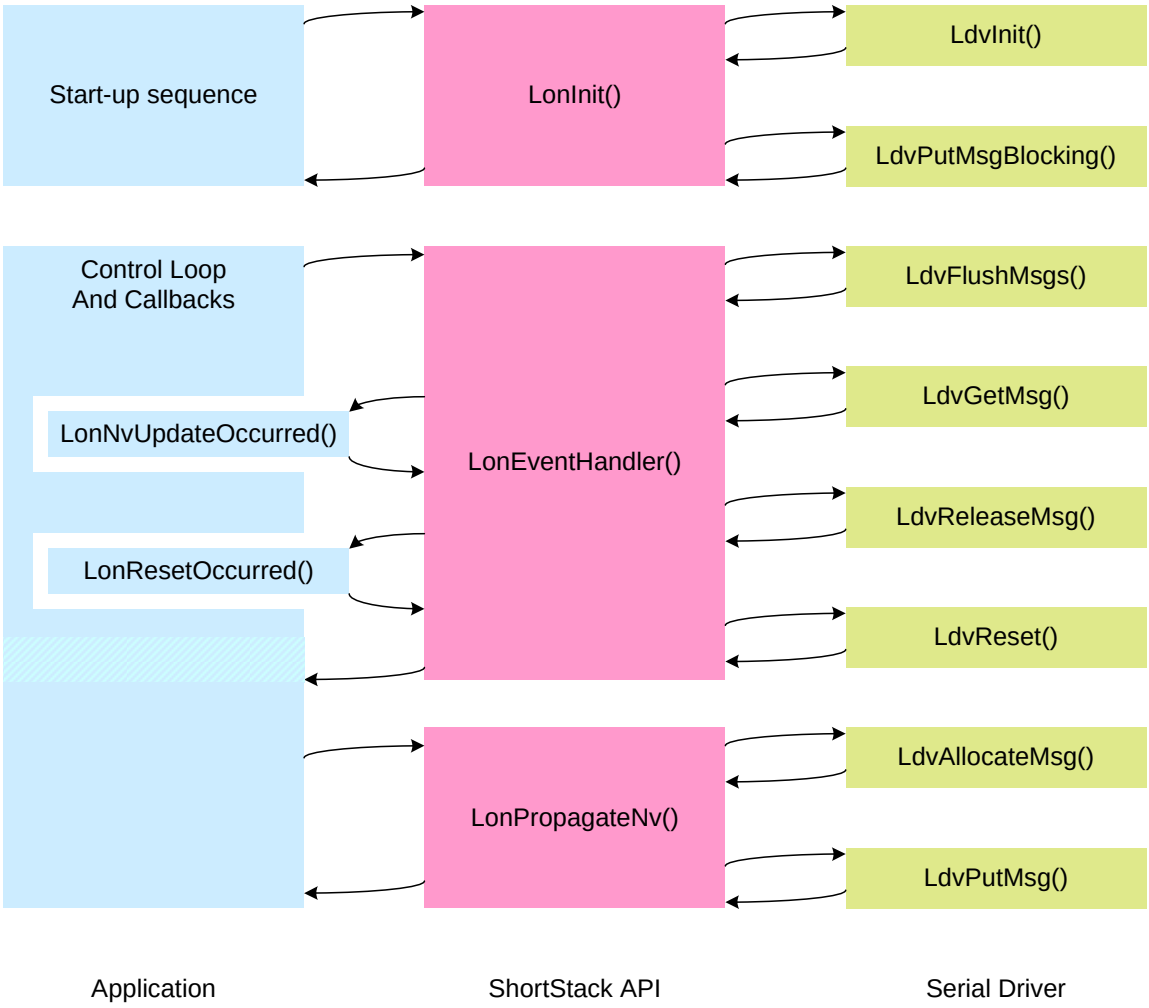

Figure 58. A ShortStack Application Communicates with the API and the Serial Driver

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

Initializing the ShortStack Device

Your application must call the **LonInit()** function once during device startup. This function initializes the ShortStack LonTalk Compact API, driver, and Micro Server.

The **LonInit()** function copies the ShortStack device interface data to the ShortStack Micro Server. This data defines the network parameters and device interface for the ShortStack Micro Server. Your application can call this function after device startup to reinitialize and restart the ShortStack Micro Server, to change the network parameters, or to change the device interface.

Recommendation: Add a call the **LonInit()** function in the **main()** function of your application (or to your host platform equivalent of that function).

During initialization, the Micro Server enters quiet mode until the initialization is complete. Quiet mode ensures that only a complete and fully functioning protocol stack attaches to the network. While the Micro Server is in quiet mode,

the host processor can use local commands to communicate with the Micro Server, such as query status or ping, but the Micro Server cannot communicate with other devices on the network.

Example:

```
void main(void) {
   // Initialize host-side hardware
   ...
   // Initialize host software
   ...
   LonInit();

   // Enter the main loop:
   while (TRUE) {
      LonEventHandler();
      // Process your application
      ...
   }
}
```

Periodically Calling the Event Handler

Your ShortStack application must periodically call the **LonEventHandler()** function to check if there are any LONWORKS events to process. You can call this function from your application's control (or idle) loop, or from any point in your application that is processed periodically (if your application meets the execution context requirements described in *Using the ShortStack LonTalk Compact API* on page 164).

The host application should be prepared to process the maximum rate of LONWORKS traffic delivered to the device. To prevent any possible backlog of incoming messages, use the following formula to determine the minimum call rate for the **LonEventHandler()** 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.

Recommendation: In the absence of measured data for the network, assume 90 packets per second arriving for a TP/FT-10 ShortStack device, or 9 packets arriving per second for a PL-20 ShortStack device. These packet rates meet the channels' throughput figures, assuming that most traffic uses the 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 the ShortStack device.

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

Again using the formula, a typical PL-20 power-line device would call the **LonEventHandler()** function once every 100 ms. However, to ensure low network latency, all ShortStack devices should call the **LonEventHandler()** function at least once every 10 ms.

When an event occurs during a call to the **LonEventHandler()** function, the function calls the appropriate callback function for your host application to handle the event. Your callback handler functions must be designed for this minimum call rate, and should defer time-consuming operations (such as lengthy flash writes) whenever possible.

See Appendix C, *ShortStack LonTalk Compact API*, on page 287, for a list of the available callback functions.

Sending a Network Variable Update

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

LonNvIndex*Name*

Example: A network variable named **nvoValue** has the **LonNvIndexNvoValue** index name.

The **LonPropagateNv()** function forwards the update to the ShortStack Micro Server, which in turn transmits the update to the network. This function returns an error flag that indicates whether the update was delivered to the Micro Server, but does not indicate successful completion of the update itself. For example:

```
LonApiError error = LonPropagateNv(LonNvIndexNvoValue);
```

After the update is complete, the ShortStack Micro Server informs the **LonEventHandler()** function in the ShortStack LonTalk Compact API, which in turn calls your **LonNvUpdateCompleted()** callback handler function, which notifies your application of the success or failure of the update. You can use this function for any application-specific processing of update completion. **Figure 59** on page 172 shows the control flow for processing a network variable update.

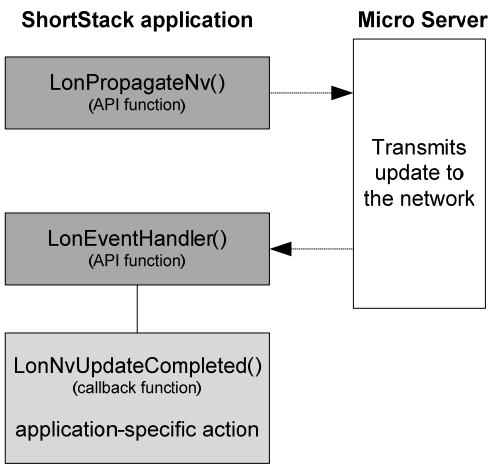

Figure 59. Control Flow for Sending a Network Variable Update to the Network

Perhaps the most frequent cause of propagation failure for a device that frequently sends network variable updates or application messages is the **LonApiTxBufIsFull** error (defined in the **LonApiError** enumeration) from the **LonPropagateNv()** function.

If all output buffers are in use at the time of the API call, the application must wait until at least one of the outstanding transactions completes, and frees an output buffer. Because this wait can take a significant amount of time, subject to the device's network configuration, networking environment, and the nature of the outstanding transactions, your application should return to its main processing control algorithm to process other work before it retries propagation.

Some applications require that the propagation be initiated before processing can continue. Such an application could support a wrapper around the **LonPropagateNv()** function that tests for this particular failure reason, and calls the API's periodic service entry point until propagation succeeds. For example:

```
LonBool lonPreemptionMode = FALSE;

LonApiError myPropagateNv(const unsigned index) {
  LonApiError error = LonApiNoError;

  while((error=LonPropagateNv(index))==LonApiTxBufIsFull) {
    lonPreemptionMode = TRUE;
    LonEventHandler();
  }
  lonPreemptionMode = FALSE;
  return error;
}
```

This example wrapper supports a global variable, **lonPreemptionMode**, which is true while the function waits for an output buffer to become availabile, in order to satisfy the original request. Until the buffer becomes availabile, the routine makes frequent calls to the API's periodic service entry point, **LonEventHandler()**. Because the **LonEventHandler()** function calls callback handler functions, which in turn could trigger network events, signal the buffer-unavailable state to the application so that it can avoid further propagation of network variables or application messages while in this state.

In the case of an unacknowledged or repeated service type, the Micro Server considers the update complete when it has finished sending the update to the network. In the case of an acknowledged service type, the Micro Server considers the update complete when it receives acknowledgements from all receiving devices, or when the retry timer expires.

In case of an unbound network variable (an output network variable that is not currently connected to any input network variables), propagating a network variable update always succeeds. This behavior is consistent with that of other LONWORKS devices, and allows you to create applications without having to track the device's network configuration.

Depending on the device's current network configuration and its networking environment, completion of any locally initiated transaction, such as the propagation of an updated output network variable, can take a significant amount of time until the success or failure of the transaction can be determined.

To process an update failure, edit the **LonNvUpdateCompleted()** callback handler function in the **ShortStackHandlers.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; use the retry count to do that automatically. 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 be 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 "ShortStackDev.h"
```

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

Receiving a Network Variable Update from the Network

When the ShortStack Micro Server receives a network variable update from the network, it forwards the update to the ShortStack LonTalk Compact API, which writes the update to your network variable, and then calls the **LonNvUpdateOccurred()** callback 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 corresponding variable declared in the **ShortStackDev.c** file.

To receive notification of a network variable update, modify the **LonNvUpdateOccurred()** callback handler function (in the **ShortStackHandlers.c** file) to call the appropriate functions in your application. The API calls this function with the index of the updated network variable. **Figure 60** shows the control flow for receiving a network variable update.

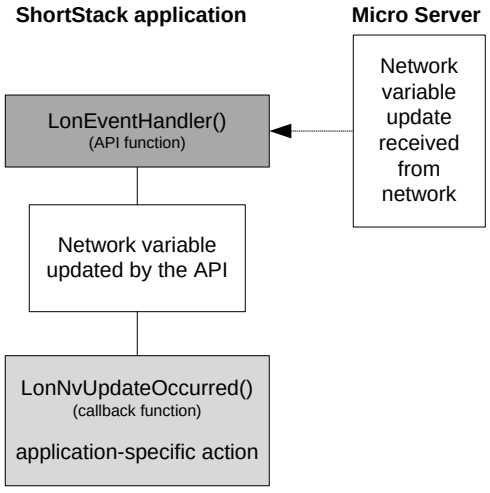

Figure 60. Control Flow for Receiving a Network Variable Update

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 171. That example demonstrated how to read the network variable inputs asynchronously

by reading the latest values from the network variables declared in the **ShortStackDev.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 **ShortStackHandlers.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 "ShortStackDev.h"

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

This modification calls the **myController()** function defined in the example in the previous section, *Sending a Network Variable Update*, on page 171.

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;
   nvoWatt /= nciGain.divider;
   if (LonPropagateNv(LonNvIndexNvoWatt) != LonApiNoError)
   {
      // handle error here
      ...
   }
}
```

Configuration network variables must be persistent, that is, their values must withstand a power outage. You must implement suitable hardware or software to achieve non-volatile data storage for this data. See *Providing Persistent Storage for Non-Volatile Data* on page 192 for more information.

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 ShortStack Micro Server responds to poll or fetch requests by sending the current value of the requested network variable. The **LonEventHandler()** function processes the request and sends the network variable value to the network. **Figure 61** shows the control flow for handling a network variable poll or fetch request.

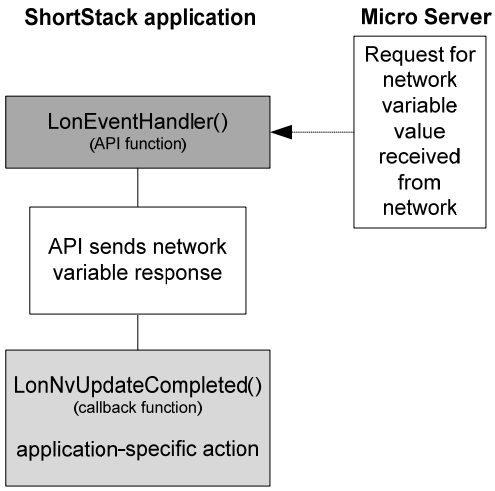

Figure 61. Control Flow for Handling a Network Variable Request

Handling Changes to Changeable-Type Network Variables

When a network management tool plug-in, such as 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 ShortStack FX ARM7 Example Port 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. This file must be explicitly referenced (**#include**) in your host application. You can use the NodeBuilder Resource Editor to determine the file that you need, which is generally in the [*LonWorks*]**\NeuronC\Include** directory. The enumeration is defined 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 any of the 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, a network management tool must ensure that all related network variables change to the new type. You cannot share a type-inheriting configuration property among both changeable and non-changeable network variables.

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 a network management tool 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 ShortStack Micro Server about the current length of the changeable-type network variable. The current length information must be kept in non-volatile memory.

Because the application must also ensure that the **SCPTnvType** configuration property reports the current and correct type, you can use the configuration property's **type_size** field to store that information.

The ShortStack LonTalk Compact 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 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.

Communicating with Other Devices Using Application Messages

You can use application messages to create a proprietary (that is, non-interoperable) interface for a device. For example, you can use application messages 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), network management messages, network diagnostic messages, and explicitly addressed network variable messages.

There are two interoperable uses for application messages: the Interoperable Self-Installation (ISI) protocol and the LONWORKS file transfer protocol (LW-FTP). The ISI protocol is used in self-installed networks; see Chapter 11, *Developing a ShortStack Application with ISI*, on page 197, for more information about ISI. LONWORKS FTP 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. Message code values are listed in **Table 22**. For application messages, you typically 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).

Table 22. Message Code Values

Message Type	Message Code	Description
User Application Messages	0 to 47 (0x0 to 0x2F)	Generic application messages. The interpretation of the message code is left to the application.
Standard Application Messages	48 to 62 (0x30 to 0x3E)	Standard application messages defined by LONMARK International.
Responder Offline	63 (0x3F)	Used by application message responses. Indicates that the sender of the response was in an offline state and could not process the request.
Foreign Frames	64 to 78 (0x40 to 0x4E)	Used by application-level gateways to other networks. The interpretation of the message code is left to the application.
Foreign Responder Offline	79 (0x4F)	Used by foreign frame responses. Indicates that the sender of the response was in an offline state and could not process the request.

Message Type	Message Code	Description
Network Diagnostic Messages	80 to 95 (0x50 to 0x5F)	Used by network tools for network diagnostics.
Network Management Messages	96 to 127 (0x60 to 0x7F)	Used by network tools for network installation and maintenance.
Network Variables	128 to 255 (0x80 to 0xFF)	The lower six bits of the message code contain the upper six bits of the network variable selector. The first data byte contains the lower eight bits of the selector.

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.

Each message tag is represented by a global variable declared by the LonTalk Interface Developer utility in the **ShortStackDev.c** file, with **extern** declarations provided in the **ShortStackDev.h** file. This file defines an index value for each message tag in the **LonMtIndex** enumeration. The index names use the following format:

LonMtIndex*Name*

Example: A message tag named **cpFilePtr** has the **LonMtIndexCpFilePtr** index name.

Sending an Application Message to the Network

Call the **LonSendMsg()** function to send an application message. This function forwards the message to the ShortStack Micro Server, which in turn transmits the message on the network. After the message is sent, the ShortStack Micro Server informs the **LonEventHandler()** function in the ShortStack LonTalk Compact API, which in turn calls your **LonMsgCompleted()** callback 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 ShortStack 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 ShortStack Micro Server receives a response, it forwards the response to the **LonEventHandler()** function in the ShortStack LonTalk Compact

API, which in turn calls your **LonResponseArrived()** callback handler function for each response it receives.

Figure 62 shows the control flow for sending an application message.

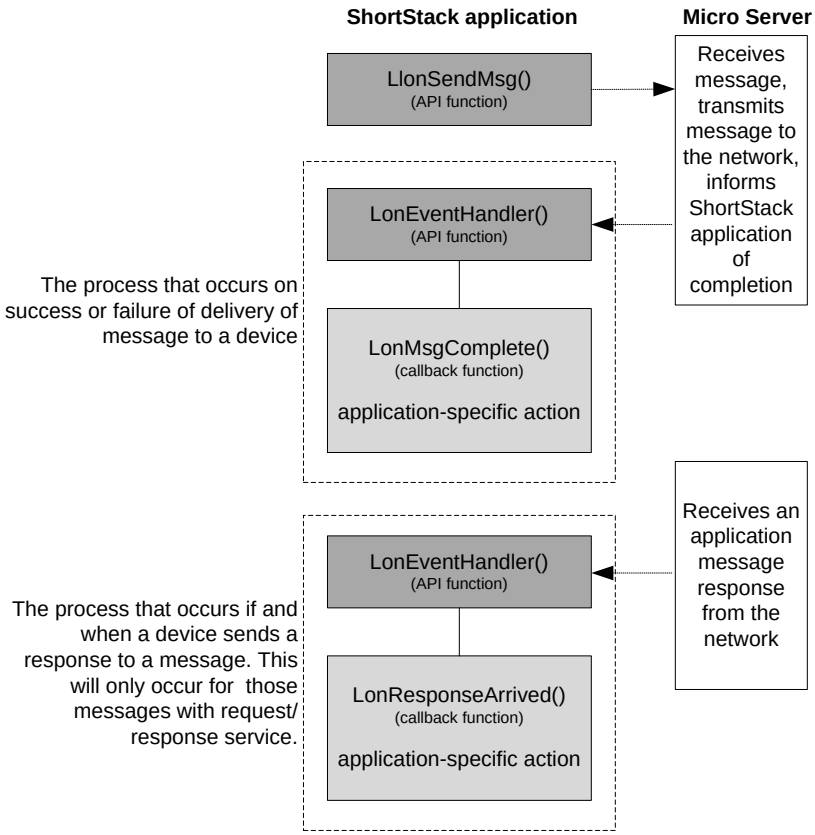

Figure 62. Control Flow for Sending an Application Message

Receiving an Application Message from the Network

When the ShortStack Micro Server receives an application message from the network, it forwards the message to the **LonEventHandler()** function in the ShortStack LonTalk Compact API, which in turn calls your **LonMsgArrived()** callback handler function. Your implementation of this function must process the application message, and can optionally notify your ShortStack application about the message.

The ShortStack Micro Server does not call the **LonMsgArrived()** callback handler function if an application message is received while the ShortStack 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.

Figure 63 on page 184 shows the control flow for receiving an application message.

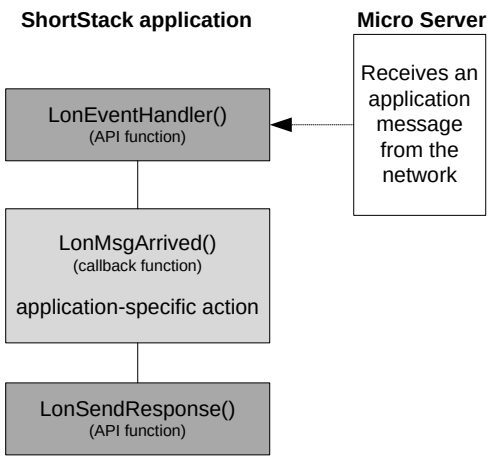

Figure 63. Control Flow for Receiving an Application Message

Handling Management Tasks and Events

LONWORKS installation and maintenance tools use network management commands to set and maintain the network configuration for a device. The ShortStack Micro Server automatically handles most network management commands that are received from these tools. A few network management commands are application-specific, and are forwarded by the Micro Server to the **LonEventHandler()** function in the ShortStack LonTalk Compact API, which in turn forwards the request to your application through the network management callback handler functions. These commands are requests for your application to wink, go offline, go online, handle pressed or held service pin events, or reset, and must be handled by your **LonWink()**, **LonOffline()**, **LonOnline()**, **LonServicePinPressed()**, **LonServicePinHeld()**, 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 ShortStack application and implemented by the ShortStack Micro Server. Local network management tasks are never propagated to the network. The optional Network Management Query and Update ShortStack APIs allow you to include handling of these local network management commands if your ShortStack application requires it.

Many of these commands are called by your ShortStack application and then handled by the ShortStack Micro Server with no additional notification through callback handler functions. These functions include: **LonClearStatus()**, **LonSetNodeMode()**, **LonUpdateAddressConfig()**, **LonUpdateAliasConfig()**, **LonUpdateConfigData()**, **LonUpdateNvConfig()**, and **LonUpdateDomainConfig()**.

A few of the extended local network management commands are requests for information. After the ShortStack Micro Server receives these requests, it makes the response information available to the ShortStack LonTalk Compact API. When the Micro Server makes this information available, the **LonEventHandler()** function calls the appropriate callback handler function, which you can customize to handle the information in an application-specific way. **Figure 64** on page 185

through **Figure 67** on page 186 show the control flow for handling these kinds of network management commands.

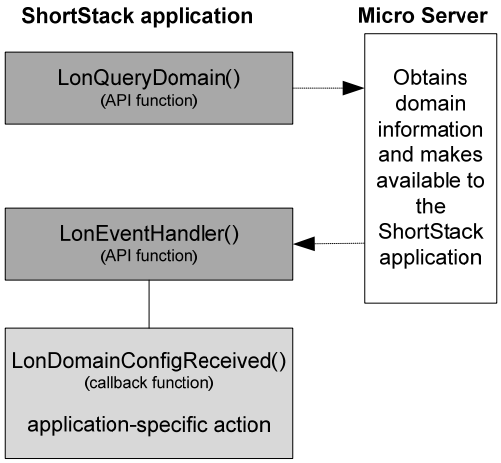

Figure 64. Control Flow for Query Domain Network Management Command

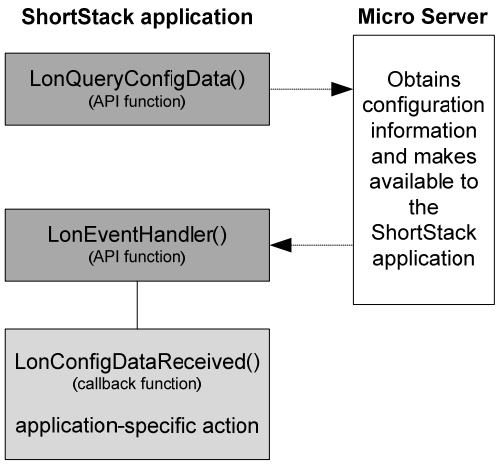

Figure 65. Control Flow for Query Configuration Data Local Network Management Command

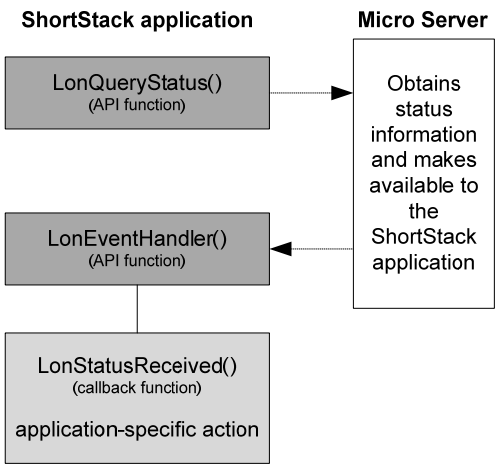

Figure 66. Control Flow for Query Status Local Network Management Command

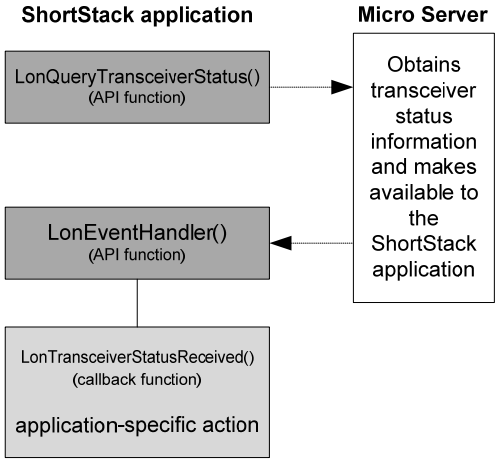

Figure 67. Control Flow for Query Transceiver Status Local Network Management Command

Handling Reset Events

A ShortStack Micro Server can reset for a variety of reasons. To determine the cause of a Micro Server reset, you can use the **LonGetLastResetNotification()** function of the ShortStack Network Management Query API. This function returns a pointer to the **LonResetNotification** structure, which is defined in the **ShortStackTypes.h** file. The **LonResetNotification** structure is also provided with the **LonReset()** callback handler function.

The **LonResetNotification** structure contains the following information:

- The State of the Micro Server
- The Version of the link layer protocol (3 for ShortStack 2.1; 4 for ShortStack FX)
- Information about availability and state of the static IO9 input signal on the Micro Server (see *Using the IO9 Pin* on page 68)

- Information about whether the Micro Server is initialized
- The Micro Server Key (see *Using the ShortStack Micro Server Key* on page 58)
- The cause for the most recent reset, encoded in a value from the **LonResetCause** enumeration
- The most recent system error, encoded in a value from the **LonSystemError** enumeration
- The Micro Server's 48-bit unique ID (also known as its Neuron ID)
- The current number of address table records, domains, and aliases supported by the Micro Server

Querying the Error Log

The ShortStack Micro Server writes application errors to the system error log. The reset notification contains the most recent system error code, but you can use the **LonQueryStatus()** function to query the complete error and statistics log.

The **LonStatus** structure, which is provided in response to the **LonQueryStatus()** call through the **LonStatusReceived()** callback handler function, contains complete statistics information, such as the number of transmit errors, transaction timeouts, missed and lost messages, and so on.

In addition to the standard system error codes (129 and above), a ShortStack FX Micro Server can log ShortStack-specific system error codes that help you diagnose problems.

Table 23 lists the ShortStack-specific system error codes. All system error codes are provided by the **LonSystemError** enumeration in **ShortStackTypes.h**.

Table 23. LonSystemError Enumeration Values for ShortStack

Value	Condition	Description
1	Smart Transceiver lock	Unsupported Micro Server hardware. Use an Echelon Smart Transceiver for the Micro Server. This error condition also changes the Micro Server's state to applicationless.
2	niSiData message received	This message is unsupported for ShortStack FX. See *Converting a ShortStack 2 Application to a ShortStack FX Application* on page 257 for information about migrating to ShortStack FX.
3	Network variable processing with host selection is not supported	The Micro Server was created with the **#pragma netvar_processing_off** directive, which is not supported. This error condition also changes the Micro Server's state to applicationless.

Value	Condition	Description
4	Transceiver not supported	This error occurs when the host tries to configure the Micro Server for a transceiver that is neither special-purpose mode, nor single-ended at 78 kbps. Unlike the Smart Transceiver lock, the Micro Server is not changed to the applicationless state. This error is logged and the node enters quiet mode.
5	Message too big	An outgoing message cannot be sent because it exceeds the available buffer size.
6	Unknown link-layer command	The Micro Server received an unknown link-layer command from the host.
7	Malformed NVINIT message	The NVINIT message specified a number of network variables, but provided data for fewer network variables.
64	RPC callback timeout	The Micro Server attempted a remote procedure call to call an ISI callback on the host, but the host failed to acknowledge the uplink message for 15.5 seconds (31*500 ms).
65	RPC callback NACK	The Micro Server attempted a remote procedure call to call an ISI callback on the host, but the host replied with an unexpected negative response.
66	RPC out of sequence	An out-of-sequence reply from the host has been received. The out-of-sync reply is ignored.
67	RPC nothing to acknowledge	A positive or negative RPC acknowledgement has been received, but was unexpected. The acknowledgement is ignored.
68	Interleaving RPC call attempted	An RPC call to the host was attempted while a previous call was still outstanding. The Micro Server resets.

Error conditions that change the state to applicationless also invalidate the cached signature, thus enforcing a complete re-initialization after Micro Server reload.

Reinitializing the ShortStack Micro Server

For ShortStack devices that sense their configuration and alter their device interface at runtime (for example, a hot-pluggable modular I/O system), the host must re-initialize the Micro Server with a new interface. Such changes can alter a device's interoperability, and thus should be done carefully.

To re-initialize the Micro Server at run-time:

- Each interface must have its own, unique, program ID.
- A unique XIF file must be provided for each supported program ID.

- Interface changes must be initiated only when the device is in the unconfigured state (use the **LonQueryStatus()** function to determine the current state).

When the host application reinitializes the Micro Server with a new application, the Micro Server automatically enters quiet mode until the initialization is successfully completed.

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 (LW-FTP) protocol
- The host direct memory file (DMF) access method

The LW-FTP protocol is appropriate when large amounts of data need to be transferred between the host processor and Smart Transceiver. The host DMF access method is appropriate for most other cases. The LW-FTP protocol supports configuration files and configuration network variables (CPNVs). The host DMF access method supports only configuration files. You can use both the LW-FTP protocol and the DMF access method within a single application.

By supporting direct memory files, your application allows the network management tool to use standard memory read and write network messages to access configuration files located on the host. Direct memory files appear to the network management tool as if they were located within the Micro Server's native address space, but the Micro Server routes memory read and write requests within the DMF memory window to the host processor. The ShortStack LonTalk Compact API in turn forwards these requests to code that handles the request. This code is generated by the LonTalk Interface Developer utility.

You do not generally need to modify the code that the LonTalk Interface Developer utility generates, unless your application requires support for non-volatile storage for writeable configuration value files. See *Providing Persistent Storage for Non-Volatile Data* on page 192 for more information about managing non-volatile data storage.

Important: The host DMF access method requires Version 16 system firmware, or later, and thus is not available for current PL 3120 Smart Transceivers, which are based on Version 14 system firmware. All other standard Micro Server images have this feature enabled. See *Custom Micro Servers* on page 241 for information about how to create custom Micro Servers that can support the host DMF access method.

When the host DMF access method is enabled, the Micro Server relays to the host all memory read or write requests for configuration files that cannot be locally satisfied. These requests are those that relate to memory that is not declared in the Micro Server's memory map, including areas that are declared as *memory-mapped I/O*.

Example: An FT 3120 Smart Transceiver has no memory in the 0xA100..0xCEFF address range, and relays all memory read or write requests concerning this area to the host processor, if the DMF feature is enabled. Without the DMF access method, the same memory read or write request would receive a failure code.

The standard Micro Servers for 3150 Smart Transceivers use 64 KB (or larger) flash memory. The memory maps of this memory are declared such that the same 0xA100..0xCEFF area is available for the DMF access method.

You can create a custom Micro Server with a larger DMF window, and you can use the LonTalk Interface Developer utility to override the default start address and size of the DMF memory window.

