ECHELON

IzoT® ShortStack® SDK User's Guide

Quickly add control networking and Internet accessibility to any device with a microprocessor or microcontroller

Version 4.30 078-0540-01B

Echelon Corporation
www.echelon.com

Welcome

Echelon's IzoT ShortStack® Software Development Kit (SDK) enables any product that contains a microprocessor or microcontroller to quickly and inexpensively become a networked and Internet connected smart device. The ShortStack Micro Server provides a simple way to add LonTalk/IP and LON networking to new or existing smart devices. The IzoT ShortStack SDK 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 networked device using Echelon's IzoT ShortStack SDK. It describes the architecture of a ShortStack device and how to develop a ShortStack device. To develop a ShortStack device you will interface the ShortStack Micro Server with your microprocessor, create your ShortStack serial driver and adapt the portable API code provided to your compiler and runtime environment. The application is created with definitions for interoperable network interface items such as datapoints and blocks using the *IzoT Markup Language* (IML), and using the LonTalk/IP Compact API functions to program your ShortStack application.

Audience

This document assumes that the reader has a good understanding of the IzoT platform and development of embedded devices.

What's New in the IzoT ShortStack SDK

The IzoT ShortStack SDK extends previous versions of the ShortStack SDK. It includes most features and functions of earlier versions, and features a new method for defining and managing the application's network interface.

Installation

The IzoT ShortStack SDK is available on a shared repository on GitHub at github.com/izot/shortstack. To obtain access to the private IzoT ShortStack SDK repository on GitHub, fill out the IzoT ShortStack SDK registration form at echelon.com/shortstack.

You can obtain source code for the API and examples from the shared repository, which also includes all required binaries and a number of optional utilities. This makes it easy to the integrate updates provided by Echelon.

IzoT Markup Language (IML)

The IzoT Markup Language replaces the Neuron C model file used by earlier versions of the ShortStack SDK. IML provides a new and easier method of defining and managing your application's network interface. See *Introduction to the IzoT Markup Language* for more information.

To define your application's network interface, you will include IML in the source files for your application. To implement the network interface you define with IML, you will pre-process your source code with the IzoT Interface Interpreter. The IzoT ShortStack SDK includes a Windows installer for the IzoT Interface Interpreter for Windows.

You can continue to use a Neuron C model file to define your network interface by using the LonTalk Interface Developer (LID) included with earlier versions of the ShortStack Developer's Kit.

Older Versions of ShortStack

The IzoT ShortStack SDK co-exists with installations of earlier releases of the ShortStack software. You can continue using older versions of ShortStack, and the LonTalk Interface Developer (LID), for existing projects.

You can update those projects to take advantage of the new IzoT Micro Server, see *Application migration from the ShortStack FX SDK to the IzoT ShortStack SDK* for more information.

New Hardware Support

The IzoT ShortStack SDK adds standard Micro Servers for the FT 6050 Smart Transceiver.

Other Neuron Chips and Smart Transceivers are also supported. Standard Micro Servers are included for various configurations of FT 3120, FT 3150, PL 3120, PL 3150, PL 3170, FT 5000, and FT 6050 processors, and you can use the IzoT NodeBuilder tool to create a custom Micro Server if your require a configuration not provided by one of the standard Micro Servers.

Micro Server Enhancements

All standard IzoT ShortStack SDK Micro Servers include enhancements for improved host synchronization and start-up. You must rebuild any custom Micro Servers that you created with earlier versions of the ShortStack tool by using the new IzoT ShortStack Micro Server core libraries to take advantage of these enhancements.

Improvements include a configurable post-reset delay to allow the host more time to re-synchronize the serial link layer driver, and improved validation of an important initialization message.

ShortStack LonTalk/IP Compact API

The ShortStack LonTalk/IP Compact application programming interface (API) includes new functions and callback handler functions, including a function to configure the IzoT ShortStack SDK Micro Servers' post-reset pause and including improved support for persistent storage of non-volatile data.

Some functions of the ShortStack FX LonTalk Compact API have been taken over by the IzoT Interface Interpreter, including the commonly used callback handlers.

Some new callback functions have been added, while some other callback prototypes have changed.

The API also supports an enhanced interface to the serial driver.

ShortStack FX LonTalk Compact API Compatibility

The IzoT ShortStack SDK LonTalk/IP Compact API is similar to the ShortStack FX LonTalk Compact API, but differs in some details of API functions and callback function prototypes.

Porting an existing ShortStack FX application to IzoT ShortStack requires some code changes. While these changes are discussed in *Application Migration From*

ShortStack FX to IzoT ShortStack, your compiler and linker will point-out where the required changes were not made correctly.

Porting an existing ShortStack FX application to the IzoT ShortStack SDK also requires a change from the Neuron C model file to the IzoT Markup Language (IML). This requires a different way to express your preferences, such as your selection of the Micro Server, and your selection of profiles or datapoints to implement. However, you can use one of the new IzoT ShortStack Micro Servers with existing ShortStack FX projects and the LonTalk Interface Developer. (Access to and control over some of the added features may be limited.)

Example Ports

The IzoT ShortStack SDK includes a set of simple example host applications based on a port of the ShortStack LonTalk/IP API and a driver for embedded Linux on a Raspberry Pi (or compatible) platform.

IzoT ShortStack SDK User's Guide

The *IzoT ShortStack SDK User's Guide* describes the IzoT Interface Interpreter and the IzoT Markup Language; detailed information on both can be found at echelon.com/docs/izot.

Related Documentation

The documentation for the IzoT ShortStack SDK includes this *IzoT ShortStack SDK User's Guide* which includes documentation on the *The ShortStack LonTalk/IP Compact API and Callback Handler Functions*.

The IzoT ShortStack SDK includes an example port for an embedded Linux platform, including an example implementation of the serial driver, API, and callback handlers, and several sample applications. You can access these example ports from the IzoT ShortStack SDK GitHub repository at github.com/izot/shortstack, where you can obtain the examples and optionally contribute enhancements to the IzoT ShortStack SDK community.

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

- *Series 6000 Chip Data Book* (005-0199-01C). This manual provides detailed specifications on the electrical interfaces, mechanical interfaces, and operating environment characteristics for the FT 6050 and FT 6010 Smart Transceivers, and the Neuron 6050 Processor.
- *Series 5000 Chip Data Book* (005-0199-01C). 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.
- *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.
- *PL 3120 / PL 3150 / PL 3170 Power Line Smart Transceiver Data Book* (005-0193-01C). 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.

- *Introduction to the LONWORKS Platform* (078-0183-01B). This manual provides an introduction to the ISO/IEC 14908-1 Control Network Protocol, and provides a high-level introduction to LONWORKS networks and the tools and components that are used for developing, installing, operating, and maintaining them.

- *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, lonmark.org.

- *IzoT Commissioning Tool User's Guide* (078-0509-01A). This manual describes how to use the IzoT Commissioning Tool to design, commission, monitor and control, maintain, and manage a network.

- *IzoT NodeBuilder® User's Guide* (078-0516-01A). This manual describes how to develop a LONWORKS device using the IzoT NodeBuilder tool.

  You use the IzoT NodeBuilder Tool to create a custom ShortStack Micro Server. See *Custom Micro Servers*, for more information about custom Micro Servers. Most ShortStack developers will not need to create a custom ShortStack Micro Server.

  You can also use the IzoT NodeBuilder Tool to create a model file for a ShortStack FX application using the LonTalk Interface Developer.

  You do not need IzoT NodeBuilder with IzoT ShortStack when using one of the standard IzoT ShortStack Micro Servers provided.

- *FT 5000 EVB Hardware Guide (078-0390-01B).* This manual describes how to develop, prototype, test and debug applications using the FT 5000 EVB.

- *FT 6000 EVB Hardware Guide (087-0504-01A).* This manual describes how to develop, prototype, test and debug applications using the FT 6000 EVB.

- *IzoT Markup Language* section *of the IzoT Manual* at echelon.com/docs/izot

The following useful documents are available in the IzoT ShortStack SDK repository:

- *Eclipse Configuration*. This document describes how to configure Eclipse Luna for the example applications. It can be found in the *docs* folder within the IzoT ShortStack SDK repository.
- *ShortStack Goes Raspberry Pi Wiring Instructions*. This document describes the wiring instructions between a Raspberry Pi Model B+ and an Echelon FT 6000 EVK. It can be found in the *example/rpi/doc* folder within the IzoT ShortStack SDK repository.

Table of Contents

1

Introduction to the IzoT ShortStack SDK

This chapter introduces the IzoT ShortStack SDK. It describes the architecture of a ShortStack device, the requirements and restrictions of an IzoT ShortStack Micro Server, and the IzoT ShortStack components that are available from Echelon.

Overview

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

The communications protocol used for LonTalk/IP networks is the ISO/IEC 14908-1 Control Network Protocol combined with IP. The ISO/IEC 14908-1 protocol is an international standard seven-layer protocol that has been optimized for control applications. The seven layers are described in **Table 1**

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

Layers 4 through 7 of the ISO/IEC 14908-1 protocol are implemented in the protocol stack for all LonTalk/IP devices. The services provided by these layers are called the *LonTalk/IP Control Services*. Layers 1 through 3 are media and link dependent.

For LonTalk/IP devices implemented as native ISO/IEC 14908-1 devices, Layers 2 and 3 are defined by ISO/IEC 14908-1, and Layer 1 is defined by other standards such as ISO/IEC 14908-2 for FT channels, ISO/IEC 14908-3 for PL channels, and ISO/IEC 14908-4 for IP-852 channels. As a result, LonTalk/IP devices implemented as native ISO/IEC 14908-1 devices are fully compatible with and interoperable with classic LON devices. LonTalk/IP devices support additional services such as UDP messaging, ICMP support, and SNMP support. These services are provided in a way that is compatible with classic LON messaging.

For native LonTalk/IP devices on Ethernet, Layers 1 through 3 are defined by the Ethernet and IP standards. For native LonTalk/IP devices on Wi-Fi, Layers 1 through 3 are defined by the Wi-Fi and IP standards.

You can use the IzoT ShortStack SDK to implement a LonTalk/IP device with an Echelon Series 6000 processor and a host processor of your choice. A device implemented with the IzoT ShortStack SDK and a Series 6000 processor is fully compatible with both LonTalk/IP and classic LON devices.

You can also use the IzoT ShortStack SDK to implement a classic LON device with an Echelon Series 5000 or Series 3100 processor and a host processor of your choice. A device implemented with the IzoT ShortStack SDK and a Series 5000 or Series 3100 processor is a classic LON device, and is fully compatible with both LonTalk/IP and classic LON devices.

IzoT Device Architectures

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

These components can be combined in a physical device. For example, you can use Echelon's FT 6050 Smart Transceiver as a single-chip solution that combines all four components in a single chip that communicates on a free topology (FT) twisted pair channel and implements the LonTalk/IP protocol. You can use Echelon's CPM 4200 Wi-Fi Module as a single module solution that combines all four components in a single module for Wi-Fi-based communication using the LonTalk/IP protocol. You can create a single-device solution with Echelon's IzoT Device Stack EX running on versatile embedded computers with a Linux operating system, such as the Raspberry Pi or Beaglebone Black, and communicate via Ethernet and the LonTalk/IP protocol.

The IzoT ShortStack SDK supports an IzoT device architecture where these components are split between two processors, a *host processor* that runs the device application, and a co-processor that implements the LonTalk/IP or LON protocol and provides the network interface. The co-processor is implemented with a Smart Transceiver or Neuron Chip running the ShortStack firmware. The

combination of the Smart Transceiver or Neuron Chip with the ShortStack firmware is called a ShortStack *Micro Server*. The Micro Server connects to your microcontroller through a synchronous (SPI) or asynchronous (SCI) serial link.

You can use one of many embedded operating systems or you may not use an operating system at all ("bare metal" design). The IzoT ShortStack SDK does not require an operating system on the target host, but the IzoT ShortStack example applications and driver are designed for use with a Linux operating system.

IzoT ShortStack SDK Application Development

For a ShortStack device, you write the application program in C or C++ using a common application framework and application programming interface (API). This API is called the *LonTalk/IP Compact API*. You select a suitable host processor and use the host processor's application development environment to develop the application.

The general architecture of a ShortStack device is shown in **Figure 1**. Because a ShortStack Micro Server can work with any host processor, you must provide the serial driver implementation for the host. The ShortStack software includes an example driver for the Raspberry Pi platform using the Raspbian Linux operating system.

To define the LonTalk/IP interface for your device, you embed markup in your code that is contained in C comments within one of your C or C++ source files. This markup is defined by the *IzoT Markup Language (IML)*.

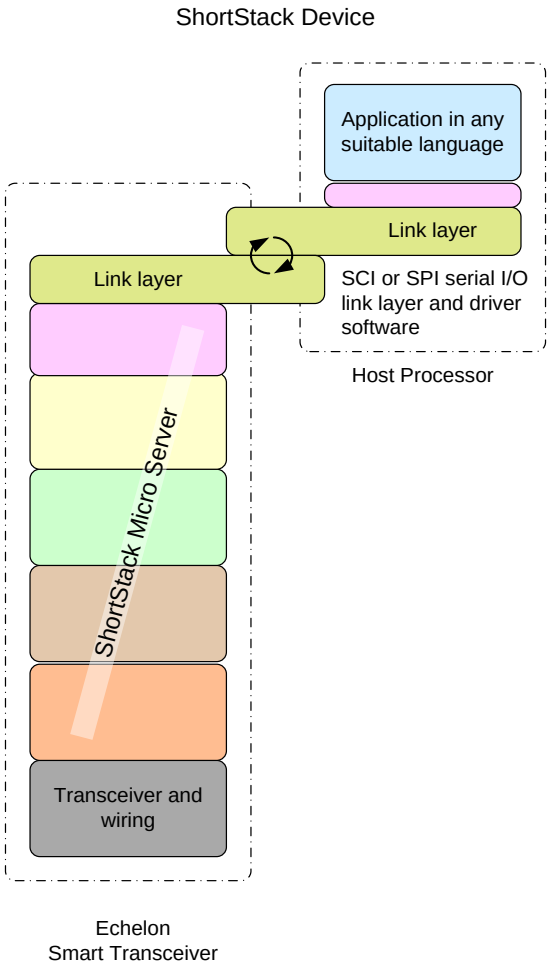


Figure 1. A ShortStack Device

Neuron Hosted and IzoT ShortStack Device Characteristics

Table 2 compares some of the key characteristics of the Neuron hosted and host-based solutions for LonTalk/IP and LON devices.

Table 2. Comparing Neuron Hosted and Host-Based Solutions

Characteristic	Neuron Hosted Solution	ShortStack Solution
Maximum number of network variables	254[1][2] or 62	254[1][2] or 62
Maximum size of network variable	225[3] or 31 bytes	225[3] or 31 bytes

Maximum number of NV config table entries	254[1][2] or 62	254[1][2] or 62
Maximum number of address table entries	254[2][3] or 15	254[2][3] or 15
Maximum number of aliases	127[1][2] or 62	127[1][2] or 62
Maximum number of dynamic network variables	0	0
Maximum size of application messages	228 bytes	228 bytes
Maximum number of receive transaction records	16	16
Maximum number of transmit transaction records	2	2
Support for the 14908-1 Extended Command Set	No	No
File access methods supported	LW-FTP [4], DMF[4]	LW-FTP[4], DMF [4]
Link-layer type	N/A	4- or 5-line SCI or 6- or 7-line SPI
Typical host API runtime footprint	N/A	5-6 KB code with 1 KB RAM (includes serial driver, but does not include optional API or ISI API) in a bare-metal target.
Host processor type	N/A	Most microprocessors or microcontrollers
Application development language	Neuron C	C or C++ with IML

Notes:

1. Neuron firmware version 16 or greater.
2. Dependant on available resources.
3. Series 6000 Smart Transceivers and Neuron Chips
4. The file access methods listed are:
    - Direct memory file (DMF); see *Supporting Direct Memory Files*
    - The LONWORKS file transfer protocol (LW-FTP); see the File Transfer engineering bulletin at echelon.com/docs

IzoT ShortStack SDK and CPM 4200 Wi-Fi SDK

The IzoT ShortStack SDK and CPM 4200 Wi-Fi SDK solutions are both built on the LonTalk/IP platform using the IzoT Interface Interpreter, and they share very similar declarations of the device's network interface and use similar API and application frameworks. This simplifies migrating applications from one solution to the other. In addition, you can create applications that share a common code base for devices that use both solutions.

Requirements and Restrictions for ShortStack

The ShortStack LonTalk/IP 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 in a bare metal design.The API does not require additional non-volatile memory, but most applications implement properties and require persistent, modifyable storage of such data. The API does not require an operating system—memory requirements may be higher if the host includes an operating system.

The ShortStack firmware requires a Smart Transceiver or Neuron Chip with a minimum of 4 KB of application memory and 2 KB of RAM. The IzoT ShortStack SDK includes a variety of standard Micro Server images, which support FT 3120, FT 3150, FT 5000, FT 6050, PL 3120, PL 3150, and PL 3170 Smart Transceivers in various configurations. You can create a custom Micro Server to support other chips and hardware configurations

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 availability and speed of the interface depends on the type of serial interface, the clock speed of the ShortStack Micro Server, and the specific processor used for the Micro Server:

- The highest bit rate for the SCI interface is approximately 1.2 Mbps for a ShortStack Micro Server running on an FT 6050 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 6050 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* and *Setting the SPI Bit Rate*.

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.

Network Installation

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 IzoT Commissioning Tool (CT) or another tool that can install and monitor LonTalk/IP or LON devices. See the *IzoT Commissioning Tool User's Guide* for more information about IzoT CT. However, if your ShortStack device supports the Interoperable Self-Installation (ISI) protocol, a network management tool is not required.

For network diagnostics and troubleshooting, you can use the Wireshark network protocol analyzer. The Wireshark network protocol analyzer collects and displays

low-level protocol packets, and often provides important diagnostics. See www.wireshark.org for more information.

For ShortStack device development a logic analyzer, serial communications analyzer, or digital storage oscilloscope is also useful for diagnosing and troubleshooting the serial communication between the Micro Server and your host processor.

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/IP Compact API for the host processor
4. A ShortStack application that uses the ShortStack LonTalk/IP Compact API

Figure 2 shows the basic software architecture of a ShortStack device.

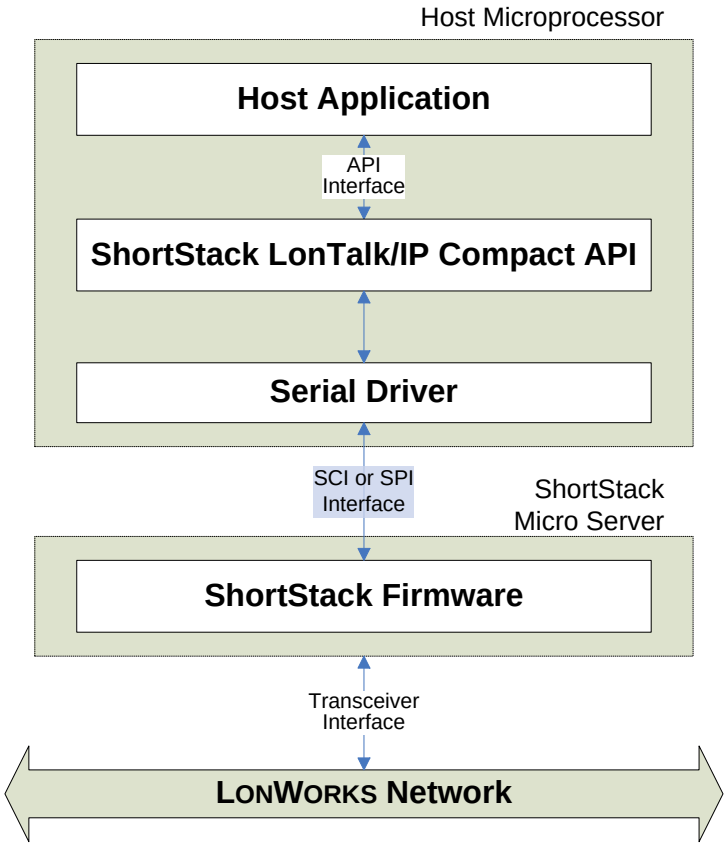


Figure 2. ShortStack Architecture

The ShortStack Serial Driver

The ShortStack serial driver provides the hardware-specific interface between the LonTalk/IP Compact API and ShortStack Micro Server. If you use a

standard operating system, the serial driver itself may be portable across different platforms, as is the case with the driver example included with the example for the Raspberry Pi computer and the Raspbian Linux operating system.

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, typically derived from the example driver for the Raspberry Pi. You can use or modify the example driver, or create your own driver for a different processor or operating system.

SCI Architecture

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) as shown in Figure 3. You can use standard UART or USART hardware to implement this link.

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

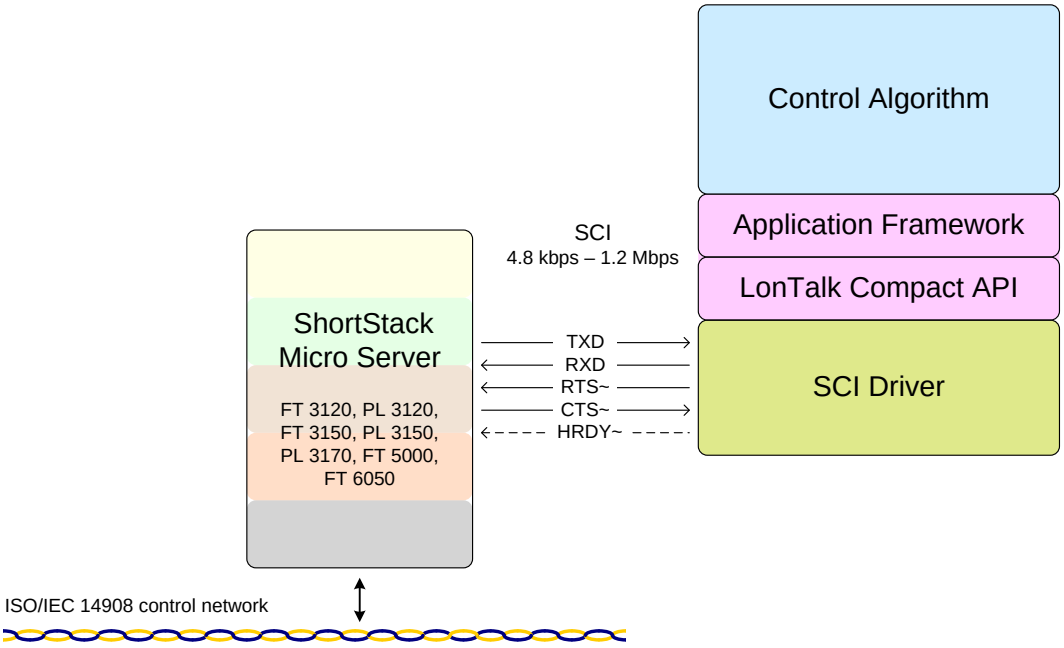


Figure 3. SCI Architecture for a ShortStack Device

SPI Architecture

The SPI interface is a half-duplex synchronous serial interface, where the Micro Server acts as the master, as shown in Figure 4. Most ShortStack devices use the SCI interface because of the need for fewer I/O lines for the asynchronous link, and because the requirements for the SPI driver are more complex. The SPI interface is useful if all SCI resources on the host processor are already in use or if an SPI interface is more readily available on the host processor.

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

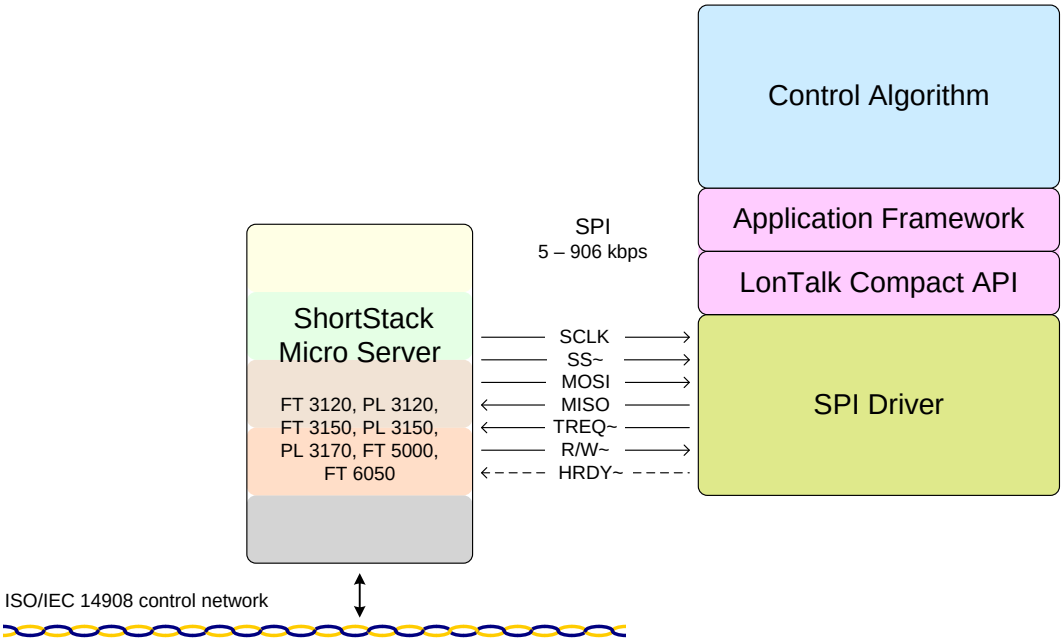


Figure 4. SPI Architecture for a ShortStack Device

The ShortStack LonTalk/IP Compact API

The IzoT ShortStack SDK includes source code for the ShortStack LonTalk/IP 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 LonTalk/IP or LON network. The API code is written in ANSI C. You must port the code for your host processor.

The ShortStack LonTalk/IP 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.
- Event dispatchers for common events, such as those signaling the arrival of network variable data or an error in the propagation of an application message.
- Callback handler functions for advanced and less common events.
- 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**.

Table 3. Tasks for Developing a ShortStack Device

Task	Additional Considerations	Reference
Obtain the IzoT ShortStack SDK and become familiar with it		*Getting Started with the IzoT ShortStack SDK*
Select hardware for the ShortStack Micro Server and prepare it by loading the ShortStack firmware into it	You need to 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.	*Selecting and Creating a ShortStack MicroServer* *Selecting the Host Processor*
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.	*Designing the Hardware Interface*
Create the serial driver for the host processor	You need to 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.	*Creating a ShortStack Serial Driver*
Port the ShortStack LonTalk/IP Compact API to the host processor	You need to port the ShortStack LonTalk/IP 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.	*Porting the ShortStack LonTalk/IP Compact API* *Appendix A ShortStack LonTalk/IP Compact API*
Select and define the functional profiles and data types for your device using tools such as the IzoT Resource Editor and the SNVT and SCPT Master List	You need to select profiles and data types for use in the device's network 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.	*IzoT Markup Language* section *of* the *IzoT Manual* at echelon.com/docs/izot

Task	Additional Considerations	Reference
Structure the layout and network interface of your ShortStack device by declaring the required blocks, datapoints, and properties with the IzoT Markup Language.	Define the network interface for your device using standard C source code and the IzoT Markup Language.	*IzoT Markup Language* section *of the IzoT Manual* at echelon.com/docs/izot
Integrate the IzoT Interface Interpreter into your development tool's work flow.	Run the IzoT Interface Interpreter prior to build your project within your host development tool. You can include the IzoT Interface Interpreter as an integral step to your build process, making generation and maintenance of your application framework completely transparent.	*Integrating the IzoT Interface Interpreter*
Complete event handlers and the ShortStack LonTalk/IP Compact API callback handler functions to process application-specific network events	Complete your event handlers to connect your application's algorithm with the IzoT network. You can also implement advanced callback handlers for less frequently used events.	*Developing a ShortStack Application* Appendix A, *ShortStack LonTalk/IP Compact API*
Modify your application to interface with an IzoT network by using the ShortStack LonTalk/IP Compact API function calls	You need to 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.	*Developing a ShortStack Application* Appendix A, *ShortStack LonTalk/IP Compact API*
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, but can make your device significantly easier to install without the use of an installation tool.	*Developing a ShortStack Application with ISI* Appendix B, LonTalk/IP ISI API

Task	Additional Considerations	Reference
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.	*Custom Micro Servers*
Test, install, and integrate your ShortStack device using self-installation or a LonTalk/IP or LON network tool such as the IzoT Commissioning Tool		*IzoT Commissioning Tool User's Guide*

2

Getting Started with the IzoT ShortStack SDK

This chapter describes the IzoT ShortStack SDK and how to install it.

IzoT ShortStack SDK Overview

The IzoT ShortStack SDK is a software toolkit that contains software tools, the ShortStack LonTalk/IP Compact API, LonTalk/IP ISI API, ShortStack firmware, and documentation needed for developing applications for any microcontroller or microprocessor that uses a ShortStack Micro Server to communicate with a LonTalk/IP or LON network. You can use the software with ShortStack Micro Servers that use an Echelon Series 6000, Series 5000, or Series 3100 Smart Transceiver or an Echelon Neuron 6050 or Neuron 5000 Processor.

The kit includes the following components:

1. Portable ANSI C source code for the ShortStack LonTalk/IP Compact API and LonTalk/IP 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 6050, 5000, 3170, 3150, and 3120 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 IzoT Interface Interpreter. The IzoT Interface Interpreter translates your C source code that you have annotated with IML into device interface data and device interface files that simplify the implementation of your ShortStack application, and 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/IP Compact API and LonTalk/IP ISI API.
6. Source code for an example port for the Raspberry Pi.

Installing the IzoT ShortStack SDK

You can install the IzoT ShortStack SDK on any computer that runs Microsoft Windows 10, Windows 8, Windows 7, or Windows XP.

To install the IzoT ShortStack SDK, which is available free of charge, perform the following steps:

1. If you do not have an account with GitHub, register for a free account at github.com
2. Click the **Download Now** button at echelon.com/shortstack to register for the IzoT ShortStack SDK. Enter your GitHub account name in the registration form.
3. Wait until you receive a notification e-mail from GitHub, confirming that you have been given access to the IzoT ShortStack repository.
4. Visit the IzoT ShortStack SDK repository at github.com/izot/shortstack to view and clone the repository.

Clone the repository into your local shortstack project folder. Do not clone it to a location within the LONWORKS folder. Use one of your user locations instead, such as **My Documents\izot-shortstack**. You need to have full read and write access to that location.

5. Locate the IzoT Interface Interpreter installer in the *install* folder within the repository, and install the IzoT Interface Interpreter. The installer is named **iii***VVV***.exe**, where *VVV* is a three-digit version number.

   Run the IzoT Interface Interpreter installer on your Windows computer.

6. Explore the repository.

   A good place to start is the Simple application example for Raspberry Pi computers using the Raspbian Linux operating system. This example is located in the **example/rpi/simple** folder within your repository, and is accompanied by wiring instructions in the **example/rpi/doc** folder.

   The examples assume, but do not require, that you use Eclipse Luna. The *doc* folder within the IzoT ShortStack SDK repository contains details about the Eclipse workspace and project configuration assumed by the IzoT ShortStack SDK examples.

7. Install the IzoT Resource Editor. You can obtain the IzoT Resource Editor from the download section at echelon.com/shortstack.

   You can use the IzoT Resource Editor to view standard profiles and data types, and to create and edit your custom profiles and data types. The IzoT Resource Editor also includes a conversion tool to translate your custom resource definitions into IzoT resource packages. The IzoT Interface Interpreter only accepts resources in the IzoT resource package format.