If the model file contains a **SNVT_address** typed network variable and at least one configuration property defined in a configuration file, and the selected Micro Server supports the DMF access method, the LonTalk Interface Developer utility automatically generates all code and data that is necessary to satisfy the memory read and write requests; however, the application must still provide code for non-volatile, persistent, data storage.

The DMF Memory Window

To the network management tool, all content of the DMF memory window is presented as a continuous area of 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 Micro Server's native address space, even though it usually 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 configuration value files (if any)
- The read-only configuration value files (if any)

Figure 68 on page 191 shows how the different memory address spaces relate to each other.

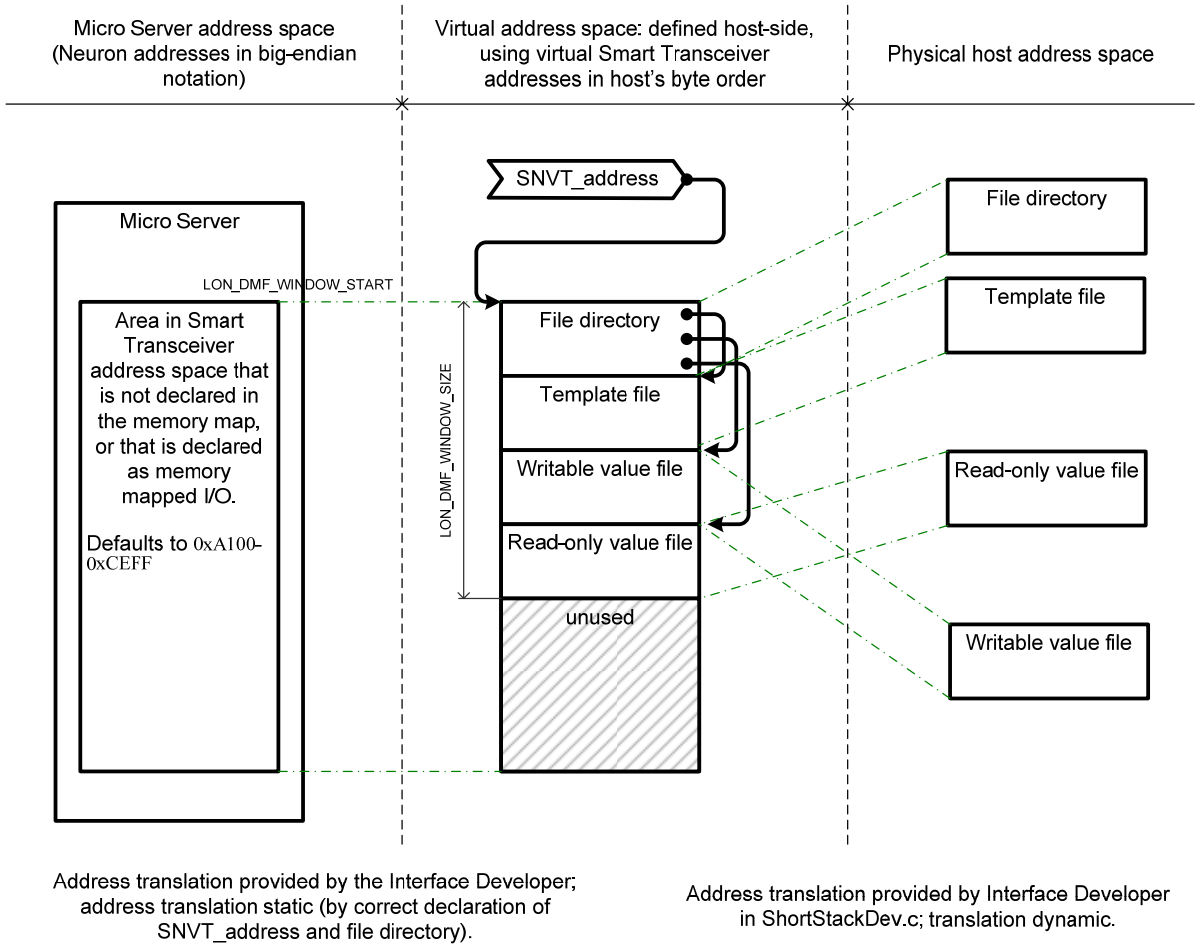

Figure 68. Relationship between Different Memory Spaces

The LonTalk Interface Developer utility defines three macros in the generated **ShortStackDev.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.

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
- Provide a larger DMF memory window by creating a custom Micro Server
- Implement the LONWORKS File Transfer Protocol (LW-FTP)

File Directory

The LonTalk Interface Developer utility produces a configurable file directory structure, which supports:

- Using named or unnamed files (it uses unnamed by default)
- Up to 64 KB of data for each file
- Up to a total of 64 KB for all files plus the file directory itself

The utility initializes the file directory depending on the chosen access method. The directory can be used with an LW-FTP server implementation or the direct memory file access method implementation. The initialization that the utility provides works for both little-endian and big-endian host processors.

The **ShortStackDev.h** header file allows you to customize the file directory structure, if needed.

Providing Persistent Storage for Non-Volatile Data

If you use configuration files, configuration network variables, network variables declared with the **eeprom** modifier, or use other, application-specific, persistent data, you must supply a mechanism to read that data into RAM during startup, preserve modifications to that data, and track any read or write errors.

The details for handling persistent storage are dependent on your 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.

DMF Memory Drivers

The LonTalk Interface Developer utility generates all the code necessary for the basic host memory driver implementation within the generated **ShortStackDev.c** file. This code is used by two callback handler functions, **LonMemoryRead()** and **LonMemoryWrite()**, which are defined within the **ShortStackHandlers.c** file. This file is copied into the project folder by the LonTalk Interface Developer utility, but is not overwritten or updated when you re-run the utility. Thus, you can modify this file for your host memory driver.

The code to handle DMF-related memory read or write requests is based on the **LonMemoryRead()** and **LonMemoryWrite()** callback handler functions that the LonTalk Interface Developer utility generates. The API calls these callback handler functions whenever a related request is received.

Both callback handler functions use the **LonTranslateWindowArea()** function (defined in **ShortStackDev.c**) to provide the translation of virtual addresses into host addresses. This translation is based on the **windowLayoutTable** array, also defined in **ShortStackDev.c**.

When the translation succeeds, the **LonTranslateWindowArea()** function supplies a pointer to a record within the **windowLayoutTable**, which describes the segment in question. The segment might be the file directory, the template file, any of the value files, or a user-defined additional file.

A segment description is based on the **LonDmfWindowSegment** type, which is defined in **ShortStackDev.h** as:

```
typedef struct
{
    LonBool  Writeable;
    LonMemoryDriver Driver;
    void*    Start;
    size_t   Size;
} LonDmfWindowSegment;
```

The **Driver** member, based on the **LonMemoryDriver** enumeration, allows the **LonMemoryRead()** or **LonMemoryWrite()** callback handler function to determine how to access the data. The default memory driver uses a simple **memcpy()** approach. This approach might be sufficient for battery-backed RAM, but most applications need to add an application-specific driver.

To add an application-specific DMF memory driver, add a new member to the **LonMemoryDriver** enumeration, or use an identifier for your memory driver that you derive from the pre-defined **LonMemoryDriverUser** enumeration member, and add code to the **LonMemoryRead()** or **LonMemoryWrite()** callback handler functions to dispatch the action to the appropriate memory driver.

Recommendation: Derive your driver identifier from **LonMemoryDriverUser**, because this approach avoids editing the **ShortStackDev.h** file, which will be overwritten when you re-run the LonTalk Interface Developer utility.

See *Application Start-Up and Failure Recovery* on page 194 for other considerations for the memory driver.

CPNV and EEPROM NV

For configuration network variables (CPNVs) and non-volatile network variables (those declared with the **eeprom** modifier), your application must provide functions for reading and writing the non-volatile data.

During processing for the **LonInit()** function, the ShortStack LonTalk Compact API calls the **LonNvdDeserializeNvs()** callback handler function for every CPNV and non-volatile network variable to read their values (if any) stored in persistent storage. This function has the following signature:

```
const LonApiError LonNvdDeserializeNvs(void);
```

Your application must obtain the most recent value for the network variable with the given index from non-volatile memory, and store it in the location provided by the **LonGetNvValue()** function. For changeable-type network variables, the application should always retrieve network-variable data that equals the initial network variable type in size. If the current size of a changeable-type network variable is less than its maximum (and initial) size, supply zeroes to fill the remaining, currently unused, memory. You can obtain the size of the initial network variable from the network variable table or by using the **sizeof()** operator with the initial (declared) network variable type, (rather than using the **LonGetNvSize()** callback handler function, which returns the current size of the network variable).

Whenever a CPNV or non-volatile network variable is updated over the network, your **LonNvUpdateOccurred()** (or **LonNvUpdateCompleted()**) callback handler function should evaluate whether to write the CPNV or network variable data to

non-volatile memory, and then call your non-volatile-memory-write function as needed.

To determine the offset of a particular non-volatile network variable value within the non-volatile storage, the application can read the network variable table (the **nvTable** array). For example, the application could add the sizes of all non-volatile network variables with index value less than the current network variable, and use that size as a pointer offset into the non-volatile storage. Different host platforms and compilers offer other ways to write and read data from non-volatile memory. For example, if your host processor supports flash memory, EEPROM, or NVRAM, you might be able to declare your non-volatile network variables directly in this memory.

Application Start-Up and Failure Recovery

Typical applications load all persistent data into RAM during startup. The ShortStack LonTalk Compact API handles that process for persistent network variables by calling the **LonNvdDeserializeNvs()** function from the **LonInit()** function, but your application must take appropriate steps to ensure correct data for all DMF window segments.

Recommendation: Your application should read data from DMF window segments prior to calling the **LonInit()** function, because the device is already attached to the network when the **LonInit()** function returns.

Because your application is responsible for loading and modifying applicable data in non-volatile memory, you should use the application signature generated by the LonTalk Interface Developer utility to ensure that the application manages its own data, rather than another application's data. Use the **LON_APP_SIGNATURE** macro defined in the **ShortStackDev.h** file to retrieve the current application's signature.

Writing non-volatile data can be error-prone and slow, depending on the type and organization of the memory. Your application must detect any failures during the write process, and to ensure that the write process completes in a timely a fashion.

Recommendation: If the write process takes too long to complete within the API's timing requirements (see *Periodically Calling the Event Handler* on page 170), your application should use queues or caches to minimize both latencies and the number of modifications.

The application should also be able to detect data corruption. If, for example, the device incurs a power loss during a write operation to non-volatile data, that data can be invalid. When the application starts up after the failure, and attempts to re-load that data, it should detect that the data is not valid. If invalid data is found, the application should cease operation and put the Micro Server into the unconfigured state.

Applications can implement any method to ensure reliable persistence of data, or to ensure detection of failure, such as hardware support (for example, battery backup, or early power-out interrupts to flush any pending write requests). Typical software support includes management of "dirty" flags and checksum protection for persistent data.

Application Migration: Series 3100 to Series 5000

A ShortStack FX application that is designed to work with a Micro Server on a Series 3100 chip (such as an FT 3150 Smart Transceiver) can work with a Micro Server on a Series 5000 chip (such as an FT 5000 Smart Transceiver), but you must re-run the LonTalk Interface Developer utility and recompile the application. If the host processor type does not change, you do not need to modify your link-layer serial driver.

Perform the following general tasks to migrate a ShortStack FX application from a Series 3100 device to a Series 5000 device:

1. Create a backup copy of your application's existing project. You will need the original files from the project for step 3.
2. Re-run the LonTalk Interface Developer utility. Select the desired standard or custom Micro Server for the Series 5000 device. The utility will overwrite the **LonDev.c** and **LonDev.h** files, among others.
3. Use a DIFF or MERGE tool to compare the backed-up versions of the **LonDev.c** and **LonDev.h** files with the newly created ones. Carefully merge your application's code (from the backed-up version) into the new **LonDev.c** and **LonDev.h** files.

   Important: Do not to replace the application initialization data blocks generated by the utility for the Series 5000 hardware.
4. Rebuild your application.
5. Load the appropriate Micro Server image into the FT 5000 Smart Transceiver or Neuron 5000 Processor.
6. Load the application into the host processor and test the device.

Because this change involves a hardware change and updated software for both the Micro Server and the host, upgrading a device to use new hardware in the field is not recommended. In addition, such a hardware change is likely to invalidate any certificates or declarations of conformity obtained for the device, given that they were obtained for the previous hardware.

11

Developing a ShortStack Application with ISI

This chapter describes how to develop a ShortStack application with Interoperable Self-Installation (ISI) support. It also describes the various tasks performed by the application.

Overview of ISI

A control network could be a small, simple network in a home or in a machine consisting of a few devices, or it could be a large network in a building, factory, or ship consisting of tens of thousands of devices. The devices in the network must be configured to become part of the common network and to exchange data. The process of configuring devices in a control network is called *network installation*.

There are two main categories of networks:

- *Managed* networks
- *Self-installed* networks

A managed network is a network where a shared *network management server* performs network installation. A user typically uses a tool to interact with the server and to define how the devices are configured and how they communicate. Such a tool is called a *network management tool*. For example, Echelon's LonMaker Integration Tool is a network management tool that uses the LNS Server network management server to install devices in a network. Although a network management tool and a server are used to establish initial network communication, they need not be present for the network to function. The network management tool and server are required only to make changes to the network's configuration.

In a managed network, the network management tool and server together allocate various network resources, such as device and data point addresses. The network management server is also aware of the network topology, and can configure devices for optimum performance within the constraints of that topology.

The alternative to a managed network is a self-installed network. There is no central tool or server that manages the network configuration in a self-installed network. Instead, each device contains code that replaces parts of the network management server's functionality, which results in a network that does not require a special tool or server to establish network communication or to change the configuration of the network.

Because each device is responsible for its own configuration, a common standard is required to ensure that devices configure themselves in a compatible way. The standard protocol for performing self-installation in LONWORKS networks is called the LONWORKS *Interoperable Self-Installation (ISI) Protocol*. The ISI protocol can be used for networks of up to 200 devices.

Larger or more complex networks must either be installed as managed networks, or must be partitioned into multiple smaller subnetworks, where each subnetwork has no more than 200 devices and meets the ISI topology and connection constraints. Devices that conform to the LONWORKS ISI protocol are called *ISI devices*.

An ISI device manages its network identity (its address) and its network variable connections with minimum impact on the network performance. These two groups of services are supported through a set of API calls, callback handlers, and notification events. See *Managing the Network Address* on page 202 and *Managing Network Variable Connections* on page 206 for more information about these services.

The ShortStack Developer's Kit includes standard Micro Servers that can be used to create ISI devices, and allows the creation of custom Micro Servers that support the ISI protocol. Such an ISI-enabled Micro Server can be used in self-installed or managed networks, but a Micro Server without built-in support for the ISI protocol cannot be used in an ISI network (unless you implement the required portions of the ISI protocol as part of your host application using the standard ShortStack messaging and self-installation APIs provided). For a detailed description of the ISI protocol, see the *LONWORKS ISI Protocol Specification*.

The ISI protocol is a licensed protocol. In addition to the ShortStack FX Developer's Kit, the ISI Developer's Kit and Mini FX Evaluation Kit each include a license for development use of the ISI library.

Using ISI in a ShortStack Application

Using the ISI protocol in a ShortStack application is similar to using the ISI protocol in a Neuron C-based application (such as ones developed with the ISI Developer's Kit or the Mini FX Evaluation Kit). The application calls ISI functions and implements some or all of the ISI callback handler functions to produce the desired ISI behavior.

There are two ways to modify the ISI behavior of a Micro Server:

- If your ShortStack device uses a Micro Server that supports the ISI protocol, you can implement most of the ISI callback handler functions within your host application. Overriding ISI callback handler functions is an important part of creating an ISI application, because these callback handlers provide essential, and typically application-specific, details to the ISI engine.
- If you create an ISI-enabled custom Micro Server, you can determine the location of most of the ISI callback handler functions. If there is sufficient space in the Smart Transceiver, you can put enough intelligence into the Micro Server Neuron C application to have a large percentage of the ISI logic in the Smart Transceiver. Alternatively, you can let the Micro Server use the ShortStack ISI RPC protocol to call callback handler functions located on the host processor.

See *Comparing ISI for ShortStack and Neuron C* on page 238 for information about the similarities and differences between ShortStack ISI applications and Neuron C ISI applications. See *Creating a Custom Micro Server with ISI Support* on page 248 for information about customizing an ISI-enabled Micro Server.

Running ISI on a 3120 Device

A standard ShortStack Micro Server on a 3120 Smart Transceiver does not include support for ISI because of resource limitations. For 3120 devices, the ShortStack LonTalk Compact API allows you to implement ISI support on the host processor.

Running ISI on a 3150 Device

A standard ShortStack Micro Server on a 3150 Smart Transceiver can be installed in an ISI-S or ISI-DA network. Support for ISI is largely handled by the Micro Server itself. However, you can also use the ShortStack LonTalk Compact API to implement ISI support on the host processor. In addition, you can create a custom Micro Server to provide custom ISI support, including support for ISI-DAS applications.

Running ISI on a PL 3170 Device

A standard ShortStack Micro Server on a PL 3170 Smart Transceiver can be installed in an ISI-S or ISI-DA network. Support for ISI is largely handled by the Micro Server itself. However, you can also use the ShortStack LonTalk Compact API to implement ISI support on the host processor. In addition, you can create a custom Micro Server to provide custom ISI support. However, a Micro Server on a 3170 Smart Transceiver cannot support ISI-DAS applications.

An ISI-enabled Micro Server for the PL 3170 Smart Transceiver has several limitations, compared to other ISI-enabled standard Micro Servers. The following limitations are permanent and cannot be overcome by creating a custom, ISI-enabled, Micro Server:

- The link layer supports SCI at the fixed bit rate of 38400 bps. In addition, the SPI/SCI~, SBRB0, and SBRB1 signals are ignored.
- The utility functions, which include local operations such as the ping or echo command, are not supported by the Micro Server.
- ISI-S and ISI-DA modes are supported, but ISI-DAS mode is not.

The following limits can be changed by creating a custom, ISI-enabled, Micro Server, and adjusting the Micro Server's properties as needed:

- Capacity is limited to 120 network variables and 75 aliases.
- The ISI connection table is 24 records, local to the Micro Server.
- Controlled enrollment is supported.

Running ISI on an FT 5000 Device

A standard ShortStack Micro Server on an FT 5000 Smart Transceiver can be installed in an ISI-S or ISI-DA network. Support for ISI is largely handled by the Micro Server itself. However, you can also use the ShortStack LonTalk Compact API to implement ISI support on the host processor. In addition, you can create a custom Micro Server to provide custom ISI support, including support for ISI-DAS applications.

Tasks Performed by a ShortStack ISI Application

A ShortStack ISI application must decide whether to start the ISI engine (based on the **SCPTnwrkCnfg** configuration property), call ISI services as needed, handle ISI events, and recover from failures.

After the ISI engine starts, it manages various aspects of your device, and makes services available to you through the ISI API. The two major aspects managed include: managing the device's network address and managing its network variable connections.

Starting and Stopping ISI

Use the **IsiStart()** function to start the ISI engine for any supported ISI type. Typically, because the ISI engine is stopped after a Micro Server reset, you start the ISI engine in your **LonResetOccurred()** callback handler function when self-installation is enabled.

The **IsiStart()** function accepts two arguments: the ISI mode of operation (defined by the **IsiType** enumeration) and a bit vector with various flags (defined by the **IsiStartFlags** enumeration).

The ShortStack ISI API does not support, or require, the host application to call the **IsiPreStart()** function. Micro Servers that support hardware which requires the use of this function automatically call this API during power-up and reset.

Use the **IsiStop()** function to explicitly stop the ISI engine at any time. Typically, you stop the ISI engine when self-installation is disabled. Because the ISI engine is always off after a power-up or reset, and needs to be started explicitly with each reset, this function is not widely used.

When you stop the ISI engine, ISI callbacks into the application no longer occur. Because most ISI functions behave appropriately when the engine is stopped, the ShortStack application does not need to track the engine's state and can issue the same set of ISI API calls in any state.

Implementing a SCPTnwrkCnfg Configuration Property

ISI applications must implement a **SCPTnwrkCnfg** configuration property that is implemented as a configuration network variable. This configuration property must apply to your application's Node Object functional block, if available, or apply to the entire device if there is no Node Object.

This configuration property provides an interface for network management tools to disable self-installation on an ISI device. By using this configuration property, the same device can be used in both self-installed and managed networks.

Typically, the **cp_info(reset_required)** attribute is used with the declaration of the **SCPTnwrkCnfg** CP. This attribute allows you to check the current ISI state in the device's **LonResetOccurred()** callback handler function.

The configuration property has two values: **CFG_LOCAL** and **CFG_EXTERNAL**. When set to **CFG_LOCAL**, your application must enable self installation. When set to **CFG_EXTERNAL**, your application must disable self installation. Network management tools automatically set this value to **CFG_EXTERNAL** to prevent conflicts between self-installation functions and the network management tool.

For a device that will use self-installation, during the first start (only) with a new application image, set the value for the **SCPTnwrkCnfg** configuration property as

CFG_LOCAL so that the ISI engine can come up running with the first power-up. Subsequent starts use the default value of **CFG_EXTERNAL**.

Example:

```
SCPTnwrkCnfg nciNetConfigLocal;
ReadNonVolatileData();

nciNetConfigLocal = nciNetConfigLastKnownGoodValue;

if (nciNetConfigLocal == CFG_NUL) {
  /* For the first application start, set nciNetConfig to
   * CFG_LOCAL, thus allow the ISI engine to run by default
   */
  nciNetConfig = CFG_LOCAL;
  bWriteNonVolatileData = TRUE;
}

nciNetConfigLastKnownGoodValue = nciNetConfig;

if (nciNetConfig == CFG_LOCAL) {
  /* We are in self-installed mode */
  if (nciNetConfigLocal == CFG_EXTERNAL) {
    /* The application has just returned to the self-
     * installed mode. Make sure to re-initialize the
     * entire ISI engine.
     * Note that running this task on the Micro Server can
     * take a significant amount of time, after which, the
     * Micro Server resets. */
    IsiReturnToFactoryDefaults();
  }

  /* Start the ISI engine */
  IsiStart(IsiTypeS, IsiFlagExtended);
}
```

Managing the Network Address

After the ISI engine is started, it manages the device's network address. The network address consists of a subnet and node ID pair plus a domain identifer.

The subnet and node ID pair is managed automatically: ISI chooses a suitable value pair, and ensures the uniqueness of that value pair within the network, making changes to that value pair as needed while the device is running.

The domain identifier and its length (generally referred to collectively as "the domain") define the logical network to which the device belongs. Several devices can share the same physical network media, for example a power line communications channel, but can be logically isolated into distinct logical networks. Each logical network is known as a "domain."

ISI devices can be part of one primary domain. All ISI devices are also part of a secondary domain for administrative purposes, but all application-specific communication is limited to the primary domain.

There are four methods to assign a domain to an ISI device:

1. The domain can be pre-defined and assigned by the device application or by the ISI implementation. All ISI devices must initially support this method because an initial application domain is assigned prior to acquiring a domain using one of the other methods. This method enables all devices to be used in an ISI-S network, the smallest form of an ISI network, which uses this method by default. All ISI-enabled ShortStack Micro Servers support installation in an ISI-S network.
2. A device that supports domain acquisition can acquire a unique domain address from a domain address server. If a domain address server is not available, domain acquisition fails, and the ISI engine continues to use the most recently assigned domain (initially, the default domain). Devices that support domain acquisition also support multiple, redundant, domain address servers. Domain address acquisition is initiated by the user and controlled by the device acquiring the domain, not by the domain address server. This method allows the device to make intelligent decisions about retries, and prevents enrollment during domain acquisition. It also allows the device to increase automatic enrollment performance following the completion of domain acquisition. All standard ISI-enabled ShortStack Micro Servers support domain-acquisition services, but custom ISI-enabled Micro Servers can choose not to support them.
3. A domain address server can assign a domain to a device without a request from the device. This method minimizes the code required in the device, and can be used with all devices. This process is called *fetching a device*. All ISI-enabled devices and all ISI domain address servers support this method. This method simplifies the implementation of the ISI application, but control of the process is no longer within the ISI application.
4. A domain address server can fetch the domain from any of the devices in a network and assign it to itself. This method keeps multiple domain address servers in a network synchronized with each other, or allows a replacement domain address server to join an existing ISI network. This process is called *fetching a domain*. All ISI-enabled devices and all ISI domain address servers support this method.

A domain address server must support all four methods. That is, it can supply a pre-defined domain (which is typically used as the domain address server's default domain), it can support a device that requests a domain (domain acquisition), it can fetch any ISI device, and it can fetch a domain from another device.

Supporting a Pre-Defined Domain

While its ISI engine is running, any ISI device is always a member of two domains: the administrative secondary domain that uses a pre-defined and fixed domain, and the application-specific primary domain.

The primary domain uses a three-byte domain ID with value 0x49.53.49 (ASCII codes for "ISI") by default. An **IsiGetPrimaryDid()** callback function is supported, which allows applications to provide a different default for the primary domain. This alternate default can be used by some devices to start in a closed, non-interoperable, ISI network. The same method can also be used by domain

address servers to assign a unique domain identifier to the server's default primary domain (typically equal to the server's own unique ID).

Acquiring a Domain from a Domain Address Server

To acquire a domain from a domain address server using domain acquisition services, start the ISI engine using the **IsiStart()** function with the **isiTypeDa** type.

A domain address server must be in device acquisition mode to respond to domain ID requests. To start device acquisition mode on a domain address server, call the **IsiStartDeviceAcquisition()** function.

To start domain acquisition on a device that supports domain acquisition, call the **IsiAcquireDomain()** function.

A typical implementation starts the domain acquisition process when the Connect button is activated and a domain is not already assigned. If **SharedServicePin** is set to **FALSE**, the **IsiAcquireDomain()** function also issues a standard service pin message, thus allowing the same installation paradigm in both a managed and an unmanaged environment. If the application uses the physical service pin to trigger calls to the **IsiAcquireDomain()** function, the system image will have issued a service pin message automatically, and the **SharedServicePin** flag should be set to **TRUE** in this case.

When calling **IsiAcquireDomain()** with **SharedServicePin** set to **FALSE** while the ISI engine is not running, a standard service pin message is issued nevertheless, allowing the same installation paradigm and same application code to be used in both self-installed and the managed networks.

After domain acquisition has been enabled by calling **IsiStartDeviceAcquisition()** on the domain address server and it has been started on the device by calling **IsiAcquireDomain()**, the device responds to the **isiWink** ISI event with a visible or audible response. For example, a device may flash its LEDs. The user confirms that the correct device executed its wink routine by activating an appropriate user interface control on the domain address server that calls the server's **IsiStartDeviceAcquisition()** function again. When confirmed, the domain address server grants the unique domain ID to the device. The device notifies its application with ISI events accordingly.

The device automatically cancels domain acquisition if it receives multiple, but mismatching, domain response messages. This mismatch can happen if multiple domain address servers with different domain addresses are in device acquisition mode, and all respond to the device's query.

Devices should support domain acquisition whenever possible (device resources permitting) rather than only supporting device fetching because the domain acquisition process provides a more robust process with features such as automatic retries and automatic connection reminders.

The **IsiCancelAcquisition()** function causes a device to cancel domain acquisition. The cancellation applies to both device and domain acquisition. After this function call is completed, the ISI engine calls **IsiUpdateUserInterface()** with the **IsiNormal** event. On a domain address server, use the **IsiCancelAcquisitionDas()** function instead.

Example 1: The following example starts domain acquisition on a domain address server when the user presses a Connect button on the server.

```
if (connect_button_pressed) {
  IsiStartDeviceAcquisition();
}
```

When started, the domain address server remains in this state for five minutes, unless cancelled with an **IsiCancelAcquisitionDas()** call. Each successful device acquisition retriggers this timeout.

Example 2: The following example starts domain acquisition on a device when the user pushes a Connect button on the device.

```
if (connect_button_pressed) {
  IsiAcquireDomain(FALSE);
}
```

Fetching a Device from a Domain Address Server

A domain address server can use the **IsiFetchDevice()** function to assign the DAS' unique domain ID to any device. Unlike the **IsiAcquireDomain()** function, the **IsiFetchDevice()** function does not require any action, or special library code, on the device. To fetch a device, call the **IsiFetchDevice()** function on the domain address server.

DAS devices must make this feature available to the user. With this feature, it is not required that devices support domain acquisition in order to participate in an ISI network that uses unique domain IDs.

Similar to the domain acquisition process, fetching a device also requires a manual confirmation step to ensure that the correct device is paired with the correct domain address server.

Example: The following example fetches a device on a domain address server when the user presses the Connect button on the server.

```
if (connect_button_pressed) {
  IsiFetchDevice();
}
```

Fetching a Domain for a Domain Address Server

A domain address server can use the **IsiFetchDomain()** function to obtain a domain ID. Unlike the **IsiAcquireDomain()** function, the **IsiFetchDomain()** process does not require a domain address server to provide the domain ID information, and does not use the DIDRM, DIDRQ, and DIDCF standard ISI messages. Instead, the domain address server uses the **IsiFetchDomain()** function to obtain the current domain ID from any device in the network, even from those that do not implement or execute ISI at all. This is typically used when installing replacement or redundant domain address servers in a network: a domain address server normally uses the **IsiGetPrimaryDid()** override to specify a unique, non-standard, primary domain ID. A replacement domain address server (or a redundant domain address server) needs to override this

preference by using the domain ID that is actually used in the network. This override is provided with the **IsiFetchDomain()** function.

Example: The following example fetches a domain on a domain address server when the user presses the Connect button on the server.

```
if (connect_button_pressed) {
  IsiFetchDomain();
}
```

If no unambiguous domain ID is already present on the network, the domain address server uses its default domain ID, as advised with the **IsiGetPrimaryDid()** callback, as a unique domain ID.

Managing Network Variable Connections

You can exchange data between devices by creating *connections* between network variables on the devices. Connections are like virtual wires, replacing the physical wires of traditional hard-wired systems. A connection defines the data flow between one or more output network variables to one or more input network variables. The process of creating a self-installed connection is called *enrollment*. Inputs and outputs join a connection during open enrollment, much like students join a class during open enrollment. Following the sucessful completion of an ISI enrollment, the ISI engines on the devices in the connection automatically create and manage the network variable connection, assign the network variable selectors and other protocol resources, monitor their suitability, and change these values as needed while the connection is active.

Other connection-related ISI services include deleting an entire connection, removing individual devices from a connection, or extending a connection by adding new participants.

Because an ISI network uses unbounded groups (group size 0), your application should not poll network variable values. Using a request-response service with unbounded groups can significantly degrade network performance.

This section describes the ISI connection model and describes the procedures required to create a connection.

ISI Connection Model

Connections are created during an *open enrollment* period that is initiated by a user, a connection controller, or a device application. When initiated, a device is selected to open enrollment—this device is called the *connection host*. Any device in a connection can be the connection host; the connection host is responsible for defining the open enrollment period and for selecting the connection address to be used by all network variables within the connection. Connection address assignment and maintenance is handled by the ISI engine, and is transparent to your application.

Even though any device in a connection can be the connection host, if you have a choice of connection hosts, pick the natural hub as the connection host. For example, in a connection with one switch and multiple lights, the switch is the natural hub, whereas in a connection with one light and multiple switches, the light is the natural hub. If there is no natural hub—multiple switches connected to multiple lights for example—you can pick any of the devices (preferably one with easy access).

A connection host opens enrollment by sending a *connection invitation*. After a connection host opens enrollment, any number of devices can join the connection.

Connections are created among *connection assemblies*. A connection assembly is a block of functionality, a grouping of one or more network variables, much like a Neuron C functional block. A simple assembly refers to a single network variable, as shown in **Figure 69**.

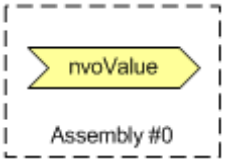

Figure 69. A Simple Assembly

A connection assembly that consists of a single network variable is called a *simple assembly*.

A single assembly can include multiple network variables in a functional block, can include multiple network variables that span multiple functional blocks, or can exist on a device that does not have any functional blocks; an assembly is a collection of one or more network variables that can be connected as a unit for some common purpose.

A connection assembly that consists of more than one network variable is called a *compound assembly*, as shown in **Figure 70**.

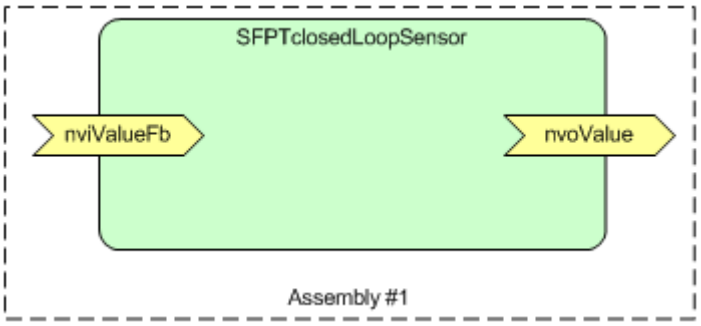

Figure 70. A Compound Assembly

For example, a combination light-switch and lamp ballast controller can have both a switch and a lamp functional block, which are paired to act as a single assembly in an ISI network, but could be handled as independent functional blocks in a managed network, as shown in **Figure 71** on page 208.

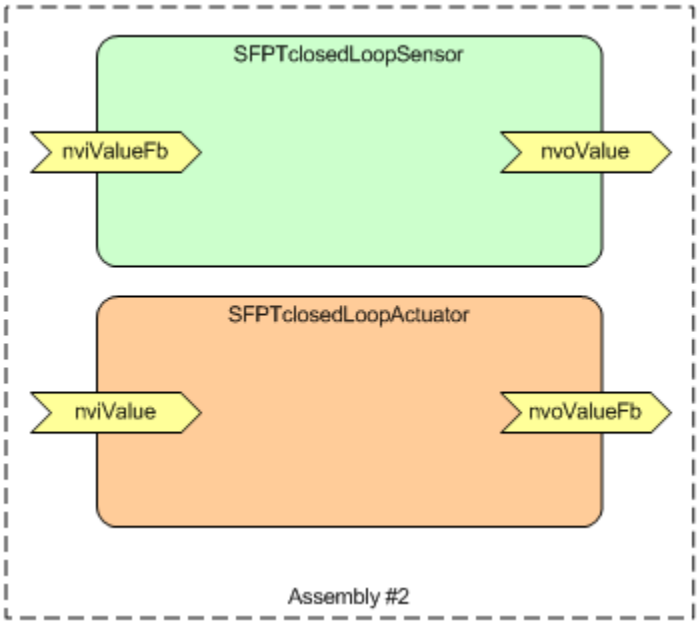

Figure 71. Multiple Functional Blocks as a Single Compound Assembly

To communicate and identify an assembly to the ISI engine, the application assigns a unique number to each assembly. This assembly number must be in the 0 to 254 range, sequentially assigned starting at 0. Required assemblies for standard profiles must be first, assigned in the order that the profiles are declared in the application. Standard ISI profiles that define multiple assemblies must specify the order in which the assemblies are to be assigned.

Each assembly has a *width*, which is equal to the number of network variable selectors used in the enrollment. Typically, but not necessarily, the number of network variable selectors in an enrollment equals the number of network variables in the assembly. In the previous figures, for example, assembly 0 has a width of 1, assembly 1 typically has a width of 2, and assembly 2 typically has a width of 4. All assemblies must have a width of at least 1. Simple assemblies have a width of 1; compound assemblies typically have a width greater than 1.

Recommendation: Keep the width of an assembly as small as possible while maintaining the functionality of the application. For example, keep the width below 10.

One of the network variables in a compound assembly is designated as the *primary network variable*. If the primary network variable is part of a functional block, that functional block is designated as the *primary functional block*. Information about the primary network variable can be included in the connection invitation.

To open enrollment, the connection host broadcasts a connection invitation that can include the following information about the assembly:

- The network variable type of the primary network variable in the assembly
- The functional profile number of the primary functional profile in the assembly
- The connection width

Other devices on the network receive the invitation and interpret the offered assembly to decide whether they could join the new connection.

In the case of assembly 0 in **Figure 69** on page 207, the connection invitation can specify a width of one and the network variable type. This is a case similar to the one employed by a generic switch device where the switch offers a **SNVT_switch** network variable that is not tied to a specific functional profile.

Assembly 1 in **Figure 70** on page 207 demonstrates a more specialized example. A switch can offer this assembly and describe it as an implementation of the **SFPTclosedLoopSensor** profile, with a width of two, and a **SNVT_switch** input and output. The ISI protocol defines how multiple network variable selectors are mapped to the individual network variables offered.

Because the invitation includes no more than one functional profile number, a compound assembly is typically limited to a single functional block on each device. To include multiple functional blocks in an assembly, a *variant* can be specified. A variant is an identifier that customizes the information specified in the connection invitation. Variants can be defined for any device category or any functional profile-member number pair.

For example, a variant can be specified with the **SFPTclosedLoopSensor** functional block offered in assembly 2 in **Figure 71** on page 208 to specify that the **SFPTclosedLoopActuator** functional block is included in the assembly. Standard variant values are defined in standard functional profiles that are published by LONMARK International, and manufacturers can specify manufacturer-specific variant values for manufacturer-specific assemblies.

Each assembly on a device has a unique number that is assigned by the application. Each network variable on a device can be assigned to an assembly. The ISI engine calls the **IsiGetNvIndex()** and **IsiGetNextNvIndex()** callback functions to map a member of an assembly to a network variable on the device.

Opening Enrollment

You can create a connection using *automatic*, *controlled*, or *manual* enrollment. When you use controlled or manual enrollment, user intervention is required to identify devices or assemblies to be connected. Controlled enrollment is initiated by a centralized tool, such as a controller or user interface panel. This centralized tool is called the *connection controller*. Most of the standard ISI profiles require support for controlled enrollment. Manual enrollment is initiated from the devices to be connected, typically with a push button called the **Connect** button. When you use automatic enrollment, connections are automatically created, and no user intervention is required.

The standard Micro Server images support controlled enrollment.

To join a connection, a device must support at least one type of enrollment. A device can support multiple types of enrollment, or a device can support all three types of enrollment. For example, a lamp actuator can support automatic enrollment to a gateway, controlled enrollment configured by a user interface panel, and manual enrollment with switch devices. Devices that support controlled enrollment must also support connection recovery as described in *Recovering Connections* on page 233. Standard functional profiles can require support for specific types of enrollment.

An event triggers your application to open enrollment. The type of event depends on the type of enrollment:

- Manual enrollment: A user input on the device itself typically triggers manual enrollment. The input can be a simple button push, or a device could have a more complex user interface that allows the user to request a connection.
- Controlled enrollment: A request from a connection controller typically triggers controlled enrollment. This request is typically initiated by some user input to the connection controller and arrives in a control request (*CTRQ*) message. The CTRQ message identifies an ISI function and an optional parameter.
- Automatic enrollment: The **isiWarm** event in the **IsiUpdateUserInterface()** callback function typically triggers automatic enrollment.

To open manual enrollment, call the **IsiOpenEnrollment()** function on the connection host, passing in the assembly number to be offered for this connection. The ISI engine then sends a connection invitation by broadcasting an *open enrollment message* (*CSMO)*. The CSMO message is the invitation for other devices to join this connection, and signals an open enrollment period. The ISI protocol also provides extended versions of the CSMO messages, which add fields to determine if the connection is acknowledged or polled, the scope of the connection and parts of the program ID, and the primary network variable member.

The ISI engine creates the CSMO message by calling the **IsiCreateCsmo()** function, which fills the relevant fields of an **IsiCsmoData** data structure with the values needed to describe the connection type and data that is offered to the network. The default implementation of this function, which is provided with the ISI libraries and is available to Neuron C applications, is not available to ShortStack devices. However, you can implement this function either within the host application or within a custom Micro Server.

After calling the **IsiCreateCsmo()** function, the ISI engine constructs the remainder of the CSMO message and broadcasts the connection invitation to the network. To create a compound connection (one with an assembly width larger then 1), you must override the **IsiGetWidth()** callback function. Sending reminders of this message also calls several callback functions, including **IsiCreateCsmo()** and **IsiGetWidth()**.

Controlled enrollment is initiated and controlled by the connection controller, which opens the controlled enrollment by sending a CTRQ message specifying the **IsiOpenEnrollment()** function, and also specifying the assembly number to be offered. The application must respond to the CTRQ message with a control response (*CTRP*) message indicating that it implements the requested operation.

If your ShortStack device needs to use controlled enrollment, you can create a custom Micro Server that includes it.

To open automatic enrollment, wait for the **IsiWarm** event from the **IsiUpdateUserInterface()** callback function, and then call the **IsiInitiateAutoEnrollment()** function, passing a pointer to an **IsiCsmoData** structure containing the invitation, and an the assembly number to be offered for this connection. The ISI engine then sends a connection invitation by broadcasting an automatic enrollment (*CSMA)* message. The ISI engine also

sends periodic reminders about the automatic connection by sending CSMR messages. The reminder ensures that new devices have an opportunity to join the automatic connections.

Whenever a CSMR is due, the ISI engine calls **IsiCreateCsmo()** to create the message. The CSMA and CSMR messages are the invitations for other devices to enroll in this connection automatically. Opening automatic enrollment through **IsiInitiateAutoEnrollment()** is an immediate action, and after the call is made, the connection is implemented for the assembly that the call was made with, regardless of whether there are any members for the connection.

The ISI engine automatically transmits the extended CSMOEX, CSMAEX, or CSMREX message (as appropriate) if **isiFlagExtended** was specified during the start of the engine. Otherwise, the ISI engine automatically clips the **Extended** sub-structure of the **IsiCsmoData** structure and issues the regular CSMO, CSMA, or CSMR message.

You can provide feedback to the user while enrollment is open, for example by starting a Connect light to flash. This is typically only done with manual enrollment. The ISI engine informs your application of significant ISI events by calling an **IsiUpdateUserInterface()** callback function.

Example 1: The following example opens automatic enrollment.

```
void IsiUpdateUserInterface(IsiEvent event, unsigned
      parameter) {
  if (event == IsiWarm && !myIsiGetIsConnected(myAssembly))
  {
    IsiInitiateAutoEnrollment(&myCsmoData, myAssembly);
  }
}
```

In this example, the **Event** is compared to **IsiWarm** and to the value returned by the **myIsiGetIsConnected()** function. Your application implements this function, which returns **TRUE** if the status for the specified assembly (**myAssembly**) is connected, and returns **FALSE** otherwise. To maintain the connection status for each assembly, the application should periodically call the **IsiQueryIsConnected()** function. Then, within the **IsiIsConnectedReceived()** callback handler function, you can update the connection status for each assembly.

The **IsiWarm** event signals that a sufficient amount of time has passed since the ISI engine has been started. This interval includes a random component to prevent all devices in the network from simulatenously starting the automatic enrollment processes and thus colliding in the event of a site-wide return to power.

Example 2: The following example opens manual enrollment for a simple assembly with one network variable, using the network variable's global index as the application-specific assembly number. This example runs within your host application.

```
void startEnrollment(void) {
  IsiOpenEnrollment(LonNvIndexNvoValue);
}
```

Example 3: The following controlled enrollment example instructs a remote device with a specified unique ID (Neuron ID) to open enrollment for its assembly number 5. The first part of this example runs within your host application, which initiates the controlled enrollment request (the host application

implements an ISI connection controller), and the second part of this example runs within a custom Micro Server that is used by the targeted remote device.

See the *Interoperable Self-Installation Protocol Specification* for information about the ISI Protocol, including its message codes and structures. For example, the **IsiControl** enumeration and the **IsiMessage** data structure are not included in the **ShortStackIsiTypes.h** file.

```
const LonApiError controlEnrollment(IsiControl control,
        unsigned parameter, LonUniqueId* pUniqueId) {

  LonSendUniqueId target;
  IsiMessage message;

  /* Use Neuron ID addressing with one of the addresses
   * gathered during device discovery */
  target.Type = LonAddressNeuronId;
  target.Domain = 0;
  target.RepeatRetry = 3 |
      (LonRpt192<<LON_SENDNID_REPEAT_TIMER_SHIFT);
  target.RsvdTransmit = LonTx96;
  target.subnet = 0;
  memcpy(target.NeuronId, pUniqueId,
      sizeof(target.NeuronId));

  /* Prepare the ISI message */
  message.Header.Code = IsiCtrq;
  message.Msg.Ctrq.Control = control;
  message.Msg.Ctrq.Parameter = parameter;

  return LonSendMsg(LonMtIndexMyTag, FALSE,
      LonServiceRequest, FALSE,
      (const LonSendAddress*)&target,
      IsiApplicationMessageCode, &message,
      sizeof(IsiMessageHeader) + sizeof(IsiCtrqMessage));
}

void myEnroll(...) {
    ...
    LonApiError error = controlEnrollment(IsiOpen, 5, ...);
    ...
}
```

Your application can evaluate success or failure of the request by using the **LonResponseArrived()** callback handler function. When the controlled enrollment request completes, the target device replies with an ISI CTRP response message, which indicates success or failure. The CTRP message includes the target device's unique ID, which allows you to correlate it with the outstanding request.

If the device fails to provide a CTRP response message, you should generally assume that the target device does not implement controlled enrollment. As the example shows, you should use network protocol features, such as the repeat counter and timer values, to configure repeated communication attempts.

On the receiving device, a **controlledEnrollmentDispatcher()** function and a **sendControlResponse()** utility function are implemented to process the controlled enrollment request.

To ensure that your custom Micro Server can control enrollment, add a call to the **controlledEnrollmentDispatcher()** function within the **IsiMsgHandler()** function in the **MicroServer.nc** file. An example for the calling the **controlledEnrollmentDispatcher()** function is provided in Example 2 in *Accepting a Connection Invitation* on page 218.

```
boolean IsiMsgHandler(void) {
  boolean result, preemptionMode;
  boolean enrolled;

  result = FALSE;
  preemptionMode = shortStackInPreempt();

  enrolled = controlledEnrollmentDispatcher();

  switch(isiType) {
#ifdef  SS_SUPPORT_ISI_S
    case isiTypeS:
      result = IsiApproveMsg() &&
          (preemptionMode
           || !IsiProcessMsgS()
           || controlledEnrollmentDispatcher());
      break;
#endif  //  SS_SUPPORT_ISI_S
#ifdef  SS_SUPPORT_ISI_DA
    case isiTypeDa:
      result = IsiApproveMsg() &&
         (preemptionMode ||
          !IsiProcessMsgDa() ||
          controlledEnrollmentDispatcher());
      break;
#endif  //  SS_SUPPORT_ISI_DA
#ifdef  SS_SUPPORT_ISI_DAS
    case isiTypeDas:
      result = IsiApproveMsgDas() &&
          (preemptionMode
           || !IsiProcessMsgDas()
           || controlledEnrollmentDispatcher());
      break;
#endif  //  SS_SUPPORT_ISI_DAS
  }
  return result;
}
```

Example 4: The following example opens manual enrollment for a compound assembly with four selectors. The **IsiGetWidth()** returns the library's default value. In this example, enrollment is being opened in response to the user's pressing a Connect button. Enrollment can only be opened when the ISI engine is in the normal state. The **ProcessIsiButton()** function is called in response to the Connect button's being pressed.

This example runs within your host application.

```
IsiEvent isiState = IsiNormal;

void IsiCreateCsmo(....) {
  // set pCsmoData as desired
}
```

```
unsigned IsiGetWidth(unsigned assembly) {
  return 4;
}

void ProcessIsiButton(unsigned assembly) {
  switch(isiState) {
    ...
    case IsiNormal:
      IsiOpenEnrollment(assembly);
      break;
    ... //Processing for other states
  } // end of switch(isiState)
}
```

The example assumes that the **IsiCreateCsmo()** and **IsiGetWidth()** callback handler functions are implemented in the same location, and implies that both are implemented in the location of the **ProcessIsiButton()** function (presumably, within your host application). When you create an ISI-enabled custom Micro Server, you can choose whether the **IsiCreateCsmo()** and **IsiGetWidth()** callback handler functions should be implemented local to the Micro Server or on the host, but these two callback handler functions would typically be implemented in the same location.

Example 5: The following refines example 1 and provides a more comprehensive example of opening automatic enrollment for a simple assembly with one network variable.

This example runs within your host application.

```
// MyCsmoData defines the enrollment details for the
// automatic ISI network variable connection offered by
// this device.
static const IsiCsmoData MyCsmoData = {
  // group
  ISI_DEFAULT_GROUP,
  // direction and width:
  IsiDirectionOutput << ISI_CSMO_DIR_SHIFT) | 1,
  // Profile number
  { 0, 2 },
  // NV type index      (76: SNVT_freq_hz)
  76,
  // Variant:
  0
};

// Call InitiateAutoEnrollment in response to isiWarm
void IsiUpdateUserInterface(IsiEvent event, unsigned
      parameter) {
  if (event == IsiWarm &&
      !myIsiGetIsConnected(myAssemblyNumber)) {
    // We waited long enough and we are not connected
    // already, so let's open an automatic connection:
    IsiInitiateAutoEnrollment(&MyCsmoData,
      myAssemblyNumber);
  }
}
```

```
void IsiCreateCsmo(unsigned assembly, IsiCsmoData* pCsmo) {
  if (assembly == myAssemblyNumber) {
    memcpy(pCsmo, &MyCsmoData, sizeof(IsiCsmoData));
  }
}

unsigned IsiGetWidth(unsigned assembly) {
  unsigned result = 0;
  if (assembly == myAssemblyNumber) {
    result = LON_GET_ATTRIBUTE(MyCsmoData, ISI_CSMO_WIDTH);
  }
  return result;
}
```

In this example, the **Event** is compared to **IsiWarm** and to the value returned by the **myIsiGetIsConnected()** function. Your application implements this function, which returns **TRUE** if the status for the specified assembly (**myAssembly**) is connected, and returns **FALSE** otherwise. To maintain the connection status for each assembly, the application should periodically call the **IsiQueryIsConnected()** function. Then, within the **IsiIsConnectedReceived()** callback handler function, you can update the connection status for each assembly.

Example 6: The following example opens automatic enrollment for a compound assembly with four selectors, offering enrollment for member network variables 1 to 4 of an implementation of the **SFPTsceneController** profile (the **nviScene**, **nvoSwitch**, **nviSetting**, and **nviSwitch** members).

This example runs within your host application.

```
// MyCsmoData defines the enrollment details for the
// automatic ISI network variable connection offered by
// this device
static const IsiCsmoData MyCsmoData = {
  // group
  ISI_DEFAULT_GROUP,
  // direction and width:
  (isiDirectionVarious << ISI_CSMO_DIR_SHIFT) | 4,
  // Profile number in big-endian notation:
  { 3251 / 256, 3251 % 256 },
  // NV type index (0: determined by SFPT)
  0,
  // Variant:
  0
};

// Call InitiateAutoEnrollment in response to isiWarm
void IsiUpdateUserInterface(IsiEvent event, unsigned
      parameter) {
  if (event == IsiWarm &&
      !myIsiGetIsConnected(myAssemblyNumber)) {
    // We waited long enough and we are not connected
    // already, so let's open an automatic connection:
    IsiInitiateAutoEnrollment(&MyCsmoData,
      myAssemblyNumber);
  }
}

void IsiCreateCsmo(unsigned assembly, IsiCsmoData* pCsmo) {
```

```
  if (assembly == myAssemblyNumber) {
    memcpy(pCsmo, &MyCsmoData, sizeof(IsiCsmoData));
  }
}

unsigned IsiGetWidth(unsigned assembly) {
  unsigned result = 0;
  if (assembly == myAssemblyNumber) {
    result = LON_GET_ATTRIBUTE(MyCsmoData, ISI_CSMO_WIDTH);
  }
  return result;
}
```

As in the previous example, the **Event** is compared to **IsiWarm** and to the value returned by the **myIsiGetIsConnected()** function. Your application implements this function, which returns **TRUE** if the status for the specified assembly (**myAssembly**) is connected, and returns **FALSE** otherwise. To maintain the connection status for each assembly, the application should periodically call the **IsiQueryIsConnected()** function. Then, within the **IsiIsConnectedReceived()** callback handler function, you can update the connection status for each assembly.

Example 7: For a complete example that implements connection management for multiple assemblies, see the self-installation example application that is included with the ShortStack FX ARM7 Example Port, which is available for free download from www.echelon.com/shortstack.

Receiving an Invitation

You can receive a connection invitation and specify which assemblies are eligible to join the ISI connection. When an ISI device receives a CSMO, CSMA, or CSMR connection invitation message, the ISI engine first checks the availability of the device resources that are required to implement the connection. If any of these resources is missing or insufficient, such as address or connection table space, the invitation is dropped.

If the ISI engine determines that there are sufficient resources, it calls the **IsiGetAssembly()** and **IsiGetNextAssembly()** callback handler functions with the received CSMO, CSMA, or CSMR message. These functions return all assembly numbers that are provisionally approved to join the connection. The **automatic** argument of **IsiGetAssembly()** and **IsiGetNextAssembly()** indicates whether the enrollment is manual or controlled (CSMO) or automatically (CSMA or CSMR) initiated, with **FALSE** meaning that the enrollment was initiated manually or by a connection controller. On devices that do not support connection removal, the assembly is ignored if it is already engaged in another connection.

When a device receives an extended CSMOEX, CSMAEX, or CSMREX message, all fields of the **IsiCsmoData** structure are passed to the application, and the fields in the **Extended** sub-structure are all valid.

When a device receives a regular CSMO, CSMA, or CSMR message, the extended fields are automatically set to all zeros, with exception of the **Extended.Member** field, which is set to one.

Applications do not need to distinguish between regular and extended incoming messages.

You can provide feedback to the user when an invitation is received and provisionally approved, for example by causing a Connect light to flash while enrollment is open. Such feedback is typically only provided for a manual connection. The ISI engine informs your application that an eligible invitation has been received and provisionally approved by calling the **IsiUpdateUserInterface()** callback function (with the **IsiPending** event code) for each assembly that is provisionally approved to join the connection. The application can indicate provisionally approved, but not yet accepted, connection invitations.

Example: The following example receives and provisionally approves a connection invitation, and blinks a Connect light until the invitation is accepted, or the connection is confirmed or canceled.

This example runs within your host application.

```
// IsiUpdateUserInterface is called with IsiPending as the
// IsiEvent parameter in response to receiving a CSMO
void IsiUpdateUserInterface(IsiEvent event, unsigned
      parameter) {
  ... //Optional event processing
  isiState = (event == IsiPending || event == IsiApproved
      || event > IsiWarm) ? event : IsiNormal;
}

unsigned IsiGetAssembly(const IsiCsmoData* pCsmo,
      LonBool automatic) {
  unsigned result = ISI_NO_ASSEMBLY;
  if (pCsmo->Group == ISI_LIGHTING_CATEGORY
      && pCsmo->Extended.Scope == isiScopeStandard
      && pCsmo->NvType == SNVT_SWITCH_2_INDEX
      && !(pCsmo->Variant & 0x60)
      && !LON_GET_ATTRIBUTE(pCsmo->Extended, ISI_CSMO_ACK)
      && !LON_GET_ATTRIBUTE(pCsmo->Extended,
          ISI_CSMO_POLL)) {
    // Recognized CSMO, return appropriate assembly
    // number
    result = myAssemblyNumber;
  }
  return result;
}

unsigned IsiGetNextAssembly(const IsiCsmoData* pCsmo,
      LonBool automatic, unsigned assembly) {
  unsigned result = ISI_NO_ASSEMBLY;

  if (assembly == myAssemblyNumber) {
    result = myAssemblyNumber + 1;
  }
  return result;
}
```

The example identifies the enrollment and specifies **myAssemblyNumber** as the first local applicable assembly for the enrollment. The **GetNextAssembly()** callback handler function then adds a second local applicable assembly to the list. Unacceptable enrollment data, or requests for additional local assemblies, receive the **ISI_NO_ASSEMBLY** constant.

Accepting a Connection Invitation

You can accept a connection invitation to join the offered connection. When you accept a connection invitation, the ISI engine sends an enrollment acceptance message (CSME) to the connection host. Accepting an invitation only sends an acceptance to the connection host; the connection is not implemented until the connection host confirms the new connection.

You can only accept enrollment for an assembly that has been provisionally approved. To provisionally approve an assembly, the **IsiGetAssembly()** or **IsiGetNextAssembly()** function must return the assembly number for the current **IsiCsmoData** structure, and the **IsiUpdateUserInterface()** callback function must identify the current assembly as being in the **IsiPending** state.

For manual enrollment, a connection invitation is typically accepted based on user input. For example, LEDs blink on a device when invitations are received and provisionally approved, and the user then pushes the related Connect button to accept a specific invitation.

For a controlled enrollment, a connection invitation is typically accepted based on a request from a connection controller. This request is typically initiated by some user input to the connection controller.

For automatic enrollment, a connection invitation is typically accepted based on some application-specific criteria. For example, a home gateway opens automatic enrollment for its inputs and outputs, and newly installed home devices automatically accept all eligible connection invitations from the home gateway.

The actual establishment of an automatic connection is handled by the ISI engine, and requires a call to **IsiCreateEnrollment()** or **IsiExtendEnrollment()**. The ISI engine extends the connection if the library supports connection extension, or creates the extension if the library does not support connection extension and the assembly is not already connected, or if the library supports connection removal. The ISI libraries that are used with the standard, ISI-enabled, ShortStack Micro Servers support connection extensions and connection removal procedures. Different ISI libraries can be used with custom Micro Server implementations; see *Creating a Custom Micro Server with ISI Support* on page 248.

For devices that support connection removal, you can create a connection that replaces all existing connections for an assembly. For devices that support connection extension, you can add a new connection to an assembly that might already be enrolled in other connections.

To create a connection that replaces all existing connections for an assembly, call **IsiCreateEnrollment()**. To add a connection to an assembly without overriding any existing connections associated with the same assembly, call **IsiExtendEnrollment()**. You can extend a nonexistent connection; **IsiExtendEnrollment()** has the same functionality as **IsiCreateEnrollment()** if no connection exists for the assembly.

Extending a connection consumes additional device and network resources, compared with the initial connection. Each extension to a connection requires one or more new aliases and connection table entries, and results in additional network transactions for every update to the connection. You can eliminate this additional resource usage by deleting and re-creating a connection instead of extending it.

You can provide feedback to the user when an invitation is accepted, for example by changing the state of the Connect light when the connection invitation is accepted from flashing to solid on. Such feedback is typically only provided for manual enrollment. The ISI engine informs your application that a connection invitation has been accepted by calling the **IsiUpdateUserInterface()** callback function, assigning the **IsiApproved** or **IsiApprovedHost** state to the respective assembly. The application indicates the accepted connection invitation.

Example 1: The following manual enrollment example accepts a connection invitation when the user presses a Connect button.

This example runs within your host application.

```
IsiEvent isiState;

void ProcessIsiButton(unsigned assembly) {
  switch(isiState) {
  ...
    case IsiPending:
      IsiCreateEnrollment(assembly);
      break;
      ... //Processing for other states
  } // end of switch(state)
}
```

After the host accepts the connection, your application receives the **IsiUpdateUserInterface()** callback with the **Event** set to **IsiApproved**. Your application can use this event status to update the device interface, for example, by illuminating an LED.

Example 2: The following example opens controlled enrollment when requested by the connection controller.

This example runs within a custom Micro Server.

```
void sendControlResponse(boolean success) {
  IsiMessage ctrlResp;

  ctrlResp.Header.Code = isiCtrp;
  ctrlResp.Ctrp.Success = success;
  memcpy(ctrlResp.Ctrp.NeuronID, read_only_data.neuron_id,
      NEURON_ID_LEN);

  resp_out.code = isiApplicationMessageCode;
  memcpy(resp_out.data, &ctrlResp,
      sizeof(IsiMessageHeader)+sizeof(IsiCtrp));
  resp_send();
}

boolean controlledEnrollmentDispatcher(void) {
  boolean isProcessed;
  IsiMessage inMsg;

  isProcessed = FALSE;
  memcpy(&inMsg, msg_in.data, sizeof(IsiMessage));

  if (inMsg.Header.Code == isiCtrq) {
    if (inMsg.Ctrq.Control == isiOpen) {
      sendControlResponse(TRUE);
```

```
        IsiOpenEnrollment(inMsg.Ctrq.Parameter);
        isProcessed = TRUE;
      } else if (inMsg.Ctrq.Control == isiCreate) {
        sendControlResponse(TRUE);
        IsiCreateEnrollment(inMsg.Ctrq.Parameter);
      } else if (inMsg.Ctrq.Control == isiFactory) {
        sendControlResponse(TRUE);
        IsiReturnToFactoryDefaults();
      } else {
        sendControlResponse(FALSE);
      }
    } else {
      // Other requests deleted for this example
      ...
    }
    return isProcessed;
  }
```

Implementing a Connection

In a manual or controlled enrollment, when a connection host sends a connection invitation by broadcasting an open enrollment message, one or more devices can accept the connection invitation and respond with an enrollment acceptance message (CSME). When the connection host receives at least one CSME message, the host application receives the **IsiApprovedHost** event through the **IsiUpdateUserInterface()** callback function. Typically, the application changes the state of the related Connect light from flashing to solid on.

When the connection host's assembly is in the **IsiApprovedHost** state, the connection can be cancelled or implemented. See *Canceling a Connection* on page 221 for information about cancellation.

To implement a connection on a connection host, call either **IsiCreateEnrollment()** or **IsiExtendEnrollment()**. The connection host joins the connection and issues a connection enrollment confirmation message (CSMC). When calling **IsiCreateEnrollment()**, any connection that exists for the same assembly is removed; see *Deleting a Connection* on page 222 for more information. When calling **IsiExtendEnrollment()**, the new connection is added to any existing connections for the same assembly, consuming an alias table entry for each NV in the assembly.

After the connection host confirms the connection, devices that have previously accepted the connection invitation join the connection by replacing or extending an existing connection, depending on the function that was used to accept the invitation.

When a device joins a connection, the ISI engine on that device updates the network configuration for the device, and the accepted connection becomes active.

The ISI engine automatically implements the connections for the accepted assembly. To determine the network variables to be connected, the ISI engine calls the **IsiGetNvIndex()** and **IsiGetNextNvIndex()** functions for each selector used with the connection.

You can provide feedback to the user when a connection has been joined, for example by turning off the Connect light. Such feedback is typically only provided for manual connections. The ISI engine informs your application that a

connection has been implemented by providing the **IsiImplemented** event through the **IsiUpdateUserInterface()** callback function. The application indicates the new connection. Your application will receive one **IsiImplemented** event for each network variable that belongs to the assembly.

Example: The following manual enrollment example implements a connection on a connection host when the user presses the Connect button a second time. The complete application also turns off the Connect light to indicate the acceptance on the host.

```
void ProcessIsiButton(unsigned assembly) {
  switch(isiState) {
  ...
    case IsiApprovedHost:
      if (bCancelEnrollment)
        IsiCancelEnrollment();
      else
        IsiCreateEnrollment(assembly);
      break;
      ... // Processing for other states
  } // End of switch(state)
}
```

After the host accepts the connection, your application receives the **IsiImplemented** event through the **IsiUpdateUserInterface()** callback handler function once for each local network variable associated with the assembly. Your application can use this event status to update the device interface, for example, by illuminating an LED.