   The IzoT Interface Interpreter already includes all standard and IoT resource definitions. You need to use the IzoT Resource Editor and the conversion tool to the IzoT resource package format only for your user-defined resources.

ShortStack LonTalk/IP Compact API Files

The ShortStack LonTalk/IP Compact API is provided as a set of portable ANSI C files, which are listed in **Table 4**. These files are contained in the [*ShortStack*]**\api** directory (where [*ShortStack*] is the directory in which you cloned the IzoT ShortStack SDK repository.

You must port the API to your host processor; for more information about porting the API, see Porting the ShortStack LonTalk/IP Compact API.

Table 4. ShortStack LonTalk/IP Compact API Files

File Name	Description
LonBegin.h **LonEnd.h**	Optional definitions for implementing data packing and alignment preferences
Ldv.h	Definition of the driver API functions. These are used by the ShortStack API, but you need to supply the implementations of these functions. See Creating a ShortStack Serial Driver.
LonPlatform.h	Definitions for adjusting the ShortStack LonTalk/IP Compact API to your compiler and environment. Definitions for several common compilers are provided, but you need to review and possibly add definitions to match your toolchain.
ShortStackApi.c **ShortStackApi.h**	Function definitions for the ShortStack LonTalk/IP Compact API
ShortStackHandlers.c	Function definitions for the ShortStack callback functions
ShortStackInternal.c	Internal functions and utilities that are used by the ShortStack LonTalk/IP Compact API, but not exported to the host application
ShortStackIsiApi.c **ShortStackIsiApi.h**	Function definitions for the LonTalk/IP ISI API
ShortStackIsiHandlers.c	Function definitions for the ShortStack ISI callback handler functions
ShortStackIsiInternal.c	Internal functions and utilities that are used by the LonTalk/IP 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 Micro Server Firmware Images

The IzoT ShortStack SDK includes several standard ShortStack Micro Server firmware images provided as pre-compiled image files that you can program into serial flash memory chips for use with FT 6050 Smart Transceivers, into serial EEPROM memory chips for FT 5000 Smart Transceivers, into on-chip memory

for FT or PL 3120 Smart Transceivers or PL 3170 Smart Transceivers, or into flash memory chips to be used with FT or PL 3150 Smart Transceiversor.

You can use the ShortStack Micro Server only with an Echelon Smart Transceiver or an Echelon Neuron Chip. 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 IzoT Interface Interpreter or the LonTalk Interface Developer
- For 3120 and 3170 devices, an NFI file for a programmer device or in-circuit programmer for programming a Smart Transceiver
- For 3150 devices, an NEI file for a universal chip programmer or in-circuit programmer for programming a flash memory chip.
- For Series 5000 devices, an NME file for a universal chip programmer or in-circuit programmer for programming a serial EEPROM memory chip
- For Series 6000 devices, an NMF file for a universal chip programmer or in-circuit programmer for programming a serial flash 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 LonTalk/IP ISI API; see *Developing a ShortStack Application with ISI*, 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).

When using the IzoT Interface Interpreter, you need to provide the set of image files for your selected Micro Server, and describe your choice of Micro Server in your application's C source code.

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* for a description of the firmware file naming convention.

All standard ShortStack Micro Server images are located within the [*ShortStack*]**/microserver/standard** folder within your ShortStack project folder

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
FT 6050	TP/FT-10	20	21	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 and 6000 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 *Supporting Direct Memory Files*
   - The LONWORKS file transfer protocol (LW-FTP); see the File Transfer engineering bulletin at echelon.com/docs
   - Configuration network variables (CNVs); see *Persistent NVs*
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.

You can create a custom Micro Server image to support a combination of hardware, channel type, and ISI features that is not supported by the pre-compiled Micro Server images. Specifically, you must create a custom Micro Server image:

- If your device uses a different Echelon Smart Transceiver or Neuron Chip than the ones listed in **Table 5** (such as a Neuron 6050 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*.
- If your Micro Server device requires ISI-DAS support, or a different level of ISI support.
- If you require an application-specific custom Micro Server that supports 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 *Supporting Direct Memory Files* for more information.
- If your device requires a Micro Server with different properties than those used for the standard Micro Server images, such as the buffer configuration or maximum number of addresses, aliases or network variables.

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

3

Introduction to the IzoT Markup Language

This chapter introduces the IzoT Markup Language (IML) and provides code examples using IML.

Overview

Your program interacts with the LonTalk/IP or LON network and other devices through input and output *datapoints*, which implement *network variables* for dynamic, non-persistent, data, or *properties* for persistent configuration data which changes infrequently.

A datapoint is an extension of a simple network variable; a datapoint contains the network variable. The network variable is the datum which receives an update from the network, or which can be propagated onto the network. In addition to the network variable, the datapoint also contains the corresponding **global_index** and can be associated with **onUpdate** and **onComplete** events.

Multiple datapoints are typically combined into logical units called **blocks**. A block is an implementation of a profile, and defines a particular functionality. For example, a block implementing the standard **SFPTrealTimeKeeper** profile consists of one mandatory output datapoint (reporting current date and time), and a number of optional network variables and properties. For example, the optional **nciUpdateRate** property can be implemented to allow configuration of the rate by which the date and time information is transmitted on the network.

The IzoT ShortStack SDK includes a large selection of standard network variable and property data types and profile definitions for a multitude of different applications. You can also create and use your own definitions as *user-defined* data types and profiles.

The set of datapoints, properties, and profiles implemented by your program, along with related attributes, is collectively known as the *device interface*. You will specify and implement the device interface within your source code. The IzoT Interface Interpreter (III) produces a framework which interacts with the ShortStack API to exchange data and other events with the control network. The IzoT Interface Interpreter also generates configuration data required to configure the Micro Server.

Your definitions of blocks, datapoints and properties alongside your specification of related preferences such as the selection of the Micro Server type or the inclusion of optional runtime features, all occur within your C source code.

The basic principle of the IzoT Markup Language (IML) is a simple *as-if* paradigm: when editing your C source code for your program, you declare IzoT interface items *as if* your standard C or C++ compiler knew what those meant, and you annotate your declaration with a special tag. Your C compiler ignores the tag because it appears within a C comment, but when you execute the IzoT Interface Interpreter in a pre-compilation step, the IzoT Interface Interpreter recognizes the tag and creates the required framework, ready for submission to your compiler.

This chapter introduces IML. See the *IzoT Markup Language* section *of* the *IzoT Manual* at echelon.com/docs/izot for more information.

Selected Introductory Examples

The following example implements the standard **SFPTpressureSensor** profile and all its mandatory member network variables and properties within your program's C source code.

Example 1

This example implements a single SFPTpressureSensor block.

```
#include “ShortStackDev.h”

SFPTpressureSensor(pressure) pressure; //@IzoT block
```

The **//@IzoT block** tag specifies a *block*, with the profile specified by the variable declared to its left. When this code is processed by the IzoT Interface Interpreter, it generates a standard C type (typedef) within the **ShortStackDev.h** file, which your code must include.

Example 2

This example implements an array of two blocks, both based on the standard **SFPTpressureSensor** profile:

```
#include “ShortStackDev.h”

SFPTpressureSensor(pressure) pressure[2]; //@IzoT block
```

Example 3

This example adds a completion update to monitor success or failure of transactions in relation to the block’s principal **nvoPress** output, and adds an optional *nciPressOffset1* property, which is defined by the profile to apply to the block’s *nvoPress* output.

```
#include “ShortStackDev.h”

SFPTpressureSensor(pressure) pressure[2]; //@IzoT block \
//@IzoT onComplete(nvoPress, onPressureCompletion), \
//@IzoT implement(nvoPress.nciPressOffset1, init=3.5) // kPa
```

Continuation lines of IML declarations must begin with the //@IzoT tag to prevent your C or C++ compiler from processing these.

Example 4

This example implements the **onPressureCompletion** event handler. The event handler must not be static, and must meet the event’s specific prototype.

```
void onPressureCompletion(
    const unsigned index, const LonBool success
)
{
    if (index == pressure[0].nvoPress.global_index) {
```

```
        … track success or failure for your first pressure sensor
…
    } else {
        … and the same for the second …
    }
}
```

Example 5

This example implements two blocks of the **SFPTpressureSensor** profile, one with, and one without, the optional **nciPressOffset1** property:

```
#include "ShortStackDev.h"

SFPTpressureSensor(a) simple; //@IzoT block
SFPTpressureSensor(b) configurable; //@Izot block \
//@IzoT implement(nvoPress.nciPressOffset1, init=3.5) // kPa
```

Example 5 implements the **simple** and **configurable** blocks. Both implement the standard **SFPTpressureSensor** profile, but only **configurable** implements the optional **nvoPressOffset1** property. This difference leads to different C data types used for the **simple** and **configurable** variables; one to include, one not to include, the optional property.

Each generated profile and type name must be unique. In Example 5, there are two instances of the **SFPTpressureSensor** profile. To make the profile name unique for repeated instances, you must add an IML *type name modifier*. In Example 5, the type name modifiers are shown as **a** and **b**.

The type name modifier acts as an appendix to the name of the generated type. These are declared in **ShortStackDev.h** and will be **SFPTpressureSensor__a** and **SFPTpressureSensor__b** in this example, but you can reference them as **SFPTpressureSensor(a)** and **SFPTpressureSensor(b)** in your program.

Examples 1 to 3 use **pressure** for the type name modifier. The technical requirement is that the type name modifier must be a valid part of an identifier and, when combined with the resource type name, must yield a type name unique within your program. You can use the name of the variable declared, as shown in Examples 1 through 3, as a simple way to meet this requirement.

IzoT Interface Interpreter also generates **ShortStackDev.c**, which you must compile and link with your program. This file implements the code to initialize your blocks, datapoints and properties, and all data and functions necessary to configure the LonTalk/IP Compact API and ShortStack Micro Server.

The IzoT Interface Interpreter also generates an interface file with a XIF file name extension, which shares the base name of your C source file. The XIF file is useful when integrating your device in a managed network of devices.

The Simple Application Example

This following is a commented review of the complete **simple** application example that can be found in the **example/rpi/simple** folder within the IzoT ShortStack repository.

Part 1: Include files and global preferences

```
#include "ShortStackDev.h"
#include "ShortStackApi.h"
#include "ldv.h"

//@IzoT Option target("shortstack-classic")
//@IzoT Option programId("9F:FF:FF:08:16:01:04:00")
//@IzoT Option \
//@IzoT Server \
//@IzoT ("../../../microserver/standard/SS430_FT6050_SYS20000kHz")
```

Part 2: Interface definition

The application implements two blocks based on the standard closed loop actuator profile, each implementing an input and output pair of network variables. This profile does not stipulate a particular data type for these network variables. In this example, the standard **SNVT_volt** type is used.

Because the application implements more than one block, it is also required to implement a standard Node Object block. The Node Object must be declared first, and provides general diagnostics and housekeeping services.

```
SFPTnodeObject(node) nodeObject; //@IzoT block \
//@IzoT external("nodeObject"), \
//@IzoT onUpdate(nviRequest, onNviRequest),\
//@IzoT implement(nciLocation), implement(nciNetConfig), \
//@IzoT implement(nciDevMajVer, init=1), \
//@IzoT implement(nciDevMinVer, init=0)

SFPTclosedLoopActuator(volt, SNVT_volt) driver[2]; //@izot block \
//@IzoT external("volts"), \
//@IzoT onUpdate(nviValue, onDriverUpdate), \
//@IzoT implement(nciLocation, init="room 101")
```

Part 3: Minimum Node Object behavior

The **onNviRequest** update event handler, associated with the Node Object's **nviRequest** input, implements the Node Object's behavior. The following code shows the minimum behavior required.

Most applications implement additional Node Object features such as support for disable, override, or self-test functionality. The source code within the repository contains more comments to guide you in the process of supporting those.

```
void onNviRequest(
    const unsigned index,
    const LonReceiveAddress* const pSourceAddress
)
{
    uint32_t flags = LON_GET_UNSIGNED_DOUBLEWORD(
        nodeObject.nvoStatus.data.flags
    );
    uint16_t object_id = LON_GET_UNSIGNED_WORD(
        nodeObject.nviRequest.data.object_id
    );
    object_request_t request =
nodeObject.nviRequest.data.object_request;

    flags &= ~(ST_REPORT_MASK | ST_INVALID_ID |
ST_INVALID_REQUEST);
```

```
    if (object_id >= LON_FB_COUNT) {
        object_id = 0;
        flags |= ST_INVALID_ID;
    } else if (request ==
            RQ_REPORT_MASK) {
        flags = ST_REPORT_MASK | ST_INVALID_ID |
ST_INVALID_REQUEST;
    } else if (request == RQ_NORMAL) {
        flags = 0;
    } else if (request ==
            RQ_UPDATE_STATUS) {
        flags = 0;
    } else if (request ==
            RQ_CLEAR_STATUS) {
        flags = 0;
    } else {
        flags = ST_INVALID_REQUEST;
    }

    LON_SET_UNSIGNED_WORD(
        nodeObject.nvoStatus.data.object_id, object_id
    );
    LON_SET_UNSIGNED_DOUBLEWORD(
        nodeObject.nvoStatus.data.flags, flags
    );
}
```

Part 4: Application algorithm.

This simple application implements a trivial algorithm: when one of the volt inputs receives an update, the application adds 3 to its value, assigns the result to the corresponding output, and propagates the output.

Trivial as this might be, this is a good way to start with IzoT ShortStack. You can use a network tool such as the *NodeUtil* command-line utility to write a value to the input and observe the computed value in the output. This confirms a complete round trip from your network tool though the Micro Server and the link layer into your **onDriverUpdate** event handler, and the return trip back down through the link layer and Micro Server and back to your tool.

```
void onDriverUpdate(
    const unsigned index,
    const LonReceiveAddress* const pSourceAddress
)
{
    for (int i = 0; i < sizeof(driver) / sizeof(driver[0]); ++i) {
        if (index == driver[i].nviValue.global_index) {
            LON_SET_UNSIGNED_WORD(
                driver[i].nvoValueFb.data,
                3 + LON_GET_UNSIGNED_WORD(driver[i].nviValue.data)
            );
            LonPropagateNv(driver[i].nvoValueFb.global_index);
            break;
        }
    }
}
```

Your device must be in the online and configured state in order to receive input data. You can use a network tool or the NodeUtil console utility to set this state.

Part 5: Driver configuration

Your serial driver defines an **LdvCtrl** control block type for optional driver configuration data. The example driver for Raspberry Pi uses this to be informed of serial device selection and GPIO pin assignments for the link layer control signals.

The LonTalk/IP Compact API makes no assumptions on the **LdvCtrl** type, but simply passes this through to your driver.

```
static LdvCtrl ldvCtrl = {
    ”/dev/ttyAMA0”, 38400,
    { 10, 9, 11 } // RTS, CTS, HRDY GPIO pins
};
```

Part 6: The main() function

```
int main(int argc, char* argv[], char* env[])
{
    LonApiError sts = LonInit(&ldvCtrl);

    while (sts == LonApiNoError) {
        LonEventHandler();
    }

    LonExit();
    return LonApiNoError;
}
```

4

Integrating the IzoT Interface Interpreter

This chapter discusses how you can integrate the IzoT Interface Interpreter with your build process or development environment.

Input and Output Files

The IzoT Interface Interpreter consumes input from one of your project's C or C++ source files. This is called the IML source file for the purpose of this discussion, however, the IML source file is a regular C source file with IML contained within C comments. The IML source file must be processed by the IzoT Interface Interpreter prior to processing the file with your C or C++ compiler.

The IzoT Interface Interpreter generates the following output files:

- **ShortStackDev.h** must be included in all your application source code which uses the ShortStack API or accesses portions of the device application's network interface, such as datapoints or blocks. The IML source file must also include the **ShortStackDev.h** file.
- **ShortStackDev.c** includes all code generated by the IzoT Interface Interpreter. This file must be compiled and linked with your application after the IzoT Interface Interpreter finished with success.
- A device interface file is also generated. This file shares the basename of the IML source file and carries a .XIF file extension. The .XIF file is used for integration with some network tools, and plays no role in the C compilation and link process.

Pre-build Step or Script Method

The easiest method to integrate the IzoT Interface Interpreter with your development environment is to configure the interface interpreter as a pre-build step, for example, when using a script-driven build process. Many integrated development environments such as the Eclipse IDE also support user-defined pre-build steps.

To launch the IzoT Interface Interpreter as a pre-build step, configure your build steps to point to **iii.exe** and execute with the **--target shortstack-classic** option, followed by the path to the IML source file, as shown in the following example:

```
iii.exe --target shortstack-classic main.c
```

The Interface Interpreter generates output in the location of the IML source file.

The IzoT Interface Interpreter is installed into the **bin** sub-folder of your LONWORKS folder, **C:\Program Files (x86)\LonWorks** by default for 64-bit versions of Windows.

Makefile Method

Projects using explicitly declared makefiles (or similar build managers) can link **ShortStackDev.o** with the other object files, but use special rules to implement a make-driven pre-compilation execution of the IzoT Interface Interpreter.

The principle of operation is outlined in the following sketch:

```
all: pre-build main-build
main-build: myapp.elf

# III output
pre-build: rm ShortStackDev.c ShortStackDev.h myapp.xif

myapp.elf: ShortStackDev.o $(OTHER_OBJS)
      # link

ShortStackDev.o: ShortStackDev.c
      # compile

# IML source
ShortStackDev.c:
      iii.exe --target shortstack-classic myapp.c

%.o: %.c
```

compileOutput Files in Detail

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

- **ShortStackDev.h**
- **ShortStackDev.c**
- ***main*.xif**

Together, the generated files and the files for API and driver code form the ShortStack application framework, which includes everything you need to begin device development.

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.

You do not normally need to edit any of the generated files.

The following sections describe the generated files.

ShortStackDev.h

The **ShortStackDev.h** file is the main header file that the IzoT Interface Interpreter produces. This file provides the definitions that are required for your application code to interface with the application framework and the ShortStack LonTalk/IP Compact API, including C **extern** references to public functions, variables, and constants generated by the IzoT Interface Interpreter.

Include this file with all source files that make your application. You do not normally have to edit this file. Any manual changes to this file are not retained when you re-run the IzoT Interface Interpreter. 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 IzoT Interface Interpreter produces. This file includes the **ShortStackDev.h** file header file, declares the network variables, configuration properties, and blocks, event dispatchers and initialization functions.

It defines variables and constants, including the network variable table and 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 6 describes the fields of this data structure.

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 6. Fields of the appInitData Structure

Field	Description
appInitData.signature	A 16-bit number that identifies the current application. The IzoT Interface Interpreter 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. The example implementations also use the signature to sign persistent data to ensure the applicability of this data when loaded into the application at startup.
appInitData.programId	The 48-bit program ID in binary form.

Field	Description
appInitData.communication	The 96-bit communication parameter record that is used to correctly initialize communications with the LonTalk/IP or LON network. This data is optional in the IzoT ShortStack SDK, and normally set to all zeroes. All-zero communication parameters have no effect, causing the Micro Server to use its default parameters. A new **LonCustomCommunicationParameters()** callback is supported in **ShortStackHandlers.c** which allows advanced applications to select a communication parameter record for use with the Micro Server. This is controlled with the 0x40 flag in the preferences byte.
appInitData.preferences	An 8-bit vector of flags. Includes 0x20 to enable explicit addressing, 0x40 to select the Micro Server's default communication parameters, and a 5-bit value for the service-pin-held delay in seconds (mask 0x1F), where zero disables the feature. The remaining flag 0x80 is reserved for future use, and should be kept cleared (zero).
appInitData.nvCount	One byte for the total number of network variables in the application. This number should 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 includes the following flags: priority (0x80), output (0x40), service type (acknowledged [0x00], repeated [0x10], unacknowledged [0x20]), and authenticated (0x08).

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 IzoT Interface Interpreter, but the file contains comments about how you can override some of the preferences and assumptions made by the utility.

main.xif

The IzoT Interface Interpreter generates the device interface file for your project in the XIF format.

For this file, *main* is the name of the C source file which is processed by the IzoT Interface Interpreter.

These files comply with the LONMARK device interface format.

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

5

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 IzoT ShortStack SDK, and how to load them into a Smart Transceiver or Neuron Chip.

Overview

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*.

Selecting the Micro Server Hardware

You can build a ShortStack Micro Server using any Echelon Smart Transceiver or Neuron Chip. The Echelon Smart Transceiver integrate a Neuron core with a free topology (FT) twisted pair transceiver or a power line (PL) transceiver. The Echelon FT Smart Transceivers include the FT 6050, FT 6010, FT 5000, FT 3150, and FT 3120. The Echelon PL Smart Transceivers include the PL 3170, PL 3150, and PL 3120. The Echelon Neuron Chips include a Neuron core and do not include an integrated transceiver. The Neuron Chips can be used with other transceivers such as RS-485 transceivers. The Echelon Neuron Chips include the Neuron 6050 and the Neuron 5000.

You can develop custom hardware using an Echelon Smart Transceiver or Neuron Chip. For device evaluation and development with the Smart Transceivers, you can use the Echelon FT 6000 EVB or FT 5000 EVB evaluation boards, which include optional Electronic Industries Alliance (EIA) standard RS-232-C level shifters, jumpers, and I/O connectors that you can use to prototype a ShortStack interface to a host with an RS-232 interface.

More information about Echelon's evaluation boards is available from the Echelon Web site at echelon.com.

If you are developing a new FT device that must be compatible with existing LON FT devices, the FT 6050 provides the best performance and lowest implementation cost, and requires the smallest board space. If you are developing a new FT device that does not require interoperability with existing LON FT devices, the FT 6010 provides the best performance and lowest implementation cost, and requires the smallest board space. If you are developing a new PL device, the PL 3120 provides the best performance and lowest implementation cost for applications that do not require a large number of network variables. For applications that require many network variables, the PL 3150 provides the best performance and lowest implementation cost. If you are developing a new device that will not be using an FT or PL interface, the Neuron 6050 provides the best performance and lowest implementation cost, and requires the smallest board space.

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 6000 or 5000 devices, the clock rate is determined by the internal system clock rate. You can specify a Series 6000 or 5000 device's internal system clock rate within the device's hardware template when you create a custom Micro Server. For Series 3100 devices, the clock rate is determined by the external crystal or oscillator. 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 6050 or 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* 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 Series 6000 Smart Transceivers use an auto-tuning link algorithm which automatically configures the effective memory map to meet the application requirements. 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 6050, 5000, or 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 6050, FT 5000, FT 3150, or PL 3150 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 *Supporting Direct Memory Files* for more information.

For FT 5000 devices, the standard ShortStack Micro Server images require an SPI or I²C EEPROM or SPI flash 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 *Supporting Direct Memory Files* for more information.

For devices based on a Series 6000 chip such as the FT 6050 Smart Transceiver, a serial flash memory part is required Echelon's *Series 6000 Chip Data Book* provides detailed specifications on the electrical interfaces, mechanical interfaces, and operating characteristics for the FT 6050 Smart Transceiver, FT 6010 Smart Transceiver, and Neuron 6050 Processor.

Preparing the ShortStack Micro Server

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

You typically load the Micro Server only once. 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.

After you load a new Micro Server image, the first 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 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 the an earlier version of the ShortStack Developer's Kit to the IzoT ShortStack SDK 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.

The serial link layer communications used with IzoT Shortstack are compatible with ShortStack FX.

Table 7 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* for a description of the Micro Server image file extensions and file naming convention.

Table 7. 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 IzoT Commissioning Tool
		NFI	PROM programmer	A universal programmer, such as one from BPM Microsystems or HiLo Systems
FT 3150 or PL 3150 Smart Transceiver	Off-chip flash	APB or NDL	Network management tool	NodeLoad utility IzoT Commissioning Tool
		NEI	Universal chip programmer or in-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 IzoT Commissioning 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
FT 6050 Smart Transceiver	Off-chip flash memory	APB or NDL	Network management tool	NodeLoad utility IzoT Commissioning Tool

Smart Transceiver	Memory Type	Micro Server Image File Extension	Micro Server Image Programming Tool	Example Programming Tools
		NMF	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 IzoT Commissioning tool is available from echelon.com.
- Information about BPM Microsystems programmer models is available from 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 hilosystems.com.tw.
- Information about TotalPhase programmers is available from totalphase.com.

For device production, you typically use a universal chip programmer (where the chip is programmed prior to soldering it to the device circuit board) or an in-circuit programmer. 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:

SS430_ + *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 or Series 6000 devices, because the external clock speed is fixed at 10 MHz, 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: **SS430_PL3120E4_10000kHz.nfi**.
- The NodeLoad standard image for the ISI-enabled FT 6050 Smart Transceiver has the following name: **SS430_FT6050ISI_SYS20000kHz.ndl**.

The firmware images with these names are located in the [*ShortStack*]**microserver\standard** directory.

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.

The IzoT Interface Interpreter does not copy or rename the Micro Server image files. Instead, the IzoT Interface Interpreter assumes that you provide the repository of applicable Micro Server image files in a suitable location, and that your application's C source code references the chosen Micro Server. The IzoT Interface Interpreter supports a project-specific Micro Server repository as well as one or more repositories in different locations on your computer or computer network.

Table 8 lists the valid *file extension* values for the firmware image files.

Table 8. Micro Server Image File Extensions

Extension	Description
APB	Micro Server firmware image file for network management tools, such as the IzoT Commissioning 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.
NME, NMF	Micro Server programmable firmware image file for a universal chip programmer. Applies only to Series 5000 or 6000 chips.

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

- XIF – The Micro Server's device interface (XIF) file (used by the IzoT Interface Interpreter and the LonTalk Interface Developer tools)

- SYM – The Micro Server's device symbol file (used by the IzoT Interface Interpreter and the LonTalk Interface Developer tools)
- 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 universal chip programmer that supports the 3120 or 3170 Smart Transceiver.

To load the ShortStack Micro Server firmware using a universal chip programmer or in-circuit programmer, 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 IzoT Commissioning tool. See *Using a Network Management Tool for In-Circuit Programming* for information about using these tools to load a ShortStack Micro Server.

Do not use the IzoT Commissioning tool for the *initial* load of a ShortStack Micro Server into a power line Smart Transceiver. You can use the IzoT Commissioning 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 IzoT Commissioning Tool with* ShortStack.

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 use an appropriate flash programmer to load a ShortStack Micro Server's NEI file into the 3150 Smart Transceiver's off-chip memory.

Although you can reload the FT 3150 or PL 3150 Micro Server using in-circuit programming, you need to perform an *initial* load for the Micro Server firmware

using a universal chip programmer or in-circuit programmer. 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 IzoT Commissioning tool. See *Using a Network Management Tool for In-Circuit Programming* for information about using these tools to load a ShortStack Micro Server.

Loading a Blank Application

IzoT ShortStack SDK 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 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. ShortStack devices that use an FT 5000 Smart Transceiver with a different memory map or different non-volatile memory types (such as flash memory), need to use a custom Micro Server for the intended configuration. Note that there is no standard Micro Server image for a Neuron 5000 Processor.

To load the ShortStack firmware 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 IzoT Commissioning tool. See *Using a Network Management Tool for In-Circuit Programming* for information about using these tools to load a ShortStack Micro Server.

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

Loading an FT 6050 Smart Transceiver

A device based on a Series 6000 Chip always has non-volatile off-chip serial flash memory. The ShortStack firmware needs to reside in the non-volatile memory. The standard Micro Server for an FT 6050 Smart Transceiver supports all of the supported flash memory parts listed in the Series 6000 Data Book.

To load the ShortStack firmware use an appropriate flash programmer (such as the Total Phase Aardvark I2C/SPI Host Adapter) to load a ShortStack Micro Server's NMF file into the FT 6050 Smart Transceiver's off-chip memory. For the FT 6050 EVB, connect the programmer to the **JP23** header, as described in the *FT 6050 EVB Hardware Guide*.

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

Do not use the IzoT Commissioning Tool for the *initial* load of a ShortStack Micro Server. You can use the IzoT Commissioning tool for any subsequent loads as long as the Micro Server's system clock multiplier does not change. See *Using the IzoT Commissioning Tool with ShortStack*.

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 NodeUtil or NodeLoad utilities, or the IzoT Commissioning 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 the initial loading of a ShortStack Micro Server, this health check is likely to fail. 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 enters 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 IzoT Commissioning 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 IzoT Commissioning Tool for the *initial* load of a ShortStack Micro Server. You can use the IzoT Commissiong 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

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, U20, or U60 DIN Network Interface or the PCLTA-21 PCI Network Interface.