Canceling a Connection

You can cancel a pending enrollment on the connection host at any stage, and on any device that has accepted the connection invitation. However, cancellation is no longer possible after the connection is implemented; see *Deleting a Connection* on page 222 for information about deleting an implemented connection.

Pending enrollment sessions are automatically cancelled if:

- On the connection host, if no connection enrollment acceptance message (CSME) is received within the open enrollment period after the **IsiOpenEnrollment()** function call.
- On the connection host, if the connection is not implemented by a **IsiCreateEnrollment()** or **IsiExtendEnrollment()** function call within the open enrollment period after the receipt of a connection enrollment confirmation message (CMSE).
- On an accepting device, if the connection has been accepted and no connection enrollment confirmation message (CMSC) has been received within the open enrollment period after the acceptance.

To explicitly cancel a pending enrollment, call the **IsiCancelEnrollment()** function.

When a connection host cancels a pending enrollment session, it issues a connection enrollment cancellation message (CSMX). Devices that have accepted the related connection invitation automatically cancel when they receive a related CSMX message.

When a connection member cancels a pending enrollment session, the cancellation only has local effect—the approved assembly changes to the **IsiCancelled** state. Because the connection host can re-send invitation messages (CSMOs), the same device can, once again, conditionally approve the assembly and move it to the **IsiPending** state. The user can now accept the connection invitation again (by causing the application to call **IsiCreateEnrollment()** or **IsiExtendEnrollment()**), or simply do nothing. The pending assembly remains pending until the enrollment is closed, and automatically returns to the **IsiNormal** state.

Deleting a Connection

You can delete an implemented connection using one of three methods:

- The device can restore factory defaults by calling the **IsiReturnToFactoryDefaults()** function. This function clears all system tables, stops the ISI engine, and resets the Micro Server. See *Deinstalling a Device* on page 237 for more information about this function.
- The device can delete a connection by calling the **IsiDeleteEnrollment()** function. This function causes the connection information to be removed from the local device, as well as on all other devices that are members of the same connection. The **IsiDeleteEnrollment()** function can be called on the connection host, and on any other device that has joined the connection.
- The device can opt out of an existing connection, leaving other devices that have joined the same connection unchanged. To leave a connection locally, call the **IsiLeaveEnrollment()** function. Calling this function on the connection host has the effect of **IsiDeleteEnrollment()**, that is, a connection host cannot leave a connection, but must always delete the connection.

The ISI engine calls the **IsiUpdateUserInterface()** function with the **IsiDeleted** event to notify the application of the completion of a deletion.

Handling ISI Events

You can signal the progress of the enrollment process to the device user. Such feedback is typically only provided for devices that use manual connections, because automatic and controlled connections do not require user interaction from the connected devices. User feedback could be as simple as a single Connect light and button, possibly shared with the Service light and button. A more complex gateway or controller could have a more sophisticated user interface.

To receive status feedback from the ISI engine, override the **IsiUpdateUserInterface()** callback function. The ISI engine calls this function with the **IsiEvent** parameter set to one of the values listed in **Table 24** on page 223 when the associated event occurs. Some of these events carry a meaningful value in the numeric parameter, as shown in the table.

Table 24. ISI Event Types

IsiEvent	Value	Description
IsiNormal	0	The ISI engine has returned to the normal, or idle, state for an assembly. The related assembly is encoded in the parameter; a parameter value of **ISI_NO_ASSEMBLY** indicates that the event applies to all assemblies.
IsiRun	1	The ISI engine has been successfully started (**parameter** is **TRUE**) or stopped (**parameter** is **FALSE**).
IsiPending	2	The connection related to the assembly given with the numerical parameter has entered the pending state. The event means that the device has received, and provisionally approved, a connection invitation, but has not yet accepted the connection invitation. This event only applies to a connection member. For a connection host, see **IsiPendingHost**. Devices often signal the **IsiPending** (or **IsiPendingHost**) state with a flashing LED.
IsiApproved	3	The connection related to the assembly given with the numerical parameter changed from the pending state to the approved state. This event occurs when a connection invitation has been provisionally approved and accepted. This event only applies to a connection member. For a connection host, see **IsiApprovedHost**. Devices often signal the **IsiApproved** (or **IsiApprovedHost**) state by turning on an LED (which was flashing before, coming from the **IsiPending** or **IsiPendingHost** state).
IsiImplemented	4	The connection related to the assembly given with the numerical parameter has been implemented. This event occurs on a connection host after calling **IsiCreateEnrollment()** or **IsiExtendEnrollment()** to implement a connection and close enrollment, and on a connection member after receiving an enrollment confirmation message (CSMC). The application receives one **IsiImplemented** event for each network variable that is part of the assembly.
IsiCancelled	5	The connection related to the assembly given with the numerical parameter has been cancelled by a timeout, user intervention, or network action. An assembly number of **ISI_NO_ASSEMBLY** indicates that all pending enrollments are cancelled.
IsiDeleted	6	The connection related to the assembly given with the numerical parameter has been deleted.

IsiEvent	Value	Description
IsiWarm	7	The ISI engine has warmed up (that is, a predetermined time, with a random component, has passed since the last reset). After this time, the application can call the **IsiInitiateAutoEnrollment()** function. This event occurs no sooner than the expiry of the T_{auto} ISI protocol timer, but can occur later.
IsiPendingHost	8	The connection related to the assembly given with the numerical parameter has entered the pending state. This event occurs on a connection host after it has issued a connection invitation (CSMO), but not yet received any enrollment acceptance messages (CSMEs). This event only applies to a connection host. For a connection member, see **IsiPending**.
IsiApprovedHost	9	The connection indicated with the numerical parameter changed from the pending state to the approved state. This event occurs on a connection host at the receipt of the first connection enrollment acceptance message (CSME). This event only applies to a connection host. For a connection member, see **IsiApproved**.
IsiAborted	10	The device stopped domain or device acquisition. The parameter is a member of the **IsiAbortReason** enumeration, and indicates the reason for the abort.
IsiRetry	11	The device is retrying the device acquisition procedure. The parameter is the remaining number of retries.
IsiWink	12	The device should perform its wink function. The specific function is application-dependent, but should provide some visible or audible feedback to the user. For example, the application blinks an LED on the device.
IsiRegistered	13	This event indicates either acquisition start or successful acquisition completion on either a device that supports domain acquisition or a domain address server. The parameter indicates either a successful start (**parameter = 0**) or completion (**parameter = 0xFF**).

You typically override the **IsiUpdateUserInterface()** callback function with an application-specific function to provide application-specific user feedback. The default implementation of this function does nothing, and is only useful for devices that exclusively use automatic enrollment.

Figure 72 on page 225 summarizes the typical sequence of events for a connection host using manual or controlled enrollment. The sequence of events is similar for a connection host using automatic enrollment, except that the connection host skips the **IsiApprovedHost** event and goes straight to the **IsiImplemented** event.

Although the sequence of events shown in this figure is typical, the actual sequence of events passed to the **IsiUpdateUserInterface()** callback can vary.

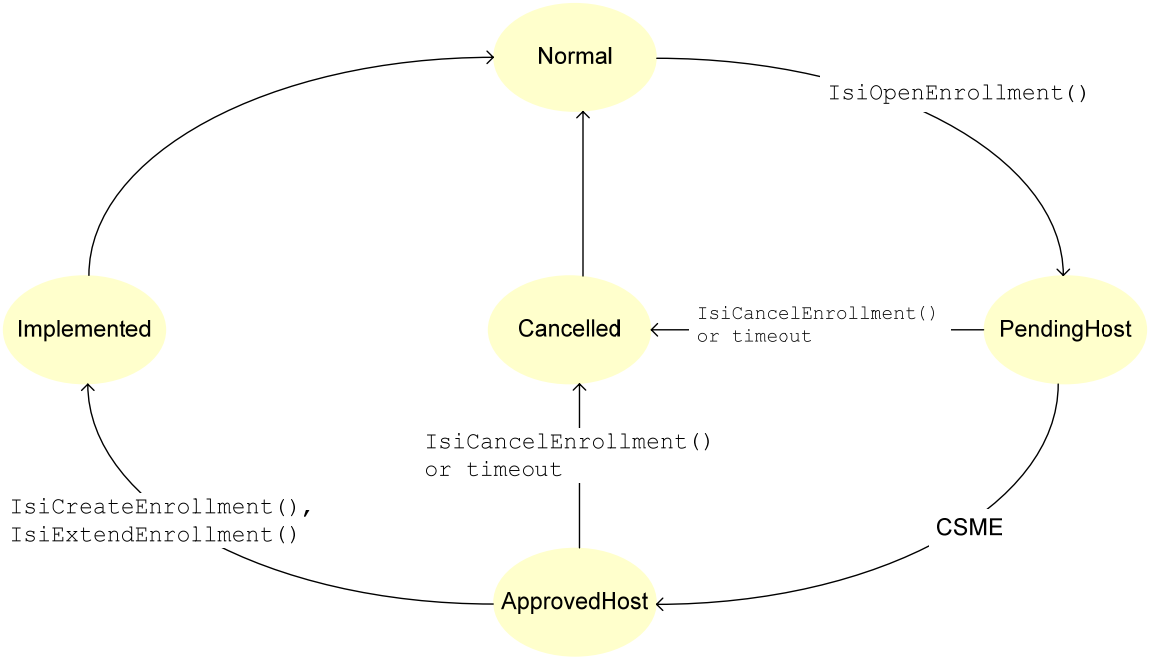

Figure 72. Sequence of Events for a Connection Host

Figure 73 summarizes the typical sequence of events for a connection member. Although the sequence of events shown in this figure is typical, the actual sequence of events passed to the **IsiUpdateUserInterface()** callback can vary.

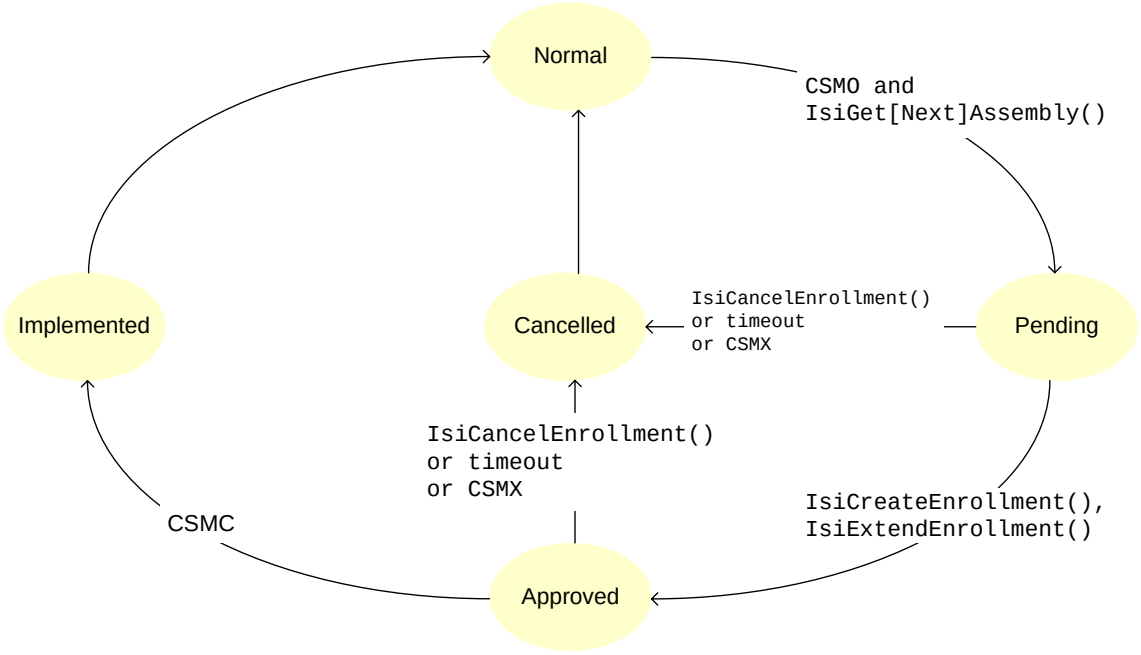

Figure 73. Sequence of Events for a Connection Member

Domain Address Server Support

None of the standard ShortStack Micro Servers supports the creation of an ISI domain address server (DAS) because of resource limitations on all supported hardware platforms.

To implement a domain address server as a ShortStack device, perform either of the following tasks:

- Create a custom Micro Server on a 3150 Smart Transceiver that supports more RAM through the external memory interface, or create a custom Micro Server on an FT 5000 Smart Transceiver. The ISI memory requirement is approximately 0.5 KB.

  Ensure that this Micro Server has sufficient external RAM for buffers (a DAS typically needs fairly large buffer counts) and any DAS-specific code that requires external RAM (such as device lists and lookup-tables on the Micro Server). Typically, external RAM of a few kilobytes suffices.

- Use a standard Micro Server on a 3120 or 3170 Smart Transceiver, or a custom Micro Server on a 3150 or 5000 Smart Transceiver, that does not have built-in ISI support, and implement ISI with DAS-features on the host processor.

Discovering Devices

You can discover all devices in an ISI network. All devices in an ISI network periodically broadcast their status by sending out Domain Resource Usage Message (DRUM) messages. To discover devices, you can monitor these status messages. Gateways and controllers that need to maintain a table of all devices in a network, or provide unique capabilities for specific types of devices in a network, should monitor these messages.

To discover devices, monitor the DRUM messages being sent on the network by other devices and store the relevant information in a *device table*. A device table is a table that contains a list of devices and their attributes including their network addresses. The DRUM messages contain all of the relevant information for explicit messaging. To create a device table, store the relevant DRUM fields, such as subnet ID, node ID, and Neuron ID, in a table that you can use to communicate directly with other devices. To detect deleted devices, monitor the time of the last update for each entry in the table and detect devices that have not recently sent a DRUM.

You can implement the code to maintain the device table within a custom Micro Server or within the host application. For either implementation, you must create a custom Micro Server.

Maintaining a Device Table within the Micro Server

To implement device discovery local to the Micro Server, perform the following steps:

1. Add code to the **MicroServer.nc** file that defines a data structure for the device table.

2. Implement the **ProcessDrum()** function.
3. Create a function that decrements credits from each device in the device table.
4. In the **ShortStackIsiHandlers.h** file, define the **IsiCreatePeriodicMsg()** callback handler function to be implemented within your custom Micro Server.
5. In the **MicroServerIsiHandlers.c** file, call the function that decrements credits from the **IsiCreatePeriodicMsg()** callback handler function.
6. In the **MicroServer.nc** file, modify the **IsiMsgHandler()** function to call your DRUM dispatcher.
7. Create a utility function that informs the host of newly discovered or removed devices.
8. Add code to your host application to process the user-defined remote procedure call for the utility function.

Each of these steps is described in the following sections.

Define the Data Structure

Define a **Device** data structure to hold information about a discovered device, and create a **devices** table to hold information about all discovered devices. You can add the following code to the **MicroServer.nc** file or add it to a separate file (perhaps called **DeviceDiscovery.c**) that you reference (**#include**) from **MicroServer.nc**.

```
#include <mem.h>

#define MAX_DEVICES 16
#define MAX_CREDITS 5

unsigned deviceCount;

// Struct to hold device information
typedef struct {
  unsigned credits;
  unsigned subnetId;
  unsigned nodeId;
  unsigned neuronId[NEURON_ID_LEN];
} Device;

Device devices[MAX_DEVICES];
```

Implement the ProcessDrum() Function

Add the **ProcessDrum()** function to **MicroServer.nc** (or to your **DeviceDiscovery.c**). This function is called from the ISI message handler whenever it sees an ISI DRUM message. We'll add the code that makes this call later.

The function also uses a utility function, **ReportDevice()**, that is described in *The ReportDevice() Utility Function* on page 231.

```
void ProcessDrum(const IsiDrum* pDrum) {
  unsigned i;
  extern ReportDevice(boolean, unsigned);
```

```
  // Iterate through the device list and see if the Neuron
  // ID of the stored device matches that of the new
  // device; if it does, then update the related details
  for (i = 0; i < deviceCount; i++) {
    if (memcmp(devices[i].neuronId, pDrum->NeuronId,
          NEURON_ID_LEN) == 0) {
      devices[i].credits = MAX_CREDITS;
      devices[i].subnetId = pDrum->SubnetId;
      devices[i].nodeId = pDrum->NodeId;
      break;
    }
  }

  // If i is equal to the device count, then the device
  // was not found, so add it to the device table if
  // possible
  if (i == deviceCount && deviceCount < MAX_DEVICES) {
    memcpy(devices[i].neuronId, pDrum->NeuronId,
        NEURON_ID_LEN);
    deviceCount++;
    devices[i].credits = MAX_CREDITS;
    devices[i].subnetId = pDrum->SubnetId;
    devices[i].nodeId = pDrum->NodeId;

    ReportDevice(TRUE, i);
  }
}
```

Create the Decrement Function

Add the **DetectStale()** function to **MicroServer.nc** (or to your **DeviceDiscovery.c**). This function slowly decrements credits from each device in the **devices** table.

If the device is functioning, it continues to send DRUM messages, and thus is maintained in the table. If a device disappears from the network, it is eventually removed from the table.

The function also uses a utility function, **ReportDevice()**, that is described in *The ReportDevice() Utility Function* on page 231.

```
void DetectStale(void) {
  unsigned i;
  extern ReportDevice(boolean, unsigned);

  for (i = 0; i < devicecount; i++) {
    devices[i].credits--;
    if (devices[i].credits == 0) {
      ReportDevice(FALSE, i);
      devicecount--;
      if (devicecount != i) {
        // Move device from end to this spot's location
        memcpy(devices+i, devices+devicecount,
            sizeof(Device));
      }
    }
  }
}
```

Call the **DetectStale()** function at a rate roughly equal to the expected DRUM rate. One way to ensure an appropriate call rate is to call this function from the **IsiCreatePeriodicMsg()** callback handler function, although in this case, you must implement the **IsiCreatePeriodicMsg()** callback handler function local to the Micro Server.

Define IsiCreatePeriodicMsg() in ShortStackIsiHandlers.h

In the **ShortStackIsiHandlers.h** file, define the **IsiCreatePeriodicMsg()** callback handler function to be implemented within your custom Micro Server.

```
/*
 * Callback: IsiCreatePeriodicMsg
 * Standard location: default
 *
 * The IsiCreatePeriodicMsg() callback enabled an optional
 * and advanced feature, through which the application can
 * claim a slot in the ISI broadcast scheduler.
 * This callback is rarely overridden.
 */
/*#define ISI_DEFAULT_CREATEPERIODICMSG */
#define ISI_SERVER_CREATEPERIODICMSG
/*#define ISI_HOST_CREATEPERIODICMSG    */
```

Call the Decrement Function

Within the **MicroServerIsiHandlers.c** file, locate the implementation of the **IsiCreatePeriodicMsg()** callback handler function, and call the **DetectStale()** function from this callback handler function.

```
// ------------------------------------------------------
//  Callback:   IsiCreatePeriodicMsg
// ------------------------------------------------------
#ifndef ISI_DEFAULT_CREATEPERIODICMSG
boolean IsiCreatePeriodicMsg(void) {
#ifdef  ISI_SERVER_CREATEPERIODICMSG

  extern void DetectStale(void);

  boolean result;
  result = FALSE;

  DetectStale();

  // TODO: Add code implementing the actual
  // IsiCreatePeriodicMsg() callback, if needed.

  return result;

#else
#ifdef  ISI_HOST_CREATEPERIODICMSG
  // DO NOT MODIFY - This code redirects the callback to
  // the host
  return IsiRpc(LicIsiCreatePeriodicMsg, 0, 0, NULL, 0);
#endif  //  ISI_HOST_CREATEPERIODICMSG
#endif  //  ISI_SERVER_CREATEPERIODICMSG
}   // IsiCreatePeriodicMsg
```

```
#pragma ignore_notused  IsiCreatePeriodicMsg
#endif  //  ISI_DEFAULT_CREATEPERIODICMSG
```

Call Your DRUM Dispatcher from IsiMsgHandler()

Within the **MicroServer.nc** file, locate the **IsiMsgHandler()** function. After each message has been approved, and you have confirmed that **preemptionMode** is **FALSE**, call your DRUM dispatcher. This routine determines whether the newly arrived ISI message is a DRUM message, and calls **ProcessDrum()** if necessary.

The **ProcessDrum()** function is defined to return **FALSE** so that it can easily be inserted into the **IsiMsgHandler()** routine.

```
boolean ProcessDrum(void) {
  IsiMessage message;

  memcpy(&message, msg_in.data, sizeof(IsiMessage));
  if (message.Header.Code == isiDrum ||
        message.Header.Code == isiDrumEx) {
    ProcessDrum(&message.Msg.Drum);
  }
  return FALSE;
}

// IsiMsgHandler() is a utility function used by the
// ShortStack Micro Server core to identify and process ISI
// messages.  This function returns true if the message was
// handled by this function.

extern boolean shortStackInPreempt(void);

boolean IsiMsgHandler(void) {
  boolean result, preemptionMode;

  result = FALSE;
  preemptionMode = shortStackInPreempt();

  switch(isiType) {
#ifdef  SS_SUPPORT_ISI_S
    case isiTypeS:
      result = IsiApproveMsg() && (preemptionMode ||
          ProcessDrum() || !IsiProcessMsgS());
      break;
#endif  //  SS_SUPPORT_ISI_S
#ifdef  SS_SUPPORT_ISI_DA
    case isiTypeDa:
      result = IsiApproveMsg() && (preemptionMode ||
          ProcessDrum() || !IsiProcessMsgDa());
      break;
#endif  //  SS_SUPPORT_ISI_DA
#ifdef  SS_SUPPORT_ISI_DAS
    case isiTypeDas:
      result = IsiApproveMsgDas() && (preemptionMode ||
          ProcessDrum() || !IsiProcessMsgDas());
      break;
#endif  //  SS_SUPPORT_ISI_DAS
  }
```

```
  return result;
}
#pragma ignore_notused IsiMsgHandler
```

The ReportDevice() Utility Function

The **ReportDevice()** utility function informs the host application of newly discovered or removed devices by implementing a user-defined remote-procedure call (RPC). This call is handled by the **IsiRpc()** function, which supplies the related **Device** data structure and the information about whether this device was newly added or removed from the **devices** table. To reduce overhead, this remote procedure call is implemented as an unacknowledged call.

```
void ReportDevice(boolean added, unsigned index) {
  (void)IsiRpc(LicIsiUserCommand|LicIsiNoAck, added, index,
       devices+index, sizeof(Device));
}
```

Process Your User-Defined RPC

Your host application must process the information about newly discovered or removed devices. The Micro Server's **IsiRpc()** function supplies this information to your host application. You add code to your host application to process this information by extending the **IsiUserCommand()** callback handler function in the **ShortStackIsiHandlers.c** file.

A typical use for this callback is to update an advanced device's graphical user interface with a representation of all devices that are located on the same ISI network. The same device table information can also be used to implement advanced connection scenarios with ISI.

Maintaining a Device Table within a Host Application

As an alternative to implementing the device table within the Micro Server, you can implement most of the device discovery process within the host application. For this implementation, the host receives a DRUM message through a user-defined remote procedure call (RPC) and maintains the device table on the host. You must create a custom Micro Server to forward DRUM messages to the host.

To implement device discovery local to the host application, perform the following steps:

1. Add code to the host application that receives a DRUM message through a user-defined remote procedure call
2. Add code to your host application to process the user-defined remote procedure call

Each of these steps is described in the following sections.

Implement the ProcessDrum() Function

Within the **MicroServer.nc** file, locate the **IsiMsgHandler()** function. After each message has been approved, and you have confirmed that **preemptionMode** is **FALSE**, call your DRUM dispatcher. This function determines whether the

newly arrived ISI message is a DRUM message, and forwards the DRUM message to the host application, using a user-defined unacknowledged remote procedure call.

The **ProcessDrum()** function is defined to return **FALSE** so that it can easily be inserted into the **IsiMsgHandler()** routine.

```
boolean ProcessDrum(void) {
  IsiMessage message;

  memcpy(&message, msg_in.data, sizeof(IsiMessage));
  if (message.Header.Code == isiDrum ||
      message.Header.Code == isiDrumEx) {
    (void)IsiRpc(LicIsiUserCommand|LicIsiNoAck, 0, 0,
        &message.Msg.Drum, sizeof(IsiDrum));
  }
  return FALSE;
}

// IsiMsgHandler() is a utility function used by the
// ShortStack Micro Server core to identify and process ISI
// messages.  This function returns true if the message was
// handled by this function.

extern boolean shortStackInPreempt(void);

boolean IsiMsgHandler(void) {
  boolean result, preemptionMode;

  result = FALSE;
  preemptionMode = shortStackInPreempt();

  switch(isiType) {
#ifdef  SS_SUPPORT_ISI_S
    case isiTypeS:
      result = IsiApproveMsg() && (preemptionMode ||
          ProcessDrum() || !IsiProcessMsgS());
      break;
#endif  //  SS_SUPPORT_ISI_S
#ifdef  SS_SUPPORT_ISI_DA
    case isiTypeDa:
      result = IsiApproveMsg() && (preemptionMode ||
          ProcessDrum() || !IsiProcessMsgDa());
      break;
#endif  //  SS_SUPPORT_ISI_DA
#ifdef  SS_SUPPORT_ISI_DAS
    case isiTypeDas:
      result = IsiApproveMsgDas() && (preemptionMode ||
          ProcessDrum() || !IsiProcessMsgDas());
      break;
#endif  //  SS_SUPPORT_ISI_DAS
  }
  return result;
}
#pragma ignore_notused IsiMsgHandler
```

Process Your User-Defined RPC

The Micro Server's **IsiRpc()** function supplies DRUM messages to your host application, which must evaluate these DRUM messages to maintain an accurate list of devices that are available on the ISI network at any given time. You add code to your host application to process this information by extending the **IsiUserCommand()** callback handler function in the **ShortStackIsiHandlers.c** file.

A typical use for this callback is to update an advanced device's graphical user interface with a representation of all devices that are located on the same ISI network. The same device table information can also be used to implement advanced connection scenarios with ISI.

Recovering Connections

A connection controller can display connections that it created but that are no longer in its database, and it can display connections that it did not create. To recover connections, a connection controller must first discover all the devices in the network, as described in *Discovering Devices* on page 226. To recover the connections, the controller uses the read connection table request (RDCT) message, which allows it to read a device's connection table over the network. Support for this message is required for devices that support controlled enrollment, and is optional for other devices.

The RDCT message includes optional host and member assembly fields that specify which connection table entries are requested:

- If the host and member assembly fields are not supported by the device, or are both set to 0xFF, the connection table entry indicated by the index is requested.
- If the host and member assembly fields are supported by the device, and the host or member field is not 0xFF, the index provided is the starting index. The first matching connection table entry is returned, if any.
- If both host and member fields are set to a value different from 0xFF, connection table entries are returned that match either the host or the member fields, if any.

This message allows a connection controller to read the entire connection table, or to read the table selectively to provide quick answers to questions like "is assembly *Z* on device *X* connected, and is it the host of the connection?"

If the requested data is available, the response to an RDCT message is a read connection table success (RDCS) message. This message contains the requested connection table index and data. If the connection table index does not exist, or if the requested assemblies do not exist, the response is a read connection table failure (RDCF) message.

A connection controller can determine if a device does not support the optional host and member assembly fields by comparing the assembly numbers in the read response to the requested assembly number, or by receiving an RDCF message that indicates a failed read. If a device does not support the host and member assembly fields, the connection controller must read every entry in the connection table individually. Reading every entry has minimal impact for devices with one or two connection table entries, but increases network traffic for devices with many connection table entries.

You can implement much of the code for ISI connection recovery either within your custom Micro Server or in your host application.

The following sections describe example implementations for supporting connection recovery. The first example shows a custom Micro Server implementation, where the Micro Server recovers the ISI connections and relays the results to the host application. The second example shows a host-based implementation.

Example 1: Custom Micro Server Implementation

The following connection controller example uses code implemented within a custom Micro Server to recover all the connections from a device.

Add the following code to the **MicroServer.nc** file or add it to a separate file (perhaps called **ConnectionRecovery.c**) that you reference (**#include**) from **MicroServer.nc**.

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

#define RETRY_COUNT 3
#define ENCODED_TX_TIMER 11 // 768ms
#define ENCODED_RPT_TIMER 2
#define PRIMARY_DOMAIN 0

// This structure holds information required while reading
// a remote device's connection table
struct {
  unsigned neuronId[NEURON_ID_LEN];
  unsigned index;
} recoveryJob;

// Issue one read connection table request using the global
// recoveryJob variable for destination address and current
// connection table index information. Increment the index
// kept in that global variable.
void RequestConnectionTable(void) {
  IsiMessage request;
  msg_out_addr destination;

  request.Header.Code = isiRdct;
  request.Msg.Rdct.Index = recoveryJob.index++;
  request.Msg.Rdct.Host = request.Msg.Rdct.Member =
        ISI_NO_ASSEMBLY;

  destination.nrnid.type = NEURON_ID;
  destination.nrnid.domain = PRIMARY_DOMAIN;
  destination.nrnid.rpt_timer = ENCODED_RPT_TIMER;
  destination.nrnid.subnet = 0;
  destination.nrnid.retry = RETRY_COUNT;
  destination.nrnid.tx_timer = ENCODED_TX_TIMER;
  memcpy(destination.nrnid.nid, recoveryJob.neuronId,
        NEURON_ID_LEN);
```

```
  IsiMsgSend(&request,sizeof(IsiMessageHeader)
        +sizeof(IsiRdct), REQUEST, &destination);
}

// Handle receipt of incoming responses. This example
// focuses on isiRdcs and isiRdcf responses.
boolean processRdc(void) {
  boolean processed;
  IsiMessage response;

  processed = FALSE;

  if (resp_in.code == isiApplicationMessageCode) {
    // This is an ISI response
    memcpy(&response, resp_in.data, resp_in.len);
    if (response.Header.Code == isiRdcf) {
    // The remote device rejected our request, probably
    // because we have queried all available connection
    // table entries already (bad index). Notify the user
    // interface, if needed.
    ...
    processed = TRUE;
  } else if (response.Header.Code == isiRdcs) {
    // The remote device replied to our request with the
    // connection table entry requested, in
    // response.Msg.Rdcs. Notify the UI and/or process
    // this data further, as needed by the application:
    (void)IsiRpc(LicIsiUserCommand|LicIsiNoAck, ....);

    // Because we received a positive response, let's try
    // for the next index
    RequestConnectionTable();
    processed = TRUE;
  }
  return processed;
}
```

In the **processRdc()** function, use the **IsiRpc()** function to notify your host application of any results. If you have already used the **IsiRpc()** function with the **LicIsiUserCommand** code for device discovery, use the first numerical parameter to this function to specify a sub-command so that your host application can correctly interpret the data delivered.

When you notify the host application about a connection recovery, you also have to include information about the remote device, the connection table index, and the remote connection table record. Add that information to a structure (that you define) that is shared between your host application and your custom Micro Server. The call to the **IsiRpc()** function should include the data within that structure to the host application.

The **processRdc()** function returns **TRUE** to allow for simple integration within the Micro Server code, as shown below.

```
// Initiate the process of reading a remote device's
// connection table.  The function kick-starts the process,
// where the majority of the work is done in the processRdc
// function. Calling this function before the previous
// connection table read job completes causes the previous
```

```
// job to abort, and the new one to start
void ReadRemoteConnectionTable(const unsigned*
      remoteNeuronId) {
  memcpy(recoveryJob.neuronId, remoteNeuronId,
      NEURON_ID_LEN);
  recoveryJob.index = 0;
  RequestConnectionTable();
}
```

Most likely, you call the **ReadRemoteConnectionTable()** function from within your code that implements device discovery, either when device discovery is complete or whenever a new device is discovered.

Finally, within the **IsiRespHandler()** function in the **Micro Server.nc** file, add a call to the **processRdc()** function.

```
boolean IsiRespHandler(void) {
  boolean processed;
  processed = processRdc();

#ifdef  SS_SUPPORT_ISI_DAS
  return processed || (isiType == isiTypeDas &&
    !IsiProcessResponse());
#else
  return processed;
#endif  // SS_SUPPORT_ISI_DAS
}
```

Example 2: Host Implementation

You can use the standard ShortStack LonTalk Compact API to implement ISI connection recovery within your host application. If your application has knowledge of other ISI devices within the same network, for example as a result of device discovery, you can issue RDCT requests using the standard **LonSendMsg()** API function, using the remote device's unique ID (Neuron ID) or its current subnet and node ID for addressing. See the *Interoperable Self-Installation Protocol Specification* for more information about the RDCT, RDCS, and RDCF message codes and formats.

One of the parameters that the **LonSendMsg()** function requires is the message data to send. In this case, the message data to send is an **IsiMessage** structure, using the **isiRdct** command and the RDCT data block. To send this message, you need to port the **IsiMessage** structure, and fill in the RDCT data block and ISI message header, as appropriate. Then, in the **LonSendMsg()** function, use **IsiMessage &msg** instead of **LonByte *pData** for the message data.

An example for calling the **LonSendMsg()** function is shown below. The message code for ISI messages is 0x3D. The actual data to send and the remote address to send it to are dependent on the application.

```
LonBool msgPriority = FALSE;
LonBool msgAuth = FALSE;
LonByte msgCode = 0x3d;

IsiMessage msg;
msg.Header = ...
msg.Rdct = ...
```

```
LonSendUniqueId remoteAdr;
remoteAdr.Type = LonAddressNeuronId;
remoteAdr.... = ...

LonApiError msgResp;

msgResp = LonSendMsg(LonMtIndexMyTag, msgPriority,
    LonServiceType.LonServiceRequest, msgAuth,
    (LonSendAddress*)&remoteAdr, msgCode, &msg,
    sizeof(IsiMessageHeader)+sizeof(IsiRdct));

if (msgResp != LonApiNoError) {
  /* do something about the error */
}
```

In this case, the **IsiRespHandler()** function that runs on the Micro Server will not recognize the response, or pass it to your **LonResponseArrived()** callback handler function, implemented in **ShortStackHandlers.c**.

Deinstalling a Device

You can deinstall a device to remove all network configuration data, including the domain addresses, network addresses, and connection configurations. For devices that do not provide direct connection removal, this is the only way to remove a device from a connection. You can use this procedure to re-enable self-installation for an ISI device that was installed in a managed network. You can also use this procedure to return a device to a known state. You can deinstall a device to move it from a managed network to a self-installed network, or to move a self-installed device to a new self-installed network. All ISI devices must support deinstallation.

To deinstall a device, set the **SCPTnwrkCnfg** configuration property to **CFG_LOCAL** to enable self-installation and then call the **IsiReturnToFactoryDefaults()** function. You typically deinstall a device in response to an explicit user action. For example, the user might be required to press and hold the service pin for five seconds to trigger deinstallation.

The **IsiReturnToFactoryDefaults()** function clears and reinitializes all system tables, stops the ISI engine, and resets the Micro Server. Because of the Micro Server reset, the call to the **IsiReturnToFactoryDefaults()** function never returns when it runs on the Micro Server. When it runs in the host application, the ISI host API's implementation of **IsiReturnToFactoryDefaults()** does return to the caller, but the Micro Server can take up to one minute to re-initialize. When initialization is complete, the Micro Server resets and establishes communications with the host application.

Example: The following example deinstalls a device after the service pin is held for a long period.