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 Smart Transceiver for the first time. 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 **SS430_FT6050ISI_SYS20000kHz.ndl** over a LON 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 –LSS430_FT6050ISI_SYS20000kHz.ndl
```

If you copy this command and paste it to a Windows command prompt and it does not work, try re-typing the dashes before the command switches. The NodeLoad utility might not recognize the dashes as copied from this PDF

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

```
nodeload -DLON1 -W20 -M –X –LSS430_FT6050ISI_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 SS430_FT6050ISI_SYS20000KHZ.NDL
NodeLoad Result: Success; NID=04c5c5e20100.
```

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

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

Using the IzoT Commissioning Tool with ShortStack

You can use the IzoT Commissioning 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 IzoT Commissioning tool. Otherwise, you need to load the Smart Transceiver using a Universal chip programmer or in-circuit programmer.

You cannot use the IzoT Commissioning Tool for the *initial* load of a Micro Server. Because IzoT Commissioning Tool cannot adjust the 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 Universal chip programmer or the NodeLoad utility), you can use the IzoT Commissioning Tool for subsequent loading as long as the system clock multiplier or transceiver selection remains unchanged.

Use a universal chip programmer or in-circuit programmer to perform the initial load for the Micro Server. You can use a universal chip programmer, in-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 exits quiet mode quickly as initialization completes much faster than the first time.

To load the ShortStack firmware using in-circuit programming using IzoT Commissioning Tool:

1. Add a Device shape to your IzoT Commissioning Tool drawing.
2. Optional: Ensure that the host processor is loaded with the ShortStack LonTalk/IP Compact API, 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 LonTalk/IP or LON network.
5. Complete the information required by the IzoT Commissioning Tool Device Wizard.

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

After you add the device to the IzoT Commissioning Tool 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 IzoT Interface Interpreter or the LonTalk Interface Developer utility in the **XIF Name** field. Do not use the Micro Server's XIF file.

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

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* for additional information about verifying the operational status of the Micro Server.

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

See the *IzoT Commissioning Tool User's Guide* for more information about using this tool.

Working with FT 6000 EVB or FT 5000 EVB Evaluation Boards

You can use an Echelon FT 6000 EVB or FT 5000 EVB evaluation board to develop your ShortStack application. However, you need to 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 6000 or 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)*.

  For specific wiring instructions between an Echelon FT 6000 or FT 5000 EVB and a Raspberry Pi mini-computer (or compatible device) see the wiring instructions located in [*ShortStack*]**/example/rpi/doc**, where [*ShortStack*] is your local folder into which you cloned the IzoT ShortStack SDK repository.

- 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)*.

To enable the FT 5000 EVB or FT 6050 EVB to support a ShortStack application, you need to 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 6000 EVB Hardware Guide* for more information about these jumpers.

General Jumper Settings for the FT 5000 EVB and FT 6000 EVB

Verify and set the following jumpers to run a ShortStack Micro Server on an FT 6000 EVB or FT 5000 EVB

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

JP1

For the FT 5000 EVB only, leave the jumpers for the JP1 header mounted as shown in Figure 5. This header connects the Smart Transceiver to the onboard serial flash and serial EEPROM memory.

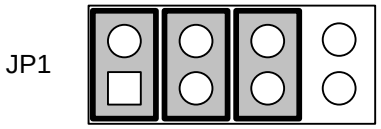


Figure 5. EVB Serial Memory Connections Header (JP1)

JP31

Dismount all of the jumpers from the **JP31** header, as shown in Figure 6. The settings shown in the figure disconnect the Smart Transceiver's I/O lines from the onboard I/O.

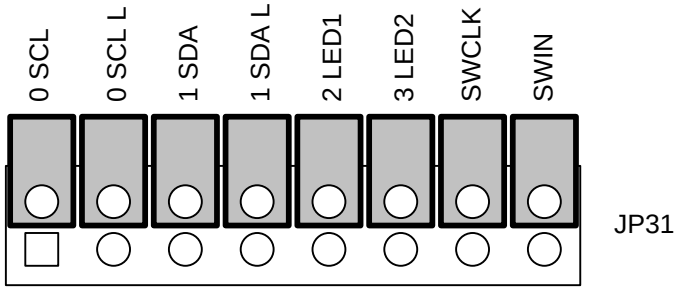


Figure 6. EVB I/O Connections Header (JP31)

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 7. This jumper setting is optional.

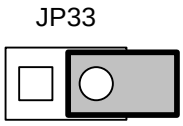


Figure 7. EVB LCD Display Power Header (JP33)

Using the Gizmo Interface (SCI or SPI)

To use the **P201** Gizmo interface on an FT 6000 EVB or 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 8 and Figure 9. The settings for pins 1-10 of the header shown in the figure disconnect the Smart Transceiver's I/O lines from the onboard I/O.

The **3 PD** jumper setting in Figure 10 specifies the SCI interface for the ShortStack Micro Server. The **3 PD** jumper setting in Figure 11 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 un-mounted, 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.

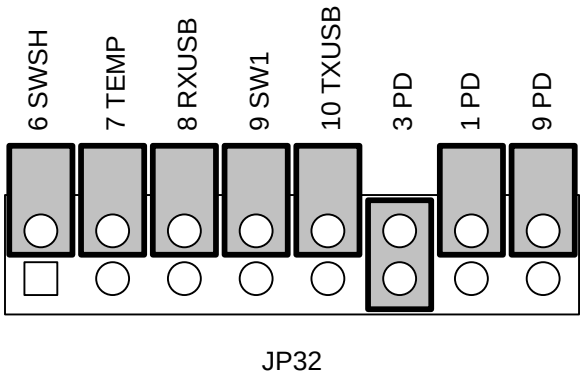


Figure 8. EVB I/O Connections Header (JP32) – SCI

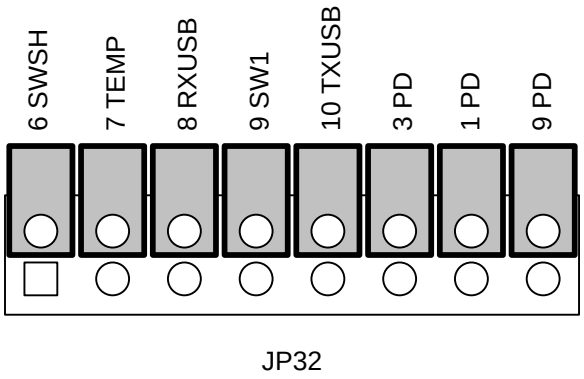


Figure 9. EVB I/O Connections Header (JP32) – SPI

JP201

Dismount all of the jumpers on the **JP201** header, except the **10T1IN** jumper, as shown in Figure 10. 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).

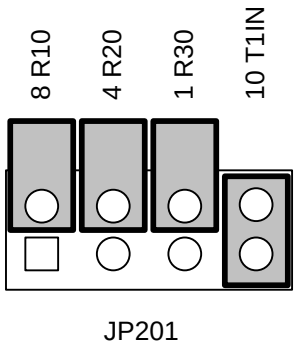


Figure 10. EVB EIA-232 Communications Header (JP201)

JP202

Mount the jumper for the **JP202** header to determine the external power source for the EVB, as shown in Figure 11.

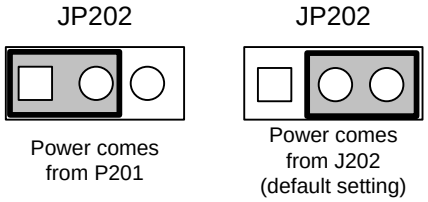


Figure 11. EVB Power Selector Header (JP202)

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

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

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

1. **Important**: Do not connect the external power supply to **J202** connector of the EVB.
2. Set the EVB's **JP202** jumper to use power from the **P201** Gizmo header (the left-hand image of **Figure 12**).
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 EVB).
5. Supply power to the host processor board.

JP203

Dismount the **0 T2IN** and **FON PD** jumpers, as shown in Figure 12. 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).

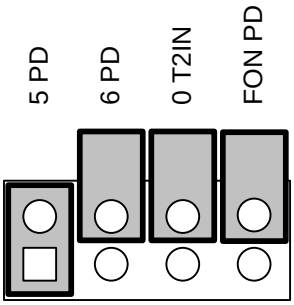


Figure 12. EVB ShortStack Header (JP203)

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*, and then mount the 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* or *Setting the SPI Bit Rate* for the correct settings for the IO5 and IO6 pins.

JP204

Mount the jumper for the **JP204** header, as shown in Figure 13, 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.

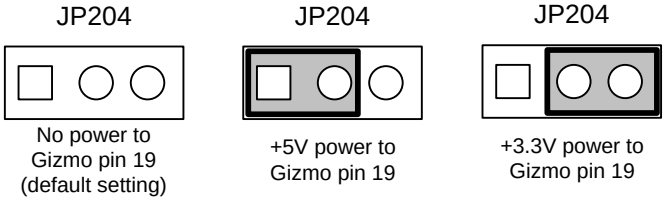


Figure 13. EVB Gizmo Pin 19 Power Selector Header (JP204)

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 14. The default setting is to provide no power to pin 17, but you can supply +3.3 V.

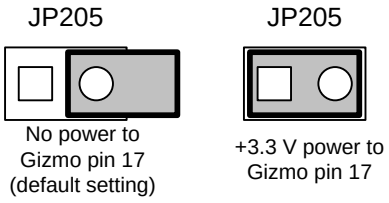


Figure 14. EVB Gizmo Pin 17 Power Selector Header (JP205)

P201

To connect your host evaluation board or Micro Server custom board to the **P201** Gizmo header, you need to create a custom connection cable. For rapid prototyping, you can use short 0.25" (0.635 mm) square socket test leads for these connections. Figure 15 shows the Gizmo header (**P201**) on the FT 5000

EVB. The figure shows the signal names as used by the FT 6000 EVB and 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 6000 or 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.

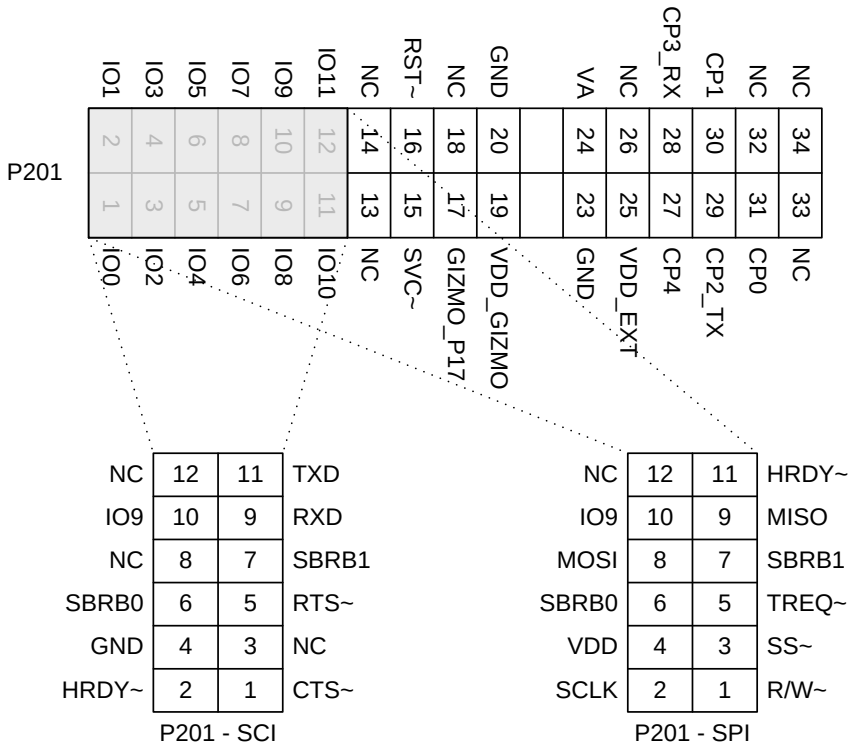


Figure 15. The Gizmo Header (P201) with the SCI and SPI Micro Server Signals

Using the EIA-232 Interface (SCI)

To use the EIA-232 interface on the FT 6000 or 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 16. The settings for pins 1-10 of the header shown in the figure disconnect the 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.

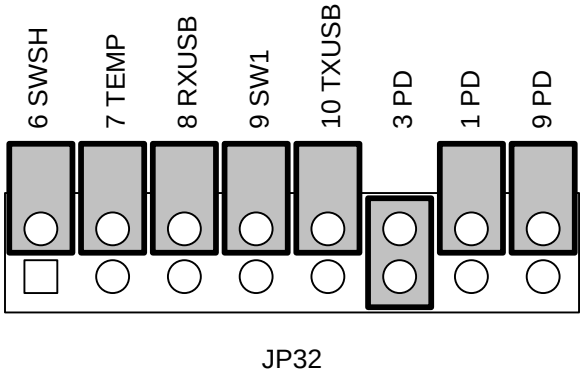


Figure 16. EVB I/O Connections Header (JP32)

JP201

Mount all of the jumpers on the **JP201** header, as shown in Figure 17. 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**.

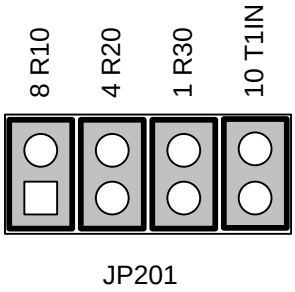


Figure 17. EVB EIA-232 Communications Header (JP201)

JP203

Mount the **0 T2IN** and **FON PD** jumpers on the **JP203** header, as shown in Figure 18. 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.

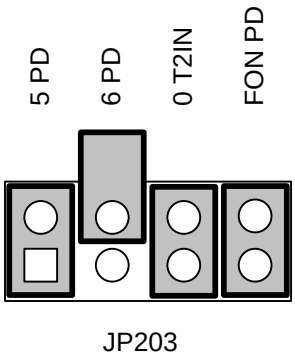


Figure 18. EVB ShortStack Header (JP203)

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.

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 18. Alternatively, your SCI serial driver can 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*, and then mount the 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* 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 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

Clearing the Non-Volatile Memory

In general, if you have a working device, you should not need to clear the onboard non-volatile memory. For a working device, you can receive a Service message and reload the non-volatile memory as needed.

However, if it should become necessary to clear the non-volatile memory:

For the FT 5000 EVB

See *External Serial Non-Volatile Memory Device Connection (JP1)* in the *FT 5000 EVB Hardware Guide*.

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). Never use the IzoT Commissioning Tool to load an image following this procedure

For the FT 6000 EVB

Reload the .NMF file using a suitable in-circuit SPI flash programmer. See *Performing In-Circuit Programming of the External Serial Memory Device* in the *FT 6000 EVB Hardware Guide*.

Using a Logic Analyzer

During device development, you can 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*. You can use the **JP24** header (see Figure 19) on the EVB to connect a logic analyzer to the EVB.

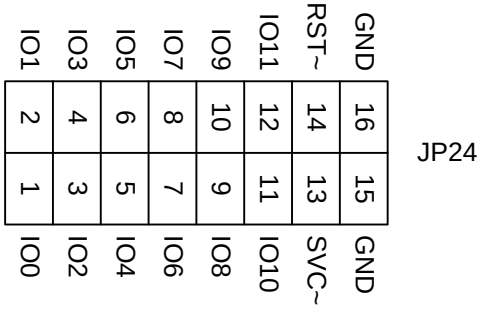


Figure 19. Logic Analyzer Header (JP24)

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 need to 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 5V signal level. By default, this connection supports only SCI serial driver connections. 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)*.

- 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).*

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 20. 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.

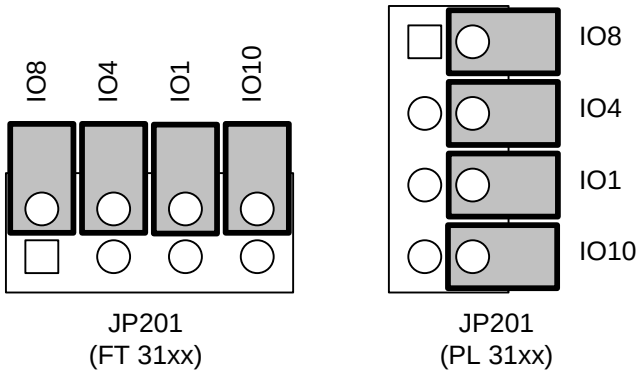


Figure 20. Mini EVB EIA-232 Enable Jumpers (JP201)

JP203

Dismount the IO0 jumper as shown in Figure 21; 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.

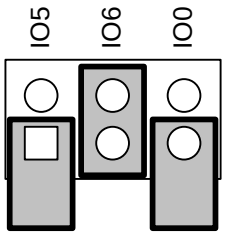


Figure 21. PL 3150 Mini FX ShortStack Enable Jumper (JP203)

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* 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* for the correct settings for the IO5 and IO6 pins.

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.

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 5V 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 need to create a custom connection cable. For rapid prototyping, you can use short 0.25" (0.635 mm) square socket test leads for these connections. Figure 22 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).

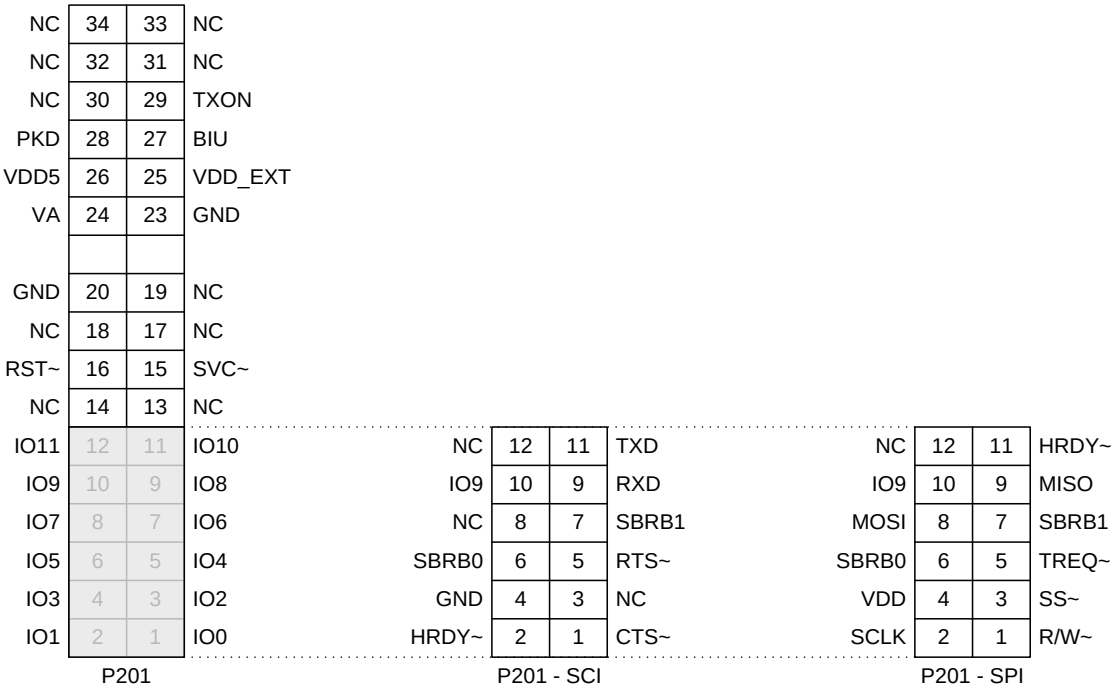


Figure 22. The Gizmo Header (P201) with the SCI and SPI Micro Server Signals

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 23. 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.

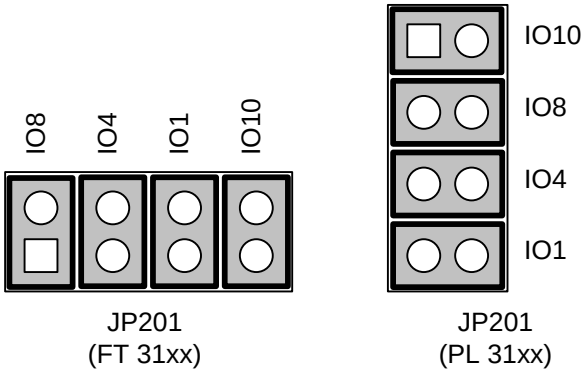


Figure 23. Mini EVB EIA-232 Enable Jumpers (JP201)

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.

To prevent the MAX3387E RS-232 transceiver from entering autoshutdown mode, your serial driver can 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 24 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.

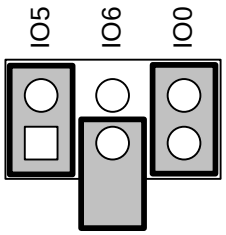


Figure 24. PL 3170 Mini FX ShortStack Enable Jumper (JP203)

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*, 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* 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 10. Keep the total cable length to a maximum of 24 inches (0.6 meters).

Table 10. 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

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 a while (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 beginning of this step.

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 or 6000 device; 210 to 840 ms after reset for a Series 3100 device, depending on the external clock rate). For some host platforms, 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 LonTalk/IP or LON device. 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, optional communication parameters, network variable configuration data, and miscellaneous preferences.

The ShortStack LonTalk/IP Compact API automatically resends this registration data whenever the Micro Server reports a reset and indicates that no application is registered.

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 LonTalk/IP or LON device.

See *Initializing the ShortStack device* for more information about the initialization ShortStack LonTalk/IP Compact API function and the *IzoT Markup Language* section *of* the *IzoT Manual* at echelon.com/docs/izot.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, its clock rate, whether it supports ISI, and its channel type. The key value is a 16-bit number that is reported to the host whenever the Micro Server sends a reset notification; Table 11 defines the bit values that comprise the key for standard Micro Servers.

Table 11. 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
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

Micro Server Type	Bit Values						Key Value
	Custom	Revision	Chip Type	Clock Speed	ISI Support	Channel Type	
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
FT 5000	0	0001	0101	011	0	000	0x0AB0
FT 6050	0	0001	0111	011	1	000	0x0BB8
FT 6050	0	0001	0111	011	0	000	0x0BB0

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[1] 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
  - 0b0101 indicates an FT 5000 Smart Transceiver
  - 0b0110 indicates a Neuron 5000 Processor[2]

[1] "0b0" represents a binary literal or constant value of 0 (zero).

    - 0b0111 indicates a Series 6000 chip
- *Clock speed* is a three-bit field that identifies the clock speed for the Smart Transceiver or Neuron Processor[3]:
    - 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 transceiver 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*]**\microserver\custom\MicroServer.h** header file:

```
#define MICRO_SERVER_KEY 0x8000ul
```

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

---

[2] The Neuron 5000 or 6050 Processor is not supported by the standard Micro Servers that are included with the IzoT ShortStack SDK. You need to create a custom Micro Server to support a Neuron 5000 Processor.

[3] For a Series 3100 Smart Transceiver, this value is the external crystal or oscillator frequency value. For a Series 5000 or 6000 chip, this value is twice its system clock value (from the device's hardware template), to represent an equivalent Series 3100 clock rate.

6

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* or *Setting the SPI Bit Rate*.

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

If your application uses SPI, the host processor needs to 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 must assert the **HRDY~** signal briefly. A typical driver implementation deasserts this signal 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 is 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, the SCI interface is easier to implement. 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

A processor with a big-endian (most significant byte at low address) architecture is easier to implement than a processor with a little-endian (least significant byte at low address) for a ShortStack device. Network data in a LonTalk/IP or LON 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/IP 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/IP Compact API requires frequent periodic servicing through the **LonEventHandler()** API function (see *Periodically Calling the Event Handler*). 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 bare-metal 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 and ShortStack LonTalk/IP Compact API]). See *API Memory Requirements* for a description of the memory requirements for the ShortStack LonTalk/IP Compact API and optional APIs.

If your application uses large network variables, application messages, or foreign frame messages, which include larger payload data and therefore require larger buffers or additional buffers in the host application, the RAM requirement could increase significantly.

Modifiable Non-Volatile Memory

Although the ShortStack LonTalk/IP 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/IP Compact API provides tools and code that can help manage non-volatile memory.

Compiler and Application Programming Language

The IzoT ShortStack SDK provides the ShortStack LonTalk/IP Compact API and application framework as portable ANSI C source code. A standard ANSI C (or C++) compiler for application development is appropriate. Other development tools and languages are possible, but you need to then port the driver, API, and application framework to the other language.

You can use the ShortStack LonTalk/IP Compact API and application framework 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/IP 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 can generate the exact memory image of these structures and unions without inserting any padding bytes. By exclusively using single-byte entities, the ShortStack LonTalk/IP Compact API allows most compilers to be used with an IzoT ShortStack SDK application.

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

Selecting the Development Environment

The ShortStack LonTalk/IP 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/IP 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/IP Compact API. Other solutions can call and process data from the API from a timer-based interrupt service handler routine.

Although the ShortStack LonTalk/IP 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/IP Compact API should ensure that all ShortStack LonTalk/IP Compact API access is within a single thread (or task or interrupt context).

See *Appendix A, ShortStack LonTalk/IP Compact API* for additional considerations and recommendations regarding threading and execution context.

7

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 or Neuron Chip. 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*, the *Series 5000 Chip Data Book* or the *Series 6000 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 **examples** folder of the IzoT ShortStack SDK repository on github.com/izot/shortstack for example code and documentation. 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 11** and **Table 13**, 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, you can have 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 must 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 will be approximately 50 Ω) at the driving pin. In addition, the trace must run on the top layer of the PCB, over the inner ground plane, and cannot 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*, the *Series 5000 Chip Data Book,* or the *Series 6000 Chip Data* book for information about the requirements for this pin.

However, because a ShortStack device uses two processor chips, the Smart Transceiver or Neuron Chip 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*); 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); thus, 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 IzoT ShortStack SDK 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 can use a jumper or configuration switch to select, or deselect, 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 influence the required bit rate is support for explicit addressing, an optional feature that the ShortStack application can enable and disable.

The following minimums apply to general-use LONWORKS devices:

- ShortStack Micro Server external clock frequency
    - 10 MHz or higher for TP/FT-10 devices (for Series 5000 or 6000 devices, specify a minimum 5 MHz system clock rate)
    - 5 MHz or higher for power-line devices
- Bit rate
    - 38400 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, set the device bit rate at a value above the minimum calculated from the formula. Thus, for this example, a bit rate of at least 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 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.

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 or FT 6050 Smart Transceivers can operate with up to an 80 MHz system clock rate, but the standard Micro Server for Series 5000 and 6000 chips use 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 the availability of standard bit rates.

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 or 6000 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 or 6000 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, but a faster Micro Server will provide the best performance.

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 or 6000 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 384ms.

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. You can connect these signals to the host processor, but they do not have 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 12 summarizes these pin assignments for the SCI interface.

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 12. 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*)
IO10	TXD	Output
IO11	N/A	No connection

Notes:

- Signal direction is from the point of view of the Smart Transceiver or Neuron Chip (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 13. The rates are listed as bits per second; the values are also approximate and rounded to the nearest 100 bits per second.

Table 13. SCI Serial Bit Rates

Series 3100 External Clock	Series 5000 or 6000 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 or 6000 system clock rate in the hardware template for a custom Micro Server. The standard Series 5000 or 6000 Micro Server images use a 20 MHz system clock. The external crystal clock frequency for a Series 5000 or 6000 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 13. The standard Series 5000 or 6000 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* for more information about creating a custom Micro Server image.

Some of the higher bit rates listed in Table 13 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.

For implementations with higher bit rates, be sure that the link-layer hardware provides low impedance and correct termination. Also add 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 proceeding to other debugging procedures.

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

SCI Communications Interface

The SCI communications interface shown in Figure 25 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.
- 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, tie it low so that it is continually asserted (to specify that the host is always ready to accept data transfers). See *Serial Communications* 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 13).

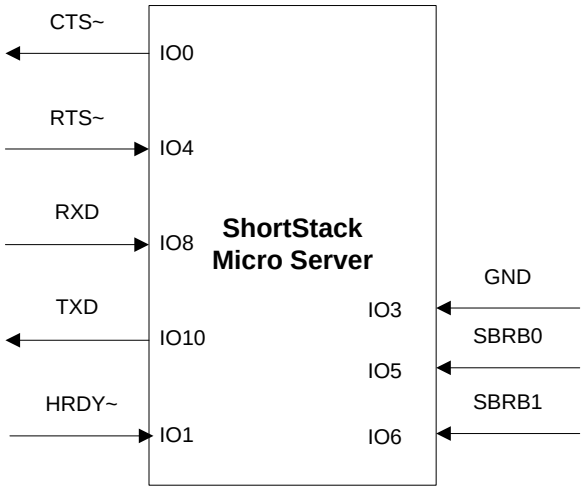


Figure 25. ShortStack SCI Communications Interface

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 must 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 26 shows an example for the Micro Server to host SCI control flow, including the states of the various I/O pins.

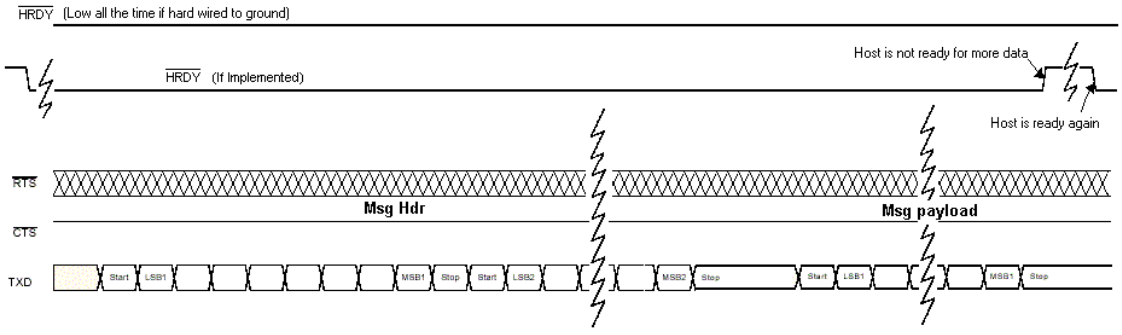


Figure 26. SCI Micro Server to Host Transfer Control Flow Diagram

SCI Host to Micro Server (Downlink) Control Flow

The Micro Server uses the **CTS~** pin to enforce a half-duplex interface. Every downlink transfer needs to 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 must include a timeout guard that can accommodate this wait period, 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 *Creating a ShortStack Serial Driver*, for more information about the serial driver.

The IzoT ShortStack SDK application and driver example for use with the Raspberry Pi -computer and the Raspbian Linux operating system includes an example SCI driver, which implements the recommended timeout guards. See [*ShortStack*]**/example/rpi/driver/rpi.c** for the implementation of those timeouts, and for extensive discussion about each timeout's duration.

Figure 27 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.

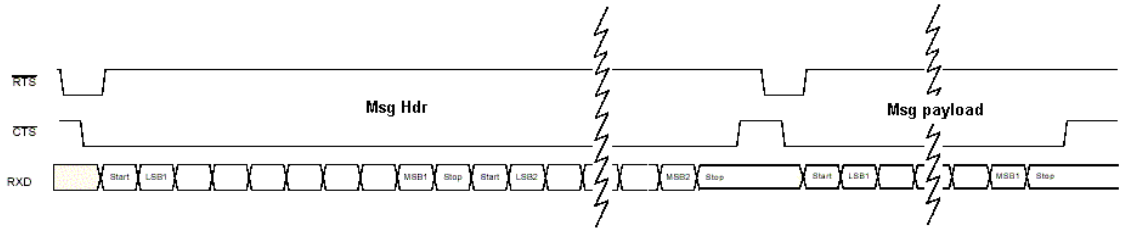


Figure 27. SCI Host to Micro Server Transfer Control Flow Diagram

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 14 summarizes these pin assignments for the SPI interface.

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 14. 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*)
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. You control the effective SPI bit rate with the **SCLK** output from the ShortStack Micro Server, but you preselect the desired bit rate using the **SBRB0** and **SBRB1** (IO5 and IO6) input signals. 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 Tables 15 and 16. 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 15. SPI Serial Bit Rates for Uplink

Series 3100 External Clock	Series 5000 or 6000 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 or Series 6000 system clock rate in the hardware template for a custom Micro Server. The standard Series 5000 or 6000 Micro Server images use a 20 MHz system clock. The external crystal clock frequency for a Series 5000 or 6000 chip is 10 MHz.

Table 16. SPI Serial Bit Rates for Downlink

Series 3100 External Clock	Series 5000 or 6000 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	

Series 3100 External Clock	Series 5000 or 6000 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}
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 must create a custom Micro Server image to use the 5 MHz clock rates listed in Tables 15 and 16. The standard Series 5000 or 6000 ShortStack Micro Server images support only the 20 MHz system clock rate; you must create a custom Micro Server image to use the other clock rates listed in Tables 15 and 16. See *Custom Micro Servers* for more information about creating a custom Micro Server image.

Some host processors or UART/USART implementations might not be able to process data at some of the higher bit rates listed in Tables 15 and 16. 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.

For implementations with higher bit rates, be sure that the link-layer hardware provides low impedance and correct termination. Also add 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 proceeding to other debugging procedures.

SPI Communications Interface

The SPI communications interface shown in Figure 28 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 rising clock edge, and sampled at the falling 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 communications. 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 deassert the **R/W~** signal.
- Read/Write (**R/W~**): Indicates which direction is active during a byte transfer (low indicates write). The **R/W~** signal is low during a transfer from the Micro Server to the host (**MOSI**); the **R/W~** signal is high during a transfer from the host to the Micro Server (**MISO**). See *SPI Host to Micro Server Control Flow (MISO)* 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.

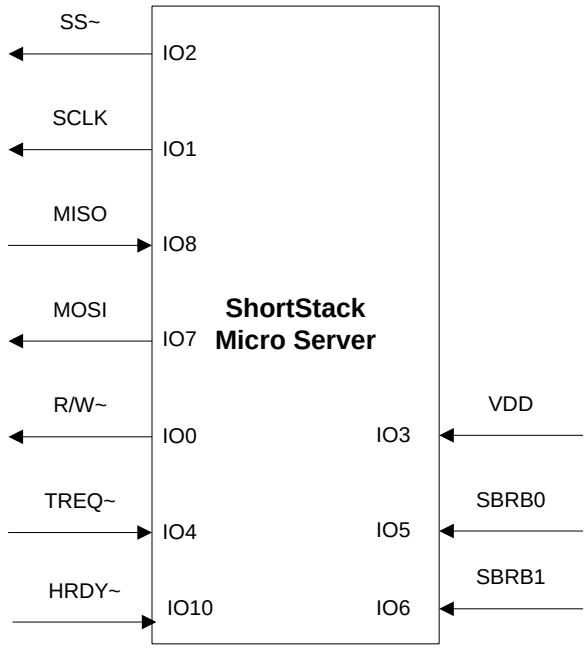


Figure 28. ShortStack SPI Communications Interface

SPI Micro Server to Host Control Flow (MOSI)

The host driver asserts the **HRDY~** signal low to indicate that it is ready to receive data. Because the Micro Server has a limited set of buffers, the host driver must deassert the **HRDY~** signal for only a short duration. A typical driver 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 driver 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~** signal to be asserted low, then it sets the **R/W~** signal low to indicate the direction of the data transfer. The Micro Server presents data on each rising edge of the **SCLK** signal; the host samples the data on each falling edge.

During MOSI transmissions, the **MISO** signal 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 cannot be used for any other purpose than ShortStack SPI interface data transfers.

Figure 29 shows an example for the Micro Server to host SPI control flow.

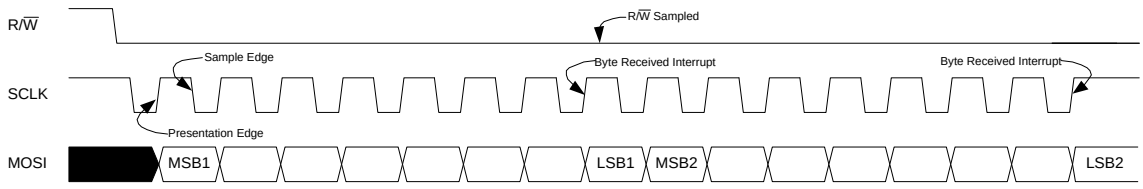


Figure 29. SPI Micro Server to Host (MOSI) Transfer Control Flow Diagram

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~** signal. Asserting the **TREQ~** signal causes the Micro Server to start the data transfer by driving the **SCLK** signal. Loading the data byte before asserting the **TREQ~** signal 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~** signal 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~** signal 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~** signal and it is still high after the transfer of the first byte, it immediately de-asserts the **TREQ~** signal 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 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 30 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.

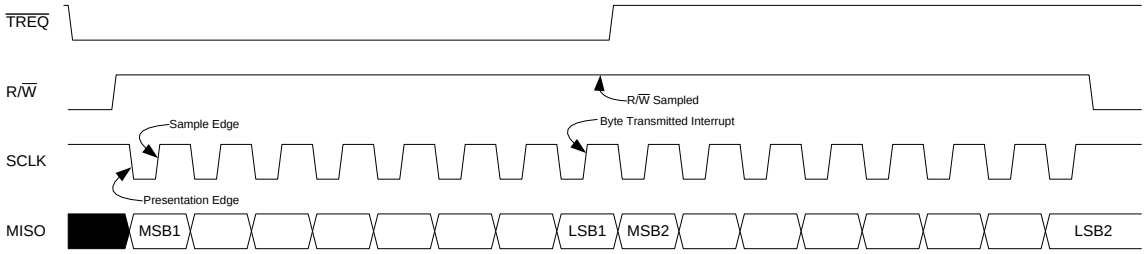


Figure 30. SPI Host to Micro Server (MISO) Transfer Control Flow Diagram without Write Collision

Figure 31 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~** signal 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~** signal is high after the attempted transfer of the first byte. The figure shows that the host successfully transmits the data on the second attempt.

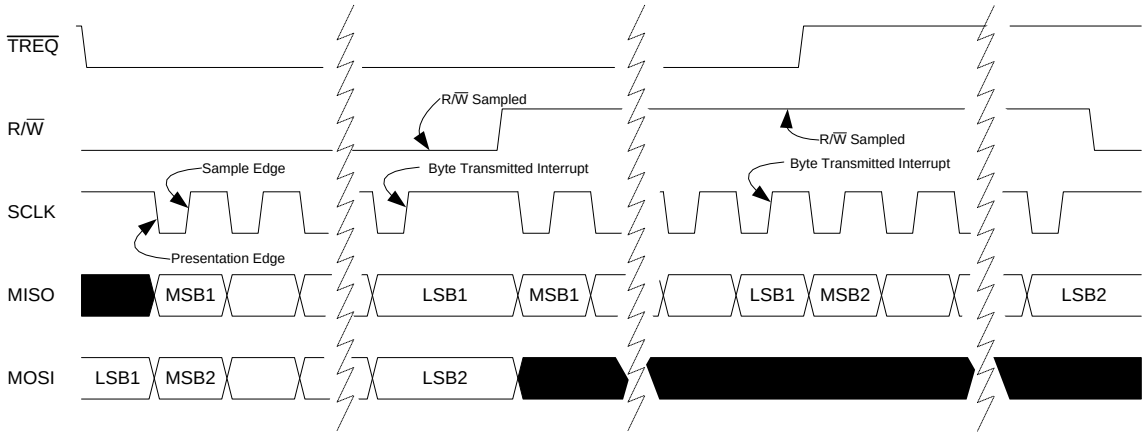


Figure 31. SPI Host to Micro Server (MISO) Transfer Control Flow Diagram with Write Collision

SPI Resynchronization

The Micro Server resynchronizes the ShortStack SPI interface by de-asserting the **SS~** signal during a byte transfer, or by de-asserting the **SS~** signal 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 LonTalk/IP or LON 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) signal or **MOSI** (IO7) signal that reflects the uplink **LonNiReset** message transfer that follows a Micro Server reset, as shown in Figure 32 for SCI and Figure 33 for SPI.

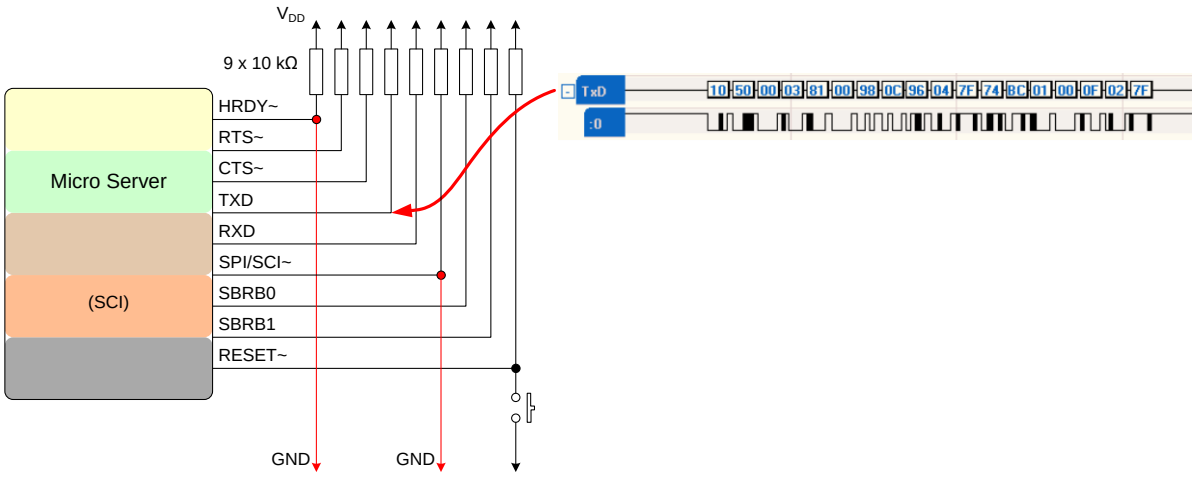


Figure 32. Uplink LonNiReset Message Transfer – SCI

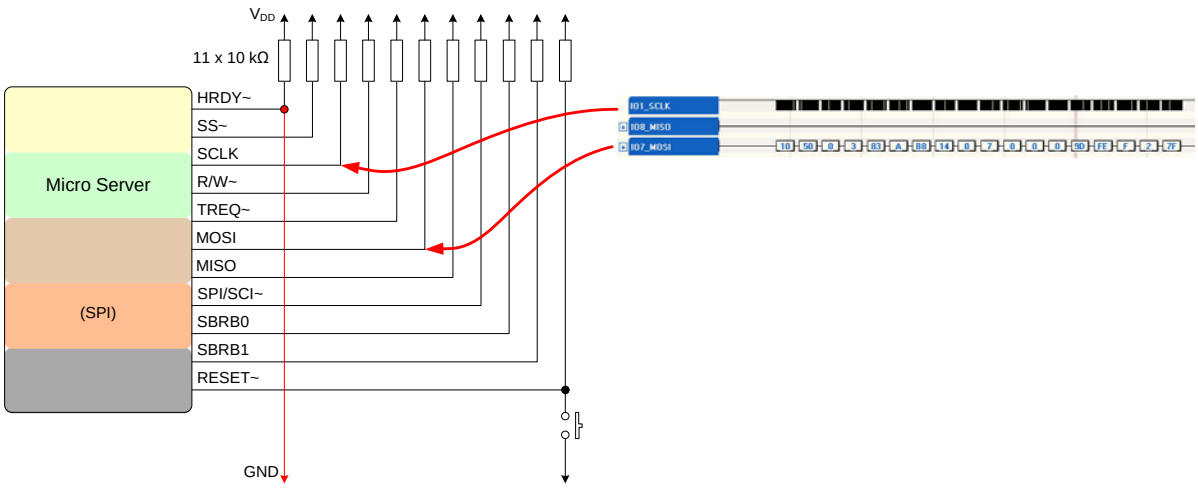


Figure 33. Uplink LonNiReset Message Transfer – SPI

The Micro Server's service 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.

Ensure that all communication and handshake lines are connected to V_{DD} with 10kΩ pull-up resistors. For the initial hardware test, the **HRDY~** input signal should be grounded (asserted). If you use the SCI interface, the **SPI/SCI~** input signal should also be grounded; if you use the SPI interface, the **SPI/SCI~** input signal must be connected to V_{DD}. Your hardware design can include a button that connects the **RESET~** pin to ground; you press this button to reset the Micro Server.

When you press the Reset button 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* 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/IP 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 either the **CTS~** signal or the SPI signals):

- 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.

Setting Up a Logic Analyzer for ShortStack

Within your logic analyzer software, specify the capture options for each signal, as shown in Table 17 for SCI and Table 18 for SPI.

Table 17. Logic Analyzer Signal Definitions for SCI

Signal Name	Signal Type	Communications Settings
IO1 – HRDY	Boolean	—
IO0 – CTS~	Boolean	—
IO4 – RTS~	Boolean	—
IO8 – RXD	Asynchronous	Data Bits: 8 Parity: None Baud Rate: Depends on **SBRB0** and **SBRB1** settings
IO10 – TXD	Asynchronous	Data Bits: 8 Parity: None Baud Rate: Depends on **SBRB0** and **SBRB1** settings

Table 18. Logic Analyzer Signal Definitions for SPI

Signal Name	Signal Type	Communications Settings
IO2 – SS	Boolean	—
IO10 – HRDY	Boolean	—
IO1 – SCLK	Boolean	—
IO4 – TREQ	Boolean	—
IO0 – RW	Boolean	—
IO8 – MISO	Synchronous: CLK = IO1 – SCLK DATA = IO8 – MISO ENABLE = IO0 – RW	Data Bits: 8 Enable State: High Clock Edge: Rising MSB first