```
void LonServicePinHeld(void) {
   nciNetConfig = CFG_LOCAL;
   IsiReturnToFactoryDefaults();
}
```

Comparing ISI for ShortStack and Neuron C

The ShortStack ISI implementation differs from the Neuron C ISI implementation in the following ways:

- A ShortStack ISI device must have at least two application output buffers.
- The ISI types and definitions follow the ShortStack rules for portable types (see **ShortStackIsiTypes.h**), and are binary compatible with the equivalent data structures defined in **isi.h**.
- All ShortStack ISI API functions return a **LonApiError** code for success or failure of the remote procedure call request. This code does not indicate successful completion of the requested function; see **IsiApiComplete()** for more information.
- The **IsiApiComplete()** callback handler function is supported with the ShortStack ISI API to provide success or failure completion codes, and possible results, of previous ISI API calls. A negative completion code indicates that the function could not be called, either at that time or within the current context. The ISI operation itself signals its success or failure through state changes, indicated with the **IsiUpdateUserInterface()** callback handler function (as in the Neuron C implementation).
- Most ISI callback handler functions are synchronous. That is, they cannot return to their caller until the return value is known. In many cases, the ISI function requires interaction with the host processor. While waiting for a function call to complete, the Micro Server can handle only one ISI request from the host processor. Similarly, all ISI requests from the host are also synchronous. That is, the host waits for a response to an ISI request before it can issue another one.
- Predicates are synchronous in the Neuron C implementation, but are necessarily asynchronous in the ShortStack ISI API. Affected predicates are: **IsiQueryIsConnected()**, **IsiQueryImplementationVersion()**, **IsiQueryProtocolVersion()**, **IsiQueryIsRunning()**, and **IsiQueryIsBecomingHost()**. The predicates' results are delivered asynchronously through: **IsiIsConnectedReceived()**, **IsiImplementationVersionReceived()**, **IsiProtocolVersionReceived()**, **IsiIsRunningReceived()**, and **IsiIsBecomingHostReceived()**.
- The following functions and callback handler functions that are included with the Neuron C implementation are not supported by the ShortStack ISI API: **IsiMsgDeliver()**, **IsiMsgSend()**, **IsiUpdateDiagnostics()**, **IsiGetAlias()**, **IsiSetAlias()**, **IsiGetNv()**, **IsiSetNv()**, **IsiSetDomain()**, **IsiGetFreeAliasCount()**, and **IsiIsConfiguredOnline()**.
- The following functions and callback handler functions that are included with the Neuron C implementation are supported by (but not exposed to) the ShortStack ISI API: **IsiStart*()**, **IsiTick*()**, **IsiProcessMsg*()**, and **IsiApproveMsg*()**. Wrapper functions and ShortStack-specific handler functions are provided in the **MicroServer.nc** file; you can edit these handler functions to allow a custom Micro Server to intercept ISI messages, if needed.

- The **IsiPreStart()** function is not supported because the Micro Server automatically handles calls to **IsiPreStart()** as needed.
- The **IsiCancelAcquisitionDas()** function is not supported. Use the **IsiCancelAcquisition()** function when calling from your host application, even when operating an ISI-DAS device.
- Callback forwardees are only available to callback overrides that are local to the Micro Server. Callback overrides that reside on the host processor must provide a complete implementation, and cannot fall back to the forwardee.
- You should not normally call the ISI API from within an ISI callback override. With the ShortStack ISI API, you can call exactly one ISI API function from within a callback override that runs on the host processor. The API call is buffered, and runs after the callback itself completes. The Micro Server rejects subsequent API calls from within the callback override, and returns a negative response.

Because most of the ShortStack ISI API is asynchronous, your host application typically receives control from a ShortStack host API function while the Micro Server is still busy executing the related action. While most ISI operations complete quickly, some operations can take a significant amount of time. For example, calls to the **IsiCreateEnrollment()** or **IsiExtendEnrollment()** functions on an enrollment host for a connection that involves a large number of network variables are time-consuming operations.

The Micro Server can appear unresponsive while performing the requested task. However, most ISI operations include a series of callbacks, including remote procedure calls to callback overrides implemented within your host application. The Micro Server processes most of its normal tasks in this state, and honors incoming and outgoing message queues.

However, you can monitor the **IsiApiComplete()** callback handler function (implemented in **ShortStackIsiHandlers.c**) to determine completion of the more complex ISI operations, and suspend network communications until the task completes. Failure to suspend network operations in this case could cause inconsistent results.

As an example of such an inconsistency, consider the case of a very wide connection. The enrollment host initiates the implementation of a network variable connection including, for example, ten output network variables. While the Micro Server performs all the necessary steps to implement that connection, the host application could enqueue ten network variable updates in an attempt to inform the newly connected destination devices of the output network variables' current values.

If the Micro Server has not yet completed the implementation of the connection (as signalled through the **IsiApiComplete()** callback handler function), some of the related network variables will not yet be bound at the time that the host application attempts to send the network variable update messages. Only devices that are already connected will receive the update messages, and update messages for output network variables that are not yet connected will not be sent on the network.

Any network device must be designed to handle partial and transient failure. Thus, the remote device connected to these output network variables should not rely on updates to network variables to occur within a specific time or order.

However, a robust ShortStack ISI application should monitor the completion of the operation, and avoid producing inconsistent and potentially confusing data.

12

Custom Micro Servers

This chapter describes custom Micro Servers and how to create and use one. Using a custom Micro Server allows you to modify the operating parameters for the Micro Server. You need either the NodeBuilder Development Tool or the Mini kit to create a custom Micro Server.

Overview

The ShortStack Developer's Kit includes standard Micro Server firmware images for 3120 and 3150 Smart Transceivers running on TP/FT-10 or PL-20 channels, PL 3170 Smart Transceivers, and FT 5000 Smart Transceivers, in some common hardware configurations (see **Table 5** in *Standard ShortStack Micro Server Firmware Images* on page 22 for a list of the standard Micro Server images).

If your ShortStack device needs to support different operating parameters from those provided by the standard Micro Server images, you can create a *custom* Micro Server for the device. See *Custom Micro Server Benefits and Restrictions* for a description of the kinds of parameters that you can modify.

Because a ShortStack Micro Server can run only on an Echelon Smart Transceiver or the Echelon Neuron 5000 Processor, the modifications that you make to the operating parameters for a custom Micro Server must be supported by the Smart Transceiver or Neuron Processor that your device uses.

To create a custom Micro Server, you must have one of the following tools so that you can compile the custom image:

- NodeBuilder Development Tool 3.13 or later for Series 3100 Micro Servers
  NodeBuilder FX Development Tool or later for Series 5000 Micro Servers
  (see www.echelon.com/nodebuilder for more information)
- Mini EVK Evaluation Kit 1.02 or later for Series 3100 Micro Servers
  Mini FX Evaluation Kit or later for Series 5000 Micro Servers
  (see www.echelon.com/mini for more information)

If your version of the development tool does not include the Interoperable Self-Installation (ISI) protocol and current libraries, and you want to create a Micro Server that supports ISI, you will also need to get version 3.03 or later of the ISI Developer's Kit at www.echelon.com/isi.

Custom Micro Server Benefits and Restrictions

When you create a custom Micro Server, you can provide support for any of the following operating parameters:

- Custom hardware configurations, such as different clock speeds or memory maps. For example, you can support off-chip RAM for an FT 3150 or PL 3150 device, which can increase the number of buffers that the device supports. You can also support a Neuron 5000 device.
- Increased buffer counts or alternate buffer sizes for network and application buffers (within the limits of available hardware resources)
- Maximum number of network variables or network variable aliases. For example, you could support a lower maximum to optimize processing speed. However, you cannot support more than 254 network variables and 127 aliases.
- Alternate levels of support for direct memory files (DMF), including enabling or disabling DMF. If DMF is enabled, you can define the maximum size of the DMF window to customize the code and data space that is local to the Micro Server.

- Alternate levels of support for ISI and ISI network types. You can customize the implementation of many ISI callback functions, which allows you to create both general-purpose Micro Servers and application-specific Micro Servers.

When you create a custom Micro Server, there are certain operating parameters that you **cannot** control or change:

- The firmware's core algorithms or basic behavior.
- The link-layer protocol for communications between the Micro Server and the host processor.
- The Micro Server's processing for network variables or application messages. That is, you cannot provide application-specific processing within the Micro Server for network variables or application messages.
- Support for transceivers other than Echelon Smart Transceivers and the Echelon Neuron 5000 Processor. A ShortStack Micro Server can only run on an FT 3120, PL 3120, FT 3150, PL 3150, PL 3170, or FT 5000 Smart Transceiver, or the Echelon Neuron 5000 Processor. ShortStack does not support the FTXL 3190 Free Topology Transceiver.
- Capacity for more network variables, aliases, domains, or address tables than are supported by an FT 3150-based or FT 5000-based Micro Server. That is, a custom Micro Server cannot support more than 254 network variables, 127 network variable aliases, 2 domains, and 15 address table entries.

Configuring and Building a Custom Micro Server

To configure and build a custom Micro Server, you must create a project for either the NodeBuilder Development Tool or the Mini kit. This project must include the main Micro Server Neuron C application and associated source files, and the ShortStack library. The ShortStack library contains the majority of ShortStack Micro Server executable code.

Table 25 lists the files that are included with the ShortStack Developer's Kit for custom Micro Server development. These files are installed in the [*ShortStack*]**\Custom MicroServer** directory.

Note: The [*ShortStack*]**\Custom MicroServer** directory does not include a pre-built custom Micro Server development project file for either the NodeBuilder Development Tool or the Mini kit. The Mini kit does not support project files, and a NodeBuilder project file would be empty because developing a custom Micro Server requires that you make decisions about hardware templates and other project preferences during project creation.

Table 25. Files for Custom Micro Server Development

File Name	Description
ShortStack400.lib	This C library contains the majority of the Micro Server implementation.

File Name	Description
ShortStack400Isi.lib	This C library provides the same basic functionality as the **ShortStack400.lib** library, but this library also includes ISI support. Use this library when you create a custom Micro Server with ISI support. For a custom Micro Server without ISI support, use the **ShortStack400.lib** library instead.
ShortStack400CptIsi.lib	This C library provides the same basic functionality as the **ShortStack400Isi.lib** library, but with the following limitations: • ISI-DAS mode is not supported. Also, all API calls related to DAS mode are not available. • The link-layer must use the SCI protocol, and must use a 38400 bit rate. Therefore, you must use either a Series 3100 device with a 10 MHz external clock or a Series 5000 device with a 5 MHz system clock. • The local utility functions (and their callback handler functions) are not available. See *Local Utility Functions* on page 294 for more information about these functions. Use this library when you create a custom Micro Server with ISI support for a PL 3170 Smart Transceiver, or other resource-constrained device.
MicroServer.nc	This file is the main Neuron C source file for developing a custom Micro Server. Although you can edit this file, you should not need to edit it unless you implement modified ISI behavior locally in your Micro Server.
MicroServer.h	This header file adjusts the features and capabilities of the custom Micro Server. This file contains numerous compiler **#pragma** directives and macro definitions (with descriptive comments to describe their functions), such as: • Compiler directives to set application and network buffer counts and sizes • Compiler directives to set the size of the receive transaction database • Compiler directives to set the maximum number of network variables (0..254), aliases (0..127), address table entries (1..15), and domain table entries (1..2) • Macros for conditional compilation This file includes all of the preferences for a custom Micro Server that you might need to modify, except those included in the **ShortStackIsiHandlers.h** file.

File Name	Description
ShortStackIsiHandlers.h	This header file adjusts the implementation details for the various ISI callback handler functions. You need this file only if your custom Micro Server supports ISI.
MicroServerIsiHandlers.c	This file contains the override callback handler function implementations for ISI support. You might need to edit this file for a custom Micro Server to match the changes you make to the **ShortStackIsiHandlers.h** file. You need this file only if your custom Micro Server supports ISI.

Overview of Custom Micro Server Development

A custom Micro Server can include or exclude support for the ISI protocol. A Micro Server that includes support for the ISI protocol does not necessarily need to use the ISI protocol, but to use the ISI protocol through the ShortStack ISI API, the Micro Server must support the ISI protocol. Applications that are designed to work with a variety of Micro Servers can determine the level of ISI support needed by inspecting the Micro Server's uplink reset notification; see *Handling Reset Events* on page 186.

A Micro Server that does not include support for the ISI protocol requires less space and can support some of the more resource-limited hardware platforms. However, if your target hardware provides sufficient resources, you should generally include support for the ISI protocol within your custom Micro Server, even if you do not immediately plan to use ISI. If the Micro Server supports the ISI protocol, you have the flexibility to add ISI support to your host application at a later time, without requiring an update to your Micro Server firmware image. The processing overhead for the ISI protocol within the Micro Server is minimal if the ISI processing engine is not running (which is its default state).

The process of creating a custom Micro Server without ISI support is simpler than creating one with ISI support.

The general process of creating a custom Micro Server involves the following tasks:

1. Copy the files in the [*ShortStack*]**Custom MicroServer** directory to a project directory for your development tool (NodeBuilder or Mini kit).
2. Edit the **MicroServer.h** file to define your custom Micro Server's operating parameters.
3. Edit the **MicroServer.nc** file as necessary. Generally, you should not need to edit this file, unless you implement modified ISI behavior locally within your Micro Server.
4. For a Micro Server that supports ISI, edit the **MicroServerIsiHandlers.c** file and **ShortStackIsiHandlers.h** files as necessary.

5. Compile the project and link with the **ShortStack400.lib**, **ShortStack400Isi.lib**, or **ShortStack400CptIsi.lib** library. For a Micro Server that supports ISI, you also link the project with the appropriate ISI library, such as the **IsiFull.lib** or **IsiCompactS.lib** library.

The generated image and interface files define your custom Micro Server. The image files can be loaded into an appropriate Smart Transceiver, as described in *Preparing the ShortStack Micro Server* on page 31.

The following sections describe the process for creating a custom Micro Server in more detail.

Creating a Custom Micro Server without ISI Support

Figure 74 shows the files that are required to create a custom Micro Server that does not support the ISI protocol. You edit the **MicroServer.h** and **MicroServer.nc** files, and compile and link the project with the **ShortStack400.lib** library to create your custom Micro Server.

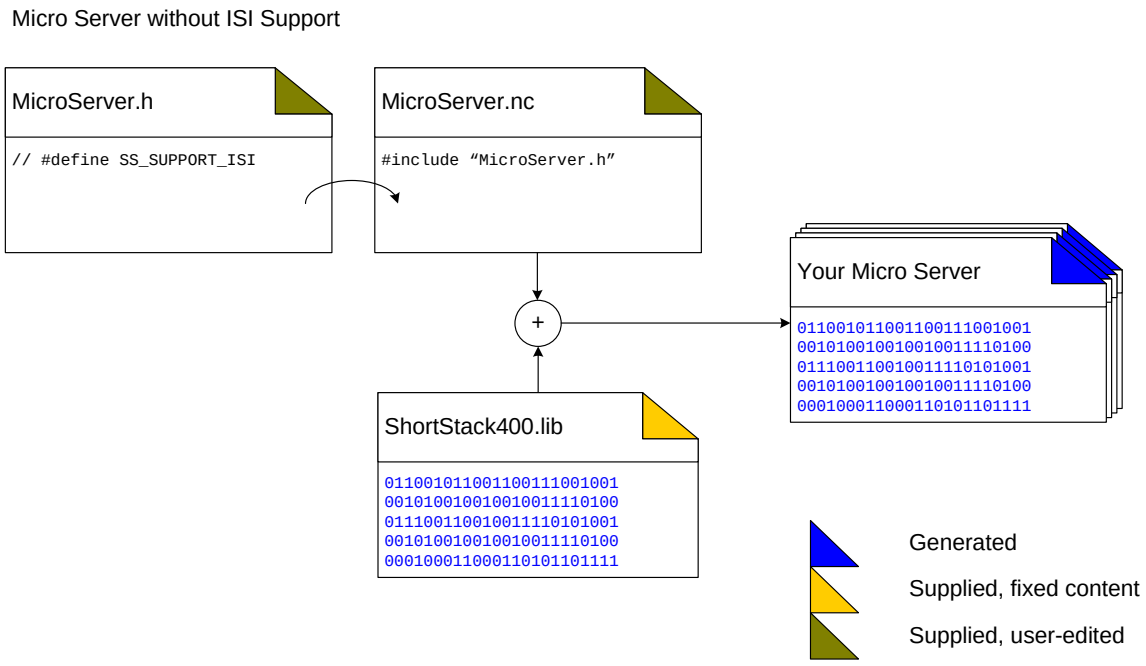

Figure 74. Files for Creating a Custom Micro Server without ISI Support

To configure and build a custom Micro Server without ISI support, perform the following tasks:

1. Create a NodeBuilder or Mini kit project, using the files described in **Table 25** on page 243.

   For the NodeBuilder tool:

   - Expand the **Device Templates** folder in the Workspace window, right-click the **Release** target folder (debugging the ShortStack firmware is not supported, so you cannot use the **Development** target), and select **Settings** to open the NodeBuilder Device Template Target Properties dialog.

- Select the **Linker** tab. Select **Generate symbol file**.
- Also from the **Linker** tab, you can optionally select **Generate map file** and select **Verbose.** A map file is optional, but recommended.
- Select the **Exporter** tab. Select **Automatic** for boot ID generation. Also select **Checksum all code**. For the reboot options, select **Communications Parameters** from the **Category** dropdown list box to select what should be rebooted, and select **Type/rate mismatch** to specify when a reboot should occur. However, do not enable rebooting of communication parameters on communication parameter mismatch for Micro Servers that use a PL 3120, PL 3150, or PL 3170 Smart Transceiver, unless you are certain that the optional features of the PL-20 transceiver will not change (such as CENELEC mode or low-power mode).
- If you use an off-chip flash memory part for the ShortStack and system firmware, do not enable rebooting the EEPROM, and do not enable rebooting on a fatal application error. If you are using a ROM (PROM or EPROM) part for the ShortStack and system firmware, you can enable these reboot options to allow possible recovery in the event of a fatal error.
- Select the **Configuration** tab. Ensure that **Export configured** is **not** selected. The option to export a device with a pre-defined configuration does not apply to a ShortStack Micro Server.

- Click **OK** to save the settings and close the NodeBuilder Device Template Target Properties dialog.

2. Specify an appropriate program ID. The program ID is not exposed to the network, because the Micro Server remains in quiet mode until the application initialization (which includes the application's program ID) is complete, but a mismatching channel type identifier might trigger warnings when using your Micro Server with the LonTalk Interface Developer utility.

   For the NodeBuilder tool, right-click the device template and select **Settings** to open the NodeBuilder Device Template Properties dialog. From the **Program ID** tab, specify an appropriate program ID.

   For the Mini kit application, click **Calculate** within the Standard Program Identifier area to open the LonMark Standard Program ID Calculator to specify the program ID.

3. Specify your target hardware correctly:

   - Always build your Micro Server for the correct clock speed. If your hardware supports multiple clock rates, build one Micro Server for each. Mismatching clock rates can cause problems during the initial link-layer connection.
   - Always build your Micro Server for the correct transceiver family. If your hardware supports both TP/FT-10 and PL-20 power line transceivers, build one Micro Server for each. Within each

transceiver family, the exact details can be configured during ShortStack application initialization.

- Select the memory map that meets your direct memory files requirements. See *Supporting Direct Memory Files* on page 253 for more information about direct memory files.

4. Review the preferences specified in the **MicroServer.h** file. See *Managing Memory* on page 254 for information about configuring the Micro Server's resources within the **MicroServer.h** file.
5. Build the Micro Server. Link your project with the **ShortStack400.lib** library.

Be sure to keep the following files for the custom Micro Server:

- The Micro Server's device interface file (XIF file extension)
- The Micro Server's symbol table (SYM file extension)
- The Micro Server's application image files (APB, NDL, NEI, NFI, NXE, NME, or NMF file extensions)

Important: All Micro Server files must share the same base name, which can be any valid set of characters. However, to avoid confusion with standard Micro Server images, do not use names that start with `SS400_` or similar pattern.

Creating a Custom Micro Server with ISI Support

You can create a custom Micro Server that supports the ISI protocol. However, a custom Micro Server with ISI support can run only on an FT 3150, PL 3150, PL 3170, or FT 5000 Smart Transceiver. An FT 3120 or PL 3120 Smart Transceiver does not have sufficient memory to accommodate a Micro Server with ISI support.

For an ISI device that is not a domain address server, you can use a standard Micro Server with an FT 3150, PL 3150, PL 3170, or FT 5000 Smart Transceiver. For a domain address server, you must create a custom Micro Server. A DAS-enabled Micro Server must run on hardware with at least 512 bytes of additional, off-chip RAM (or extended RAM for FT 5000 Smart Transceivers). For more flexibility, supply at least 2 KB RAM (or extended RAM for FT 5000 Smart Transceivers) for a DAS Micro Server to provide sufficient buffer configurations.

The process for creating a custom Micro Server that supports ISI is similar to the process described in *Creating a Custom Micro Server without ISI Support* on page 246, but includes additional files and additional considerations. **Figure 75** on page 249 shows the files that are required to create a custom Micro Server that supports the ISI protocol.

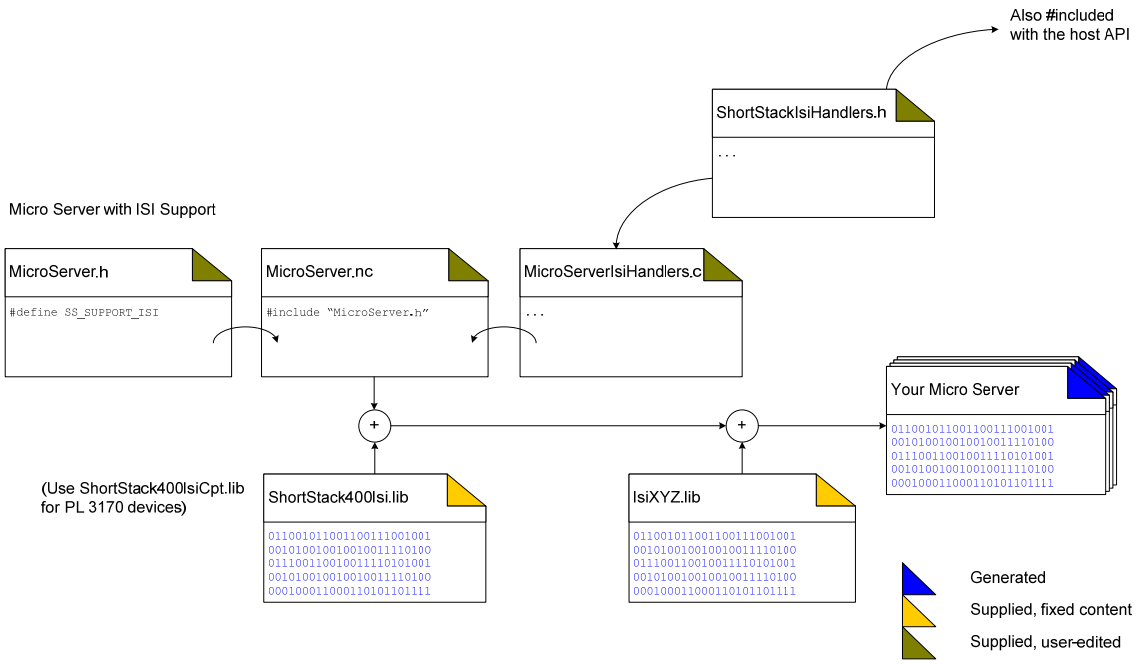

Figure 75. Files for Creating a Custom Micro Server with ISI Support

You edit the **MicroServer.h, MicroServer.nc**, **ShortStackHandlers.h**, and **MicroServerIsiHandlers.c** files, and compile and link the project with the **ShortStack400Isi.lib** (or **ShortStack400IsiCpt.lib**) library and an appropriate ISI library (typically, **IsiFull.lib**) to create your custom Micro Server. Be sure to select an ISI library that supports all of the functionality that your device requires; for example, if your device requires that automatic enrollment be able to replace connections, do not select a small ISI library that does not support connection removal.

To configure and build a custom Micro Server with ISI support, perform the following tasks:

1. Create a NodeBuilder or Mini kit project, using the files described in **Table 25** on page 243.

   For the NodeBuilder tool:

   - Expand the **Device Templates** folder in the Workspace window, and right-click one of the target folders (such as **Development** or **Release**), and select **Settings** to open the NodeBuilder Device Template Target Properties dialog. In this dialog, select the **Linker** tab and select **Generate symbol file.** Click **OK** to save the setting and close the dialog.
   - Also in the Linker tab of the NodeBuilder Device Template Target Properties dialog, you can optionally select **Generate map file.** A map file is optional, but recommended.
   - For Micro Servers that support authentication, you should export a configured custom Micro Server, including pre-defined authentication keys. In the NodeBuilder Device Template Target Properties dialog, select the **Configuration** tab and select **Export configured.** See the

NodeBuilder FX User's Guide for information about exporting a configuration.

2. Specify your target hardware correctly:
   - Always build your Micro Server for the correct clock speed. If your hardware supports multiple clock rates, build one Micro Server for each. Mismatching clock rates can cause problems during the initial link-layer connection.
   - Always build your Micro Server for the correct transceiver family. If your hardware supports both TP/FT-10 and PL-20 transceivers, build one Micro Server for each. Within each transceiver family, the exact details can be configured during the ShortStack initialization phase.
   - Select the memory map to meet your direct memory file requirements. See *Supporting Direct Memory Files* on page 253 for more information about direct memory files.
3. Review the preferences in the **MicroServer.h** file. In particular, you must uncomment the **#define SS_SUPPORT_ISI** macro. See *Configuring MicroServer.h for ISI* on page 251 for more information.
4. Review the preferences in the **ShortStackIsiHandlers.h** file.
5. If you implement one or more ISI callback handler functions local to the Micro Server, review and edit the callback handler functions in the **MicroServer.nc** file, as needed.
6. Build the Micro Server:
   - Link your project with the **ShortStack400Isi.lib** (or **ShortStack400IsiCpt.lib**) library.
   - Link your project with a suitable standard ISI library, such as **IsiFull.lib** or **IsiCompactDaHb.lib**. If resources permit, use the **IsiFull.lib** library.

A custom Micro Server that supports the ISI protocol can be used either with an application that supports ISI or with one that does not. If the application does not support ISI, it simply does not start the ISI engine (that is, it does not call the **IsiStart()** API function). There is minimal performance penalty for a Micro Server to support a disabled ISI engine.

Be sure to keep the following files for the custom Micro Server:

- The Micro Server's device interface file (XIF file extension)
- The Micro Server's symbol table (SYM file extension)
- The Micro Server's application image files (APB, NDL, NEI, NFI, NXE, NME, or NMF file extensions)
- The **ShortStackIsiHandlers.h** file, but rename it to match the Micro Server image file (be sure to keep the **.h** extension)

Important: All Micro Server files must share the same base name, which can be any valid set of characters. However, to avoid confusion with standard Micro Server images, do not use names that start with `SS400_` or similar pattern.

Configuring MicroServer.h for ISI

The **MicroServer.h** configuration file includes comments that describe how to use the file. The file provides five ISI-related preferences:

- The **SS_SUPPORT_ISI** macro enables ISI support.
- The **SS_SUPPORT_ISI_S** macro controls inclusion of support for an application that does not support domain acquisition.
- The **SS_SUPPORT_ISI_DA** macro controls inclusion of support for an application that supports domain acquisition.
- The **SS_SUPPORT_ISI_DAS** macro controls inclusion of support for a domain address server (DAS) application.
- The **SS_COMPACT** macro specifies that the Micro Server will use the **ShortStack400IsiCpt.lib** library, and will have the limitations described in **Table 25** on page 243.
- The **SS_CONTROLLED_ENROLLMENT** macro specifies that the Micro Server will support controlled enrolment.
- The **SS_ISI_IN_SYSTEM_IMAGE** macro indicates that the Micro Server firmware includes ISI support as part of the Smart Transceiver's system image. This macro is independent of the **SS_SUPPORT_ISI** macro, and is relevant even if ISI support is not configured.
- The **SS_5000** macro indicates that the Micro Server will be used with an FT 5000 Smart Transceiver or Neuron 5000 Processor.

Recommendation: In addition to the **SS_SUPPORT_ISI** macro, specify both the **SS_SUPPORT_ISI_S** and the **SS_SUPPORT_ISI_DA** macros to support ISI applications with or without domain acquisition. Because ISI domain address servers require additional hardware resources (primarily more RAM), specify the **SS_SUPPORT_ISI_DAS** macro only if it is needed.

See *Managing Memory* on page 254 for additional information about configuring the Micro Server's resources within the **MicroServer.h** file.

Configuring ShortStackIsiHandlers.h

For an ISI callback handler function, you can control the location of its implementation. Specifically, you can choose one of the following actions for almost every ISI callback handler function:

- Use its default implementation (delivered with the ISI library), and not override the callback handler function.

  Using the default implementation for a callback handler function is the simplest option, but provides the least customized behavior.

- Implement the callback override within a copy of the [*ShortStack*]**\Custom MicroServer\MicroServerIsiHandlers.c** file (which runs on the Micro Server).

  Implementing a callback override local to the Micro Server can provide the most responsive ISI implementation, but such a specialized Micro Server might work only with your specific ISI-enabled host application.

- Implement the callback override within a copy of the [*ShortStack*]**Api\ShortStackIsiHandlers.c** file (which is part of your host application).

  Implementing a callback override on the host allows you to create a general-purpose Micro Server, but can require more traffic across the ShortStack link layer because the Micro Server routes callbacks to the host using a simple remote procedure call protocol (ISI-RPC).

You control the location of each of the supported ISI callback handler functions in the [*ShortStack*]**Custom MicroServer\ShortStackIsiHandlers.h** file. This file includes comments that describe how to override a callback handler function, and includes recommendations for each callback handler function's location. Some callback handler functions are subject to certain restrictions, which are described in the **ShortStackIsiHandlers.h** file. For example, some callbacks have fewer choices for the location of the callback handler, and certain callback handlers form groups that must always reside in the same location.

Recommendations:

- Implement the ISI connection table local to the Micro Server. The ISI connection table is a fairly frequently accessed resource; implementing this table on the host processor can require a high number of ISI-RPC messages to access this table.
- Implement the **IsiUpdateUserInterface()** callback handler function within your host application, so that your application can synchronize its user interface with the ISI engine.

Important: The **IsiGetNvValue()** callback handler function must be overridden within the host application.

The LonTalk Interface Developer utility copies the **ShortStackIsiHandlers.h** file to your project directory only if you select a standard Micro Server from the ShortStack Micro Server Selection page. If you edit this file and re-run the utility, changes to the file are overwritten. However, if your project directory has a **ShortStackIsiHandlers.h** file that you created for a custom Micro Server, the LonTalk Interface Developer utility does not overwrite the file.

Implementing ISI in MicroServerIsiHandlers.c

The **MicroServerIsiHandlers.c** file contains implementations for the Micro Server-side ISI callback overrides. For callback overrides that run on the host, the code in the **MicroServerIsiHandlers.c** file is complete, and contains all the processing required for the remote procedure call. You must implement the override within your host application (in **ShortStackIsiHandlers.c**), but you do not need to edit the **MicroServerIsiHandlers.c** file.

For callback overrides that run on Micro Server, you typically need to provide application-specific code in the **MicroServerIsiHandlers.c** file. Only those callback functions that relate to the connection table have a meaningful default implementation (which implements an ISI connection table with 32 records).

Using a Custom Micro Server

For the LonTalk Interface Developer utility, you have two options for a ShortStack device that uses a custom Micro Server:

- With all of your Micro Server's image files in a single folder, use the **Browse** button on the utility's ShortStack Micro Server Selection page to specify your Micro Server's interface file by name.
- You can create a custom Micro Server database file, which the LonTalk Interface Developer utility reads when it displays the Micro Server image files in the **Firmware image** field of the ShortStack Micro Server Selection page.

The LonTalk Interface Developer utility reads a standard ShortStack Micro Server database file (**StdServers.xml**) to display the values for each of the supported Micro Servers. This file is in the [*ShortStack*]**\MicroServers** directory.

The ShortStack Micro Server database file contains information that is human-readable as well as machine-readable for each of the supported Micro Servers. You can view this file in any Web browser that supports the Extensible Markup Language (XML) with Extensible Stylesheet Language Transformations (XSLT), such as Windows Internet Explorer 6 or later.

You can create a custom Micro Server database file (**UserServers.xml**) in the [*ShortStack*]**\MicroServers** directory. The LonTalk Interface Developer utility can read the user database to display information about custom Micro Servers in the ShortStack Micro Server Selection window. Use the standard database file as a template for creating a user database file.

Using a custom Micro Server database file allows you to specify predefined lists of supported clock rates or transceivers, along with an additional description, so that you can select the custom Micro Server and be sure of selecting the correct operating interface for it.

Supporting Direct Memory Files

To allow a custom Micro Server to support the direct memory file (DMF) access method, you must specify the **#pragma enable_dmf** compiler directive when you create the custom Micro Server. Specify this directive, along with other preferences, in the **MicroServer.h** configuration file.

You then use the LonTalk Interface Developer utility to specify whether a specific ShortStack application that uses the custom Micro Server should enable or disable the DMF access method.

See *Using Direct Memory Files* on page 189 for information about the benefits and basic mechanics of the DMF access method.

A Micro Server can receive a memory read or write network management request that relates either to its own local memory or to non-existent memory (memory that corresponds to a gap in the Micro Server's own memory map).

When the Micro Server receives a memory read or write network management request that can be satisfied from the Micro Server's own local memory, the Micro Server responds to the request by accessing its memory. These kinds of requests allow for normal management tasks, including the loading of a revised Micro Server image over the network.

For a memory read or write request that does not relate to local memory, but instead relates to a "gap" in the hardware memory map or to an area declared as memory-mapped I/O, the Micro Server can have two responses:

- With the DMF access method disabled (or not supported), the Micro Server replies to such a request with a negative response.
- With the DMF access method enabled, these requests are relayed to the host processor. It is the responsibility of the host processor to satisfy the request, or to reply with a failure code.

To allow a custom Micro Server to use the DMF access method, you must leave an area within the Smart Transceiver's 64 KB memory space unused. You need to define your hardware memory map such that it contains an area of undeclared memory. The standard Micro Servers use the 0xA100..0xCEFF area, but you can change the size or location of this DMF window in your hardware design.

ShortStack supports only one DMF window. The Micro Server relays all memory read or write requests that cannot be satisfied locally to the host (if the DMF access method is enabled), including those relating to disjoint gaps in the memory map, but the DMF presentation and address translation provided by the LonTalk Interface Developer utility supports only one DMF window.

Important: The DMF access method requires Version 16 Neuron firmware or later, and thus is not available for current PL 3120 Smart Transceivers, which are based on Version 14 firmware. All other standard Micro Server images have this feature enabled. For custom Micro Servers, if you attempt to enable the DMF access method for a Smart Transceiver running Version 15 or earlier firmware, the Neuron C compiler issues a linker error.

Managing Memory

The LonTalk Interface Developer utility's Neuron C compiler generates four tables that affect memory usage in on-chip EEPROM within a Smart Transceiver. The ShortStack Micro Server firmware and network management tools use these tables to define the network configuration for a device. The four tables include:

- The address table.
  By default, this table is generated at its maximum size, which is 15 entries.
- The alias table.
  This table has no default size, and you must specify a size using the **#pragma num_alias_table_entries** compiler directive. You can set the size of the alias table to zero, or any value up to 127.
- The domain table.
  By default, this table is generated at its maximum size, which is 2 entries. You should not normally change this default.
- The 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.

See the *FT 3120 / FT 3150 Smart Transceiver Data Book*, the *PL 3120 / PL 3150 / PL 3170 Power Line Smart Transceiver Data Book*, or the *Series 5000 Chip Data Book* for detailed descriptions of these tables.

Address Table

The address table contains the list of network addresses to which the device sends implicitly addressed 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 address table contains 15 entries. Each address table entry uses five bytes of on-chip EEPROM (extended RAM for a Series 5000 Micro Server). Use the following compiler directive to specify the number of address table entries:

#pragma num_addr_table_entries *nn*

where *nn* can be any value from 0 to 15.

Recommendation: Whenever possible, specify the maximum size of 15 entries for the address table.

Alias Table

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

The alias table has no default size, and can contain between 0 and 127 entries. Each alias entry uses four bytes of on-chip EEPROM (extended RAM for a Series 5000 Micro Server). Use the following compiler directive to specify the number of alias table entries:

#pragma num_alias_table_entries *nnn*

where *nnn* can be any value from 0 to 127 (or 0 to 62 for PL 3120 Micro Servers with Version 14 firmware). Subject to the Micro Server's preferences and hardware capabilities, it might not be possible to implement the maximum number of aliases.

Recommendation: Specify the number of entries for the alias table, within the amount of available on-chip EEPROM. The number of required entries is typically fewer than the maximum of 127. The following calculation provides a useful starting point for the alias table size, *nnn*:

nnn = 0; for nv_count = 0

nnn = 10 + (nv_count / 3); for nv_count > 0

The number of aliases defined here is fixed, and cannot be changed from the ShortStack application. You should use any special knowledge that you have about the application to set the size of the alias table appropriately. A small number of aliases can prevent you from using the device in a complex network,

but a large number of unused aliases can reduce the Micro Server's throughput and the overall device performance.

Domain Table

By default, the domain table is configured for two domains. Each domain uses 15 bytes of on-chip EEPROM (extended RAM for a Series 5000 Micro Server). The number of domain table entries is dependent on the network in which the device is installed; it is not dependent on the application.

Use the following compiler directive to specify the number of domain table entries:

#pragma num_domain_entries *n*

where *n* can be either 1 or 2.

Recommendation: Specify the maximum of 2 domain table entries. LONMARK International requires all interoperable LONWORKS devices to have two domain table entries. Reducing the size of the domain table to one entry will prevent certification.

Network Variable Configuration Table

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

The maximum size of the network variable configuration table is 254 entries, provided that there are sufficient available EEPROM resources (extended RAM resources for a Series 5000 Micro Server). Each entry uses three bytes of EEPROM (or extended RAM). You cannot change the size of this table, except by adding or deleting network variables.

You can use the following compiler directive to specify the maximum number of network variables that the Micro Server supports, which in turn, affects the size of the network variable configuration table:

#pragma set_netvar_count *nnn*

where *nnn* can be any value from 0 to 254 (or 0 to 62 for PL 3120 Micro Servers with Version 14 firmware). Subject to the Micro Server's preferences and hardware capabilities, it might not be possible to implement the maximum network variable capacity.

The actual number of network variables is set by the application. Unlike for the alias table, providing support for more network variables than are needed does not affect the device's throughput. However, the total number of network variables declared for a device does affect its overall throughput and the time that the device might require for reset; also the maximum number of network variables declared with this directive affects the amount of memory required by your custom Micro Server.

13

Converting a ShortStack 2 Application to a ShortStack FX Application

This chapter describes the steps that are required to migrate a ShortStack application that uses the ShortStack 2 API to one that uses the ShortStack FX LonTalk Compact API with a ShortStack FX Micro Server.

For your application to benefit from the new features and capabilities introduced with ShortStack 2.1 or ShortStack FX, you must upgrade your ShortStack 2 application.

Overview

Because there are a number of changes to the ShortStack FX LonTalk Compact API and link-layer protocol compared to ShortStack 2, and because there are new features provided by ShortStack FX, you cannot use an unmodified ShortStack 2 application with a ShortStack FX Micro Server. That is, you must migrate the ShortStack host application from ShortStack 2 to ShortStack FX. However, you do not need to migrate from ShortStack 2 to ShortStack 2.1 before migrating to ShortStack FX; you can migrate from ShortStack 2 directly to ShortStack FX.

Important: To complete the migration for a ShortStack 2.1 host application and use a ShortStack FX Micro Server, you need only run the LonTalk Interface Developer utility from the ShortStack FX Developer's Kit and recompile the application. No changes to the host application or link-layer driver are required. However, the Micro Server attributes must be the same (except for its version number), that is, it must use the same clock setting, transceiver type, and so on.

If you are migrating an existing ShortStack device from a Series 3100 Micro Server to a Series 5000 Micro Server, see *Application Migration: Series 3100 to Series 5000* on page 195 for additional considerations.

A typical migration from ShortStack 2 to ShortStack FX consists of the following steps:

1. Save your original ShortStack 2 design for reference.
2. Update your device's Micro Server to use a ShortStack FX standard or custom Micro Server.

   Important: Ensure that you load the appropriate Micro Server image for your device's Smart Transceiver. For a standard Micro Server, the image files are in the [*ShortStack*]**MicroServers** directory.
3. Run the LonTalk Interface Developer utility to generate the ShortStack FX application framework files based on your existing model file and device characteristics.
4. Update the serial driver to use the ShortStack FX initialization sequence, link-layer message types, and the naming conventions.
5. Update the host API files by manually merging the new host API with the old ported API, preserving any application-specific changes.
6. Update the application code to meet ShortStack FX naming conventions and API changes.

A general estimate for the effort required to migrate a ShortStack 2 application to ShortStack FX is two to three days per application or port. This estimate does not include any additional effort that is required to support new features, but porting multiple devices that share the same or similar hardware will likely be faster after you have completed the process for the first device.

Important: If you use a ShortStack solution with a generic Neuron Chip, you can continue to use the ShortStack 2 Micro Server. Echelon does not plan to release updates or fixes to the ShortStack 2 Micro Servers. The discontinued images are available for download from www.echelon.com/downloads, in the Archived Downloads section.

Reorganization of API Files

The ShortStack FX LonTalk Compact API file structure is significantly different than the ShortStack 2 API file structure.

A ShortStack 2 application required many include files and two API implementation files. The organization of these files was simplified for ShortStack 2.1 (ShortStack FX uses the same organization), and now includes two implementation files (one for the API implementation and one for utilities used by the API implementation), and a small number of include files.

Table 26 lists the correspondences between the ShortStack 2 and ShortStack FX LonTalk Compact API files. You should incorporate all of the new ShortStack FX files into the application, and merge any application-specific changes that you made to the old API files into the new ones.

Table 26. ShortStack 2 API Files and ShortStack FX LonTalk Compact API Files

ShortStack 2 File	ShortStack FX File	Description
LonApi.c	ShortStackApi.c	Principal API implementation file
—	ShortStackInternal.c	Utilities functions used by the API.
LonApp.c	ShortStackHandlers.c	Templates for application-specific callback handler function implementations
platform.h	LonPlatform.h	Portability support
—	LonBegin.h LonEnd.h	Optional files that are typically used for processor-specific packing and byte alignment control (if necessary)
—	ShortStackTypes.h	Defines all type definitions, structures, and enumerations that are used by the API
—	ShortStackIsiHandlers.h	Optional file (ISI only) that allows control over the location of ISI callback functions
—	ShortStackIsiTypes.h ShortStackIsiApi.h ShortStackIsiApi.c ShortStackIsiInternal.c ShortStackIsiHandlers.c	Optional files (ISI only) that implement the ISI API for ShortStack

ShortStack 2 File	ShortStack FX File	Description
lonaccess.h lonaddr.h lonapp.h lonerr.h lonmgmt.h lonmodel.h lonmsg.h lonopts.h lonsicb.h lonstate.h	—	Now integrated into ShortStackApi.h and ShortStackTypes.h
filedir.h	—	Now integrated into ShortStackDev.h
LonDev.c LonDev.h	ShortStackDev.c ShortStackDev.h	Generated by LonTalk Interface Developer utility (ShortStack Wizard for ShortStack 2)

Support for Added Features

Several features were added to ShortStack 2.1, and new ones have been added for ShortStack FX. See *What's New for ShortStack 2.1* on page iv and *What's New for ShortStack FX* on page iii for a brief summary of these new features. Most of these features required changes to the ShortStack LonTalk Compact API, its implementation, and the link layer protocol.

New API Naming Conventions

The names of types and of functions that are used with the API were changed for ShortStack 2.1 (ShortStack FX uses the same names). This name change serves three primary goals:

- The new names are more consistent and aligned with current recommendations and conventions, which makes the API more consistent and easier to learn.
- The new names use name prefixes that provide unique names, rather than relying on explicit namespaces (which are not provided by the ANSI C language). These prefixes make it easier to integrate the ShortStack LonTalk Compact API and the ShortStack application framework with your application and environment.
- The new names are consistent with the LonTalk API used by the FTXL 3190 Free Topology Transceiver chip, which simplifies sharing code between applications written for a ShortStack Micro Server and applications written for the FTXL Transceiver.

When migrating your existing application to ShortStack FX, you must change these names, wherever they are used within your application, to meet the new guidelines.

In the source code comments for the ShortStack LonTalk Compact API, most of the functions and data structures provide the ShortStack 2 name so that you can search the API source code for the ShortStack 2 name and find the equivalent ShortStack FX name. For example, if you search for *config_data_struct*, you will find *LonConfigData*.

See *ShortStack FX Naming Conventions* on page 288 for more information on the naming convention.

Improved Portability Support

Portability for the ShortStack LonTalk Compact API and ShortStack application framework has been greatly improved; see *Portability Overview* on page 110 for a description of related changes.

All types defined for use with the ShortStack LonTalk Compact API have been redefined to meet these guidelines. Thus when you migrate an application that accesses members of these types, you will likely also need to change the related code.

Recommended Migration Process

The following process is recommended to perform this migration:

1. Save your ShortStack 2 existing design, including the Micro Server image files and all other data that you might need to reproduce the device from the ShortStack 2 baseline.
2. Update your device's Micro Server to a ShortStack FX Micro Server. See *Preparing the ShortStack Micro Server* on page 31 for more information. Because the FX Micro Server uses the same pin-out as the ShortStack 2 Micro Server, no hardware changes should be required.

   Important: If your current device does not use an Echelon FT or PL Smart Transceiver, you must change your Micro Server hardware. In addition, ensure that you load the appropriate Micro Server image for your device's Smart Transceiver.
3. Run the LonTalk Interface Developer utility with a copy of your original application's model file to generate the ShortStack FX application framework files. See *Using the LonTalk Interface Developer Utility* on page 145 (or the utility's online help) for information about using the LonTalk Interface Developer utility.
4. Migrate your ShortStack 2 serial driver to a ShortStack FX serial driver. See *Modifying the Serial Driver* on page 262 for more information. If your ShortStack device uses a host processor for which a ShortStack FX example application is available, you might be able to derive your ShortStack FX driver from one of these example implementations.
5. Move code that implements your ShortStack callback functions from the ShortStack 2-based application and its **lonapp.c** file to the new **ShortStackHandlers.c** file.

Modifying the Serial Driver

Many of the changes for ShortStack FX (such as the new initialization sequence and the new link-layer messages) should not affect the serial driver. However, because of the change in naming conventions (see *ShortStack FX Naming Conventions* on page 288 or the **ShortStackApi.h** file), you do need to make at least minor changes to the serial driver, including changing the names of the functions implemented by the driver.

One change for ShortStack FX that does require a change to the serial driver is the support for more than 62 network variables. If your ShortStack application uses more than 62 network variables, the driver requires an extra handshake to process an extra two bytes of header information. See Chapter 6, *Creating a ShortStack Serial Driver*, on page 89, for more information about the serial driver.

In addition, some of the serial driver functions return success or error codes that you might need to update to comply with the FX API.

The serial driver code should follow the ShortStack FX naming conventions listed in *ShortStack FX Naming Conventions* on page 288 when defining types and variables; however, following these conventions within the driver is not required.

Example Conversion

As an example of the tasks required for a conversion of a ShortStack 2 application and serial driver to use the ShortStack FX LonTalk Compact API, this section describes a conversion for the ShortStack 2 Nios II Example Port (available on the Echelon Web site at www.echelon.com/shortstack). This example port provides a simple example for a 3120 device that uses an Altera Nios II processor. For more information about the example port, see the *ShortStack 2 Nios II Example Port User's Guide* (078-0354-01A).

To enable the ShortStack 2 Nios II Example Port to use the ShortStack FX LonTalk Compact API and feature set, you must make modifications to the following parts of the example port:

- The properties and files defined within the Nios integrated development environment (IDE)
- The serial driver
- The application, including callback handler functions

In general, for a conversion from the ShortStack 2 API to the ShortStack FX LonTalk Compact API, you should not need to modify any files that are generated by the LonTalk Interface Developer utility.

The changes described in this section are specific to the ShortStack 2 Nios II Example Port, but the changes are representative of the kinds of changes that you would make for any ShortStack 2 driver and application.

Changes within the Nios II IDE

Because ShortStack FX uses different file names and assumes a different file directory structure than the ShortStack 2 Nios II Example Port uses, you cannot leave all of the example port's files in the project's workspace. You can either

create a new workspace and copy the necessary files into it (including the files generated by the LonTalk Interface Developer utility), or exclude certain existing files from being compiled and built with the project.

To exclude files from the build, right-click the file and select **Properties** to open the Properties window. Select **C/C++ Build** from the left-hand pane. In the right-hand pane, select **Exclude from build** from the Active Resource configuration area. Click **OK** to apply the change and close the Properties window.

The files that you need to exclude from the build are all ***.c** files in the **\api** folder and the **LonDev.c** file in the **\wizard** folder. See the *ShortStack 2 Nios II Example Port User's Guide* for a description of the example port's directory structure.

Because the **LonPlatform.h** file defines the **GCC_NIOS** symbol rather than the **GCC** symbol for the GNU Compiler Collection (GCC) compiler that the Nios IDE uses, you must add the symbol to the project's properties. Right-click the application (**Application_FT** or **Application_PL**) in the Projects pane, and select **Properties** to open the Properties window. Select **C/C++ Build** from the left-hand pane. On Tool Settings tab, select **Preprocessor**. In the Defined Symbols area, click the **Add** button to add the **GCC_NIOS** symbol. You should move the new symbol to appear directly below the **ALT_DEBUG** symbol. You can leave or delete the **GCC** symbol (if defined).

Changes to the Serial Driver

The implementation of the serial driver for the ShortStack 2 Nios II Example Port is contained in the **\driver** directory. As described in **Table 27**, half of the serial driver's source files require changes for the conversion to the ShortStack FX LonTalk Compact API. The majority of the changes are in the primary file for the serial driver, **ldvintfc.c**.

Table 27. ShortStack 2 Nios II Example Port Serial Driver Files

File	Changes Required for ShortStack FX?	Description
hndshk.h	No	Function prototypes to access reset and handshake lines
ldvintfc.h	Yes	Function prototypes for the serial driver
ldvqueue.h	Yes	Data structure definitions for the receive and transmit buffers used by the serial driver
ldvsci.h	Yes	Function prototypes and data structure definitions for the lower-layer serial driver (SCI interface)
londrv.h	No	Conditional compilation definitions that control whether driver uses SCI or SPI interface
lonsystem.h	No	Definitions for literals that control compile-time options

File	Changes Required for ShortStack FX?	Description
hndshk.c	No	Functions to access reset and handshake lines
ldvintfc.c	Yes	Functions for the serial driver
ldvqueue.c	Yes	Utility functions to access buffer queues used by the driver
ldvsci.c	Yes	Lower-layer serial driver (SCI interface)
londrv.c	No	Conditional compilation definitions that control whether driver uses SCI or SPI interface

The following sections describe the changes that are needed for the conversion to the ShortStack FX LonTalk Compact API in the ShortStack 2 Nios II Example Port serial driver.

ldvintfc.h

The **ldvintfc.h** file can be deleted from the project. This file contains function prototypes for the ShortStack 2 serial driver API functions (**ldv_init()**, **ldv_flush_msgs()**, **ldv_allocate_msg()**, **ldv_put_msg()**, **ldv_put_msg_init()**, **ldv_get_msg()**, and **ldv_release_msg()**). However, the function prototypes for the equivalent ShortStack FX serial driver API functions (**LdvInit()**, **LdvFlushMsgs()**, **LdvAllocateMsg()**, **LdvPutMsg()**, **LdvPutMsgBlocking()**, **LdvGetMsg()**, **LdvReleaseMsg()**, and **LdvReset()**) are in the **ShortStackApi.h** file.

ldvqueue.h

The **ldvqueue.h** file requires the following types of changes, as listed in **Table 28**:

- Change the included header files.
- Change buffer size definitions (the size definitions correspond to the buffer sizes defined in **ShortStackDev.h**).

Table 28. Changes for ldvqueue.h

From	To
`#include "platform.h"` `#include "LonDev.h"`	`#include "ShortStackDev.h"` `#include "LonPlatform.h"`
`#define SYSRXBUFSIZE MIP_APP_INPUT_BUFSIZE` `#define SYSTXBUFSIZE MIP_APP_OUTPUT_BUFSIZE`	`#define SYSRXBUFSIZE LON_APP_INPUT_BUFSIZE` `#define SYSTXBUFSIZE LON_APP_OUTPUT_BUFSIZE`

ldvsci.h

The **ldvsci.h** file requires changes to the included header files, as listed in **Table 29**.

Table 29. Changes for ldvsci.h

From	To
`#include "platform.h"` `#include "system.h"` `#include "lonmsg.h"`	`#include "LonPlatform.h"` `#include "system.h"` `#include "ShortStackTypes.h"`

ldvintfc.c

The **ldvintfc.c** file is the primary file for the serial driver, and requires the most changes. The changes required include the following types of changes, as listed in **Table 30**:

- Change the included header files.
- Change function calls for the **ldv_*** functions (some of these functions have return values for the ShortStack FX LonTalk Compact API).
- Change references to **Bool** and **Byte** types (the FX types are defined in **LonPlatform.h**).
- Modify the **ldv_put_msg_init()** function so that it returns a value (a more correct implementation would return a meaningful return value and would include a timeout within the **SysPutMsgInit()** function; see *Add Timeout Detection* on page 272).
- Change the **TRUE** and **FALSE** return values in the **ldv_allocate_msg()** function (the FX return values are defined in **ShortStackTypes.h**).
- Change the **TRUE** and **FALSE** return values in the **ldv_get_msg()** function (the FX return values are defined in **ShortStackTypes.h**).
- Add the **LdvReset()** function.

Table 30. Changes for ldvintfc.c

From	To
`#include "platform.h"` `#include "lonmsg.h"` `#include "ldvqueue.h"` `#include "londrv.h"` `#include "ldvsci.h"`	`#include "ShortStackDev.h"` `#include "LonPlatform.h"` `#include "ShortStackTypes.h"` `#include "ldvqueue.h"` `#include "londrv.h"` `#include "ldvsci.h"`

From	To
`void ldv_flush_msgs(void)` `void ldv_init(void)` `Bool ldv_get_msg(LonSmipMsg** ppMsg)` `void ldv_release_msg(const LonSmipMsg* pMsg)` `Bool ldv_allocate_msg(LonSmipMsg** ppMsg)` `void ldv_put_msg(const LonSmipMsg* pMsg)` `void ldv_put_msg_init(const LonSmipMsg* pMsg)` —	`void **LdvFlushMsgs**(void)` `void **LdvInit**(void)` `**LonApiError LdvGetMsg**(LonSmipMsg** ppMsg)` `void **LdvReleaseMsg**(const LonSmipMsg* pMsg)` `**LonApiError LdvAllocateMsg**(LonSmipMsg** ppMsg)` `void **LdvPutMsg**(const LonSmipMsg* pMsg)` `**LonApiError LdvPutMsgBlocking**(const LonSmipMsg* pMsg)` `**void LdvReset(void)**`
`Bool` `Byte`	`**LonBool**` `**LonByte**`
`void ldv_put_msg_init(const LonSmipMsg* pMsg)` `{` `  SysPutMsgInit(pMsg);` `}`	`**LonApiError LdvPutMsgBlocking**(const LonSmipMsg* pMsg)` `{` `  SysPutMsgInit(pMsg);` `  **return LonApiNoError;**` `}`
`Bool ldv_allocate_msg (LonSmipMsg** ppMsg)` `{` `  QElement* element;` `  SysDisableInterrupts();` `  element = DeQueue(qfreeout);` `  SysEnableInterrupts();` `  if (element != NULL)` `  {` `  *ppMsg = (LonSmipMsg*)(((SysTxBuf*)element)->data);` `  return TRUE;` `  }` `  else` `  {` `  return FALSE;` `  }` `}`	`**LonApiError LdvAllocateMsg**(LonSmipMsg** ppMsg)` `{` `  QElement* element;` `  SysDisableInterrupts();` `  element = DeQueue(qfreeout);` `  SysEnableInterrupts();` `  if (element != NULL)` `  {` `  *ppMsg = (LonSmipMsg*)(((SysTxBuf*)element)->data);` `  **return LonApiNoError;**` `  }` `  else` `  {` `  **return LonApiTxBufIsFull;**` `  }` `}`

From	To
void ldv_put_msg_init (LonSmipMsg** ppMsg) { QElement* element; SysDisableInterrupts(); element = DeQueue(qincoming); SysEnableInterrupts(); if (element != NULL) { *ppMsg = (LonSmipMsg*)(((SysRxBuf*)element)->data); return TRUE; } else { return FALSE; } }	**LonApiError LdvGetMsg**(LonSmipMsg** ppMsg) { QElement* element; SysDisableInterrupts(); element = DeQueue(qincoming); SysEnableInterrupts(); if (element != NULL) { *ppMsg = (LonSmipMsg*)(((SysRxBuf*)element)->data); **return LonApiNoError;** } else { **return LonApiRxMsgNotAvailable;** } }
—	**void LdvReset(void) { SysResetSCI(); /* in LdvSci.c */ }**

ldvqueue.c

The **ldvqueue.c** file requires changes to the references to **Bool** and **Byte** types (the FX types are defined in **LonPlatform.h**), as listed in **Table 31**.

Table 31. Changes for ldvqueue.c

From	To
Bool	**LonBool**
Byte	**LonByte**

ldvsci.c

The **ldvsci.c** file requires the following types of changes, as listed in **Table 32** on page 268:

- Change the included header files.
- Change references to **Bool** and **Byte** types (the FX types are defined in **LonPlatform.h**).

Table 32. Changes for ldvsci.c

From	To
`#include "platform.h"` `#include "lonmsg.h"` `#include "ldvsci.h"` `#include "ldvqueue.h"`	`#include "`**`ShortStackDev.h`**`"` `#include "`**`LonPlatform.h`**`"` `#include "`**`ShortStackTypes.h`**`"` `#include "ldvsci.h"` `#include "ldvqueue.h"`
`Bool` `Byte`	**`LonBool`** **`LonByte`**

To support the extended link-layer header for network variables with indexes greater than 62, you need to modify the state machine within the serial driver. A simple such change would be to add an additional check to the **TX_Len** state for the presence of the extended link-layer header, as shown below.

```
/* Check to see if info byte needs to be sent */
if (G_DriverStatus.pTxMsg[G_DriverStatus.tx_nextchar-1] ==
      (LonNiNv | LON_NV_ESCAPE_SEQUENCE)) {
  AssertRTS();
  WaitForCTSLow();
  DeassertRTS();

  /* Write the first Info byte */
  IOWR_ALTERA_AVALON_UART_TXDATA(SHORTSTACK_UART_BASE,
    ((LonSicb*) ((LonSmipMsg*) G_DriverStatus.pTxMsg)
    ->Payload)->NvMessage.Index);

  /* Write the second Info byte */
  IOWR_ALTERA_AVALON_UART_TXDATA(SHORTSTACK_UART_BASE,
    0x00);

}
```

This code compares the data in the driver buffer with the **LON_NV_ESCAPE_SEQUENCE** value (defined in **ShortStackDev.h**); if the value is equal, the code performs the handshake with the Micro Server and writes the two extended link-layer bytes. If the value is not equal, the code does nothing.

See the ShortStack FX ARM7 Example Port for a more complete example of a serial driver that handles the extended link-layer header.

Changes to the Application

Changes to the application include changes to the **main.c** file and changes to the callback handler functions.

For a ShortStack 2 application, the callback handler functions were often defined in the **lonapp.c** file. For a ShortStack FX application, they are defined in the **ShortStackHandlers.c** file. In many cases, you can copy the existing callback code from **lonapp.c** to **ShortStackHandlers.c** without any changes.

The following sections describe the changes that are needed for the conversion of the **main.c** file and the callback handler functions.

main.c

The **main.c** file requires the following types of changes, as listed in **Table 33**:

- Change the included header files.
- Change the calls to the **lonInit()** and **lonEventHandler()** functions.
- Modify the call to the **lonInit()** function to handle the return value (and take appropriate action, which for the example application is simply to display the result to the IDE console).

Table 33. Changes for main.c

From	To
`#include "system.h"` `#include "platform.h"` `#include "lonapi.h"` `#include "londev.h"` `#include "hndshk.h"`	`#include "system.h"` `#include "ShortStackDev.h"` `#include "LonPlatform.h"` `#include "ShortStackApi.h"` `#include "hndshk.h"`
`lonInit();` `lonEventHandler();`	`LonInit();` `LonEventHandler();`
`printf ("Initializing LON...");` `LonInit();` `printf ("done.\n");` `printf ("You can use LonMaker to test your device now.\n");`	`printf ("Initializing LON...");` `LonApiError rc = LonInit();` `if (rc == LonApiNoError) {` `  printf ("done.\n");` `  printf ("You can use LonMaker to test your device now.\n");` `}` `else {` `  printf("Failed. RC=%d\n", rc);` `}`

Callback Handler Functions

The example application has modified code for only one callback handler function, **LonNvUpdateOccurred()**. This function processes updates to the two input network variables defined for the example device.

The **LonNvUpdateOccurred()** function in the **ShortStackHandlers.c** file requires the following types of changes, as listed in **Table 34** on page 270:

- Copy the existing code from the **lonNvUpdateOccurred()** function in **lonapp.c** to the **LonNvUpdateOccurred()** function in **ShortStackHandlers.c**.
- Ensure that the **LonNvUpdateOccurred()** function has the correct parameters (they are slightly different than the parameters for the 2.0 **lonNvUpdateOccurred()** function) – if you use the function as generated by the LonTalk Interface Developer utility, no change is necessary.

- Change the network variable index values used for the **case** statements (the index values are defined in **ShortStackDev.h**).
- When setting an attribute value for a structured output network variable, use the **LON_SET_ATTRIBUTE** macro (the macro is defined in **ShortStackTypes.h** and the attribute values are defined in **LonNvTypes.h**).
- When retrieving or setting the value for a single-valued output network variable, use the **LON_GET_SIGNED_WORD** and **LON_SET_SIGNED_WORD** macros (defined in **ShortStackTypes.h**).
- Change the call to the **lonPropagateNv()** function, the network variable index value passed to it, and its return value.

Table 34. Changes for the LonNvUpdateOccurred() Callback Handler Function

From	To
`void lonNvUpdateOccurred(const Byte nvIndex, const RcvAddrDtl* const pNvInAddr)`	`void LonNvUpdateOccurred(const unsigned index, const LonReceiveAddress* const pSourceAddress)`
`case NVIDX_nviRequest:` `case NVIDX_nviVolt:`	`case LonNvIndexNviRequest:` `case LonNvIndexNviVolt:`
`nvoStatus.invalid_id = 1;` `nvoStatus.invalid_request = 1;`	`LON_SET_ATTRIBUTE(nvoStatus, LON_INVALIDID, 1);` `LON_SET_ATTRIBUTE(nvoStatus, LON_INVALIDREQUEST, 1);`
`nvoVolt = NET_SWAB_WORD(2*NET_SWAB_WORD(nviVolt) );`	`LON_SET_SIGNED_WORD(nvoVolt, LON_GET_SIGNED_WORD(nviVolt) * 2);`
`if (lonPropagateNv(NVIDX_nvoVolt) != API_NO_ERROR)`	`if (LonPropagateNv(LonNvIndexNvoVolt) != LonApiNoError)`

Complete code for the modified **LonNvUpdateOccurred()** function is shown below.

```
void LonNvUpdateOccurred(const unsigned index, const
LonReceiveAddress* const pSourceAddress)
{
  switch (index)
  {
  case LonNvIndexNviRequest:
    nvoStatus.object_id = nviRequest.object_id;
    LON_SET_ATTRIBUTE(nvoStatus, LON_INVALIDID, 1);
    LON_SET_ATTRIBUTE(nvoStatus, LON_INVALIDREQUEST, 1);
    break;

  case LonNvIndexNviVolt:
```

```
        LON_SET_SIGNED_WORD(nvoVolt,
          LON_GET_SIGNED_WORD(nviVolt) * 2);
        if (LonPropagateNv(LonNvIndexNvoVolt) != LonApiNoError)
        {
          // Handle error here, if desired.
        }
        break;

        default:
          break;
        }
    }
```

Additional Recommended Changes

In addition to the changes to the serial driver and application described in the previous sections, this section describes a few additional, optional, changes for the ShortStack FX implementation that can improve the application.

As with the changes described for the serial driver, the LonTalk Interface Developer files, and the application, the changes described in this section are specific to the ShortStack 2 Nios II Example Port, but are representative of the kinds of changes that you might make for any ShortStack 2 driver and application.

Modify the Model File

The model file (**Sample_Node.nc**) for the ShortStack 2 Nios II Example Port includes a node object of type **SFPTnodeObject**. However, the application does not use the node object (the **LonNvUpdateOccurred()** function in the **ShortStackHandlers.c** file simply returns an invalid request).

In addition, the model file uses a controller functional block that is based on an obsolete functional profile (**SFPTcontroller**). Because the controller functional block is deprecated, the example is not compliant with current LONMARK Interoperability Guidelines, which are available at www.lonmark.org.

Recommendations:

- Either remove the node object from the model file and re-run the LonTalk Interface Developer utility, or add code to the **LonNvUpdateOccurred()** function to handle the updates to the node object.
- Replace the functional profile (**SFPTcontroller**) for the functional block with a functional profile that complies with current LONMARK interoperability guidelines. For example, change the functional profile to **SFPTclosedLoopActuator**. The functional block still defines the same two network variables, **nviVolt** and **nvoVolt**.

Add Range and Error Checking

Because the ShortStack 2 Nios II Example Port is not intended to act as a production ShortStack device, it does not perform as much range checking or error checking as a production device's application would.

For example, add range checking for updates to the **nviVolt** network variable in the **LonNvUpdateOccurred()** function to ensure that the application does not set the voltage levels to invalid or erroneous values.

```
/* Limits for nviVolt */
#define MIN_VOLT                (-32768/2)
#define MAX_VOLT                (32767/2)

…

case LonNvIndexNviVolt:
{
  /* Whenever nviVolt is updated, set nvoVolt to twice
     the value of nviVolt.
   */
  int value = LON_GET_SIGNED_WORD(nviVolt);
  if (value > MAX_VOLT) {
    /* Input value is out of range.  Set it to maximum */
    value = MAX_VOLT;
    LON_SET_SIGNED_WORD(nviVolt, value);
  }
  else if (value < MIN_VOLT) {
    /* Input value is out of range.  Set it to minimum */
    value = MIN_VOLT;
    LON_SET_SIGNED_WORD(nviVolt, value);
  }
  LON_SET_SIGNED_WORD(nvoVolt, value * 2);

  /* Propagate the NV onto the network. */
  if (LonPropagateNv(LonNvIndexNvoVolt) != LonApiNoError) {
    /* Handle error here, if desired. */
  }
  break;
}
```

Add Timeout Detection

The current implementation of the **LdvPutMsgBlocking()** function (in the **SysPutMsgInit()** function) does not perform error checking to ensure that it does not wait forever to send a message to the Micro Server. It is important that this function not block indefinitely so that the **LonInit()** function can complete. Thus, a more correct implementation would add timeout detection, and return the appropriate error code:

1. Add a field to the **DriverStatus** structure (in the **ldvsci.h** file) for a timeout value for sending a message from the driver to the Micro Server. For example:
   ```
   typedef LON_STRUCT_BEGIN(LdvDriverStatus)
   {
     …
     LonUbits32 PutMsgTimeout;
   } LON_STRUCT_END(LdvDriverStatus);
   ```
2. Define an appropriate timeout value for the serial link (in the **ldvsci.h** file). For example, set the timeout to 60 seconds (as 60000 milliseconds):
   ```
   #define LDV_PUTMSGTIMEOUT       60000
   ```

3. Near the beginning of the **LdvPutMsgBlocking()** function, set the driver status for the timeout value. For example:
```
DriverStatus.PutMsgTimeout = LDV_PUTMSGTIMEOUT;
```
4. Within the first while loop (**while (!bSuccess)**) of the **LdvPutMsgBlocking()** function, add a check for the timeout value. For example:
```
/* Check the timer */
if (DriverStatus.PutMsgTimeout == 0) {
  /* The timer has expired. */
  /* Declare the Micro Server as unresponsive */
  SysEnableInterrupts();
  result = LonApiMicroServerUnresponsive;
  break;
}
```
5. Similarly, within the second while loop (**while ((G_DriverStatus.TxInit == TRUE))**) of the **LdvPutMsgBlocking()** function, add the same check for the timeout value described in item number 4. Also, remove the 50000 µs sleep from this while loop.

The changes described in this section are optional because the current implementation provides the same behavior as the ShortStack 2 serial driver, and behavioral changes might require extra testing for your ShortStack device. However, a production device should ensure that the **LdvPutMsgBlocking()** function does not block indefinitely.

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 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.

To run the LonTalk Interface Builder tool for ShortStack, open a Windows command prompt (**Start → Programs → Accessories → Command Prompt**), and enter the following command from LonWorks Interface Developer directory (**\LonWorks\InterfaceDeveloper**):

```
libs
```

Command Usage

The following command usage notes apply to running the **libs** 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: `libs –n …`

  Long form: `libs --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:

```
libs --queryapi=yes
libs --queryapi
```

- 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 ".libs" extension. For the command line, you must use quotation marks for strings that include spaces. However, do not include the quotation marks in a command file (spaces in strings are supported for command files).

  Example command file:

  ```
  ; LIBS command file for myProject
  --source=myModelFile.nc
  --basename=myProjectVer1
  --server= SS400_FT3120E4_40000kHz
  --clock=10
  --multiplier=1/2
  --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 35 lists the available command switches for the **libs** command. Only the following switches are required for the command:

- --source (-n)
- --pid (-i)
- --basename (-b)
- --server (-s)
- --clock (-c)
- --multiplier (-P)

Other command switches are optional.

Table 35. Command Switches for the **libs** Command

Command Switch		
Long Form	**Short Form**	**Description**
--applmsg	-m	Enable support for application messages
--basename	-b	Set the project's base name

Command Switch		Description
Long Form	Short Form	
--clock	-c	Set external clock rate (in MHz) for the Micro Server
--clockfactor	-f	Scale the Micro Server clock rate ('STD' (default) or 'ALT')
--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
--expladdr	-e	Enable the use and availability of explicit addresses
--file	-@	Include a command file
--help	-?	Display usage hint for command
--include	-I	Add the specified folder to the include search path
--isi		Enables support for ISI in host-side API
--mkscript		Generate command script in specified location
--multiplier	-P	Set the clock multiplier for the Micro Server (valid values are: ½, 1. 2. 4. and 8)
--nodefaults		Disable processing of default command files
--out	-o	Generate all output files in the specified location
--pid	-i	Use the specified program ID (in colon-separated format)
--queryapi	-q	Enables and includes optional Query API functions

Command Switch		Description
Long Form	**Short Form**	
--server	-s	Specifies the Micro Server image file name. For a standard Micro Server (or a custom Micro Server in the [*ShortStack*]**\MicroServers** directory), you can specify the Micro Server's base name (such as **SS400_FT3150ISI_10000kHz**). You can also provide an absolute path to the Micro Server without a file extension, for example "**C:\myServers\myCustomServer**" (where myCustomServer is the Micro Server image name without the file extension).
--silent		Suppress banner message display
--source	-n	Use the specified model file
--spdelay	-p	Set the service pin notification delay (255=default, 0=off)
--updateapi	-u	Enables and includes optional Update API functions
--utilityapi		Enables and includes the optional local utility API functions
--verbose	-v	Run with verbosity level 0 (normal), 1 (verbose), or 2 (trace)
--verbosecomments	-V	Generate verbose comments
--warning		Display specified message number as a warning
--xcvr	-x	Use the specified transceiver (by name)

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 8, *Creating a Model File*, on page 115.

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 could be affected by use of these directives. The valid *option*s that can be specified within a model file are:

cp_family_space_optimization
no_cp_template_compression

The automatic configuration property merging feature in NodeBuilder 3.1 (and later) might change the device interface for a device that was previously built with the NodeBuilder 3 tool. 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 could 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 application program.

#pragma disable_warning *number*

Controls the compiler's printing of specific warning and hint messages Warning messages are less severe than errors, yet could indicate a problem in a program, or a place where code could be improved. To disable all warning messages, specify an asterisk (*****) for the *number*.

See the **enable_warning directive** to enable disabled warnings.

The **disable_warning** directive supercedes the **warnings_off** directive.

#pragma enable_dmf

Enables the direct memory file access method for the ShortStack Micro Server.

#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. See the *Neuron C Programmer's Guide* for more information about SD and SI data.

#pragma enable_warning *number*

Controls the compiler's printing of specific warning and hint messages Warning messages are less severe than errors, yet could indicate a problem in a program, or a place where code could be improved. To enable all warning messages, specify an asterisk (*****) for the *number*.

See the **disable_warning** directive for the reverse operation.

The **enable_warning** directive supercedes the **warnings_on** directive.

#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 micro_interface

This pragma is only used with the Microprocessor Interface Program (MIP) or with ShortStack Micro Server applications. You must include this directive in your custom Micro Server source code.

#pragma no_hidden

This pragma is for use only in the **<echelon.h>** standard include file.

#pragma optimization *level*

For Neuron C applications, this pragma allows you to specify a code optimization level for optimal use of device memory. See the *Neuron C Reference Guide* for information about using this pragma for Neuron C applications.

For model file compilation, executable code is ignored. You can use this pragma to specify optimization for CP template files.

Table 36 lists the levels of optimization for model file compilation (levels that are specific to Neuron C code are omitted). For most model files, optimization level 5 is recommended.

As part of optimization, 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. 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.

Important: Configuration property re-ordering and merging can reduce the memory required for the template file, but could also result in slower access to the application's configuration properties by network tools. This could potentially cause a significant increase in the time required to commission your device, especially on low-bandwidth channel types such as power line channels. 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.

Table 36. Optimization Levels for the **#pragma optimization** Directive

Level	Optimization Performed
0	No optimization CP templates are not compressed
3	CP templates are compressed
5	Maximum optimization

You can use the following keywords instead of the numeric level indicators:

- *none* for level 0
- *standard* for level 3
- *all* for level 5

The keyword level indicators are generally preferred over their numeric counterparts because they are self-documenting.

The **#pragma optimization** directive replaces the following directives:

#pragma codegen cp_family_space_optimization
#pragma codegen optimization_on
#pragma codegen optimization_off
#pragma codegen no_cp_template_compression

The compiler issues the **NCC#589** warning message if you use these deprecated directives. If your model file uses any of these directives with the **#pragma optimization** directive, the compiler issues the **NCC#588** warning message.

#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 version 2.1 (and later) 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 **"3.4"** as the string identifying the LONMARK guidelines version that the device conforms to. 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 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 only has the effect described above, and does not change the syntax of SD strings generated.

The **set_guidelines_version** directive is typically used to specify a version string in the *major.minor* form (for example, "3.4"). The compiler issues a **NCC#604** warning message if the application-specific version string does not match that format, but permits the string.

Using this directive can prevent certification of the generated device.

#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.

These directives override the settings for the **#pragma enable_warning** *number* and **#pragma disable_warning** *number* directives.

The **warnings_off** and **warnings_on** directives are deprecated. Use the **enable_warning** and **disable_warning** directives instead.

C

ShortStack LonTalk Compact API

This appendix describes the functions and callback handler functions included with the ShortStack LonTalk Compact API. It also describes modifying the API callback handlers for use with your ShortStack application.

Introduction

The ShortStack LonTalk Compact API provides the functions that you call from your ShortStack application to send and receive information to and from a LONWORKS network. The API also defines the callback functions that your ShortStack application must provide to handle LONWORKS events from the network and Micro Server. Because each ShortStack application handles these callbacks in its own specific way, you need to modify the callback functions.

Typically, you use the API functions for ShortStack initialization and sending and receiving network variable updates. See Chapter 10, *Developing a ShortStack Application*, on page 163, for more information about using these functions.

The ShortStack LonTalk Compact API functions are implemented in the **ShortStackApi.c** file; the ShortStack callback functions are defined in the **ShortStackHandlers.c** file. See *ShortStack LonTalk Compact API Files* on page 21 for a list of the files included with the ShortStack Developer's Kit.

This appendix provides an overview of the functions and callbacks. For detailed information about the ShortStack LonTalk Compact API, see the HTML documentation that is available from the Windows Start menu: select **Programs → Echelon ShortStack FX Developer's Kit → API Documentation**.

Changes to the API

The ShortStack FX LonTalk Compact API is essentially the same as the ShortStack 2.1 LonTalk Compact API. ShortStack 2.1 applications require no changes to compile with the ShortStack FX Developer's Kit.

The ShortStack FX LonTalk Compact API is considerably different from the ShortStack 2 API. The basic functionality of the two APIs is similar, but the naming convention used for ShortStack FX (and ShortStack 2.1) is different and more regular than the ShortStack 2 API, and the ShortStack FX LonTalk Compact API includes a different set of header files.

See *Using Types* on page 154 for other changes to the ShortStack LonTalk Compact API, as implemented by the LonTalk Interface Developer utility.

ShortStack FX Naming Conventions

All ShortStack names, members of structures, unions, or enumerations (but not those for function arguments, variables, and macros) use upper case for the beginning letter of each word, and include no underscores in the names. For example: *LonDomainConfigReceived*.

All function arguments and variables use lower case for the first letter with upper case for the beginning letter of each subsequent word, and include no underscores in the names. For example: *myVariable*.

All global names, with the exception of macro names and global variables, have a "Lon" name prefix (rather than using explicit ANSI C namespaces). Global variables have a "lon" prefix, but global constants (because they are immutable) have a "Lon" prefix. For example:

- *LonEventHandler*: a global name
- *lonErrorCount*: a global variable
- *LonErrorLimit*: a global constant

Function arguments, local variables, or members of structures and unions do not include the "Lon" or "lon" prefix. Members of enumerations are prefixed with "lon". Pointer variables include a "p" prefix, for example, *pDomain*.

Macros follow standard ANSI C conventions. That is, they are all capital letters, with individual words separated with a single underscore. Macro and predefined symbol names also include a "LON_" prefix. For example: *LON_NEURON_ID_LENGTH*.

The LonTalk Interface Developer utility generates network variable and configuration property types that comply with the rules for enhanced portability; see *Portability Overview* on page 110 for more information about these rules. However, the following additional rules apply to utility-generated types:

- The generated network variable and configuration property types use their Neuron C equivalents (such as **ncuInt** or **ncsLong**). These types are defined in the **LonPlatform.h** file, and by default are defined using types such as **LonByte** and **LonWord**. The **LonPlatform.h** file maintains the indirection of types because a particular compiler might offer a better, more convenient, equivalent for these Neuron C types. In that case, you can edit the appropriate section of the **LonPlatform.h** file to use these more convenient types.
- Non-native Neuron C types (such as **s32_type** and **float_type**) are defined in terms of their true Neuron C equivalent, for example, as arrays of four bytes.
- Enumerations referenced from types are defined as signed Neuron C integers (**ncsInt**). Thus, the generated types do not use the **LON_ENUM** macros.
- Type references are defined as dereferenced types.
- For purely host-side types, such as **LonNvDescription** (formerly **TNvTable**), bit fields are avoided where possible because not all target compilers support bit fields. Another exception is the **LonNvDescription.Size** field, which is declared as the **LonByte** type (instead of the natural **size_t**) to reduce the memory footprint.
- The access macros defined for bit field replacements are generated following the type definition, rather than preceding it. The bit field identifier that is part of the access macro's name is generated by converting the bit field member name to all upper case, removing leading prefixes (UNVT_, SNVT_, SCPT_ , UCPT_, or LON_), and removing all underscores.
- Bit field access macros include a comment that clarifies the meaning of the related bit field.
- Names of network variable and configuration property types, and all their member names, that support the direct memory file access method might not meet the naming guidelines.

Customizing the API

Portions of the API are optional, in particular, application messaging, network management query support, network management update support, and network management callbacks. If you do not plan to use these functions, you can choose not to include them in your ShortStack application to reduce the footprint of the application in your host microprocessor. The LonTalk Interface Developer utility includes options that control whether to include the optional APIs in your application.

API Memory Requirements

The memory requirements for the ShortStack LonTalk Compact API depend on which parts of the API you include in your application. You control which parts of the API to include in your application from the Interface Developer Code Generator Preferences page of the LonTalk Interface Developer utility.

Table 37 lists the approximate API memory requirements. Part of the memory requirement is application specific, depending on the device interface. 10 to 20% of the memory requirements listed in the table assume a simple device interface.

Table 37. ShortStack LonTalk Compact API Memory Requirements

Included API			Memory Requirement
Standard API	Optional API	ISI API	
☑			1.8 KB
☑	☑		2.3 KB
☑		☑	3.0 KB
☑	☑	☑	3.5 KB

The memory requirements for the serial driver depend on the driver's implementation. For the ARM7 serial driver that is included with the ARM7 Example Port, the memory requirement is approximately 3 KB.

The ShortStack LonTalk Compact API and Callback Handler Functions

This section provides an overview of the ShortStack FX LonTalk Compact API functions and callback handler functions. For detailed information about the ShortStack LonTalk Compact API and the callback handler functions, see the HTML API documentation and the API source code:

- **Start → Programs → Echelon ShortStack FX Developer's Kit → Documentation → API Reference**
- **Start → Programs → Echelon ShortStack FX Developer's Kit → API Source Code**

ShortStack LonTalk Compact API Functions

The ShortStack LonTalk Compact API includes functions for managing network data and the ShortStack Micro Server.

Commonly Used Functions

Table 38 lists API functions that you will most likely use in your ShortStack application to send and receive data over a LONWORKS network.

Table 38. Commonly Used ShortStack LonTalk Compact API Functions

Function	Description
LonEventHandler()	Processes any messages received by the ShortStack driver. If messages are received, it calls the appropriate callback functions. See *Periodically Calling the Event Handler* on page 170 for more information about how to use this function.
LonInit()	Initializes the ShortStack LonTalk Compact API, the serial driver, and the ShortStack Micro Server. This function downloads ShortStack device interface data from the ShortStack application to the ShortStack Micro Server. The ShortStack application must call **LonInit()** once on startup.
LonPropagateNv()	Propagates a network variable value to the network. This function propagates a network variable if *all* of the following conditions are met: • The network variable is declared with the **output** modifier • The network variable must be bound to the network

Other Functions

Table 39 lists other ShortStack LonTalk Compact API functions that you can use in your ShortStack application. These functions are not typically used by most ShortStack applications.

Table 39. Other ShortStack LonTalk Compact API Functions

Function	Description
LonGetUniqueId()	Gets the unique ID (Neuron ID) value of the ShortStack Micro Server.
LonGetVersion()	Gets the version number of the ShortStack firmware in the ShortStack Micro Server.

Function	Description
LonPollNv()	Requests a network variable value from the network. A ShortStack 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 an input network variable, it must be declared in the model file as an input network variable and include the polled modifier. You cannot poll an output network variable with the LonPollNv() function. Do not use polling with ISI connections.
LonSendServicePin()	Broadcasts a service-pin message to the network. The service-pin message is used during configuration, installation, and maintenance of a LONWORKS device.

Application Messaging Functions

Table 40 lists the ShortStack LonTalk Compact 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. The same functions can be used for foreign frame and explicit network variable messages. Support for application messaging is optional.

If you choose not to support application messaging, this function is not available for use in your ShortStack application. You can select whether to include these functions in the LonTalk Interface Developer utility's Micro Server Preferences page.

Table 40. Application Messaging ShortStack LonTalk Compact API Functions

Function	Description
LonSendMsg()	Sends an application, foreign frame, or explicit network variable message.
LonSendResponse()	Sends an application, foreign frame, or explicit network variable message response to a request message. The ShortStack application calls LonSendResponse() in response to a LonMsgArrived() callback handler function.

Network Management Query Functions

The ShortStack LonTalk Compact API includes the optional network management query API functions that provide additional network management commands listed in Table 41 on page 293. Support for these network management API functions is optional.

The network management query API functions are asynchronous functions. They issue a downlink request and return immediately. The functions can fail if no downlink buffer is available.

If you do not plan to use these local network management commands, you do not need to include these functions in your ShortStack application. You can select whether to include these functions in the LonTalk Interface Developer utility's Code Generator Preferences page.

Table 41. Network Management Query API Functions

Function	Description
LonQueryAddressConfig()	Queries configuration data for the Micro Server's address table.
LonQueryAliasConfig()	Queries configuration data for the Micro Server's alias table.
LonQueryConfigData()	Queries configuration data on the ShortStack Micro Server.
LonQueryDomainConfig()	Retrieves domain information from the ShortStack Micro Server.
LonQueryNvConfig()	Queries configuration data for the Micro Server's network variable table.
LonQueryStatus()	Requests the status of the ShortStack Micro Server.
LonQueryTransceiverStatus()	Requests the status of the ShortStack Micro Server's transceiver. Used with power line transceivers. If this function is used with an FT transceiver, the function will appear to succeed, but the callback that contains the results will declare a failure.

Network Management Update Functions

The ShortStack LonTalk Compact API includes the optional network management update API functions that provide additional network management commands listed in **Table 42**. Support for these network management API functions is optional.

The network management update API functions can fail if no downlink buffer is available.

If you do not plan to use these local network management commands, you do not need to include these functions in your ShortStack application. You can select whether to include these functions in the LonTalk Interface Developer utility's Code Generator Preferences page.

Table 42. Network Management Update API Functions

Function	Description
LonClearStatus()	Clears a subset of status information on the ShortStack Micro Server.

Function	Description
LonSetNodeMode()	Sets the operating mode for the Micro Server: • Online: For an online device, both the host application and the Micro Server are running, and the device responds to all network messages. • Offline: For an offline device, the host application cannot propagate network variables or send network messages. The Micro Server processes network variable update requests, and updates the network variable values, but the ShortStack LonTalk Compact API does not call the **LonNvUpdateOccurred()** callback handler function. The Micro Server acknowledges application messages that the device receives, but discards them.
LonUpdateAddressConfig()	Sets configuration data for the Micro Server's address table.
LonUpdateAliasConfig()	Sets configuration data for the Micro Server's alias table.
LonUpdateConfigData()	Sets configuration data on the ShortStack Micro Server.
LonUpdateDomainConfig()	Sets domain information from the ShortStack Micro Server.
LonUpdateNvConfig()	Sets configuration data for the Micro Server's network variable table.

Local Utility Functions

Table 43 lists the ShortStack LonTalk Compact API functions that provide local utility functions for the host application. Including these functions is optional.

If you choose not to include these functions, they are not available for use in your ShortStack application. You can select whether to include these functions in the LonTalk Interface Developer utility's Code Generator Preferences page.

Table 43. Local Utility ShortStack LonTalk Compact API Functions

Function	Description
LonGoConfigured()	Puts the Micro Server in the configured state and online mode.
LonGoUnconfigured()	Puts the Micro Server in the unconfigured state.
LonMtIsBound()	Queries the ShortStack Micro Server to determine if the specified message tag is bound to the network. You can use this function to ensure that transactions are initiated only for connected message tags. The **LonMtIsBoundReceived()** callback handler function processes the reply to the query.

Function	Description
LonNvIsBound()	Queries the ShortStack Micro Server to determine if the specified network variable is bound to the network. You can use this function to ensure that transactions are initiated only for connected network variables. The **LonNvIsBoundReceived()** callback handler function processes the reply to the query.
LonQueryAppSignature()	Queries the Micro Server's current version of the host application signature.
LonQueryVersion()	Queries the version number of the Micro Server application and the Micro Server core library used for the Micro Server. With this version information and the Micro Server key, you can uniquely identify the current Micro Server.
LonRequestEcho()	Sends a six-byte message (arbitrary values defined by the application) to the ShortStack Micro Server. The Micro Server transforms this message by incrementing each of the six data bytes and returning the message to the host. You can use the echo command instead of the ping command, but the echo command takes longer to complete (because of larger messages, and because of the data transformation performed by the Micro Server). Echo tests should be performed frequently during early stages of device development or stress testing, but should be executed infrequently on a production device.
LonSendPing()	Sends a message to the ShortStack Micro Server to verify that communications with the Micro Server are functional. This function can be useful after long periods of network inactivity. **Recommendation**: Define a ping timer of at least 60 seconds. The application should reset this timer upon completion of every successful uplink or downlink communication. When this timer expires, the application issues a ping request to the Micro Server. If the Micro Server is functional, it replies to the ping request by causing the **LonPingReceived()** callback event. In general, link layer idleness of more than 1.5 times the ping timer's duration indicates a serious error. An application can recover from this error by physically resetting the Micro Server.

ShortStack Callback Handler Functions

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

Commonly Used Callback Handler Functions

Table 44 lists the callback 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 44. Commonly Used ShortStack Callback Handler Functions

Function	Description
LonGetCurrentNvSize()	Indicates a request for the network variable size. The ShortStack LonTalk Compact API calls this callback handler function to determine the current size of a changeable-type network variable. For non-changeable-type network variables, this function should return the value of the **LonGetDeclaredNvSize()** function. For changeable-type network variables, you must modify this function in the **ShortStackHandlers.c** file.
LonNvUpdateCompleted()	Indicates that either an update network variable call or a poll network variable call is completed.
LonNvUpdateOccurred()	Indicates that a network variable update request from the network has been processed by the ShortStack LonTalk Compact API. This call indicates that the network variable value has already been updated, and allows your host application to perform any additional processing, if necessary.
LonOffline()	A request from the network that the device go offline. Installation tools use this message to disable application processing in a device. An offline device continues to respond to network management messages, but the host application cannot propagate network variables or send network messages. When this function is called, the ShortStack Micro Server is still online, but changes to the offline state as soon as this callback handler completes.
LonOnline()	A request from the network that the device go online. Installation tools use this message to enable application processing in a device. When this function is called, the ShortStack Micro Server is still offline, but changes to the online state as soon as this callback handler completes.
LonReset()	A notification that the ShortStack Micro Server has been reset.