IO7 – MOSI	Synchronous: CLK = IO1 – SCLK DATA = IO7 – MOSI ENABLE = IO0 – RW	Data Bits: 8 Enable State: Low Clock Edge: Falling MSB first

For both IO7 – **MOSI** and IO8 – **MISO**, the synchronous clock is the IO1 – **SCLK** signal and the enable signal is the IO0 – **RW** signal.

Example Health Check for SCI

Figure 34 through Figure 38 show sample logic analyzer traces[4] 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** signals 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 for an ARM7 example port.

Figure 34 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

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

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

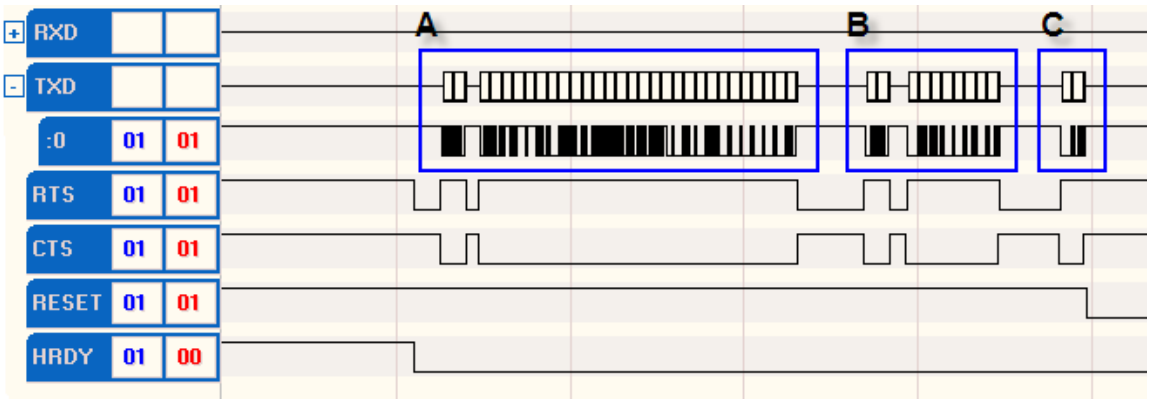


Figure 34. High-Level Logic Analyzer Trace for ShortStack Device Reset

Figure 35 shows the detailed trace for the serial driver and Micro Server interactions for sending the **LonNiAppInit** message.

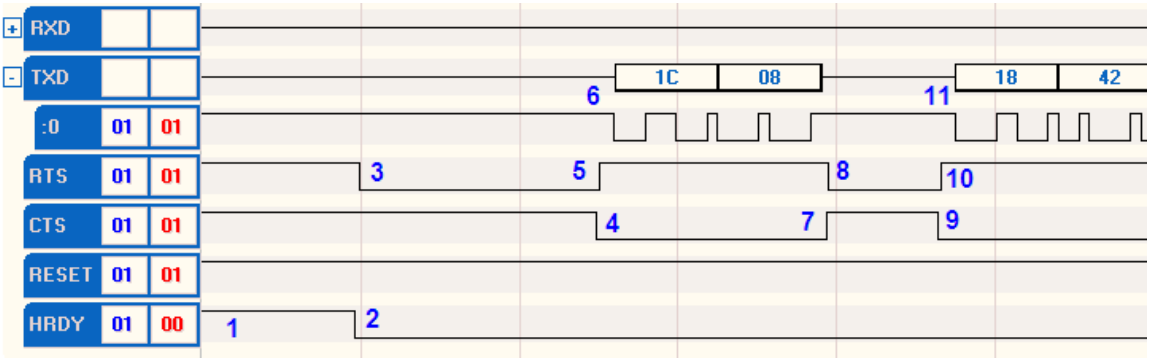


Figure 35. Detailed Logic Analyzer Trace for Sending the LonNiAppInit Message

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

1. After a device reset, the operating system, application, and driver load and initialize.
2. When the driver is ready to receive data, it asserts the **HRDY~** signal.
3. Because the driver needs to send the initialization messages, it confirms that the **CTS~** signal is not asserted, and then it asserts the **RTS~** signal 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~** signal to inform the driver that the Micro Server is ready to receive data.
5. The driver deasserts the **RTS~** signal. The handshake between the driver and the Micro Server is complete, so the driver deasserts the **RTS~** signal so that the signal can be asserted when the driver needs to send more data to the Micro Server. It is important that the driver deassert the **RTS~** signal before the last byte of data is transmitted, and deasserts the **RTS~** signal as soon as the **CTS~** signal 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~** signal to inform the driver that the Micro Server is no longer receiving 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~** signal after it has received the 2 bytes.
8. The driver confirms that **CTS~** is deasserted, and again asserts the **RTS~** signal 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~** signal to inform the driver that the Micro Server is ready to receive the payload data.
10. The driver deasserts the **RTS~** signal. 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 may vary.
12. Once the Micro Server received all payload data (according to the payload length transmitted with the header segment), it de-asserts **CTS~**, then acts on the contents of that message (not shown in the above illustration).

Figure 36 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.

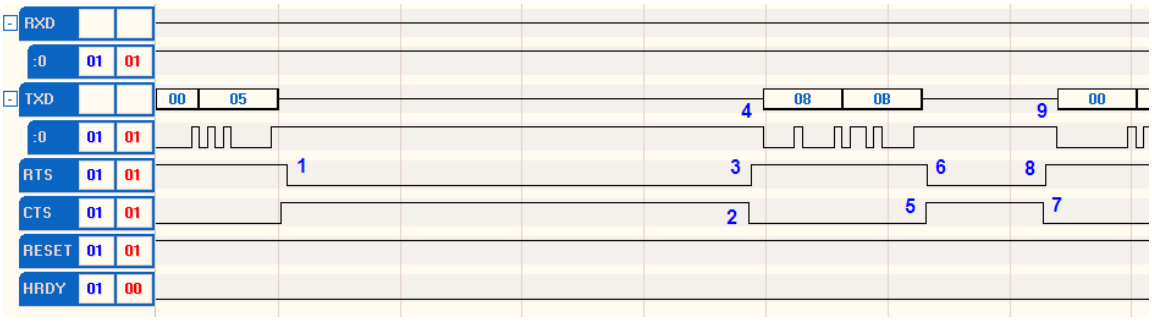


Figure 36. Detailed Logic Analyzer Trace for Sending the LonNiNvInit Message

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

1. The driver confirms that the **CTS~** signal is not asserted, and then asserts the **RTS~** signal 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~** signal 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~** signal. 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~** signal. 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~** signal after it has received the 2 bytes.
6. After confirming that **CTS~** is deasserted, the driver again asserts the **RTS~** sigal 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~** signal to inform the driver that the Micro Server is ready to receive the payload data.
8. The driver deasserts the **RTS~** signal. 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.
10. The Micro Server de-asserts **CTS~** once it received the expected number of bytes.

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~** signal. 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 37 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.

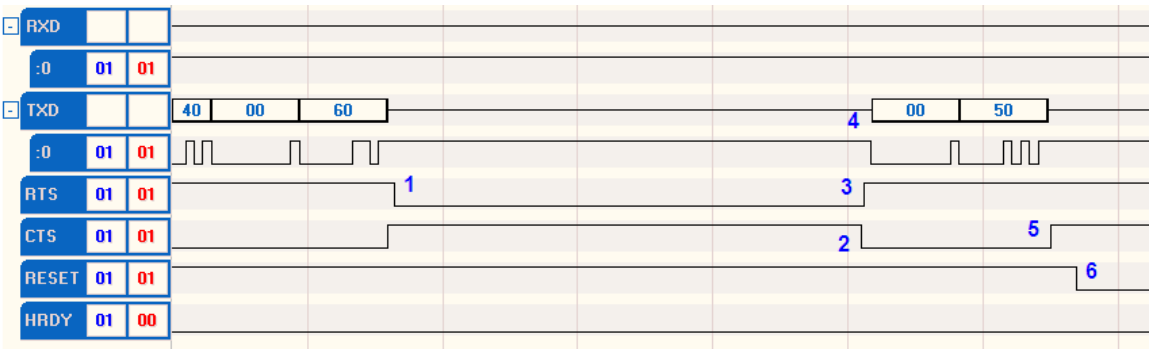


Figure 37. Detailed Logic Analyzer Trace for Sending the LonNiReset Message

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

1. The driver confirms that the **CTS~** signal is not asserted, and then asserts the **RTS~** signal 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~** signal 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~** signal. 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~** signal 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.

As shown in **Figure 38**, the driver does not re-assert the **RTS~** signal. 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~** signal 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. The **LonNiReset** message is shown on the **RXD** line because the signals are labeled from the host's point of view.

IzoT Micro Servers enforce an artificial post-reset pause *after* transmitting the uplink reset message. The Micro Server does not interact with the host in any way for the duration of this pause. The post-reset pause is 50 ms by default, and can be configured from your host application for most Micro Servers (disabled, or re-adjusted in a 1..255 ms range). The PL 3170 Micro Servers support a non-configurable 50 ms post-reset delay.

The post-reset delay grants more processing time to the host *after* it received the reset notification, which generally requires that the host aborts pending downlink operation and re-synchronizes its link layer driver.

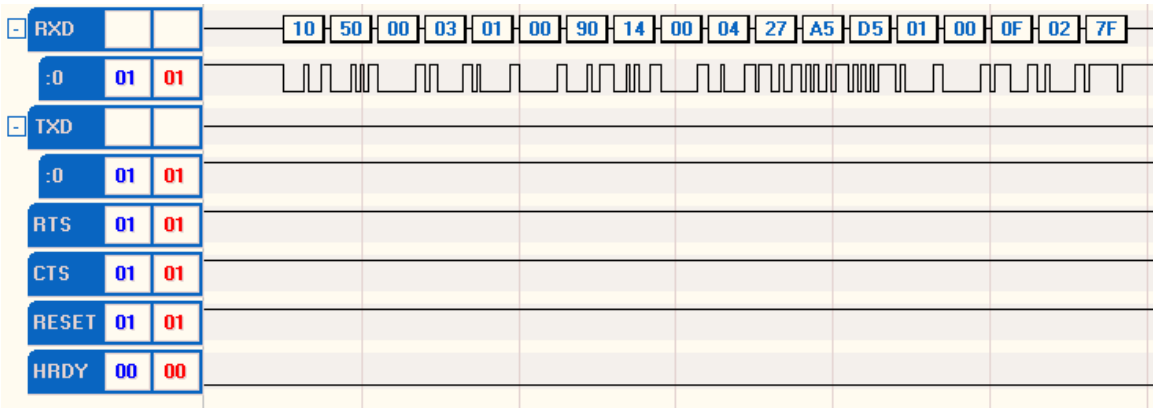


Figure 38. Detailed Logic Analyzer Trace for Receiving the Uplink Reset Message

There is no handshake through the **RTS~** and **CTS~** control signals 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 accepts 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* for more information about the link-layer header.

Example Health Check for SPI

Figure 39 and 40 show sample logic analyzer traces[5] for the communications activity between the host processor and the Micro Server during the initialization sequence after device reset. Figure 39 shows the messages from the host to the Micro Server; Figure 40 shows the response from the Micro Server after device reset is complete. This example assumes an SPI setup for a 20 MHz Series 5000 Micro Server, with both **SBRB0** and **SBRB1** connected to **GND** to set the bit rate at 172600/226600 bps. The data transmission signals (**MISO** and **MOSI**) in

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

the figures are labeled from the Micro Server's point of view as the SPI master. This example shows the reset behavior of the serial driver for the ARM7 example port SPI driver that is available from the Echelon Knowledge Base: echelon.com/support.

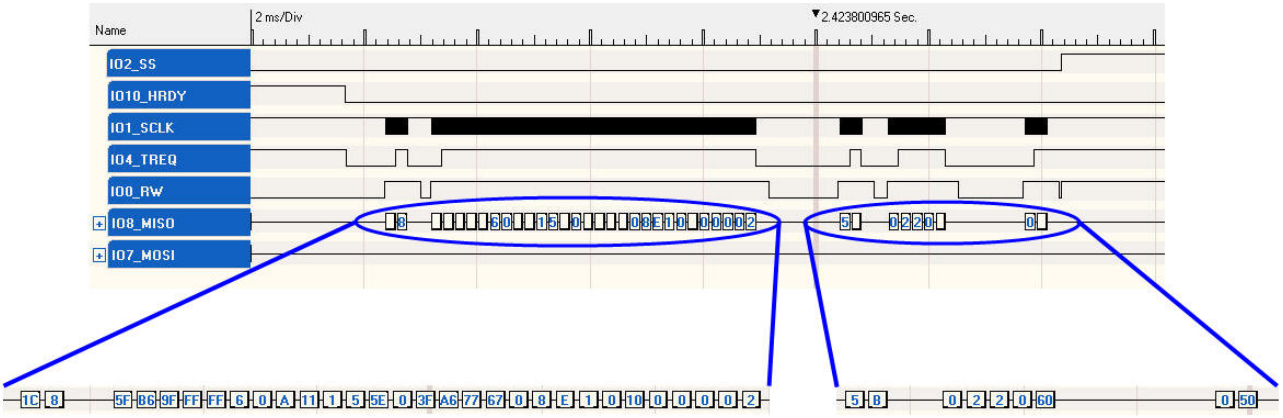


Figure 39. LonInit Communications Flow for SPI, Part 1

The basic communications flow shown in **Figure 40** includes the following steps:

1. When the host application is ready, the driver asserts the **HRDY~** signal. A low **HRDY~** signal indicates that the host is available to receive data from the Micro Server.
2. Because the **LonInit()** function needs to send data to the Micro Server, the driver asserts the **TREQ~** signal. A low **TREQ~** signal indicates that the host has data to send to the Micro Server.
3. When the Micro Server is ready to receive the host's data, it deasserts the **R/W~** signal. A high **R/W~** signal allows the host to send data. When the Micro Server sees the low **TREQ~** signal while the **R/W~** signal is high, it drives the **SCLK** signal to allow the data transfer to begin.
4. The host sends the header byte for the first message (the **LonNiAppInit** message). The data appears on the **MISO** signal.
5. As soon as the first header byte is placed within the driver's output buffer, the driver deasserts the **TREQ~** signal.
6. After the Micro Server receives the two-byte header, it stops driving the **SCLK** signal to end the data transfer. As the header byte is received by the Micro Server, it asserts the **R/W~** signal. A low **R/W~** signal indicates either a write by the Micro Server or that the Micro Server is not ready to receive data from the host.
7. Steps 2 through 6 repeat for sending the payload for the first message (the **LonNiAppInit** message).
8. Steps 2 through 7 repeat for sending the second message (the **LonNiNvInit** message) and the third message (the **LonNiReset** message), although the third message has no payload.

After the Micro Server receives the **LonNiReset** message, it resets. After the reset is complete, the Micro Server sends an uplink reset message (**LonResetNotification**) to the host processor, as shown in Figure 40.

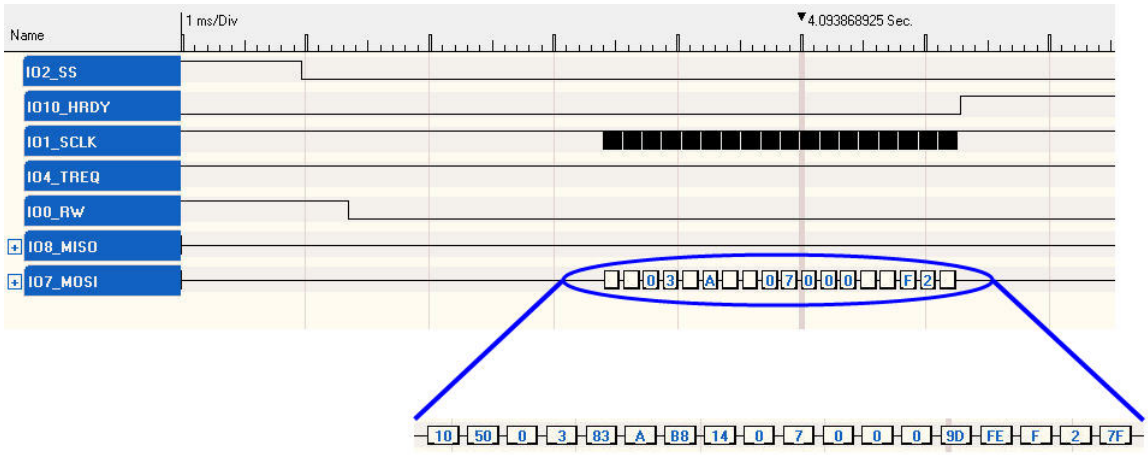


Figure 40. LonInit Communications Flow for SPI, Part 2

IzoT ShortStack Micro Servers enforce a configurable post-reset pause *after* the uplink reset message is sent. The Micro Server does not interact with the host in any way for the duration of this pause. The post-reset pause is 50 ms by default, and can be configured from your host application for most Micro Servers (disabled, or re-adjusted in a 1..255 ms range). The PL 3170 Micro Servers support a non-configurable 50 ms post-reset delay.

The post-reset delay grants more processing time to the host *after* it received the reset notification, which generally requires that the host aborts pending downlink operation and re-synchronizes its link layer driver.

Because the message originates with the Micro Server (an uplink message), it asserts the **R/W~** signal and drives the **SCLK** signal when it is ready to send the data. The message includes both the two-byte header and the message payload in a single message transfer.

8

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 all current ShortStack Micro Servers, the value of this byte is 0x00.

Figure 41 shows the structure of the link-layer message.

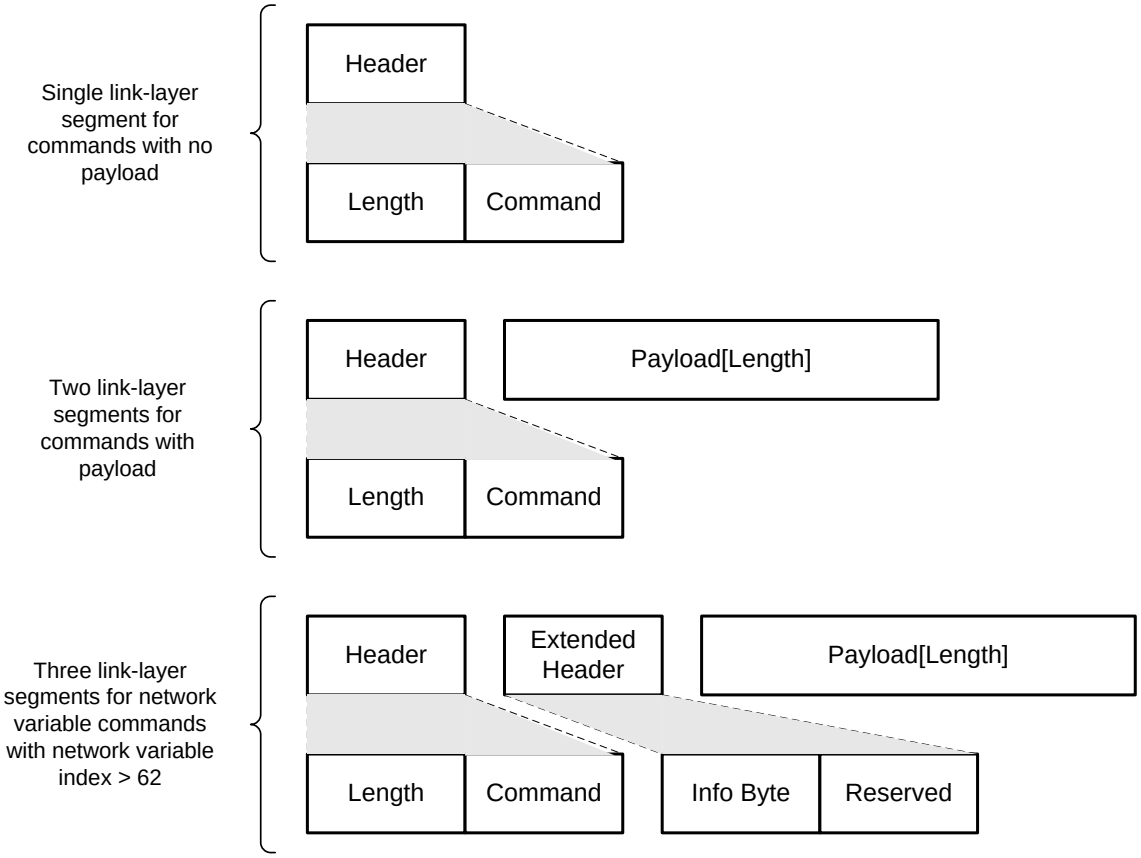


Figure 41 Link-Layer Message Structure

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 *Designing the Hardware Interface*.

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~** signal (or has its **HRDY~** signal 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 segment transmission needs to wait for the uplink to complete.

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

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 may 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/IP Compact API

One of the most important tasks performed by the ShortStack LonTalk/IP 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/IP 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/IP 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.

ShortStack LonTalk/IP Compact API Interface

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* when bare-metal designs are used. Implementations that use an operating system may find operating system support for basic serial communication, but may need to add support for some of the ShortStack link layer signals. These applications will typically use a worker thread or some other suitable means of concurrent processing to exchange data between an input and output message queue pair on the side of your application, and the Micro Server.

The ShortStack LonTalk/IP Compact API uses the functions listed in Table 19 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 **ldv.h** file.

For more information about these interface functions, see an example port's implementation of the functions; for example, see the IzoT ShortStack SDK Examples repository on github.com/izot/shortstack for example code and documentation.

Table 19. Interface Functions for the ShortStack LonTalk/IP Compact API

Function	Description
LdvOpen()	Initializes the ShortStack serial driver and the underlying communication interface. This API was previously known as *LdvInit()*.
LdvClose()	Completes all pending downlink traffic and closes the driver. This API is new with the IzoT ShortStack SDK.
LdvAllocateMsg()	Allocates a transmit buffer in the ShortStack serial driver.
LdvAllocateMsgWait()	Allocates a transmit buffer in the ShortStack serial driver in a blocking operation (until a fatal timeout occurs). This API is new with the IzoT ShortStack SDK.
LdvPutMsg()	Submits a downlink message to the driver. This is a non-blocking function. This API will typicall return to the caller before the related message transmission is complete.
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.
LdvSuspend()	Temporarily suspends the serial driver. The driver can be suspended, synchronized to the end of the next segment, or the end of the next multi-segmented frame. This API is new with the IzoT ShortStack SDK. The API is optional; your implementation of the driver can indicate whether support for this API is provided. The ShortStack API does not require this API.

Function	Description
LdvResume()	Resume a suspended driver. This API is new with the IzoT ShortStack SDK. The API is optional; your implementation of the driver can indicate whether support for this API is provided. The ShortStack API does not require this API.

Creating an SCI ShortStack Driver

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

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).

The [*ShortStack*]**/example/rpi/driver/rpi.c** example source file contains definitions and discussion of suggested values for various timeout conditions.

SCI Uplink Operation

In an SCI uplink operation, data is transferred from the ShortStack Micro Server to the host processor. Figure 42 and Figure 43 show the activity that the driver should manage for an uplink operation. The figures also show how the Micro Server, serial driver, LonTalk/IP 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~** signal is asserted. While an uplink transfer is in progress, the Micro Server does not re-sample the **HRDY~** signal. To prevent loss of uplink data, the host must assert this handshake signal whenever possible, and de-assert it for the shortest time possible.

An uplink transfer can occur between the two or three segments of a downlink transfer. Your driver must be able to receive uplink data at this time to avoid a possible deadlock condition.

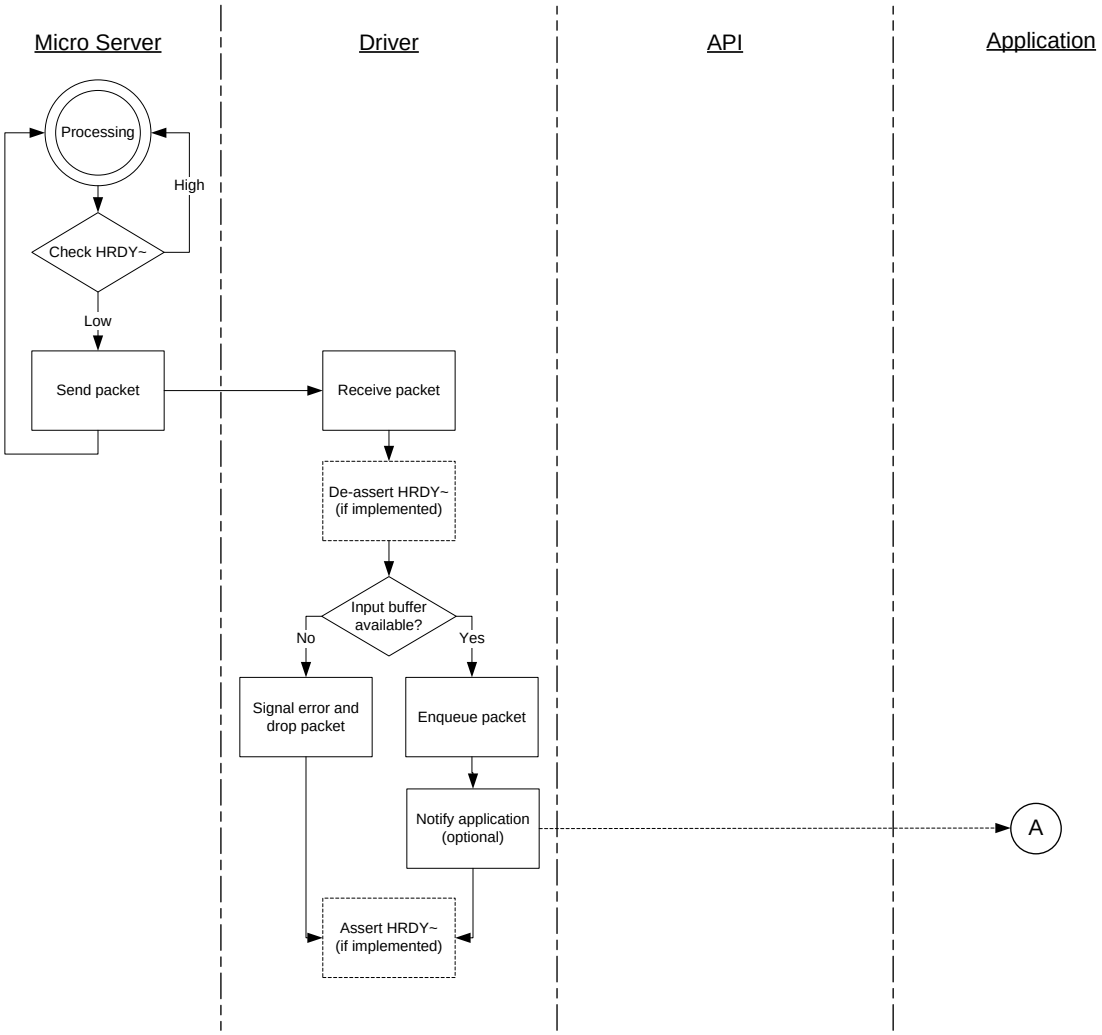


Figure 42. SCI Uplink Operation (Part 1)

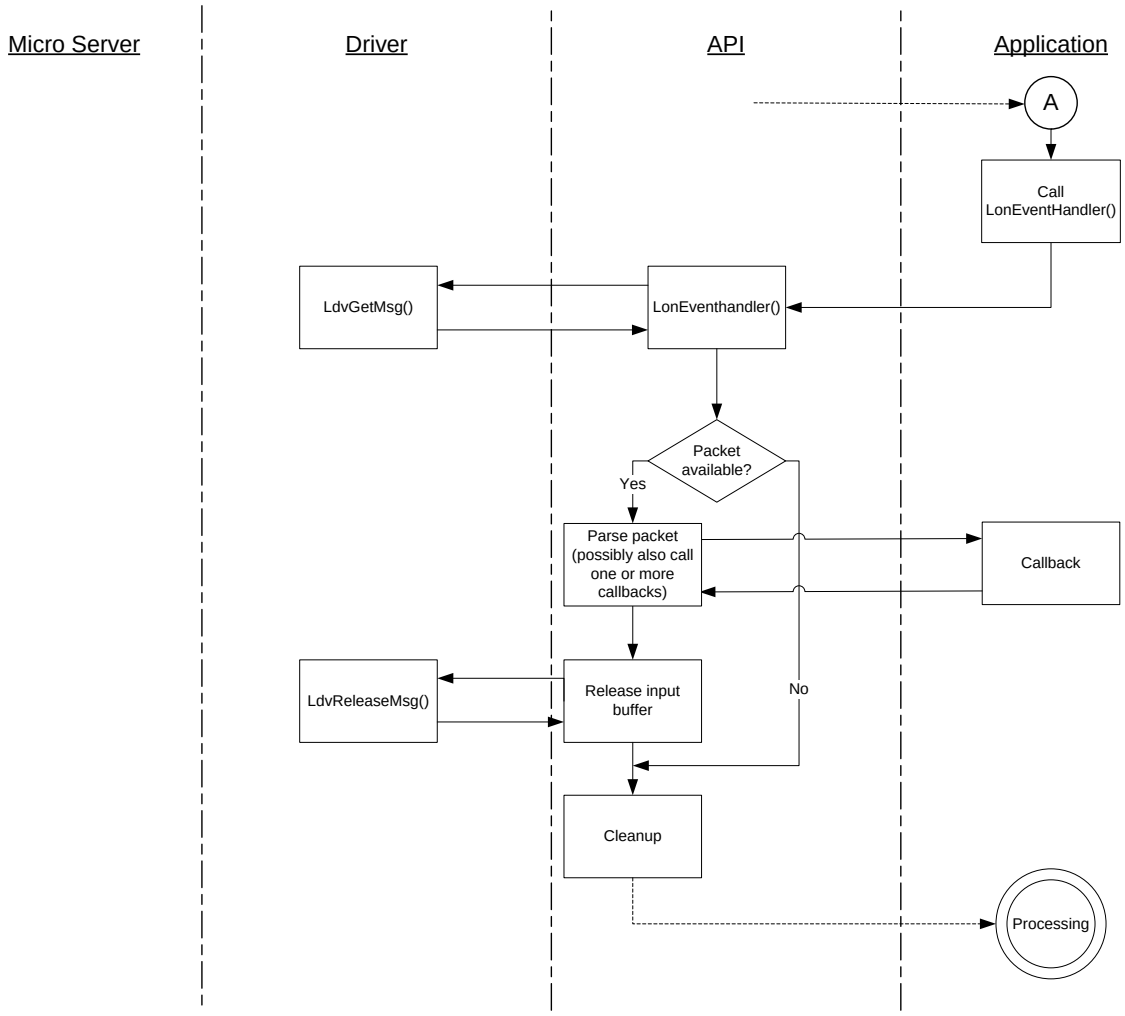


Figure 43. SCI Uplink Operation (Part 2)

SCI Downlink Operation

In an SCI downlink operation, data is transferred from the host processor to the ShortStack Micro Server. Figure 44 shows the activity that the driver should manage for a downlink operation. Figure 45 shows the SCI handshake and data transfer for the header, extended header, or payload.

To send a message downlink, the driver must 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~** signal 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~** signal 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 needs to 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. Your host driver must allow for such an uplink. Blocking such an uplink, for example by de-asserting the **HRDY~** signal, can cause a fatal deadlock to occur.

No uplink can occur while the **CTS~** signal is asserted.

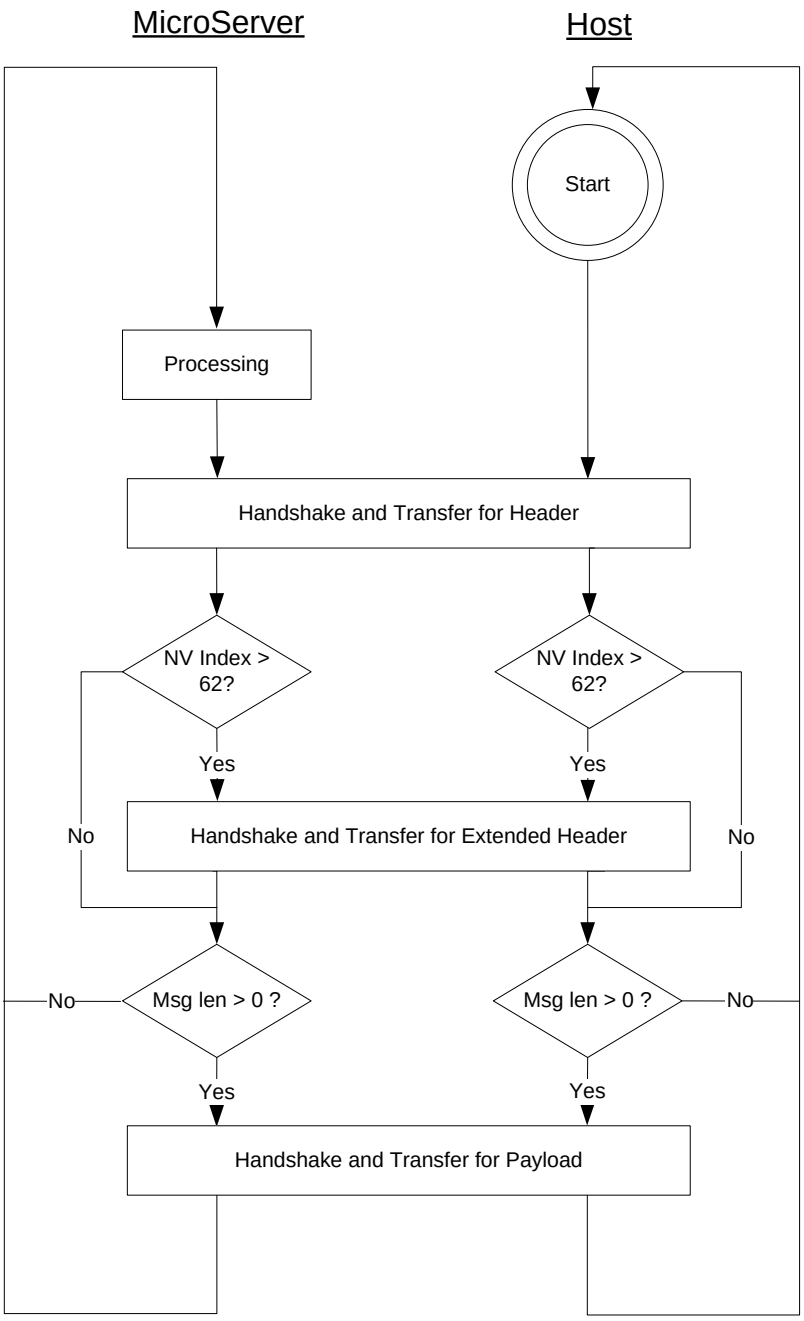


Figure 44. Downlink Operation

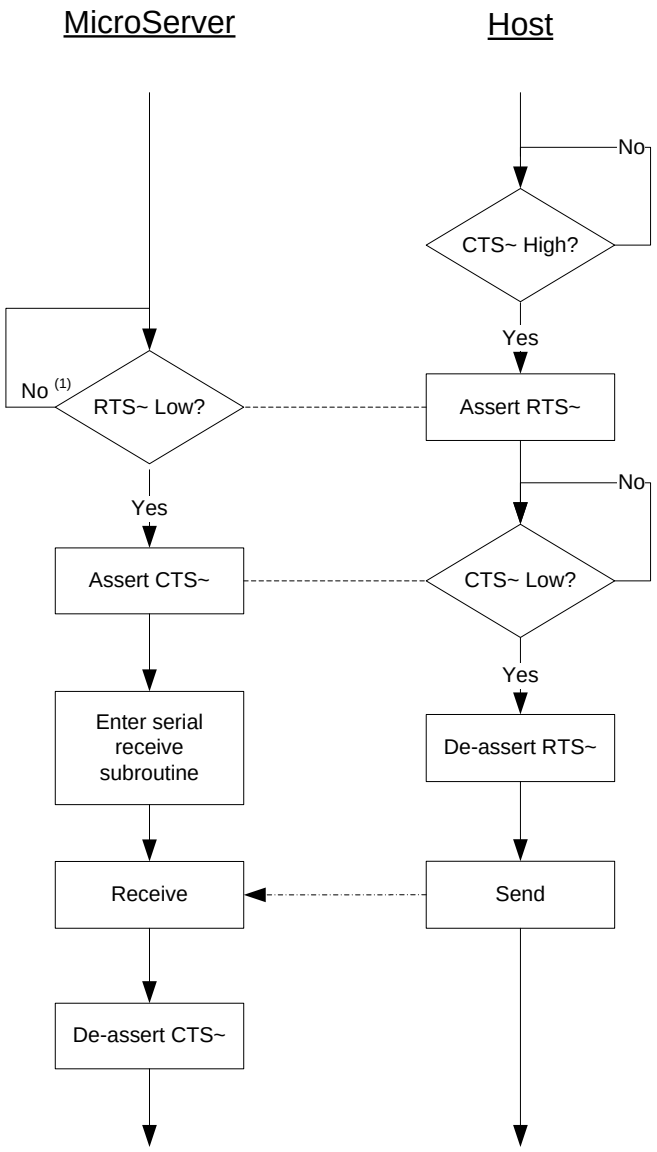


Figure 45. SCI Handshake and Data Transfer

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 6000 or 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.

Your driver must be prepared to accept uplink transaction while it awaits approval of a pending downlink handshake. This is crucial, because the uplink transfer of data might be required to make a buffer available for use by the pending downlink, and failure to accept uplink data at this time could lead to a fatal link layer deadlock situation.

Network Variable Fetch Example

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 46 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.

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

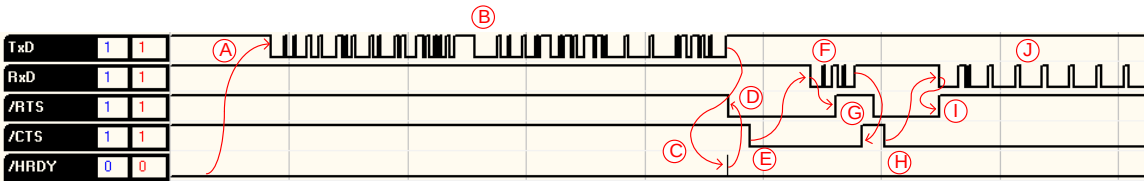


Figure 46. Logic Analyzer Trace for an NV Fetch

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*.

SPI Uplink Operation

In an SPI uplink operation, data is transferred from the ShortStack Micro Server to the host processor. Figure 47 and Figure 48 show the activity that the driver needs to manage for an uplink operation. The figures also show how the Micro Server, serial driver, ShortStack LonTalk/IP Compact API, and the application interact to process an uplink message. The driver must sense 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~** signal 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.

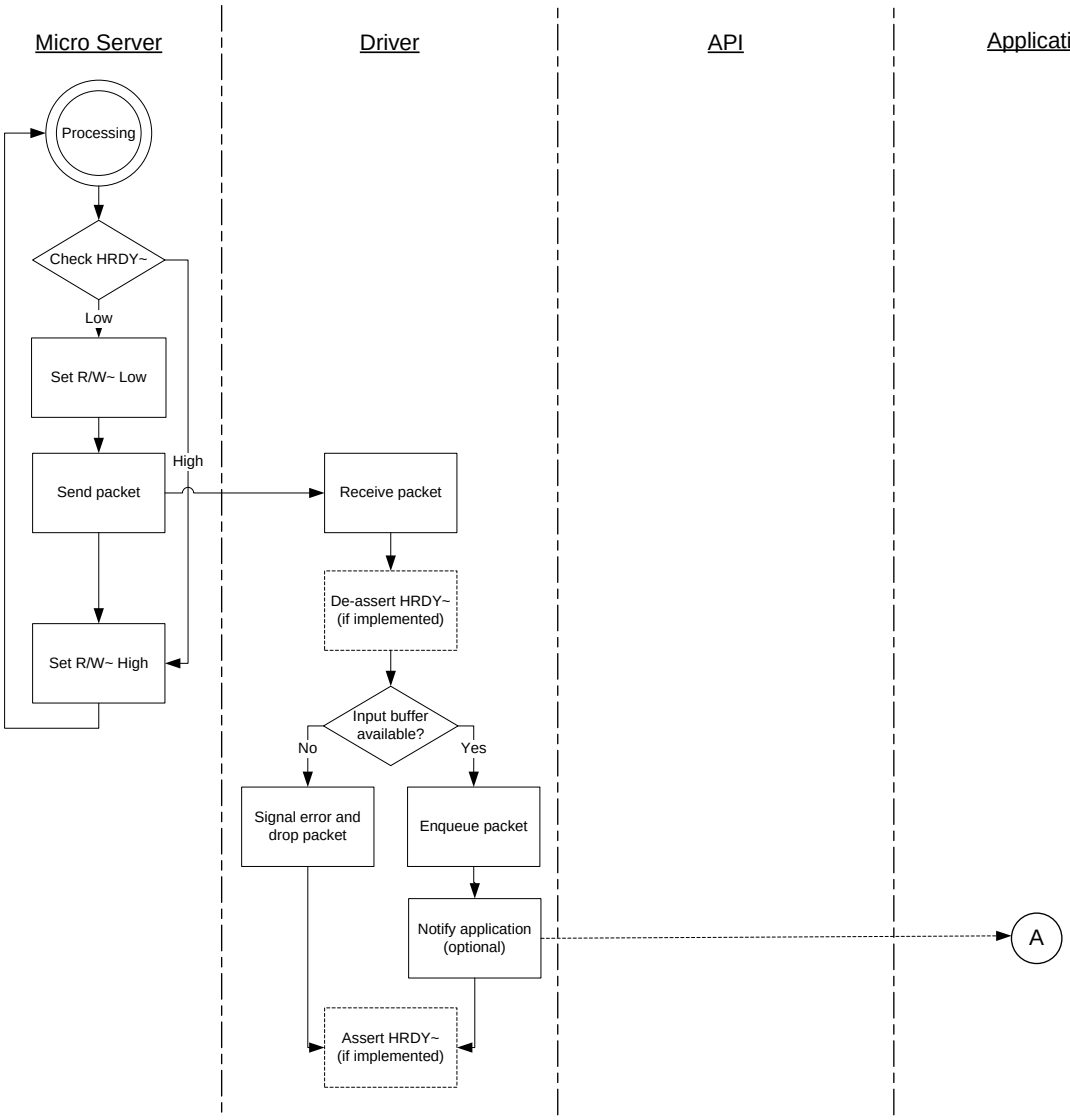


Figure 47. SPI Uplink Operation (Part 1)

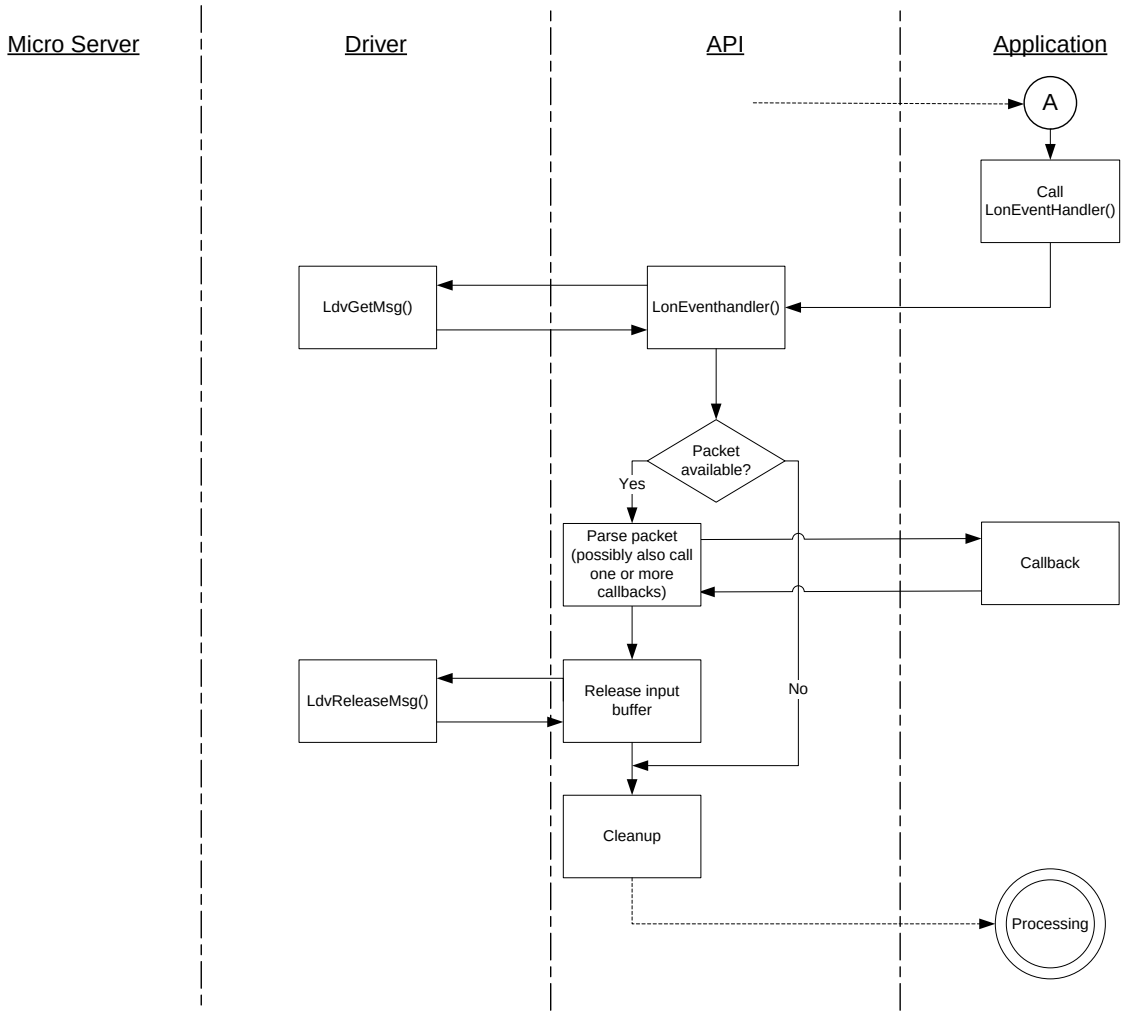


Figure 48. SPI Uplink Operation (Part 2)

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 49 shows the activity that the driver needs to manage for a downlink operation (this figure is the same as Figure 44). Figure 50 shows the SPI handshake and data transfer for the header, extended header, or payload. The driver needs to sense 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)*, the host must detect possible write collisions during data transfer.

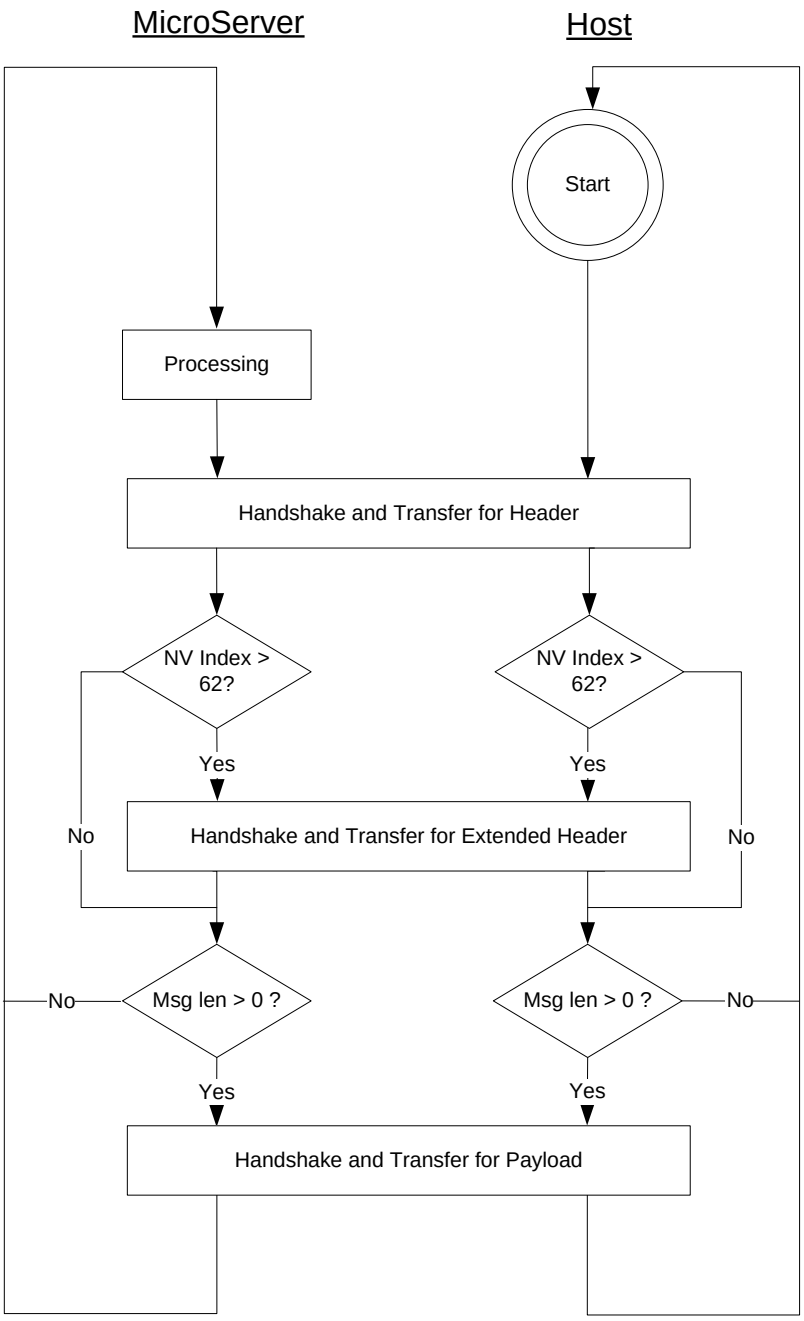


Figure 49. Downlink Operation

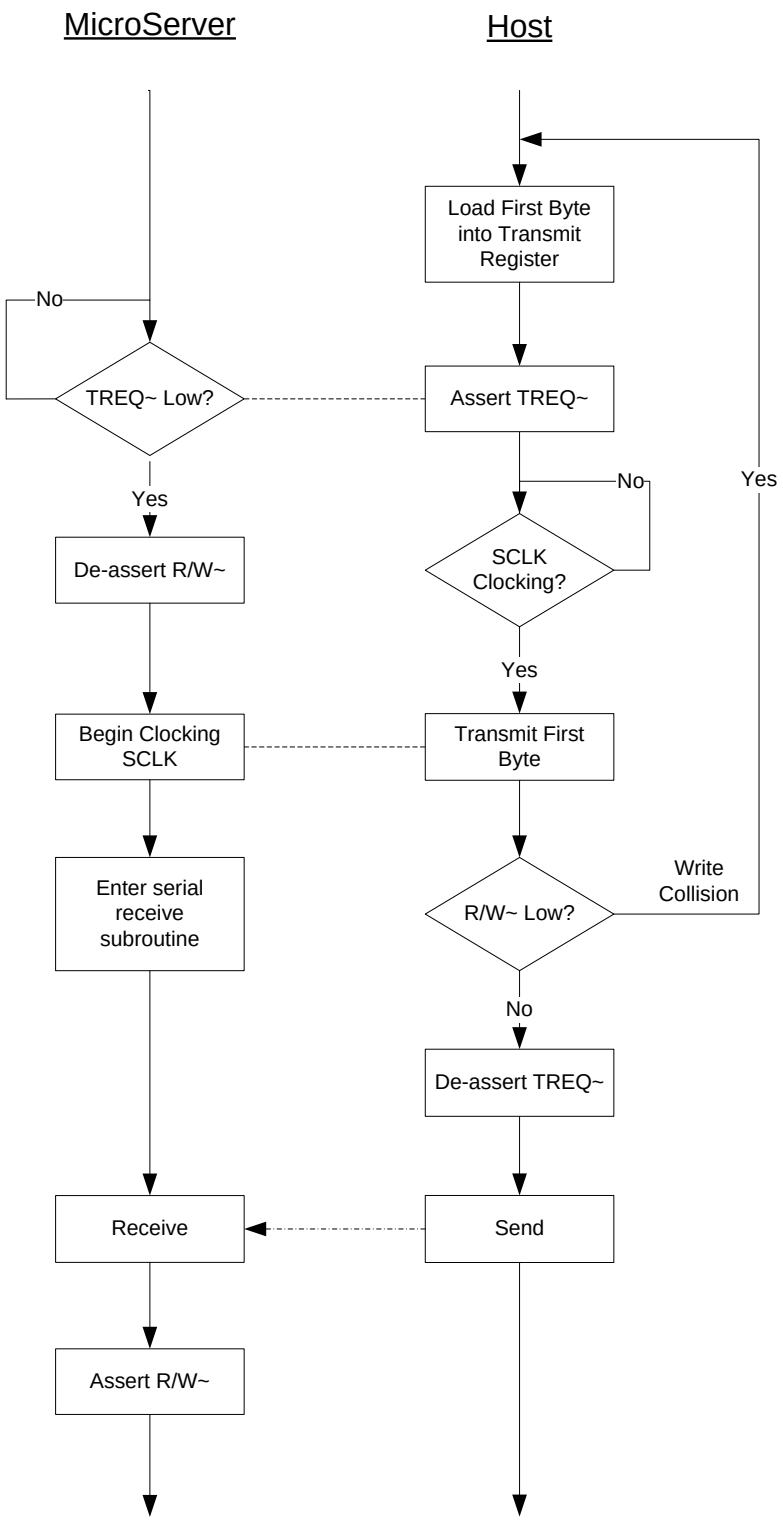


Figure 50. SPI Handshake and Data Transfer

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 handle such delays.

Transmit and Receive Buffers

The ShortStack serial driver must 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.

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
```