Function	Description
LonServicePinHeld()	An indication that the service pin on the device has been held for some number of seconds (default is 10 seconds). Use it if your application needs notification of the service pin's being held.
LonServicePinPressed()	An indication that the service pin on the device has been pressed. Use it if your application needs notification of the service pin's being pressed.
LonWink()	A wink request from the network. Installation tools use the Wink message to help installers physically identify devices. When a device receives a Wink message, it should provide some visual, audio, or other indication for an installer to be able to physically identify this device.

Application Messaging Callback Handler Functions

Table 45 lists the callback handler functions that are called by the ShortStack LonTalk Compact API for application messaging transactions. Customize these functions if you use application messaging in your ShortStack device. Application messaging is optional and only recommended for implementing the LONWORKS file transfer protocol, the ISI protocol, and for proprietary interfaces.

If you choose not to support application messaging, you do not need to customize these functions. You can select whether to include these functions in the LonTalk Interface Developer utility's Code Generator Preferences page.

Table 45. Application Messaging ShortStack Callback Handler Functions

Function	Description
LonMsgArrived()	An application or foreign frame message from the network to be processed. This function performs any application-specific processing required for the message. If the message is a request message, the function must deliver a response using the LonSendMsgResponse() function. Application messages are always delivered to the application, regardless of whether the message passed authentication. The application decides whether authentication is required for a message.
LonMsgCompleted()	Indicates that downlink transfer for a message, initiated by a LonSendMsg() call, was completed. If a request message has been sent, this callback handler is called only after all responses have been reported by the LonResponseArrived() callback handler.

Function	Description
LonResponseArrived()	An application message response from the network. This function performs any application-specific processing required for the message.

Network Management Query Callback Handler Functions

The ShortStack LonTalk Compact API includes the optional network management query API callback handler functions listed in **Table 46**. These callbacks allow you to customize the application processing for responses to local network management commands (see **Table 41** on page 293). Support of these network management query API callback functions is optional.

If you do not plan to use extended local network management commands, there is no need to customize or include these functions in your ShortStack application. You can select whether to include these functions in the LonTalk Interface Developer utility's Code Generator Preferences page.

Table 46. Network Management Query API Callback Handler Functions

Function	Description
LonAddressReceived()	Indicates that configuration data for the Micro Server's address table has been received.
LonAliasConfigReceived()	Indicates that configuration data for the Micro Server's alias table has been received.
LonConfigDataReceived()	Indicates that configuration data has been received from the Micro Server. Receipt of this data is initiated by a call to the **LonQueryConfigData()** function.
LonDomainConfigReceived()	Indicates that domain information has become available. This event is initiated by the Micro Server in response to a previous call to **LonQueryDomain()** by the ShortStack application.
LonNvConfigReceived()	Indicates that configuration data for the Micro Server's network variable table has been received.
LonStatusReceived()	Indicates that the status report has been received from the Micro Server. Receipt of this data is initiated by a call to the **LonQueryStatus()** function. Modify this function to perform application-specific handling of the status report.

Function	Description
LonTransceiverStatusReceived()	Indicates that the transceiver status report has been received from the Micro Server. Receipt of this data is initiated by a call to the LonQueryTransceiverStatus() function. Modify this function to perform application-specific handling of the transceiver status.

Local Utility Callback Handler Functions

Table 47 lists the callback handler functions for the local utility functions described in *Local Utility Functions* on page 294.

You can select whether to include the local API functions and their callback handler functions in the LonTalk Interface Developer utility's Code Generator Preferences page.

Table 47. Local Utility API Callback Handler Functions

Function	Description
LonAppSignatureReceived()	Indicates the current host application signature.
LonEchoReceived()	Provides the Micro Server's echo response, containing the transformed data from the corresponding LonRequestEcho() request. The application is responsible for verifying that the echo response meets expectations.
LonGoConfiguredReceived()	Indicates that the Micro Server has responded to the LonGoConfigured() request.
LonGoUnconfiguredReceived()	Indicates that the Micro Server has responded to the LonGoUnConfigured() request.
LonMtIsBoundReceived()	Indicates whether the specified message tag is bound to the network.
LonNvIsBoundReceived()	Indicates whether the specified network variable is bound to the network.
LonPingReceived()	Indicates whether the Micro Server received the ping message.
LonVersionReceived()	Indicates the version number of the Micro Server application and the Micro Server core library used for the Micro Server.

D

ShortStack ISI API

This appendix describes the functions and callbacks included with the ShortStack ISI API. It also describes why and how to modify the API callbacks for use with your ShortStack application.

Introduction

The **ShortStackIsiTypes.h** and **ShortStackIsiApi.h** header files include all types, enumerations, and prototypes that are needed to create an ISI-compliant host application.

This appendix provides an overview of the ShortStack ISI functions and callbacks. For detailed information about the ShortStack ISI API, see the HTML documentation that is available from the Windows Start menu: select **Programs → Echelon ShortStack FX Developer's Kit → API Documentation**.

The ShortStack ISI API

Table 48 lists the ShortStack ISI API functions. When the host application calls one of the functions listed in **Table 48**, a common function sends the downlink message. When the API completes (that is, when the API receives either an ACK or NACK response from the Micro Server for the downlink API call), it calls the **IsiApiComplete()** callback handler function to inform the host application that it can issue additional API calls.

Table 48. ShortStack ISI API Functions

Function	Description
IsiAcquireDomain()	Starts or re-starts the domain ID acquisition process in a device that supports domain acquisition. Do not use this function if the engine is started with **isiTypeS**.
IsiCancelAcquistion()	Cancels both device and domain acquisition. After this function call completes, the ISI engine calls the **IsiUpdateUserInterface()** function with the **IsiNormal** event. Do not use this function if the engine is started with **isiTypeS**.
IsiCancelEnrollment()	Cancels an open (pending or approved) enrollment. When used on a connection host, a CSMX connection cancellation message is issued to cancel enrollment on the connection members. When used on a device that has accepted (but not yet implemented) an open enrollment, this function causes the device to opt out of the enrollment locally. The function has no effect unless the ISI engine is running and in the pending or approved state.

Function	Description
IsiCreateEnrollment()	Accepts a connection invitation. This function can be called after the application has received and approved a CSMO open enrollment message. If the assembly is not already in a connection, or if the assembly is in a connection and the device supports direct connection removal, the connection is re-created. If the assembly is already in a connection, any previous connection information is replaced. This function must not be called with an assembly that is already in a connection on a device that does not support direct connection removal. On a connection host that has received at least one CSME enrollment acceptance message, this command completes the enrollment and implements the connection as new, replacing any previously existing enrollment information associated with this assembly. Calling this function on a device that does not support connection removal while indicating an assembly number that is already engaged in another connection, does not implement the new connection. The **IsiImplemented** event is not fired in this case. The application can use the **IsiQueryIsConnected()** function to determine if a given assembly is currently engaged in a connection. Where supported, and unless application requirements dictate otherwise, the **IsiExtendEnrollment()** function should be used instead. The ISI engine must be running and in the correct state when calling this function. For a connection host, the ISI engine must be in the approved state. Other devices must be in the pending state.
IsiDeleteEnrollment()	Removes the specified assembly from all connections, and sends a CSMD connection deletion message to all other devices in each connection to remove them from the connection. This function has no effect if the ISI engine is stopped.

Function	Description
IsiExtendEnrollment()	Accepts a connection invitation on a device that supports connection extension. This function can be called after the application has received and approved a CSMO open enrollment message. The connection is added to any previously existing connections. If no previous connection exists for the assembly, a new connection is created. This function must not be called on a device that does not support connection extension. Where supported, and unless application requirements dictate otherwise, call this function instead of the **IsiCreateEnrollment()** function. On a connection host that has received at least one CSME enrollment acceptance message, this command completes the enrollment and extends any existing connections. If no previous connection exists for the assembly, a new connection is created. The ISI engine must be running and in the correct state for this function to have any effect. For a connection host, the ISI engine must be in the approved state. Other devices must be in the pending state.
IsiFetchDevice()	Fetches a device by assigning a domain to the device from a domain address server (DAS). An alternate method to assign a domain to a device is for the device to use the **IsiAcquireDomain()** function. This function must be called only from a domain address server.
IsiFetchDomain()	Starts or restarts the fetch domain process in a domain address server (DAS). This function must be called only from a domain address server.
IsiInitiateAutoEnrollment()	Starts automatic enrollment. The local device becomes the connection host. Automatic enrollment can replace previous connections, if any. When this call returns, the ISI connection is implemented for the associated assembly. This function should not be called before the **IsiWarm** event has been signaled in the **IsiUpdateUserInterface()** callback. This function does nothing when the ISI engine is stopped.

Function	Description
IsiIssueHeartbeat()	Sends an update for the specified bound output network variable and its aliases, using group addressing. This function is typically called by the **IsiQueryHeartbeat()** callback handler function. This function requires that the ISI engine has been started with the **IsiFlagHeartbeat** flag.
IsiLeaveEnrollment()	Removes the specified assembly from all enrolled connections as a local operation only. When used on the connection host, the function is automatically interpreted as **IsiDeleteEnrollment()**. This function has no effect if the ISI engine is stopped.
IsiOpenEnrollment()	Opens manual enrollment for the specified assembly. The device becomes a connection host for this connection and sends a CSMO manual connection invitation to all devices in the network. The ISI engine must be running, and in the idle state.
IsiQueryImplementationVersion()	Returns the version number of this ISI implementation. This function returns its result asynchronously through the **IsiImplementationVersionReceived()** callback function. The most current ISI implementation is version 3.03. For this version, this function reports implementation version 3.
IsiQueryIsBecomingHost()	Returns **TRUE** if **IsiOpenEnrollment()** has been called for the specified assembly and the enrollment has not yet timed out, been cancelled, or confirmed. The function returns **FALSE** otherwise. This function returns its result asynchronously through the **IsiIsBecomingHostReceived()** callback function.
IsiQueryIsConnected()	Returns **TRUE** if the specified assembly is enrolled in a connection. The function returns **FALSE** if the ISI engine is stopped. This function returns its result asynchronously through the **IsiIsConnectedReceived()** callback function.
IsiQueryIsRunning()	Returns **TRUE** if the ISI engine is running and **FALSE** if the ISI engine is stopped. This function returns its result asynchronously through the **IsiIsRunningReceived()** callback function.

Function	Description
IsiQueryProtocolVersion()	Returns the version of the ISI protocol supported by the ISI engine. The number indicates the maximum protocol version supported. The ISI engine also supports protocol versions less than the number returned unless explicitly indicated. This function returns its result asynchronously through the **IsiProtocolVersionReceived()** callback function. The most current ISI protocol version is 1.
IsiReturnToFactoryDefaults()	Restores the device's self-installation data to factory defaults, causing the immediate and unrecoverable loss of all connection information. This function returns to the caller, however, calling this function resets the Micro Server.
IsiStart()	Starts the ISI engine. The ISI engine sends and receives ISI messages, and manages the network configuration of your device. This function also specifies whether domain acquisition server or client services are supported. Calls to this function with the **IsiTypeDas** parameter for a Micro Server that does not support ISI DAS are NACKed.
IsiStartDeviceAcquisition()	Starts or retriggers device acquisition mode on a domain address server. The domain address server responds to domain ID requests from devices that implement a domain acquisition client, as long as it is in device acquisition mode. Call this function only if the ISI engine has been started with the **IsiTypeDas** type.
IsiStop()	Stops the ISI engine.

Certain ISI API calls are managed by the Micro Server itself. These include the following functions:

- **IsiTick()**
- **IsiApproveMsg()**
- **IsiProcessMsg()**
- **IsiProcessResponse()**

The Micro Server automatically translates these calls according to the mode that was used when starting the ISI engine. Wrapper functions for the related ISI functions are implemented within the **MicroServer.nc** file. For a custom Micro Server, you can modify those wrapper functions, for example, to intercept ISI

messages. These wrapper functions (and any extensions that you supply) must be located on the Micro Server.

The ShortStack ISI Callback Handler Functions

Table 49 lists the ShortStack ISI callback handler functions.

In any ISI application, callback handlers provide application-specific details to the ISI engine. ShortStack ISI applications can choose whether to implement these callback handlers on the host processor or on the Micro Server. In either case, the set of callback handler functions and their prototypes remain the same.

ISI callback handler functions must return to the caller as soon as possible, providing the requested information.

Table 49. ShortStack ISI Callback Handler Functions

Function	Description
IsiApiComplete()	Indicates that the API function is complete and that the result has been received. This function is called when an API function completes. Generally, you should not call an ISI API function until the previous one completes. This callback is available only on the host processor.
IsiCreateCsmo()	Constructs the **IsiCsmoData** portion of a CSMO Message. This function is called by the ISI engine prior to sending a CSMO message. This callback can be implemented on an application-specific custom Micro Server or on the host. The standard Micro Servers expect this callback on the host. Typical applications implement this callback handler function in the same location (host or custom Micro Server) as the **IsiGetWidth()** callback handler function.

Function	Description
IsiCreatePeriodicMsg()	Specifies whether the application has any messages for the ISI engine to send using the periodic broadcast scheduler. Because the ISI engine sends periodic outgoing messages at regular intervals, this function allows an application to send a message during one of the periodic message slots. If the application has no message to send, then this function should return **FALSE**. If it does have a message to send, then this function should return **TRUE**. To use this function, you must enable application-specific periodic messages using the **IsiFlagApplicationPeriodic** flag when you call the **IsiStart()** function. The default implementation of this function does nothing but return **FALSE**. You can override this function by providing an application-specific implementation of **IsiCreatePeriodicMsg()**. Do not send any messages, start other network transactions, or call other ISI API functions while the **IsiCreatePeriodicMsg()** callback is running. To call other ISI API functions or start other network transactions, signal the application's readiness through an application-specific utility in the **IsiCreatePeriodicMsg()** callback function and evaluate the signal when appropriate. This separate utility can send the periodic message soon after the **IsiCreatePeriodicMsg()** function is completed. This callback handler can be implemented on an application-specific custom Micro Server or on the host. The standard Micro Servers use the default implementation of this callback.
IsiGetAssembly()	Returns the number of the first assembly that can join the connection. The function returns **ISI_NO_ASSEMBLY** (0xFF) if no such assembly exists, or an application-defined assembly number (0 to 254). This callback can be implemented on an application-specific custom Micro Server or on the host. The standard Micro Servers expect this callback on the host.

Function	Description
IsiGetConnection()	Returns a pointer to an entry in the connection table. The default implementation returns a pointer to a built-in connection table with 32 entries, stored in the Micro Server's on-chip EEPROM memory (extended RAM for a Series 5000 Micro Server). You can override this function to provide an application-specific means of accessing the connection table, or to provide an application table of a different size. This function is frequently called and should return as soon as possible. If you override this function, you must also override the **IsiGetConnectionTableSize()** and **IsiSetConnection()** functions. And, if you implement any of these callback handlers either on the host or on the Micro Server, you must override the other two in the same location. Assuming that the Micro Server has sufficient resources, implement all three of these functions on the Micro Server for performance reasons.
IsiGetConnectionTableSize()	Returns the number of entries in the connection table. The default implementation returns the number of entries in the built-in connection table (32). You can override this function to support an application-specific implementation of the ISI connection table. You can use this function to support a larger connection table. The ISI library supports connection tables with 0 to 254 entries. The connection table size is considered constant following a call to **IsiStart()**; you must first stop, then re-start, the ISI engine if the connection table size changes dynamically. If you override this function, you must also override the **IsiGetConnection()** and **IsiSetConnection()** functions. And, if you implement any of these callback handlers either on the host or on the Micro Server, you must override the other two in the same location. Assuming that the Micro Server has sufficient resources, implement all three of these functions on the Micro Server for performance reasons. Custom Micro Servers can change the connection table size, or its location, or both.

Function	Description
IsiGetNextAssembly()	Returns the next applicable assembly for an incoming CSMO following the specified assembly. The function returns **ISI_NO_ASSEMBLY** (0xFF) if there are no such assemblies, or an application-specific assembly number (1 to 254). This function is called after calling the **IsiGetAssembly()** function, unless **IsiGetAssembly()** returned **ISI_NO_ASSEMBLY**. This callback can be implemented on an application-specific custom Micro Server or on the host. The standard Micro Servers expect this callback on the host.
IsiGetNextNvIndex()	Returns the network variable index of the network variable at the specified offset within the specified assembly, following the specified network variable. Returns **ISI_NO_INDEX** (0xFF) if there are no more network variables or a valid network variable index (0 to 254) otherwise. This callback can be implemented on an application-specific custom Micro Server or on the host. The standard Micro Servers expect this callback on the host.
IsiGetNvIndex()	Returns the network variable index (0 to 254) of the network variable at the specified offset within the specified assembly or **ISI_NO_INDEX** (0xFF) if no such network variable exists. This function must return at least one valid network variable index for each assembly number returned by **IsiGetAssembly()** and **IsiGetNextAssembly()**. This callback can be implemented on an application-specific custom Micro Server or on the host. The standard Micro Servers expect this callback on the host.
IsiGetNvValue()	Returns the value of the specified network variable. This callback must be implemented on the host, but is only required if ISI network variable heartbeats are supported and enabled.

Function	Description
IsiGetPrimaryDid()	Returns a pointer to the default primary domain ID for the device. The function also provides the domain ID length. Domain IDs can be 1, 3, or 6 bytes long; the 0-length domain ID cannot be used for the primary domain. You can override this function to override the ISI standard domain ID value. This function is only used to define a unique primary domain when creating a domain address server, and to define a non-standard domain when creating a non-interoperable self-installed system. Both length and value of the domain ID provided are considered constant after the ISI engine is running. To change the primary domain ID at runtime using the **IsiGetPrimaryDid()** callback, stop and re-start the ISI engine. **Important**: Non-interoperable self-installed devices cannot interoperate with ISI devices. This callback is implemented on the Micro Server. By default, the default implementation is used. If you want to create an ISI domain address server with ShortStack, you must create a custom Micro Server and override the **IsiGetPrimaryDid()** function. Typically, such an overridden **IsiGetPrimaryDid()** callback returns the Micro Server's own Neuron ID.
IsiGetPrimaryGroup()	Returns the group ID for the specified assembly. The default implementation returns **ISI_DEFAULT_GROUP** (128). This callback can be implemented on an application-specific custom Micro Server or on the host. The standard Micro Servers expect this callback on the host.

Function	Description
IsiGetRepeatCount()	Specifies the repeat count used with all network variable connections, where all connections share the same repeat counter. The repeat counter value is considered constant for the lifetime of the application, and is only queried when the device powers up the first time after a new application image has been loaded, and every time **IsiReturnToFactoryDefaults()** runs. Only repeat counts of 1, 2 or 3 are supported. To take full advantage of the secondary frequency on a PL transceiver, only use a repeat count of 1 or 3. This function has no affect on ISI messages. The default implementation of this function always returns 3. This function operates whether the ISI engine is running or not. This callback can be implemented on an application-specific custom Micro Server or on the host. The standard Micro Servers use the default implementation that is provided with the ISI library, which results in 3 repeats.
IsiGetWidth()	Returns the width in the specified assembly. The width is equal to the number of network variable selectors associated with the assembly. This callback can be implemented on an application-specific custom Micro Server or on the host. The standard Micro Servers expect this callback on the host.
IsiImplementationVersionReceived()	Retrieves the version number of this ISI implementation. This callback occurs as a result of an earlier call to the **IsiQueryImplementationVersion()** function.
IsiIsBecomingHostReceived()	Reports **TRUE** if **IsiOpenEnrollment()** has been called for the specified assembly and the enrollment has not yet timed out, been cancelled, or confirmed. The function reports **FALSE** otherwise. This callback occurs as a result of an earlier call to the **IsiQueryIsBecomingHost()** API function.
IsiIsConnectedReceived()	Reports **TRUE** if the specified assembly is enrolled in a connection. The function reports **FALSE** if the ISI engine is stopped. This callback occurs as a result of an earlier call to the **IsiQueryIsConnected()** API function.

Function	Description
IsiIsRunningReceived()	Reports **TRUE** if the ISI engine is running and **FALSE** if the ISI engine is stopped. This callback occurs as a result of an earlier call to the **IsiQueryIsRunning()** API function.
IsiProtocolVersionReceived()	Retrieves the version of the ISI protocol supported by the ISI engine. The number indicates the maximum protocol version supported. The ISI engine also supports protocol versions less than the number returned unless explicitly indicated. This callback occurs as a result of an earlier call to the **IsiQueryProtocolVersion()** API function.
IsiQueryHeartbeat()	Returns **TRUE** if a heartbeat for the network variable with the specified global index has been sent, and returns **FALSE** otherwise. When network variable heartbeat processing is enabled, and the ISI engine is running, the engine queries bound output network variables using this callback (including any alias connections) whenever the heartbeat is due. This function does not send the heartbeat update—see **IsiIssueHeartbeat()**. For more details on network variable heartbeat scheduling, see the *ISI Protocol Specification*. This callback handler can be implemented on an application-specific custom Micro Server or on the host. The standard Micro Servers expect this callback to be implemented on the host.

Function	Description
IsiSetConnection()	Updates an entry in the connection table, which must be kept in persistent, nonvolatile, storage. The default implementation updates an entry in the built-in connection table with 32 entries, stored in the Micro Server's on-chip EEPROM memory. You can override this function to provide an application-specific means of accessing the connection table, or to provide an application table of a different size. This function is frequently called and should return as soon as possible. If you override this function, you must also override the **IsiGetConnectionTableSize()** and **IsiGetConnection()** functions. And, if you implement any of these callback handlers either on the host or on the Micro Server, you must override the other two in the same location. Assuming that the Micro Server has sufficient resources, implement all three of these functions on the Micro Server for performance reasons.
IsiUpdateUserInterface()	Provides status feedback from the ISI engine. These events are useful for synchronizing the device's user interface with the ISI engine. To receive notification of ISI status events, override the **IsiUpdateUserInterface()** callback function. The default implementation of this function does nothing. This callback is typically, and by default, implemented on the host.
IsiUserCommand()	Informs the host application about user-defined Micro Server events. A custom Micro Server might need to inform the host application about events that are otherwise known only to custom code that is local to a custom Micro Server. See *Discovering Devices* on page 226 for an example of using this function.

An ISI-aware host application requires an ISI-aware Micro Server, but an ISI-aware Micro Server can be used with an ISI-unaware host application and host API.

As defined in the [*ShortStack*]**\Custom MicroServer\ShortStackIsiHandlers.h** header file, an ISI callback handler function can reside in one of the following locations:

- *The ISI Library*. The callback handler is an ISI default function. No development effort is required to implement these functions, but no customized behavior is available.

- *The Micro Server application*. The callback handler is a locally overridden function. Customization of these handlers requires a custom Micro Server. Assuming the Micro Server has sufficient resources, these callback handler overrides offer the best performance and control and minimal host footprint, but can lead to application-specific Micro Server implementations.
- *The host application*. The callback handler is a remote function that uses the ShortStack ISI protocol. These callback handlers are the most flexible, but lowest performance ISI callback handlers. This type of callback handler is typically used for application-specific callbacks, and allows the use of a single Micro Server for multiple applications.

Important: A callback handler function should not call any other ISI callback handler functions, unless both the caller and the called functions reside on the same platform (host or Micro Server).

For each callback, you can choose whether the callback is handled by the ISI default, by a version local to the Micro Server, or by the host application. The [*ShortStack*]**\Custom MicroServer\ShortStackIsiHandlers.h** header file includes conditional-compilation macros for each callback handler function:

- To direct the callback to the Micro Server
- To direct the callback to the host
- To enable the default implementation

The callback control macros use the following naming convention:

ISI_*location_callback*

For example: **ISI_HOST_GETASSEMBLY** or **ISI_SERVER_GETCONNECTIONTABLESIZE**.

For a remote callback handler, the ShortStack Micro Server includes a proxy function that receives the function's parameters, packs them into a message buffer, and passes the data to the host function.

If the host application needs to send a response to a callback handler, and it is unable to do so because there are no transmit buffers, it retries sending the response until it is successful. The Micro Server's RPC guard times out after 5 seconds, after which the Micro Server logs an error and resets. See **Table 23** on page 187 in Chapter 10, *Developing a ShortStack Application*, for a list of the **LonSystemError** enumeration values.

While waiting for the response, the Micro Server continues to process downlink and uplink traffic. However, because only one downlink ISI API request can be buffered, additional requests are NACKed. Other functionality might be delayed and enqueued for later processing while waiting for the completion of an RPC.

E

Downloading a ShortStack Application over the Network

This appendix describes considerations for designing a ShortStack host application that allows host application updates over the network.

Overview

For a Neuron hosted device, you can update the application image over the network using an LNS tool, such as the LonMaker Integration tool or another network management tool. However, you cannot use the same tools or technique to update a ShortStack application image over the network. Many ShortStack devices do not require application updates over the network, but for those that do, this appendix describes considerations for adding this capability to the device.

If a ShortStack host 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 ShortStack LonTalk Compact API nor the ShortStack Micro Server directly supports updating the host application over the network, you must:

1. Define a custom host application download protocol.
2. Implement an application download utility.
3. Implement application download capability within your ShortStack host application.

For the application download process:

- The application must be running and configured for the duration of the download.
- There must be sufficient volatile and non-volatile memory to store the new image without affecting the current image.
- The application must be able to boot the new image at the end of the download. During this critical period, the application must be able to tolerate device resets and boot either the old application image or the new one, as appropriate.

This appendix decribes some of the considerations for designing a ShortStack application download function.

Custom Host Application Download Protocol

The custom host application protocol that you define for downloading a ShortStack host application over the network should support the following steps:

1. Prepare for application download.

   When the application download utility informs the current ShortStack host 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 ShortStack host application should verify that the application to be downloaded can run on the device platform (using the Micro Server key and link layer protocol version numbers 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 LONWORKS file transfer protocol (LW-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 ShortStack host application should update the flash in the background, however, it might be necessary for the protocol to define additional flow control to allow the host 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 host 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 host processor can load the new application and restart the processor with the new image. See your host processor's and operating system's documentation for recommendations and information about creating a custom boot loader.

Important: For the duration of the first three steps, the application must be running, the link-layer driver must be operational, and the ShortStack device must be configured and online.

Upgrading Multi-Processor Devices

A ShortStack device consists of at least two processor chips, each with their respective applications: a Smart Transceiver with the ShortStack Micro Server and your host processor with the ShortStack link-layer driver, ShortStack LonTalk Compact API, and your application program.

Because both processor chips must be able to communicate through the link layer, both must use the same protocol for application download, and have matching settings.

Most updates to ShortStack host applications will likely address issues within the application's control algorithm, and leave the ShortStack LonTalk Compact API and link-layer driver unchanged. To ensure that the new application is correct for the current device and its settings, the host application download protocol must ensure that at least the following requirements are met before control is handed to the new application:

- The Micro Server and the host application must support the same link-layer protocol version. The link-layer protocol version is contained in the Micro Server's reset notification message.
- The Micro Server and the host application must support matching transceiver types. You can configure the variations of the PL-20 transceiver into a Micro Server that supports any of the PL-20 channel types (PL-20N, PL-20C, PL-20C-LOW, PL-20N-LOW), but you cannot run an application designed for any of the supported power line channels on a Micro Server designed for a twisted-pair free topology (TP/FT-10) channel, nor can you run a TP/FT-10 Micro Server on a PL-20 channel. The Micro Server can report the supported channel types through its Micro Server key, which is part of the reset notification message.
- In addition to matching transceiver families, the host application could require additional Micro Server features, such as support for the ISI protocol. These settings are also contained in the Micro Server's reset notification message, if applicable.
- The Micro Server and host application must support the same physical link-layer protocol (SCI or SPI). Unless the host processor controls the Micro Server's SBRB0 and SBRB1 input signals for bitrate selection, both sides' link-layer bit rates must match.

In addition, the new application will have certain requirements for the host environment, such as availability of memory or I/O resources, or the availability or version numbers of the embedded operating system, and so on. Your host application download protocol should include an appropriate mechanism to determine and verify these requirements before passing control to the new application.

In some cases, your host application download could require an upgrade to the Micro Server image at the same time as the upgrade of the host application. The following considerations apply for designing the dual-processor application download protocol:

- Because a complete and fully operational ShortStack device is required to run the host application download protocol, the host application download must be completed first.
- The application must not reset or initialize the Micro Server until the download process has been completed for both the host application and the Micro Server image.
- Because the Micro Server will also be updated in the process, some steps of the application verification process can or must be postponed. For example, the new host application might require a Micro Server key value that is correctly implemented by the new Micro Server image, but not the current one.
- After the successful download of the Micro Server image, the Micro Server resets and enters quiet mode until the entire device has been successfully initialized. While the Micro Server is in quiet mode, no network communication is possible with the device.
- After the new Micro Server resets (after loading its new application image), it sends a reset notification to the host application. This reset

notification reports the new Micro Server's capabilities and attributes, and indicates that an application initialization is required.

- After the host application has completed initialization, the host application download protocol must perform any previously postponed verification steps and pass control to the new host application, which in turn initializes the Micro Server.

Application Download Utility

This tool needs to read the application image to be loaded, and run the application download protocol described in *Custom Host Application Download Protocol* on page 318. You can write the utility as an LNS plug-in or as any type of network-aware application.

Download Capability within the Application

Your application must implement the custom application download 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 ShortStack host 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 ShortStack host 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 perform all necessary verification tasks, and prepare the image so that the boot loader can reboot the host 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 ShortStack 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).

See your host processor and operating system documentation about guidance, recommendations, and tools that support these tasks.

F

Glossary

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

C

configuration property

A data value used to configure the application program in a device.

D

downlink

Link-layer data transfer from the host to the Micro Server.

H

handshake

The communication across the link layer between the host serial driver and the ShortStack Micro Server that confirms readiness to receive a link-layer segment. For the serial driver, the handshake involves three or four control signals.

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 ShortStack link layer.

link-layer protocol

The protocol that is used for data exchange across the link layer.

link-layer segment

A part of a message sent across the link layer that requires a handshake between the host serial driver and the ShortStack Micro Server. Examples of a link-layer segment are: the link-layer header, the link-layer extended header, and the link-layer payload.

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 ISO/IEC 14908 (ANSI/CEA-709.1 and 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 ShortStack Micro Servers

Development tools, APIs, and firmware for developing LONWORKS devices that use the LonTalk Compact API and a ShortStack Micro Server; included with the ShortStack FX 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

network variable

A data item that a particular device application program 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). Examples are a temperature value, switch value, and actuator position setting.

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.

S

ShortStack application

An application for a LONWORKS device implemented with the LonTalk Compact API and a ShortStack Micro Server.

ShortStack Developer's Kit

Software required to develop LonTalk applications for any microcontroller or microprocessor. The kit includes software tools, examples, documentation, plus the LonTalk Compact API and ShortStack firmware.

ShortStack device

A LONWORKS device based on the LonTalk Compact API and a ShortStack Micro Server.

ShortStack Driver API

A portable C language hardware driver that encapsulates platform-dependent code for transferring data between a host processor and a ShortStack Micro Server.

ShortStack Firmware

Firmware for an Echelon Smart Transceiver that enables the Smart Transceiver to be used as a network interface by a ShortStack host processor.

ShortStack host processor

Any 8-, 16-, 32-, or 64-bit host microprocessor or microcontroller that is integrated with the LonTalk Compact API, ShortStack Driver API, and a ShortStack Micro Server to create a LONWORKS device.

ShortStack link layer

The physical connection and protocol used to attach a ShortStack host processor to a ShortStack Micro Server; the hardware interface is either an SCI or SPI serial interface.

ShortStack Micro Server

An Echelon Smart Transceiver running the ShortStack Firmware.

U

uplink

Link-layer data transfer from the Micro Server to the host.

Index

D

E

F

G

H

N

O

P

Q

R

S

T

U

V

W

X

www.echelon.com