The transmit and receive buffers within the host cannot be smaller than those defined in the Micro Server.

The IzoT ShortStack SDK example application and driver for use with a Raspberry Pi computer and the Raspbian Linux operating system includes the implementation of a simple protected queue (**ldvq.h**, **ldvq.c**) with configurable limits similar to those discussed. See the source code in the **ldvq.c** file for a discussion of implementation details, configuration options, and ramifications.

Link-Layer Error Detection and Recovery

The ShortStack Micro Server and the ShortStack LonTalk/IP 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 an inter-byte timeout for both the serial receiver and transmitter. If the receiver timer expires, the current message is discarded. If the transmitter timer expires, the current message is resent later.

Your serial driver must 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 required.

The IzoT ShortStack SDK example applications and driver for use with the Raspberry Pi computer and the Raspbian Linux operating system include several configurable timeout values and extensive discussion, embedded within the **rpi.c** implementation file.

Review this example driver and the embedded commentary even if you do not plan on using a Raspberry Pi or a Linux operating system.

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 repeatedly sends echo requests with different data and confirms that the data received meets expectations. Data errors detected during such a test may 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*, *Local Utility Callback Handler Functions*, 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 can log an error and signal the event to the user. You can also optionally assert the Micro Server's reset line in an attempt to recover 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 must load the ShortStack application into the host processor.

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, or a secure shell (SSH) connection to the target device. In some cases, you may even create, manage and compile your source code on the device itself.

The IzoT ShortStack SDK examples for use with the Raspberry Pi, for example, assume that you either work locally on the Raspberry Pi, or use a cross-compilation toolchain and remote debugger from another computer.

Performing an Initial Host Processor Health Check

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

For an initial health check of the host, use a Micro Server that you tested according to the test described in *Performing an Initial Micro Server Health Check* .

To do a basic health check for the host, follow these steps:

1. When using SCI, disconnect the host from the Micro Server, and verify that your serial driver can transmit data correctly.

   Add a jumper wire or pushbutton to simulate the Micro Server's assertion of the **CTS~** signal for this test.

2. When using SCI, connect your host's transmit and receive signal and verify that your serial driver can receive the data it sent.

   Add a jumper wire or pushbutton to simulate the Micro Server's assertion of the **CTS~** signal for this test.

3. Connect the host to the Micro Server, and supply power to both
4. Issue a downlink reset command (command code 0x50)
5. Observe that the Micro Server resets
6. 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 may provide a more visible state change on the Reset LED. You can use an oscilloscope or logic analyzer to capture the Reset pulse.

During this and similar tests in the early stages of development, you can also monitor the Reset signal, because errors in the host-side driver implementation can cause the Micro Server to reset. For example, if the host asserts the **RTS~** signal, 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~** signal in a timely fashion, the Micro Server may 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 6000 or 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* 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 can 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 can 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.

9

Porting the ShortStack LonTalk/IP Compact API

If you are using a host processor and development environment that does not have an available IzoT ShortStack SDK example port, you must port the ShortStack LonTalk/IP 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/IP Compact API.

Portability Overview

The ShortStack LonTalk/IP 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/IP Compact API implements the following portability features:

- 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 following macros control structure declarations:

  ```
  #define LON_STRUCT_BEGIN(n) struct
  __attribute__((__packed__))
  #define LON_STRUCT_END(n)  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 IzoT ShortStack SDK 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/IP Compact API files, and the **LonEnd.h** file can be optionally (and automatically) included following the last type definition made by the ShortStack

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

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

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

And the **LonEnd.h** file may 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) need to be based on a signed character type (or a signed eight-bit integer). The ShortStack LonTalk/IP 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 may defined as:

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

- The ShortStack LonTalk/IP 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 **LonPlatform.h** which resides in the **api** folder within your IzoT ShortStack SDK source code repository for complier-specific definitions used by the LonTalk Interface Developer.

The definition of unions, structures, and enumerations using the **LON_BEGIN_*** and **LON_END_*** macros introduced above provide a hook to accomplish the correct definition of those items as required by the IzoT ShortStack SDK, however, these definitions can confuse other source code parsers such as automated source code formatting tools or other non-standard source code processors.

Use automatic source code formatting tools with caution in context with your IzoT ShortStack SDK application's source code.

Bit Field Members

For portability, none of the types that the IzoT Interface Interpreter or LonTalk Interface Developer generates use bit fields. Instead, the tools define bit fields with their enclosing bytes, and provide 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 IzoT Interface Interpreter and the LonTalk Interface Developer utility do not produce enumerations. The ShortStack LonTalk/IP 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 can use the **LON_ENUM_*** macros instead of the enumeration's name.

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

Table 20. Enumerations in ShortStack

ShortStack Enumeration	Equivalent ANSI C Enumeration

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

LonPlatform.h

The file within the ShortStack LonTalk/IP Compact API that helps implement the portability concepts described in *Portability Overview* is the **LonPlatform.h** include file. The ShortStack LonTalk/IP Compact API and application framework automatically include this file before any other ShortStack LonTalk/IP 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/IP Compact API, ensure that the **LonPlatform.h** file includes support for your compiler. **LonPlatform.h** resides in the **api** folder within your IzoT ShortStack SDK source code repository.

After you make the appropriate modifications to the **LonPlatform.h** file, you can compile the ShortStack LonTalk/IP Compact API files and the application framework generated by the IzoT Interface Interpreter.

Testing the Ported API Files

After the ShortStack LonTalk/IP Compact API files and the application framework generated by the IzoT Interface Interpreter compile without errors or significant warnings, you can perform a simple test to ensure that the port works correctly.

For this simple test, use your driver and API port with a very basic test application, such as the Simple application example located in your **example/rpi/simple** folder within your IzoT ShortStack SDK project folder.

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/IP Compact API and the device interface data produced by the IzoT Interface Interpreter to perform the following tasks:

- Use the ShortStack LonTalk/IP Compact API
- Use the API with a multitasking operating system
- Initialize the ShortStack LonTalk/IP Compact API
- Periodically call the ShortStack event handler
- Exchange network variable data with other devices
- 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

Most ShortStack applications 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/IP Compact API

Within the seven-layer OSI Model protocol, the ShortStack LonTalk/IP 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 51.

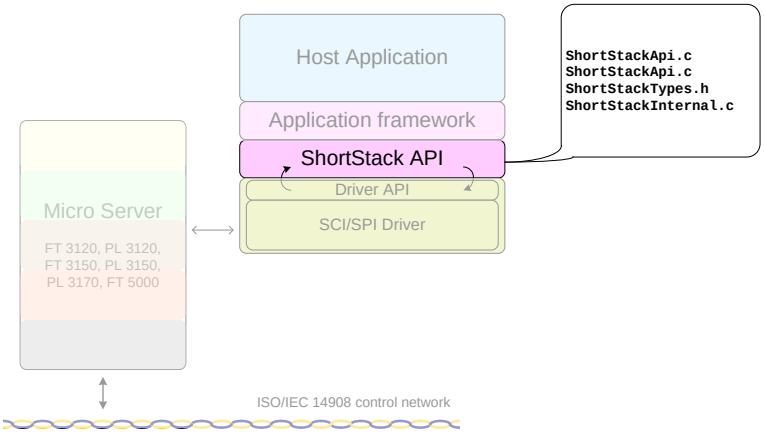


Figure 51. The ShortStack LonTalk/IP Compact API within the OSI Model

The ShortStack LonTalk/IP Compact API is implemented primarily in 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/IP Compact API, which includes functions for handling network events, propagating network variables, and responding to network variable poll requests.

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 will not have to modify the ShortStack API files for each of your applications, but you may 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 52.

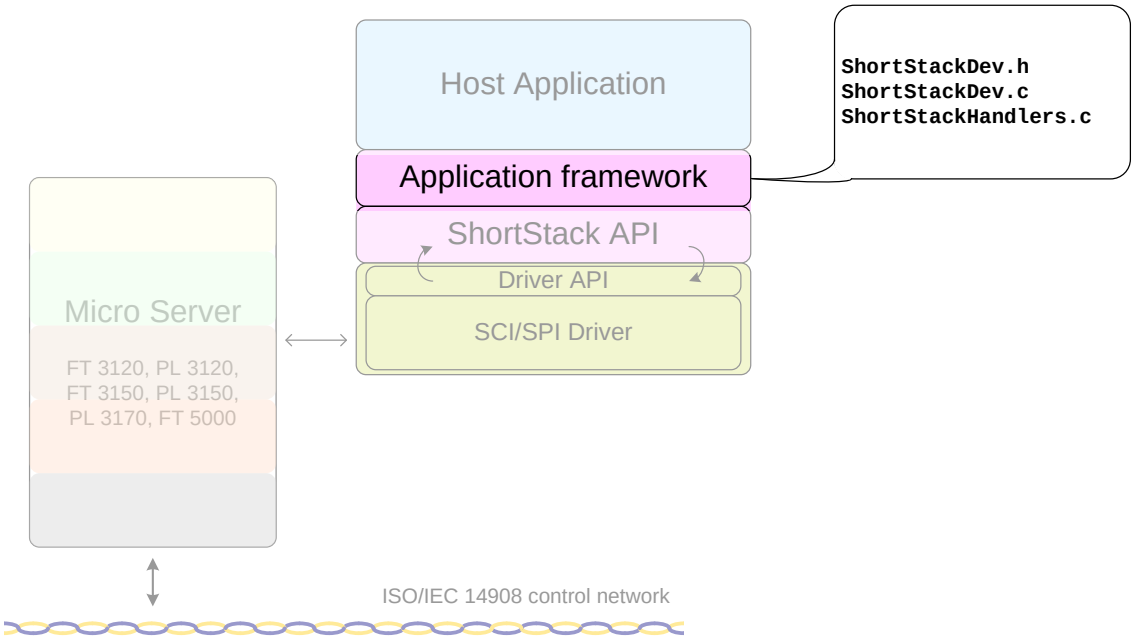


Figure 52. The ShortStack Application Framework

Figure 51 and Figure 52 do not show the API or framework files that are required for ShortStack ISI applications; see *Developing a ShortStack Application with ISI*, for information about supporting ISI in your ShortStack application.

Your main C source file contains the definition of datapoints (network variables), properties and blocks, general preferences related to your ShortStack device, and handlers for most common event types.

The **ShortStackHandlers.c** source file contains stubs for handler functions for the less common event types. Review these handlers and add code to these callback stubs if necessary.

Using Multiple System Execution Contexts

Although a ShortStack application does not require an operating system, you can use the ShortStack LonTalk/IP 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 may use one or more execution contexts for the link-layer driver, and may use a different execution context for both the ShortStack LonTalk/IP Compact API functions and callback handler functions.

The ShortStack LonTalk/IP 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/IP Compact API, you must ensure that only one task (or thread or interrupt) accesses the ShortStack LonTalk/IP Compact API. The same task that calls the **LonInit()** and **LonEventHandler()** functions must also be the only task that calls the ShortStack LonTalk/IP Compact API.

In a multi-tasking environment, the link-layer driver typically consists of USART transmit and receive interrupts or threads, possibly also using interrupts that respond to changes on the link-layer handshake lines.

The IzoT ShortStack SDK example applications use a single execution thread for the driver, serving both uplink and downlink network communications. A pair of protected input (uplink) and output (downlink) queues and a pipe is used to communicate between the main application thread and the driver.

If your application requires the use of multiple contexts, you can provide one execution context that calls the **LonEventHandler()** function. You can also supply appropriate inter-context communication and synchronization tools to guard every API function, for example by implementing a mutex requested by the LonEventHandler support context and by any other context which might call any of the API functions.

Events and callbacks execute in the context which calls the **LonEventHandler()** function. You must take additional precautions to prevent a deadlock when an event handler itself calls a protected API.

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 LonTalk/IP or LON network communication, such as the communication parameters for the physical transceiver in use, and defines the application's device interface: its functional blocks, 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 have 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. The ShortStack LonTalk/IP Compact API automatically determines 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 change acts as a notification that the application, but not necessarily the Micro Server and the ShortStack device as a whole, must reinitialize.

When the host processor powers-up or resets, you must reinitialize the ShortStack device.

When your driver recognizes an uplink reset message, ensure that any in-progress downlink activity is immediately aborted. This is required to prevent a synchronization failure for the link layer. The link layer is said to be *out of synchronization* when the Micro Server and host disagree on the type of the next downlink segment. The Micro Server might expect a header while the host transmits payload corresponding to a header sent prior to the Micro Server reset.

Be prepared to receive uplink data at all times, and particulary between the segments of a downlink transfer. The ShortStack link layer is half-duplex so that data is only transferred into one direction at a time, but a two- or three-segmented downlink transfer is not atomic, and may be interrupted by uplink transfers.

These uplink messages can be crucial for continued operation. For example, one such uplink message could deliver a completion code for a transaction started earlier. Delivery of this completion code could be required to unlock a buffer required for the next transaction.

During normal processing, the application periodically calls the **LonEventHandler()** API function, which calls the serial driver API and might call callback functions and event handlers (such as the **onUpdate** events). Other API functions allow the ShortStack application to initiate transactions. Such a transaction might in turn lead to other events, such as the **onComplete** event.

The following sections describe the tasks that an IzoT ShortStack SDK 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/IP 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.

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 LonTalk/IP or LON 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/IP Compact API*).

The host application must be prepared to process the maximum rate of LonTalk/IP or LON 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 not sufficient.

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 can call the **LonEventHandler()** function approximately once every 10 ms.

Again using the formula, a typical PL-20 power-line device can call the **LonEventHandler()** function once every 100 ms. However, to ensure low network latency, all ShortStack devices can 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 must defer time-consuming operations (such as lengthy flash writes) whenever possible.

Exchanging NV Data with Other Devices

Your application implements input and output datapoints, either as members of blocks, or as simple device datapoints. Each datapoint implements a network variable, and provides additional properties such as the *global_index* member.

Example

This example implements a generic standard closed loop actuator profile in a block called **act**. The profile has one mandatory input and one mandatory output, called **nviValue** and **nvoValueFb**, but no particular data type is stipulates for these. The following example uses the **SNVT_volt** standard data type to implement the actuator. It also declares an **onUpdate** event, which executes whenever the input received new data. Within that event, the algorithm assigns 3 plus the value of the input to the output, then triggers propagation of the output to the network and connected devices.

```
SFPTclosedLoopActuator(a, SNVT_volt) act; //@IzoT Block \
//@IzoT onUpdate(nviValue, onActuatorUpdate)

void onActuatorUpdate(
    const unsigned index,
    const LonReceiveAddress* const pSourceAddress
) {
   LON_SET_UNSIGNED_WORD(
      act.nvoValueFb.data,
      3 + LON_GET_UNSIGNED_WORD(act.nviValue.data)
   );
   LonPropagateNv(act.nvoValueFb.global_index);
}
```

Communicating with Application Messages

You can use application messages to exchange data or requests with other LonTalk/IP or LON devices. Application messages are used by applications requiring a different data interpretation model that the one used for network variables. An application message is a message packet with a 6-bit message code that identifies the packet to the receiving application or applications. The applications exchanging application messages must agree on the interpretation of

the message codes. 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 encapsulating packets using other protocols), 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 *Developing a ShortStack Application with ISI*, 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 21**. For user-defined application messages, you can 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 6. 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.
Reserved for Standard Application Messages	48 to 60 (0x30 to 0x3C)	Standard application messages defined by LONMARK International.
ISI Messages	61 (0x3D)	Standard application messges defined by the Interoperable Self Installation (ISI) protocol
FTP Messages	62 (0x3E)	Standard applications messages defined by the LONWORKS File Transfer Protocol (LW-FTP)
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.

Message Type	Message Code	Description
Foreign Responder Offline	79 (0x4F) Protocol V0	Used by foreign frame responses. Indicates that the sender of the response was in an offline state and could not process the request.
LonTalk/IP UDP Messages	79 (0x4F) Protocol V2	Used for LonTalk/IP UDP messages that are not encoded with the LonTalk/IP Control Services defined by the ISO/IEC 14908-1 Control Networking Protocol
Network Diagnostic Messages	80 to 95 (0x50 to 0x5F)	Used by network tools for network diagnostics.
Network Management Messages	96 to 115 (0x60 to 0x73)	Used by network tools for network installation and maintenance.
Router Configuration Messages	116 to 124 (0x74 to 0x7C)	Used by networks tools for router management
Network Management Escape Code	125 (0x7D)	Used by network management tools for inter-component communication
Router Far Side Escape Code	126 (0x7E)	Used by network tools to address management messages to the far side of a router
Service Messages	127 (0x7F)	Used for reporting the Neuron ID or MAC ID of a device
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 may have one byte of data in one instance and 25 bytes of data in another instance.

Each message tag is created with a **tag** IML directive. The IzoT Interface Interpreter assigns individual values to all tags during initialization.

Example

```
LonTag myTag; //@IzoT Tag
```

Sending an Application Message

You can send an application message by calling the **LonSendMsg()** function. 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/IP 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/IP Compact API, which in turn calls your **LonResponseArrived()** callback handler function for each response it receives.

Figure 53 shows the control flow for sending an application message.

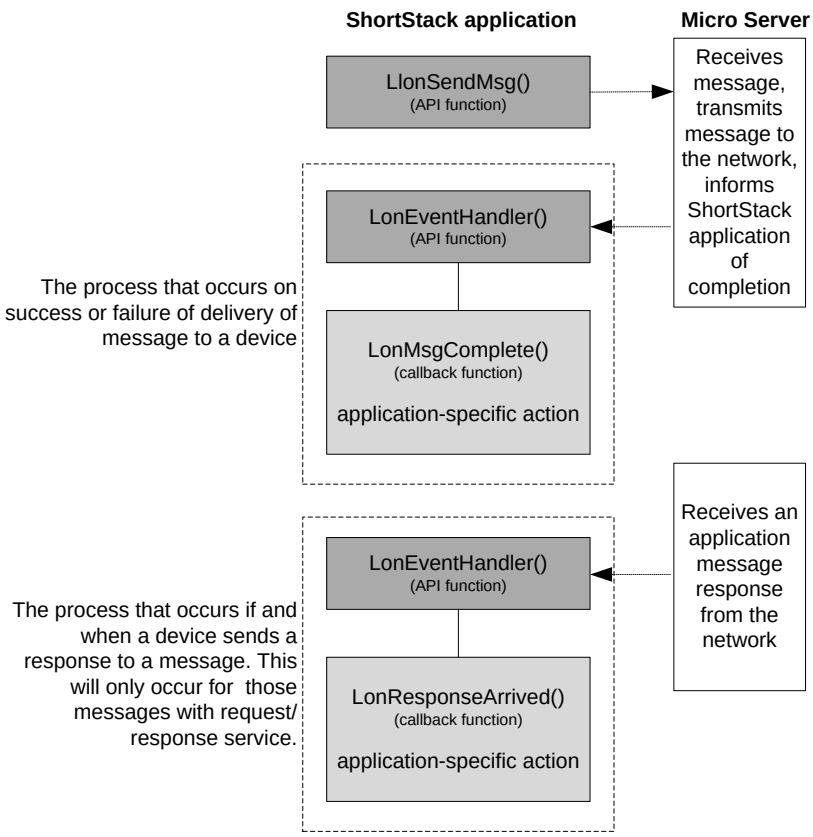


Figure 53. Control Flow for Sending an Application Message

Receiving an Application Message

When the ShortStack Micro Server receives an application message from the network, it forwards the message to the **LonEventHandler()** function in the ShortStack LonTalk/IP 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 54 shows the control flow for receiving an application message.

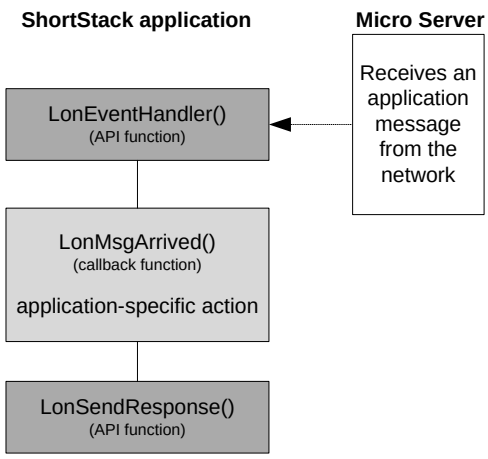


Figure 54. Control Flow for Receiving an Application Message

Handling Management Tasks and Events

LonTalk/IP and LON 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/IP 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.

The IzoT Interface Interpreter supports several event types, and automatically implements the corresponding callback functions. These are **LonNvUpdateOccurred()**, **LonNvUpdateCompleted()**, **LonWink()**, **LonOffline()**, **LonOnline()**, **LonReset()**, **LonServicePinPressed()**, **LonServicePinHeld()** and **LonGetCurrentNvSize().**

All other callback handers are defined as empty skeletons within **ShortStackHandlers.c**, which is located in your **api** source folder.

You can add your callback handler code to this file, and you can provide application-specific implementations of callback functions *outside* the standard **ShortStackHandler.c** file.

To indicate that you supply the implementation of callback X, define the **X_HANDLED** preprocessor symbol in your project preferences. For example, to indicate that you supply the **LonNvConfigReceived()** callback outside ShortStackHandlers.c, define the **LONNVCONFIGRECEIVED_HANDLED** preprocessor symbol in your project preferences.

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/IP 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 55 through 58 show the control flow for handling these kinds of network management commands.

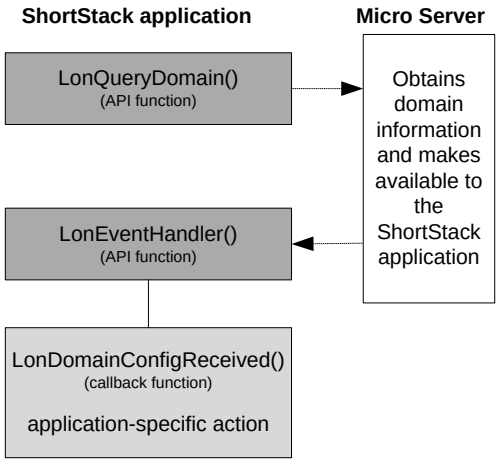


Figure 55. Control Flow for Query Domain Network Management Command

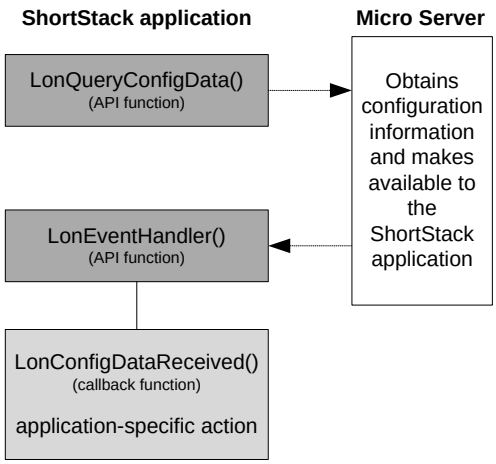


Figure 56. Control Flow for Query Configuration Data Local Network Management Command

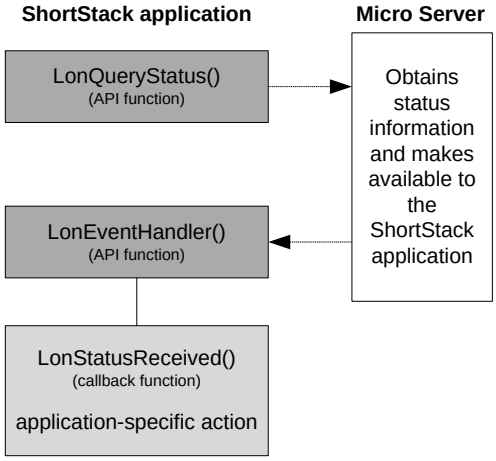


Figure 57. Control Flow for Query Status Local Network Management Command

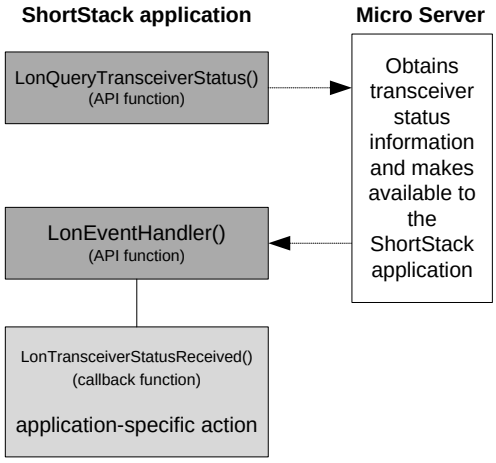


Figure 58. Control Flow for Query Transceiver Status Local Network Management Command

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 IzoT Interface Interpreter supplies reset notifications to the optional onReset event.

The **LonResetNotification** structure contains the following information:

- The State of the Micro Server
- The Version of the link layer protocol (3 for the ShortStack 2.1 SDK; 4 for the IzoT ShortStack SDK and ShortStack FX SDK)
- Information about availability and state of the static IO9 input signal on the Micro Server (see *Using the IO9 Pin*)
- Information about whether the Micro Server is initialized
- Information whether the Micro Server supports an extended address table (the Micro Server must be running on a Series 6000 processor to support an extended address table)
- The Micro Server Key (see *Using the ShortStack Micro Server Key*)
- 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, or a MAC ID for Series 6000 processors)
- 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.

In addition to the standard system error codes (129 and above), a ShortStack Micro Server can log ShortStack-specific system error codes that help you diagnose problems.

Table 22 lists the ShortStack-specific system error codes. All system error codes are provided by the **LonSystemError** enumeration in **ShortStackTypes.h**.

Table 7. 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 the IzoT ShortStack SDK.
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.
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.

Runtime Interface Selection

Most IzoT applications have one static interface; that is, the set of datapoints, properties and blocks, their attributes and relations, and a number of related aspects are defined by you when you create the application. The interface data changes during the lifetime of your application, for example, by exchanging datapoint values with other devices in the network. However, the interface itself remains static; no datapoint or block is added or removed during the lifetime of your device.

Some advanced devices implement dynamic interfaces, which support the addition, modification, and removal of datapoints or blocks either at installation time or during the lifetime of the application. Dynamic interfaces are an advanced feature and require an advanced protocol stack. The IzoT ShortStack SDK does not support creating devices with dynamic interfaces.

Applications with runtime interface selection fill the gap between static and dynamic interfaces. For example, an application which supports five different interfaces subject to different purchase options, but supports only one interface at any one time. The same application may support a low-cost entry-level model while a higher priced variant adds premium features and exposes a different interface. Alternatively, the application may support expansion with external hardware modules that require a different interface for each module.

This application could be configured at manufacture time, for example, with a sealed hardware jumper or an application message. Another application could allow the user to install node-locked license files and purchase additional features over time.

The IzoT ShortStack SDK and the IzoT Interface Interpreter do not require a specific method in which those are runtime-selectable interfaces are licensed or managed. However, the IzoT Interface Interpreter can help you define and manage applications with multiple interfaces where exactly one interface is active at any one time.

This architecture and related considerations are discussed in the remainder of this section.

The IzoT ShortStack SDK includes an application example in the **examples/rpi/ris** folder which demonstrates the features discussed here.

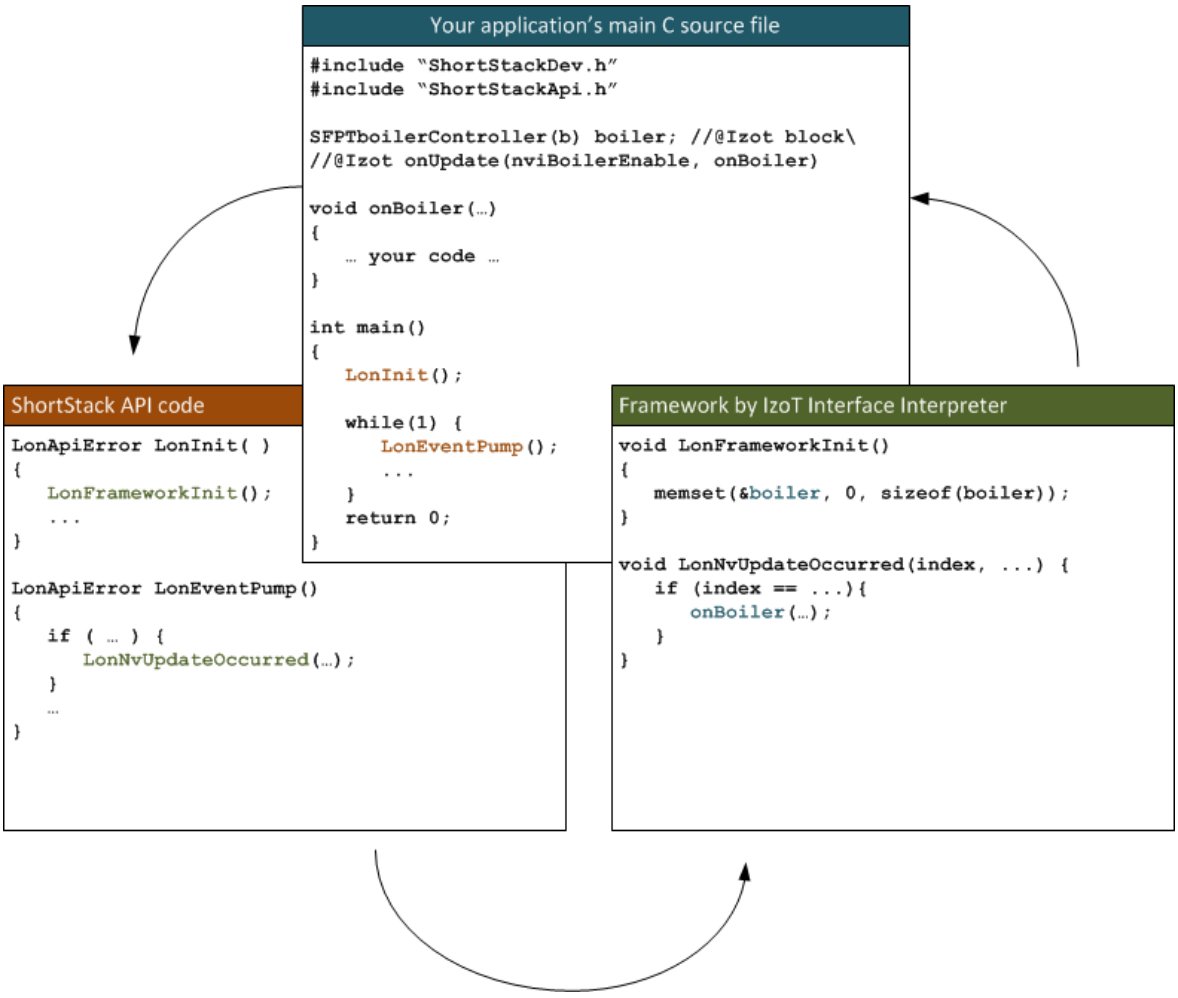
Static Interface Framework

The structure of a typical static interface is illustrated with the Simple example included in the IzoT ShortStack SDK **examples/rpi/simple** folder.

This application has a main C source file, **rpi-simple.c** in this example. This file contains the declaration of the interface in the IzoT Markup Language, and it contains the standard C **main()** function. Within the **main()** function, the code

calls the **LonInit()** API function, then makes periodic calls to the **LonEventPump()** API.

The following illustration shows some of the function call sequences in this process. The illustration is neither accurate nor complete, but is used to demonstrate the principle of operation between your source code, the ShortStack API code, and the framework code generated by IzoT Interface Interpreter.



The illustration shows how your **main()** function calls the **LonInit()** API, which in turn calls a **LonFrameworkInit()** function generated by the IzoT Interface Interpreter. This function initializes the **boiler** block defined in the example main C file.

Likewise, your application makes periodic calls to the **LonEventPump()** API. When this API detects that a network variable update occurred, it invokes the corresponding callback function, **LonNvUpdateOccurred()**, which the IzoT Interface Interpreter generates. This callback dispatches the update notifications into your event handlers, **onBoiler** in this example.

A similar mechanism applies to all other events supported by the IzoT Markup Language, and to a number of additional callbacks related to application lifetime management and initialization. For example, the data required to register your

application with the ShortStack Micro Server is defined within the framework files generated by IzoT Interface Interpreter.

Key to the the static interface framework is that the framework generated by IzoT Interface Interpreter resides in the **ShortStackDev.h** and **ShortStackDev.c** files, and includes the callback functions required by the ShortStack API.

Runtime Interface Selection Framework Architecture

The architecture of an application with runtime interface selection expands on that of a static interface application.

In an application with runtime interface selection, each interface is defined in its own source file. For example, the Runtime Interface Selection (RIS) application example defines a simple interface in **regular.c** and the premium interface version in **deluxe.c**.

Each interface contains the complete definition of the interface, including block declarations and event handlers, and each of these source files is processed by the IzoT Interface Interpreter prior to compilation.

For example, the RIS example in the IzoT ShortStack SDK configures the pre-build step, normally defined as **iii "${ProjDirPath}/${ProjName}.c"**, as

```
cmd /C iii "$(ProjDirPath}\regular.c" && iii
"$(ProjDirPath}\deluxe.c"
```

This passes the *regular* interface and the *deluxe* interface through the IzoT Interface Interpreter as two separate interfaces.

Because each interface is different, you must specify a unique program ID for each interface. The IzoT Interface Interpreter also generates device interface files (.XIF file extension) for use with network tools, one for each interface. The device interface files share the interface name. In this example, you will obtain **regular.xif** and **deluxe.xif**.

Option Output

All but one of the interfaces include the IML **option output** directive to request that the pair of generated output be named different than the **ShortStackDev** default.

Example

```
//@IzoT Option output("regularDev")
```

The most complex of your interfaces, however, does not use option output. The framework for the most complex interface will be located in the **ShortStackDev.c** and **.h** files. The ShortStack API requires that the **ShortStackDev.h** file exists, and is subject to conditional compilation and compile-time configuration based on symbols defined in this file.

In order to obtain an API configured for the superset of all features required by all your interfaces, the most complex of your interfaces will usually be a good choice for generating **ShortStackDev.c** and **.h** files.

Make sure not to re-use the interface file's name in the option output directive, because the IzoT Interface Interpreter will parse your interface and then overwrite it with the generated framework. The above example uses the name of the interface followed by **Dev** in analogy to **ShortStackDev**.

Option Namespace

All interfaces select a non-default *namespace*.

Example

```
//@IzoT Option namespace("regular")
```

The namespace is a prefix used with the generated callback functions. Without option namespace (the default), the IzoT Interface Interpreter generates callback functions including **LonResetOccurred**, **LonWink**, and **LonNvUpdateOccurred**.

Only one of each can exist in any single application, as linker errors would otherwise occur. The **Option namespace** directive solves this problem. For example, **option namespace("regular")** yields callbacks named **regularLonResetOccurred**, **regularLonWink**, and **regularNvUpdateOccurred**.

Another interface in the same application can specify the *deluxe* namespace with **option namespace("deluxe")** and thus yield **deluxeLonResetOccurred**, **deluxeLonWink**, or **deluxeLonNvUpdateOccurred**.

Callback Dispatch

The ShortStack API requires that you present implementations of the standard ShortStack API callback functions with their native names, e.g. **LonResetOccurred**, **LonWink**, or **LonNvUpdateOccurred**. You must implement all these functions with their correct native names and correct prototypes.

For applications with runtime interface selection, the **option namespace** directive redirects the implementations of these callbacks to differently named entry points, and you must provide the regular callback functions.

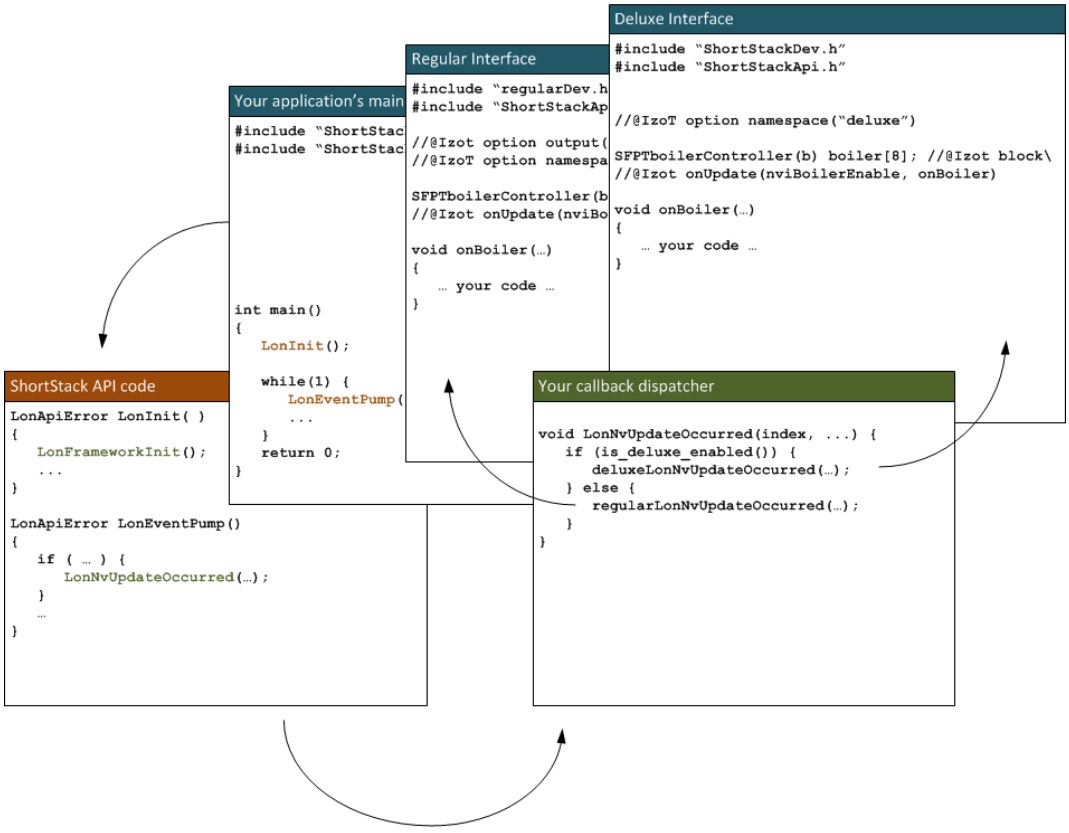
In your implementation of these callbacks, you select which interface's implementation to call, based on your knowledge of the currently selected interface type. For example, your callback dispatcher may sample a hardware input pin to select between the regular and the deluxe interface, and route the callback accordingly.

Example

```
void LonWink(void)
{
  if (is_deluxe_enabled()) {
     deluxeLonWink();
  } else {
     regularLonWink();
  }
}
```

The RIS application example included with the IzoT ShortStack SDK contains an example dispatcher implementation in the **dispatch.c** file.

This illustration shows how the callback dispatcher intercepts and re-routes the callbacks.



Interface Selection

The IzoT Interface Interpreter and the ShortStack LonTalk/IP Compact API have no requirements on how you select, license, or manage your interfaces, except the following standard requirements of all interoperable devices:

- Every interface needs to implement its own, unique, program ID
- Exactly one and only one interface can be active at all times on each device

The RIS application example, which is part of the IzoT ShortStack SDK, uses an insecure simple console input to change the interface, and stored the current interface selection in an unprotected file.

The following non-exhaustive list offers some suggestions for other methods of selecting the active interface:

- Use a simple hardware input such as a DIP switch.
- Use a software configuration tool to set a configuration property to select an alternate interface.
- Use a concealed hardware input, conditioned and sealed at production time, to select one of these interfaces.
- Automatically sense the configuration based on hardware configuration, for example by detecting which I/O modules have been installed, and automatically present the matching interface.
- Support node-locked license keys. For example, those could consist of a combination of the device's MAC-ID or the Micro Server's Neuron ID and an encoded selection of enabled application features, stored in an encrypted form using a secret algorithm and key such that the license cannot be moved to a different device.

Interface Switchover

You device can select an interface on initial startup, or at any time while running. To select the interface on initial startup, you will select the interface type prior to calling **LonInit()**. To change the interface after the **LonInit()** function has been called, use the **LonReinit()** function.

Any change of the interface requires that the device enters the *unconfigured* state. The device loses all information about network connections it was previously engaged with, and will generally obtain a new network address after being re-commissioned by the network tool when used in a managed network.

Further Steps

Much of each interfaces' functionality will be encoded within your interface source files, but you can share the same code for implementation of your interfaces' base functionality among all your interfaces, and you can support other aspects of your application's behavior as a function of the currently selected interface.

Sharing Code

You cannot share IML definitions between different interfaces, but you can share the runtime code which executes in relation to some of the interfaces' aspects.

To do so, implement a suitable processing function in any of your C source files, import the prototype of your functions into your interface definitions using the standard C **extern** keyword, and call your shared code from each of your interface definitions as required.

For example, each interface can declare an **onWink** event. You do not have to include the corresponding **onWink** event handler with your interface code so long as you present a function with the correct name and prototype to the linker.

The following illustrates a shared **onWink** event handler:

Other interfaces might share portions of the application's algorithm by calling into common functions, as illustrated in the following example.

Dispatcher Extensions

Your callback dispatcher must ensure that all callback functions required by the ShortStack LonTalk/IP Compact API and application framework are implemented. However, the dispatcher is not limited to those callbacks.

Some applications might use common code, for example to sample physical input. In the event of a significant change to such an input, your common input handler can call the current interface through a new dispatch interface defined within your application.

Use the following illustration to see the approach for a hypothetical zero-crossing detector.

Dispatched Callbacks

Here are the callbacks which need to be dispatched.

Framework Callbacks

```
void LonFrameworkInit(void);
const LonByte* LonGetSiData(unsigned* pLength);
const LonByte* LonGetAppInitData(void);
void* LonGetNvTable(void);
unsigned LonGetNvCount(void);
unsigned LonGetMtCount(void);
LonUbits32 LonGetSignature(void);
```

API Callbacks

```
void LonResetOccurred(
  const LonResetNotification* const pResetNotification
);
void LonWink(void);
void LonOffline(void);
void LonOnline(void);
void LonServicePinPressed(void);
void LonServicePinHeld(void);
void LonNvUpdateOccurred(
      const unsigned index,
      const LonReceiveAddress* const pSourceAddress
);
void LonNvUpdateCompleted(const unsigned index, const LonBool
success);
const unsigned LonGetCurrentNvSize(const unsigned nvIndex);
```

Persistent NVs

If your device interface includes any properties or non-volatile network variables, your application must provide functions for reading and writing non-volatile data for properties.

During processing for the **LonInit()** function, the ShortStack LonTalk/IP Compact API calls the **LonNvdDeserializeNvs()** callback function. This function has the following signature:

```
const LonApiError LonNvdDeserializeNvs(void);
```

Whenever the application receives an update to a persistent network variable, the ShortStack LonTalk/IP Compact API automatically calls the **LonNvdSerializeNvs()** callback function to store the new data persistently.

The IzoT ShortStack SDK's **ShortStackHandlers.c** API source file includes an example implementation for the **LonNvdSerializeNvs()** and **LonNvdDeserializeNvs()** callbacks.

When deserializing, 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 must 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 CNV or non-volatile network variable is updated over the network, your implementation of the **LonNvsSerializeNvs()** callback must write the CNV or network variable data to non-volatile memory.

Application Start-Up and Failure Recovery

Typical applications load all persistent data into RAM during startup. The ShortStack LonTalk/IP 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 other persistent data.

Because your application is responsible for loading and modifying applicable data in non-volatile memory, you can use the application signature generated by the IzoT Interface Interpreter to ensure that the application manages its own data, rather than another application's data. Use the **LonGetSignature()** function implemented in **ShortStackDev.c** 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 ensure that the write process completes in a timely a fashion.

If the write process takes too long to complete within the API's timing requirements (see *Periodically Calling the Event Handler*), your application must use queues or caches to minimize both latencies and the number of modifications.

The application must also 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 must detect that the data is not valid. If invalid data is found, the application can recover, or can 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.

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 when using ISI.

Overview of ISI

A control network may be a small, simple network in a small retail store or in a machine consisting of a few devices, or it may 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 IzoT Commissioning Tool (CT) is a network management tool that uses the IzoT Net 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 LonTalk/IP and LON networks is called the LONWORKS *Interoperable Self-Installation (ISI) Protocol*. The ISI protocol can be used for networks of up to 300 devices.

Larger or more complex networks need to either be installed as managed networks, or should be partitioned into multiple smaller subnetworks, where each subnetwork has no more than 300 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* and *Managing Network Variable Connections* for more information about these services.

The IzoT ShortStack SDK 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 that does not require any licensing fees. In addition to the IzoT ShortStack SDK, the IzoT SDK, CPM 4200 Wi-Fi SDK, and IzoT NodeBuilder Software each include a license for development use of the ISI protocol.

Using ISI in an IzoT ShortStack SDK 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 IzoT NodeBuilder Software). 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 ShortStack ISI and Neuron C ISI Implementations 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* 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/IP 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/IP

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/IP 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.
- The post-reset pause is fixed at 50 ms and cannot be configured.
- 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 Series 6000 or 5000 Device

A standard ShortStack Micro Server on an Series 6000 or Series 5000 Smart Transceiver or Neuron Chip, such as the FT 6050 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/IP 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 determine when 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 **onReset** event handler 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 LonTalk/IP 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 must 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 have to track the engine's state and can issue the same set of ISI API calls in any state.

Implementing a SCPTnwrkCnfg 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.

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

```
SFPTnodeObject(node) nodeObject;  //@IzoT block \
//@izot implement(nciNetConfig, flags=Reset, init=CFG_LOCAL) \
//@izot onUpdate(nciNetConfig, onNetConfigChange)

//@IzoT Event onReset(onResetHandler)

void onResetHandler(
    const LonResetNotification* const pResetNotification
) {
   if (*nodeObject.nciNetConfig == CFG_LOCAL) {
```

```
      /* Start the ISI engine */
      IsiStart(IsiTypeS, IsiFlagExtended);
   }
}

void onNetConfigChange(
   const unsigned index,
   const LonReceiveAddress* const pSourceAddress
) {
   if (*nodeObject.nciNetConfig == CFG_LOCAL) {
      /* The device is returned to self-installation.
       * Clear old configuration data and start again.
       * This task can take a significant amount of time,
       * after which the Micro Server resets. */
      IsiReturnToFactoryDefaults();
   }
}
```

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 with a unique domain. Each logical network is also referred to 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 typically supports 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 0x49.53.00 (ASCII codes for "IS\0") 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 DAS

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 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 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 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 can support domain acquisition to provide more robust device installation with 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 can 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 DAS

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) must 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 59.

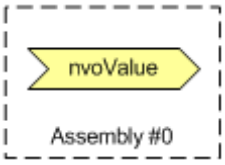


Figure 59. A Simple Assembly

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 60.

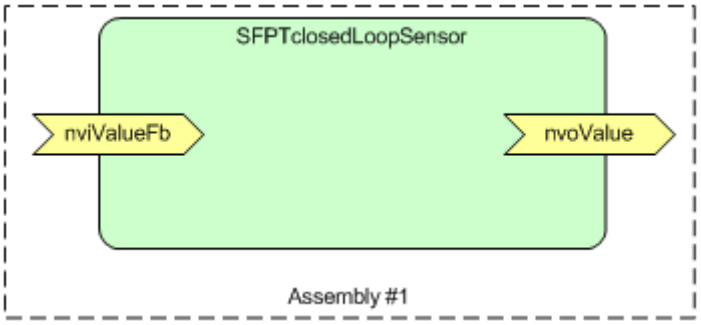


Figure 60. A Compound Assembly

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 61.

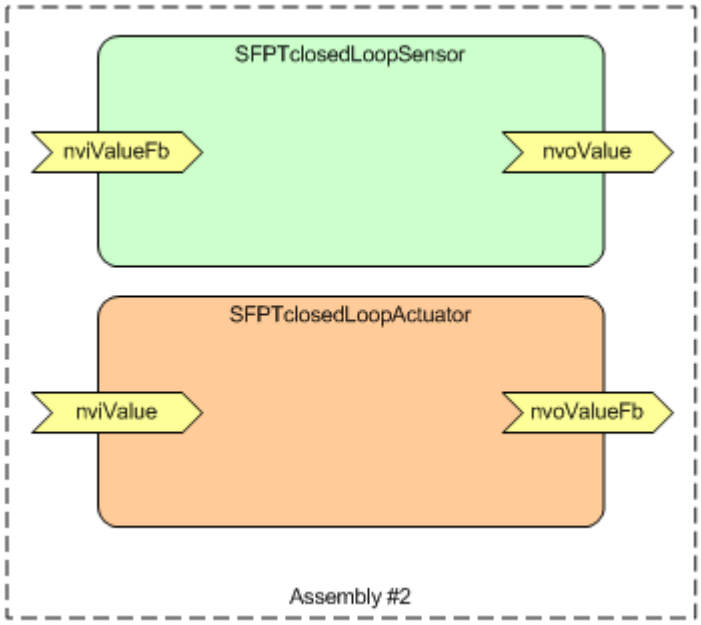


Figure 61. Multiple Functional Blocks as a Single Compound Assembly

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 typically 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 need to have a width of at least 1. Simple assemblies have a width of 1; compound assemblies typically have a width greater than 1.

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 59, 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 60 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 61, above, 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 needs to 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 need also to support connection recovery as described in *Recovering Connections*. 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 responds to the CTRQ message with a control response (*CTRP*) message indicating that it implements the requested operation.

If your ShortStack device supports 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

This 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 periodically calls 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

This 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(nvoValue.global_index);
}
```

Example 3

This 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.

```
LonTag isiTag; //@IzoT Tag bindable(No)

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(isiTag, 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 can generally assume that the target device does not implement controlled enrollment. As the example

shows, you can 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*.

```
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

This 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

This example 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 periodically calls the **IsiQueryIsConnected()** function. Then, within the **IsiIsConnectedReceived()** callback handler function, you can update the connection status for each assembly.

Example 6

This 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 periodically calls 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 included with the ShortStack FX SDK ARM7 Example Port, which is available for free download from echelon.com/downloads.

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

This 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. You can use different ISI libraries with custom Micro Server implementations; see *Creating a Custom Micro Server with ISI Support*.

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

This 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* 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* 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

This 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* 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 the following 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* 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 should 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 may be as simple as a single Connect light and button, possibly shared with the Service light and button. A more complex gateway or controller may 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 23** when the associated event occurs. Some of these events carry a meaningful value in the numeric parameter, as shown in the table.

Table 8. 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.

IsiEvent	Value	Description
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.
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 $\mathbf{T_{auto}}$ 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**.

IsiEvent	Value	Description
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 62 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.

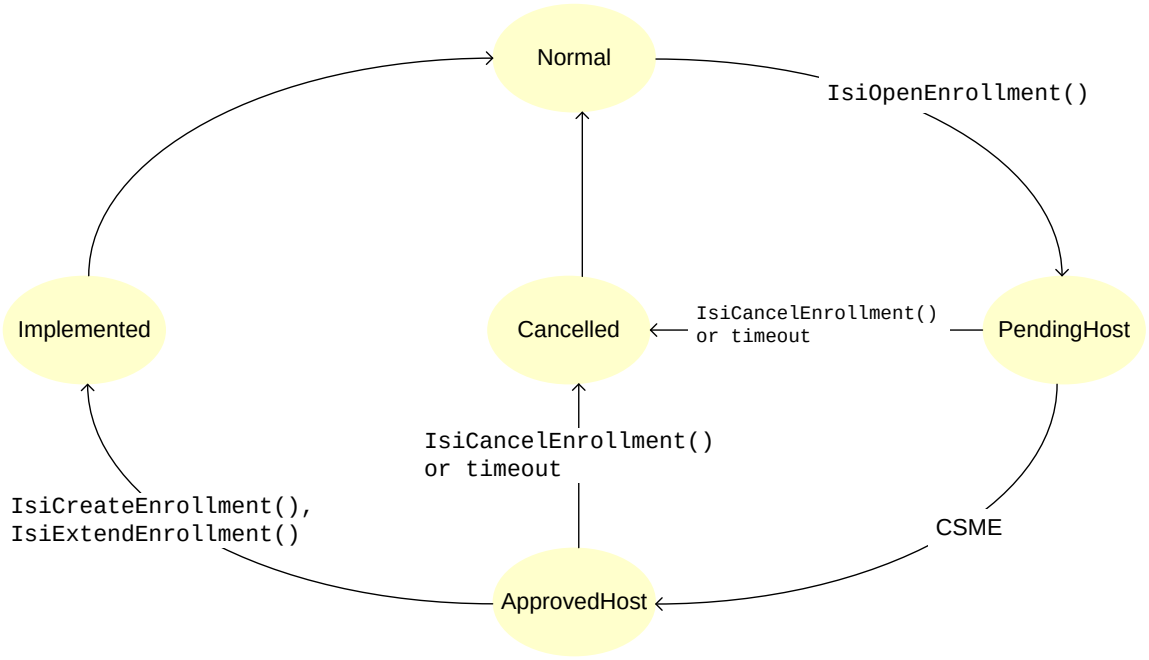


Figure 62. Sequence of Events for a Connection Host

Figure 63 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.

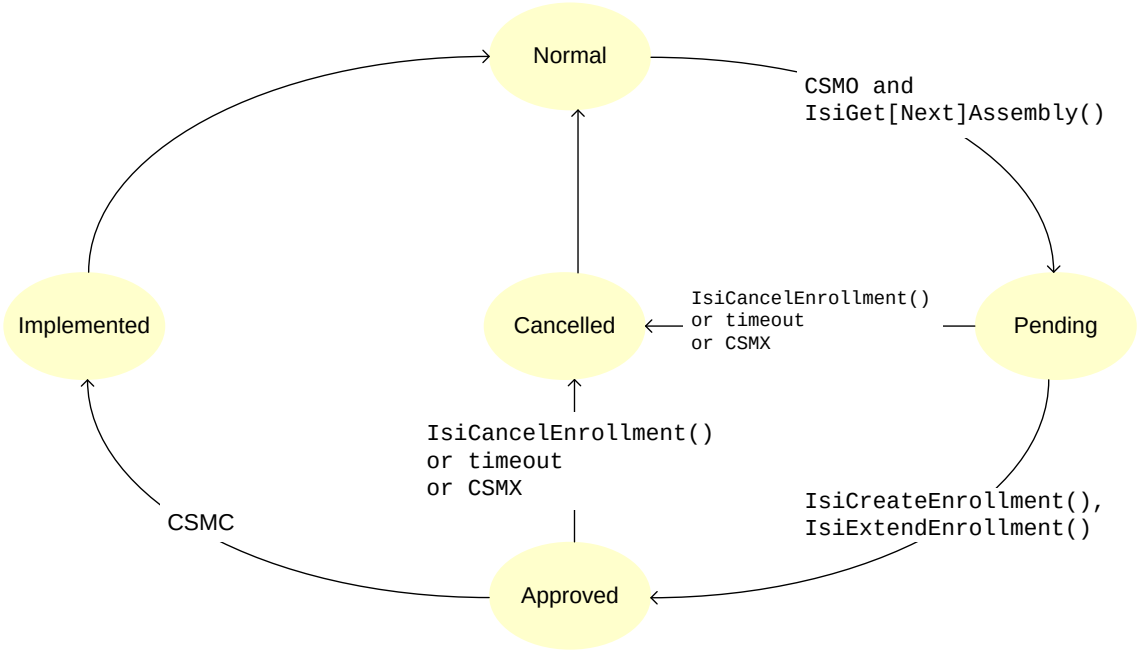


Figure 63. Sequence of Events for a Connection Member

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 or 6050 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, Series 6000 or 5000 Smart Transceiver or Neuron Chip, 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 need to 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*.

```
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*.

```
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 should 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 function 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 needs to 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 first discovers all the devices in the network, as described in *Discovering Devices*. 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 needs to 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/IP 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, 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.

```
LonTag isiTag; //@IzoT Tag bindable(No)

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(isiTag, 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

This following example deinstalls a device after the service pin is held for a long period.

```
//@IzoT Option servicebutton_held(12)
//@IzoT Event onService(onServiceHandler)

void onServiceHandler(LonBool held) {
   if (held) {
      nodeObject->nciNetConfig = CFG_LOCAL;
      IsiReturnToFactoryDefaults();
   }
}
```

Comparing ShortStack ISI and Neuron C ISI Implementations

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 LonTalk/IP 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 LonTalk/IP 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 LonTalk/IP 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 LonTalk/IP 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 LonTalk/IP 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 should provide a complete implementation, and cannot fall back to the forwardee.
- Do not call the ISI API from within an ISI callback override. With the LonTalk/IP 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 LonTalk/IP 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 failures. Thus, the remote device connected to these output network variables cannot rely on updates to network variables to occur within a specific time or order. However, a robust ShortStack ISI application monitors the completion of the operation, and avoids 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. The IzoT NodeBuilder Software is required to create a custom Micro Server.

Overview

The IzoT ShortStack SDK 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, FT 6050 Smart Transceivers and FT 5000 Smart Transceivers, in some common hardware configurations (see **Table 5** in *Standard ShortStack Micro Server Firmware Images* for a list of the standard Micro Server images).

If your ShortStack device requires support for 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 6050 and Neuron 6010 Processors, 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.

The IzoT NodeBuilder Software is required to create a custom Micro Server. The IzoT NodeBuilder Software is included with the FT 6000 EVK, and is available as a free download for developers with the NodeBuilder Development Tool.

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 6050 or 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.
- Maximum number of address table entries. Most Micro Servers support no more than 15 address table entries, but Micro Servers for Series 6000 chips support an extended address table with up to 254 address table entries.
- 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 behaviour.
- 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 6050 or 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 a Series 6000 chip, or the Echelon Neuron 5000 Processor.
- Capacity for more network variables, aliases, domains, or address tables than are supported by the Micro Server hardware and firmware. That is, a custom Micro Server cannot support more than 254 network variables, 127 network variable aliases, 2 domains. Most Micro Servers are limited to 15 address table entries, but those using a Series 6000 chip can support up to 254.

Configuring and Building a Custom Micro Server

To configure and build a custom Micro Server, create a project for the IzoT NodeBuilder Software. This project will 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 24 lists the files that are included with the IzoT ShortStack SDK for custom Micro Server development. These files are located in the **microserver/custom** folder within your IzoT ShortStack SDK repository.

Table 9. Files for Custom Micro Server Development

File Name	Description
ShortStack430.lib	This C library contains the majority of the Micro Server implementation.
ShortStack430Isi.lib	This C library provides the same basic functionality as the **ShortStack430.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 **ShortStack430.lib** library instead.

File Name	Description
ShortStack430CptIsi.lib	This C library provides the same basic functionality as the **ShortStack430Isi.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 uses the SCI protocol at a 38400 bit rate. Therefore, you must use either a Series 3100 device with a 10 MHz external clock or a Series 5000 or Series 6000 device with a 5 MHz system clock. • The post-reset pause is set to 50 ms and is not configurable. • The local utility functions (and their callback handler functions) are not available. See *Local Utility Functions* 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 typicall will not have 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 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 or 1..254), 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 will only require this file if your custom Micro Server supports ISI.
MicroServerIsiHandlers.c	This file contains the override callback handler function implementations for ISI support. You may have to edit this file for a custom Micro Server to match the changes you make to the **ShortStackIsiHandlers.h** file. You wll only require 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 have to use the ISI protocol, but to use the ISI protocol through the LonTalk/IP 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*.

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 can 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. Locate the **microserver\custom** directory within your local IzoT ShortStack SDK repository.
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 do not have 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 **ShortStack430.lib**, **ShortStack430Isi.lib**, or **ShortStack430CptIsi.lib** library. For a Micro Server that supports ISI, you also link the project with the appropriate ISI library, such as the **Isi6000.lib**, **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*.

The following sections describe the process for creating a custom Micro Server in more detail.

Creating a Custom Micro Server without ISI Support

Figure 64 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 **ShortStack430.lib** library to create your custom Micro Server.

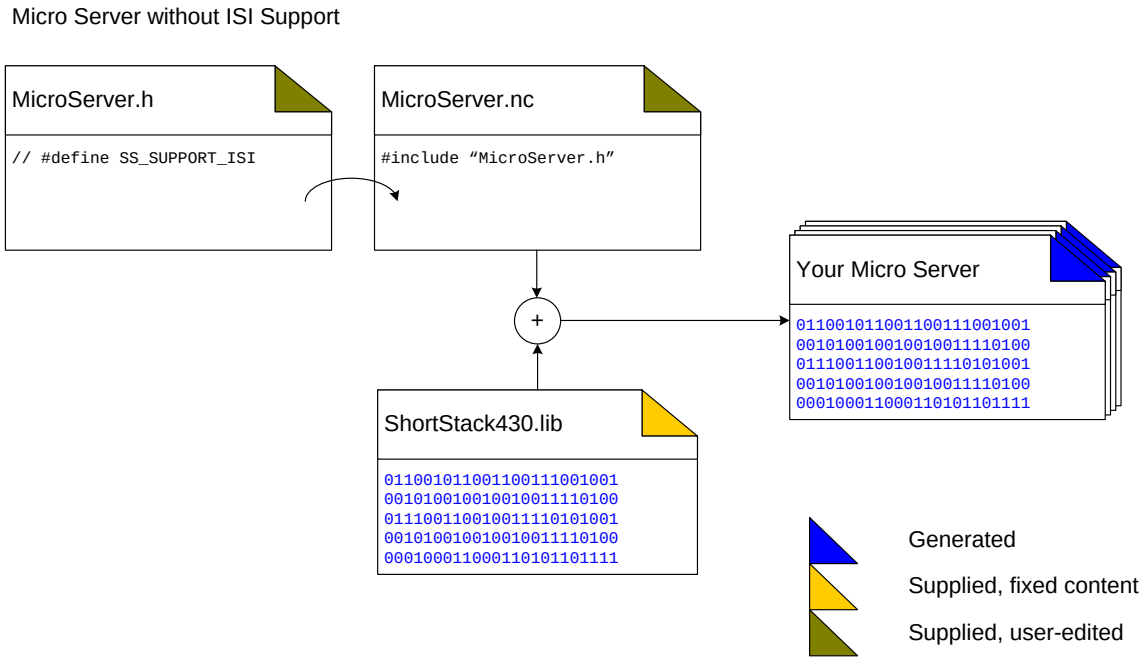


Figure 64. Files for Creating a Custom Micro Server without ISI Support

To configure and build a custom Micro Server without ISI support, perform the following tasks:

- Create a NodeBuilder project, using the files described in **Table 24**.

  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 th**e Development ta**rget), and select **Settings** to open the NodeBuilde**r Devic**e Template Target Properties dialog.

  - Select the **Linker** tab. Select **Generate symbol file.**

- **Also fr**om the **Linker** tab, yo**u can op**tionally select **Generate map file** and select **Verbose.** A map file is optional, but useful.
- 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 IzoT Interface Interpreter.

   Within the IzoT NodeBuilder Software, 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.

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* for more information about direct memory files.

4. Review the preferences specified in the **MicroServer.h** file. See *Managing Memory* for information about configuring the Micro Server's resources within the **MicroServer.h** file.
5. Build the Micro Server. Link your project with the **ShortStack430.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)

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 **SS430_** or a 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, or a Series 6000 chip. 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 or Series 6000 chip. For a domain address server, you must create a custom Micro Server. A DAS-enabled Micro Server needs to run on hardware with at least 512 bytes of additional, off-chip RAM (or extended RAM for FT 5000 Smart Transceivers and Series 6000 chips). 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*, but includes additional files and additional considerations. Figure 65 shows the files that are required to create a custom Micro Server that supports the ISI protocol.

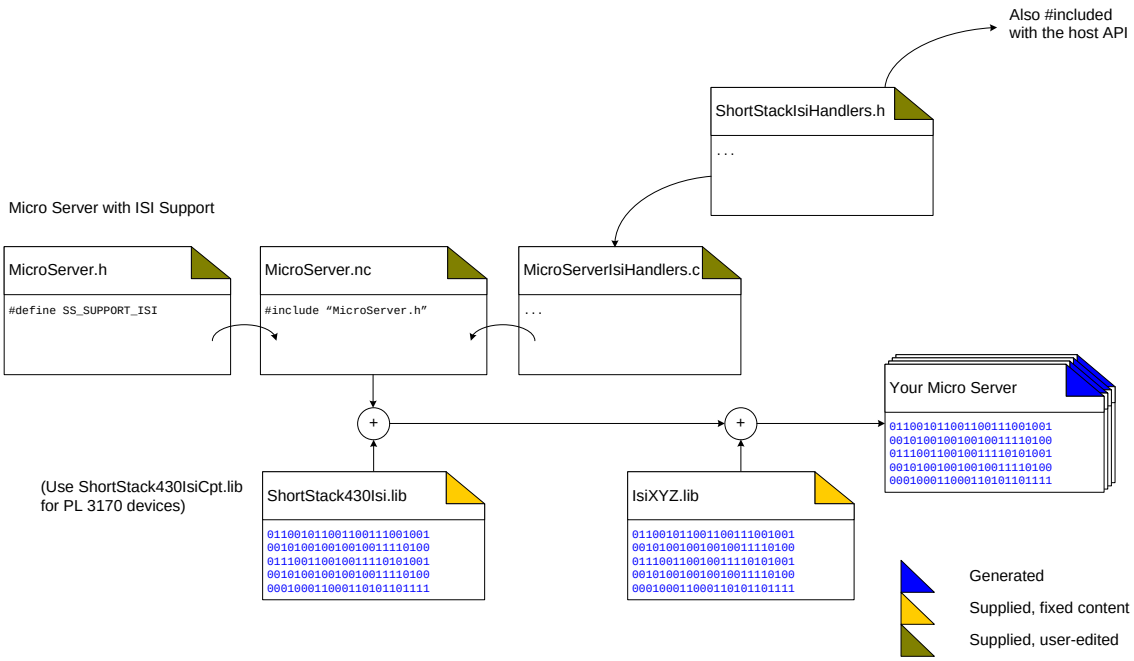


Figure 65. Files for Creating a Custom Micro Server with ISI Support

You edit the **MicroServer.h, MicroServer.nc**, **ShortStackHandlers.h**, and **MicroServerIsiHandlers.c** files, and compile and link the project with the **ShortStack430Isi.lib** (or **ShortStack430IsiCpt.lib**) library and an appropriate ISI library (typically, **Isi6000.lib** or **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 project, using the files described in **Table 24**.
   - 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 *IzoT NodeBuilder 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* 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* 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 **ShortStack430Isi.lib** (or **ShortStack430IsiCpt.lib**) library.
    - Link your project with a suitable standard ISI library, such as **Isi6000.lib, IsiFull.lib** or **IsiCompactDaHb.lib**. If resources permit, use the **Isi6000.lib** for Series 6000 devices or use the **IsiFull.lib** library otherwise.

You can use a custom Micro Server that supports the ISI protocol either with an application that supports ISI or with one that does not. If the application does not support ISI, it 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)

All Micro Server files need to 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 SS430_ or a 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 **ShortStack430IsiCpt.lib** library, and will have the limitations described in **Table 24**.
- 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.
- The **SS_6000** macro indicates that the Micro Server will be used with a Series 6000 Smart Transceiver or Neuron Processor.

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* 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 should always reside in the same location.

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.

The **IsiGetNvValue()** callback handler function needs to 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. 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 need to 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).

See *Option Server* in the *IzoT Markup Language* section *of the IzoT Manual* at echelon.com/docs/izot for more information.

Supporting Direct Memory Files

To allow a custom Micro Server to support the direct memory file (DMF) access method, 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.

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, leave an area within the Smart Transceiver's 64 KB memory space unused. 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.

The DMF access method requires Version 16 Neuron firmware or later, and 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 IzoT Interface Interpreter and 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 for Series 3100 and 5000 chips. Standard Micro Servers for Series 6000 chips, which support an extended address table with up to 254 records, are configured to support 32 address table entries.
- The alias table.
  This table has no default size, and you need to 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.
- 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*, the *Series 5000 Chip Data Book,* or the *Series 6000 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 32 entries for Micro Servers using a Series 6000 chip, and 15 entries for all others. Each address table entry uses five bytes of on-chip EEPROM (extended RAM for a Series 6000 or 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, 0 to 254 for Series 6000 chips.

Whenever possible, specify at least 15 entries for the address table. For Series 6000 chips, large address tables with over 100 entries and up to the 254 entry maximum may impact network performance due to linear address table searches performed by the Neuron firmware when network messages are received.

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.

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 can 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.

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 6000 or 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 in your application.

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

Application Migration from ShortStack FX to IzoT ShortStack

You can upgrade an existing ShortStack FX project to IzoT ShortStack and the IzoT Interface Interpreter.

Who Should Upgrade

You can use the IzoT ShortStack SDK and the IzoT Interface Interpreter for new projects, but developers with existing LID-based projects for the ShortStack FX SDK may not have to upgrade those, or may opt to upgrade those existing projects only in certain aspects.

Using an IzoT ShortStack SDK 4.30 Micro Server with the ShortStack FX SDK

Existing LID-based ShortStack FX SDK projects can take advantage of new IzoT ShortStack SDK 4.30 Micro Servers, and can use the FT 6050 standard Micro Server and take advantage of other general Micro Server enhancements.

These general enhancements include improved resilience for misconfiguration in certain error cases and an enforced delay after transmitting the uplink reset notification.

IzoT ShortStack SDK 4.30 Micro Servers which support the same hardware as supported in the ShortStack FX SDK are backwards compatible. You can, for example, use the new SS430_FT5000ISI_SYS20000kHz standard Micro Server for FT 5000 at 20 MHz system clock in place of its ShortStack FX SDK predecessor.

The standard Micro Servers for FT 5000 now support 7 normal and 3 priority buffers at 146 bytes each, while ShortStack FX SDK Micro Servers for the same Smart Transceiver supported 11 normal and 11 priority buffers at 66 bytes each.

To use a version 4.30 Micro Server in place of the ShortStack FX SDK Micro Server, point the LonTalk Interface Developer to the new IzoT ShortStack SDK 4.30 Micro Server as if it was a custom Micro Server, and re-generate your application framework. Alternatively, you can load the version 4.30 Micro Server image into your Micro Server hardware.

Upgrading a ShortStack FX SDK Project for FT 6050

You can upgrade existing LID-based ShortStack FX SDK projects to use the new Micro Server for the FT 6050 Smart Transceiver. To do so, point the LonTalk Interface Developer to the new IzoT ShortStack SDK 4.30 Micro Server as if it was a custom Micro Server, and re-generate your application framework.

This approach has the following limitations:

- The ShortStack FX LonTalk Compact API is unaware of the extended address table. The extended address table uses address table index values in the 0..254 range, which requires a change in the **nv_config** and **alias_config** data structures. As a result, the ShortStack FX LonTalk Compact API cannot be used to update or examine these data structures on a Series 6000 Smart Transceiver or Neuron Chip. Micro Servers

which support the extended address table indicate this with a new flag 0x40 in the Flags byte of the LonResetNotification structure.

- The ShortStack FX SDK and the LonTalk Interface Developer are unaware of the extended address table. The generated XIF file will not accurately reflect these capabilities. You can use your device without an XIF file, or edit a LID-generated XIF file by hand.

To manually edit the LID-generated XIF file, open the Micro Server's XIF and your application's XIF file (generated by LID) in a text editor.

Copy lines 7, 8, 9 and 10 of the Micro Server XIF file into your application's XIF file.

Copy the XIF version number from line 1 from the Micro Server XIF file into your application's XIF file.

Merge line 6 from the Micro Server XIF file into your application's XIF file:

Use field 4 (network variable record count) from your application's XIF file.

Use field 5 (message tag count) from your application's XIF file.

Set field 16 (application type) to 6.

Use field 17 (netvar count) from your application's XIF file.

Use field 23 (maximum NV count) from your application's XIF file.

For all other fields, use the values from the Micro Server's XIF file.

For all other lines, use the values generated by the LonTalk Interface Developer.

Migration From LonTalk Interface Developer to Izot Interface Interpreter

To migrate a ShortStack FX SDK (or earlier) project based on a framework generated by the LonTalk Interface Developer utility to the IzoT ShortStack SDK and a framework generated by the IzoT Interface Interpreter, follow these steps:

New IzoT ShortStack SDK Project

Create a new IzoT ShortStack SDK project. Port the API and driver, and implement a trivial implementation as a first milestone. Here is an example for such an application (not including the driver and API). Adjust the **server** selection as necessary to match your Micro Server.

```
#include “ShortStackDev.h”
#include “ShortStackApi.h”
#include “ldv.h”
```

```
//@IzoT Option target("shortstack-classic")
//@IzoT Option programId("9F:FF:FF:00:00:00:04:00")
//@IzoT Option server("SS430_FT6050_SYS20000kHz")

SFPTclosedLoopActuator(act, SNVT_volt) act; //@IzoT Block \
//@IzoT onUpdate(nviValue, onActUpdate)

void onActUpdate(
  const unsigned index,
  const LonReceiveAddress* const pSourceAddress
)
{
  LON_SET_UNSIGNED_WORD(
     act.nvoValue.data,
     3 + LON_GET_UNSIGNED_WORD(act.nviValue.data)
  )
  LonPropagateNv(act.nvoValue.global_index);
}

static const LdvCtrl ldvCtrl = {
  /* Initialize as required by your driver */
};

int main(int argc, char* argv[])
{
  LonApiError sts = LonInit(&ldvCtrl);

  while(sts == LonApiNoError) {
     sts = LonEventHandler();
  }

  LonExit();

  return sts != LonApiNoError;
}
```

The trivial test application accepts a simple input value, adds 3 and assigns the result to the output. You can test this application with a simple tool such as NodeUtil, but remember that the device must be in the configured and online state in order to receive updates to input network variables.

You can complete this experiment to be confident that your ShortStack device is working, even if the simple application does not meet your application requirements.

Select Preferences

The ShortStack FX SDK uses the LonTalk Interface Developer utility to gather your preferences. Using the IzoT ShortStack SDK, you can express your preferences directly within your source code. The application example above includes some of those expressions. For example,

```
//@IzoT Option server("SS430_FT6050_SYS20000kHz")
```

selects the SS430_FT6050_SYS20000kHz standard Micro Server for the FT 6050 Smart Transceiver.

A comparison of options and preferences supported by the LonTalk Interface Developer to their IzoT ShortStack SDK equivalent follows.

LonTalk Interface Developer	IzoT Interface Interpreter
Project file selection	Your main C source code is your project. The IzoT Interface Interpreter does not maintain data outside your main C source file.
Verbosity, Verbose Code	You can execute the IzoT Interface Interpreter with the *--verbose* option to obtain slightly more verbose console output, but an option to control verbosity of comments within the generated code is not supported, as you do not have to edit the code generated by the IzoT Interface Interpreter.
Framework Type	//@IzoT Option target("shortstack-classic") or execute the IzoT Interface Interpreter with the **--target shortstack-classic** command line option.
Micro Server selection, transceiver selection, clock selection	//@IzoT Option server(...) The IzoT ShortStack SDK uses the Micro Server's default transceiver type and communication parameter by default. To override those see the **LonCustomCommunicationParameters()** callback function in **ShortStackHandlers.c**.
Program Id	//@IzoT Option programId(...)
Model File, Preprocessor symbols, Include search path	The IzoT Interface Interpreter does not use a Neuron C model file. Your definitions of the interface are an integral part of your application's C or C++ source code.
Enable Application Messages	LonTag myTag; //@IzoT Tag The IzoT Interface Interpreter automatically enables the application messaging API when you declare one or more message tag objects.
Enable Explicit Addressing	//@IzoT Option explicit_addressing(...)
Enable Service Pin Notification, Service Pin Notification Delay	//@IzoT Option servicebutton_held(...)
Include Query	//@IzoT Option api(1)

Functions Include Update Functions	The IzoT Interface Interpreter groups both query and update functions into one API extension, API extension 1. You can select multiple API extensions by adding their numbers. For example, //@IzoT Option api(3) selects both API extensions 1 and 2.
Include Utility Functions	//@IzoT Option api(2) You can select multiple API extensions by adding their numbers. For example, //@IzoT Option api(3) selects both API extensions 1 and 2.
Include ISI	//@IzoT Option isi(...)
Enable Direct-memory Files (DMF), DMF Window Start Address, DMF Window Size	The IzoT Interface Interpreter included with the IzoT ShortStack SDK 4.30 does not support implementations of properties in property files, and therefore does not support DMF.

Migrate the Model File

The LonTalk Interface Developer utility requires that you *model* your application's network interface using the Neuron C language. By contrast, the IzoT Interface Interpreter does not require you to model your application's network interface. Instead, IzoT Interface Interpreter allows you to implement your blocks, properties, message tags and datapoints *as if* your standard C or C++ compiler knew about those items, and IzoT Interface Interpreter *makes it so.*

You will find that translating the model file into a set of IML instructions for IzoT Interface Interpreter is very easy, because IzoT Interface Interpreter includes comprehensive recognition of profiles and blocks.

In the Neuron C model file, implementing a profile as a block requires declaration of every network variable and property required declaration of the block, and declaration of mappings between the network variables and properties declared on one hand and members listed in the block's profile on the other hand.

The IzoT Interface Interpreter implements entire profiles, including all mandatory members, with a single instruction. Optional instructions to add optional members or other refinements are supported.

Interface Item	IzoT Interface Interpreter Instruction
Mandatory datapoint (network variable) members of functional blocks	Implemented automatically with block declaration.
Mandatory property members of functional blocks	Implemented automatically with block declaration

Implementation of a profile as a block	Use the **Block** directive. Example: SPFTco2Sensor(s) co2; //@IzoT Block

Implementing an optional profile member	Use the **implement** instruction with the *Block* directive.
	
	Example 1:
	
	SFPTco2Sensor(s) co2; //@IzoT Block implement(nvofloatCO2)
	
	Example 1 adds the optional **nvofloatCO2** profile member to the block. The IzoT Interface Interpreter recognizes a property's application set. The optional **nciCO2Offset** property of the CO2 sensor profile, for example, applies to the mandatory **nvoCO2ppm** member.
	
	The IzoT Interface Interpreter requires that you specify a property within its application set, and generates the block accordingly.
	
	Example 2:
	
	SFPTco2Sensor(s) co2; //@IzoT Block \
	//@IzoT implement(nvoCO2ppm.nciCO2Offset)

Device datapoint (network variable, not implementing a member of a profile)	Use the **Datapoint** directive. Example: SNVT_temp(t) nvoTemp; //@IzoT Datapoint
Device property (not implementing a member of a profile)	Use the **Property** directive. Example: SCPTlocation here; //@IzoT Property
Message Tag	Use the **Tag** directive and the **LonTag** type. Example: LonTag myTag; //@IzoT Tag

Migrate Event Handlers

The next step is to migrate your ShortStack event handlers. The IzoT ShortStack SDK ShortStackHandler.c source file looks very similar to the version included with the ShortStack FX SDK. In many cases, you can simply merge your callback function bodies into the IzoT ShortStack SDK project.

Frequently used callbacks, however, are automatically implemented by the IzoT Interface Interpreter, and are dispatched into event handlers. With the exception of the service pin-related callbacks, these event handlers have the same

prototype as their ShortStack FX SDK callback function equivalents. However, the events are declared within your main C source file.

The IzoT Interface Interpreter supports multicast events and re-usable event handlers. That is, you can declare and re-use the same **onUpdate** event handler for all your input network variables, or you can declare one unique event handler for each input network variable, or handle update events for certain groups of input network variables in one handler, others in another.

The IzoT Interface Interpreter implements event dispatchers such that events related to network variable updates or completion events only fire when the event applies to the corresponding item. That is, an **onUpdate** event which applies to a single network variable will only execute when this particular network variable received an update, an **onUpdate** event applying to two network variables will only execute when either of the two network variables received an update.

You can also create multicast events, for example by declaring three different **onWink** event handlers. Those are fired in declaration-order.

Event types supported in this fashion are **onUpdate**, **onComplete**, **onReset**, **onOnline**, **onOffline**, **onService**, and **onWink**. All other events are handled within **ShortStackHandlers.c** in the same way as with the ShortStack FX SDK.

LdvCtrl

Declare the new **LdvCtrl** structure, and pass a pointer to this structure to the **LonInit()** function. The data type of **LdvCtrl** is determined by you and your driver implementation. **LdvCtrl** provides a way for your application to pass parameters into your driver through the standard IzoT ShortStack LonTalk/IP Compact API.

Not all drivers require such parameters. Those which don't will define **LdvCtrl** as a simple dummy type, an **int**, for example. This is your choice.

Other drivers, such as the IzoT ShortStack SDK driver example for Raspberry Pi, support a selection of device names, GPIO pin assignments and other data through the **LdvCtrl** data structure.

LonExit()

The IzoT ShortStack SDK supports a new **LonExit()** API, which supports applications that can be terminated. Many embedded applications never terminate and therefore do not need to call **LonExit()**. Those which do support termination, such as the IzoT ShortStack SDK application examples for Raspberry Pi, can call this new API to support clean shut-down procedures.

LonSuspend(), LonResume()

The ShortStack LonTalk/IP Compact API supports two new *optional* functions, **LonSuspend()** and **LonResume()**, to temporarily suspend the serial driver, and resume normal operation. The underlying functionality is implemented in your driver's **LdvSuspend()** and **LdvResume()** functions. The ShortStack API does not require that you implement this functionality, but it supports it when you do.

Some applications may temporarily suspend the serial driver to allow for other critical operations. The IzoT Shortstack SDK driver example for Raspberry Pi includes an example implementation of synchronized suspend and resume operations.

LonGetCurrentNvSize()

This callback function is no longer necessary in the IzoT ShortStack SDK, because the IzoT Interface Interpreter includes knowledge of the **SCPTnvType** property implementation that applies to a changeable-type network variable. The tool automatically generates code which implements this callback.

LonNvdDeserializeNvs(), LonNvdSerializeNvs()

While the **LonNvdDeserialize()** API and callback function remains unchanged, the IzoT ShortStack SDK adds the **LonNvdSerializeNvs()** companion callback, and provides an example implementation for both within **ShortStackHandlers.c**.

In the ShortStack FX SDK, you had to implement custom code to store non-volatile network variable and property data when it was received.

In the IzoT ShortStack SDK, you must implement the **LonNvdSerialize()** and **LonNvdDeserialize()** callbacks. The ShortStack LonTalk/IP Compact API calls these functions when necessary.

14

Authentication

This chapter provides details of using authentication with the IzoT ShortStack SDK.

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 52**. 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, follow these 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 IzoT Commissioning Tool to install a key during network integration, or your application can use the **LonQueryDomainConfig()** and **LonUpdateDomainConfig()** API functions to install a key locally.

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*. If a device belongs to more than one domain, 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, perform the following tasks to ensure that the key is not exposed to the network during device installation:

1. Specify that the device uses 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. 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 a network management tool, but must specify an authentication key because the device has only a default key.

To produce highly secured ShortStack devices, create a custom Micro Server and, export the generated image with the authentication keys pre-set. See the *IzoT NodeBuilder User's Guide* for more information.

How Authentication Works

Figure 66 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 Neuron 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 sends the same message again, Device B remembers that the authentication was successfully completed and acknowledges it again.

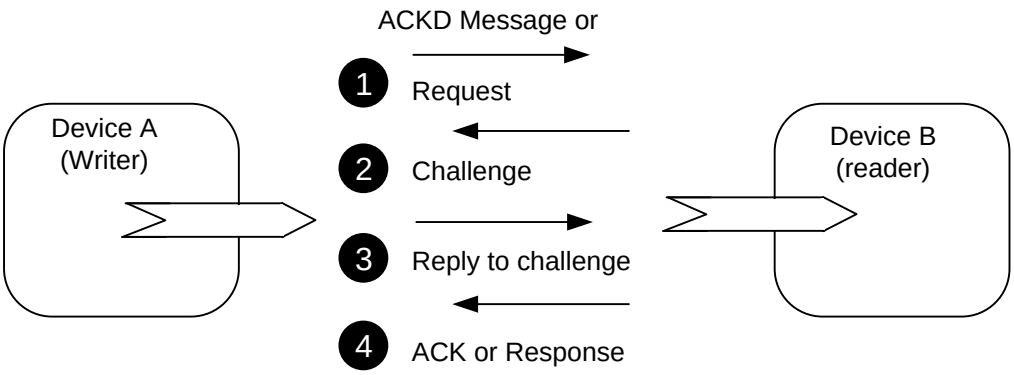


Figure 66. Authentication Process

If Device A updates 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.

A

ShortStack LonTalk/IP Compact API

This appendix describes the functions and callback handler functions included with the ShortStack LonTalk/IP Compact API. It also describes modifying the API callback handlers for use with your ShortStack application.

Introduction

The ShortStack LonTalk/IP Compact API provides the functions that you call from your ShortStack application to send and receive information to and from a LonTalk/IP or LON network. The API also defines the callback functions that your ShortStack application should provide to handle LONWORKS events from the network and Micro Server. Because each ShortStack application handles these callbacks in its own specific way, you must modify the callback functions.

Typically, you use the API functions for ShortStack initialization and sending and receiving network variable updates. See *Developing a ShortStack Application*, for more information about using these functions.

The ShortStack LonTalk/IP Compact API functions are implemented in the **ShortStackApi.c** file; the ShortStack callback functions are defined in the **ShortStackHandlers.c** file. See *ShortStack LonTalk/IP Compact API Files* for a list of the files included with the IzoT ShortStack SDK.

This appendix provides an overview of the functions and callbacks. For detailed information about the ShortStack LonTalk/IP Compact API, see the HTML documentation that is available from the **doc/api** directory within your local IzoT ShortStack SDK repository.

Changes to the API

The ShortStack LonTalk/IP Compact API is the same as the ShortStack FX LonTalk Compact API in spirit, but details have changed. The host API supports application-specific configuration data for the driver, for example to assign a serial device or specific GPIO pins for the link layer signals.

The host API has been enhanced to automatically re-initialize the Micro Server when required. This simplifies updating the Micro Server over the network, because the new API automatically detects the situation and re-initializes the Micro Server with the current application's configuration.

Several API functions have a slightly different prototype compared to earlier releases of ShortStack. All LDV functions, which implement the driver API, now return standard error codes, and use a driver-specific handle parameter. Some functions of the LDV API have been removed, some new ones added, to better support targeting modern hosts such as those using an embedded Linux operating system.

A new **LonSetPostResetPause()** API has been added. This feature is discussed under Micro Server, next.

The host API automatically detects if the Micro Server supports an extended address table (EAT), and automatically translates all affected data types and message formats, transparent to the application.

The following API types are affected: **LonNvConfig**, **LonAliasConfig**.

The following API functions are affected: **LonUpdateNvConfig()**, **LonUpdateAliasConfig()**, **LonQueryNvConfig()**, and **LonQueryAliasConfig()**.

The following callbacks are affected: **LonNvConfigReceived()** and **LonAliasConfigReceived()**.

Support for EAT-enabled Micro Servers is implemented by exposing only EAT-compatible APIs to the application. The **LonNvConfig** and **LonAliasConfig** data types have been updated to match the format required by Micro Servers which support an extended address table.

Applications reading the network variable configuration and alias configuration data will continue to function as before. When re-compiled and used with an EAT-enabled Micro Server, these applications have immediate access to the extended data, which includes the high nibble of the address table index.

Applications that write network variable configuration and alias configuration data must supply well-formed data, including the extended data which includes the address table index high nibble. To write the data, obtain the current record using **LonQueryNvConfig()** or **LonQueryAliasConfig()**, and then modify and assign using the record with **LonUpdateNvConfig()** or **LonUpdateAliasConfig()**as necessary.

The **LonCustomCommunicationParameters()** callback has been added to support applications with runtime selection of communication parameters. This supports, for example, applications for power line communication to use the same application binary to support deployment in CENELEC member countries and affiliates using the PL-20C channel type, and using the PL-20N channel type elsewhere.

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. Use the **api** option to control inclusion of optional portions of the API.

Example

```
//@IzoT Option api(3) // include all optional parts
```

API Memory Requirements

The memory requirements for the ShortStack LonTalk/IP 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 using the IML **api** option.

Table 25 lists the approximate API memory requirements based on a reference implementation using a bare-metal ARM7 target. 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 10. ShortStack LonTalk/IP Compact API Memory Requirements

Included API			
Standard API	**Optional API**	**ISI API**	**Memory Requirement**
☑			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 included in the ShortStack FX release, the memory requirement is approximately 3 KB.

The ShortStack LonTalk/IP 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/IP Compact API and the callback handler functions, see the HTML API documentation and the API source code:

- **doc/api** within the IzoT ShortStack SDK for the HTML API documentation,
- **api** within the IzoT ShortStack SDK for the API source code.

ShortStack LonTalk/IP Compact API Functions

The ShortStack LonTalk/IP Compact API includes functions for managing network data and the ShortStack Micro Server.

Commonly Used Functions

Table 26 lists API functions that you will typically use in your ShortStack application to send and receive data over a LonTalk/IP or LON network.

Table 11. Commonly Used ShortStack LonTalk/IP 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* for more information about how to use this function.
LonInit()	Initializes the ShortStack LonTalk/IP 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 calls **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 is bound

Other Functions

Table 27 lists other ShortStack LonTalk/IP Compact API functions that you can use in your ShortStack application. These functions are not typically used by most ShortStack applications.

Table 12. Other ShortStack LonTalk/IP 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.
LonPollNv()	Requests a network variable value from another device or devices. A ShortStack application can call **LonPollNv()** to request that another 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 on the ShortStack device, it must be declared as an input network variable and include the **polling** modifier. You cannot poll an output network variable on the ShortStack device with the **LonPollNv()** function. Do not use polling with ISI connections.

Function	Description
LonSendServicePin()	Broadcasts a Service- message to the network. The Service-message is used during configuration, installation, and maintenance of a LonTalk/IP or LON device.

Application Messaging Functions

Table 28 lists the ShortStack LonTalk/IP Compact API functions that are used for implementing application messaging and for responding to an application message. Application messages are used by applications requiring a different data interpretation model that the one used for network variables. The same functions can be used for foreign frame and explicit network variable update messages. Support for application messaging is automatically included when your application declares at least one message tag.

Table 13. Application Messaging ShortStack LonTalk/IP Compact API Functions

Function	Description
LonSendMsg()	Sends an application, foreign frame, or explicit network variable update message.
LonSendResponse()	Sends an application, foreign frame, or explicit network variable update 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/IP Compact API includes the optional network management query API functions that provide additional network management commands listed in **Table 29**. 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 have to include these functions in your ShortStack application. You can include these functions with api extension 1.

Example

```
//@IzoT Option api(1)
```

Table 14. 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/IP Compact API includes the optional network management update API functions that provide additional network management commands listed in **Table 30**. 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 include these functions in your source code with IML **api** extension 1.

Example

```
//@IzoT Option api(1)
```

Table 15. Network Management Update API Functions

Function	Description
LonClearStatus()	Clears a subset of status information on the ShortStack Micro Server.
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/IP 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 31 lists the ShortStack LonTalk/IP 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 include these functions in your source code by enabling IML **api** extension 2.

Example

```
//@IzoT Option api(2)
```

Table 16. Local Utility ShortStack LonTalk/IP 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.
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.

Function	Description
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. Define a ping timer of at least 60 seconds. The application typically resets 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.
LonSetPostResetPause	Disables or configures the Micro Server's post reset delay. Assign zero to disable or a value in the 1..255 ms range to enable. The Micro Server uses 50 ms by default. Assignments to this value are stored in persistent memory on the Micro Server.

ShortStack Callback Handler Functions

The ShortStack LonTalk/IP Compact API provides event handler functions for managing network and device events.

Commonly Used Callback Handler Functions

Table 32 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 have to modify these callback functions if you have no application-specific processing requirements.

Table 17. Commonly Used ShortStack Callback Handler Functions

Function	Description
LonGetCurrentNvSize()	Indicates a request for the network variable size. The ShortStack LonTalk/IP Compact API calls this callback handler function to determine the current size of a changeable-type network variable. For applications using the IzoT Interface Interpreter, this callback is automatically implemented.
LonNvUpdateCompleted()	This callback is no longer used. It indicates that either an update network variable call or a poll network variable call is completed. Applications using the IzoT Interface Interpreter declare **onComplete** events instead.
LonNvUpdateOccurred()	This callback is no longer used. It indicates that a network variable update request from the network has been processed by the ShortStack LonTalk/IP 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. Applications using the IzoT Interface Interpreter declare **onUpdate** events instead.
LonOffline()	This callback is no longer used. It represenets 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. Applications using the IzoT Interface Interpreter declare **onOffline** events instead.

Function	Description
LonOnline()	This callback is no longer used. It represenets 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. Applications using the IzoT Interface Interpreter declare **onOnline** events instead.
LonReset()	This callback is no longer used. It is a notification that the ShortStack Micro Server has been reset. Applications using the IzoT Interface Interpreter declare **onReset** events instead.
LonServicePinHeld()	This callback is no longer used. It is an indication that the Service input on the device has been activated for some number of seconds (default is 10 seconds). Use it if your application needs notification of the Service input being active. Applications using the IzoT Interface Interpreter declare **onService** events instead.
LonServicePinPressed()	This callback is no longer used. It is an indication that the Service input on the device has been activated. Use it if your application needs notification of the Service input being activated. Applications using the IzoT Interface Interpreter declare **onService** events instead.
LonWink()	This callback is no longer used. It indicates 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 can provide some visual, audio, or other indication for an installer to be able to physically identify this device. Applications using the IzoT Interface Interpreter declare **onWink** events instead.

Application Messaging Callback Handler Functions

Table 33 lists the callback handler functions that are called by the ShortStack LonTalk/IP Compact API for application messaging transactions. Customize

these functions if you use application messaging in your ShortStack device. Application messaging is optional.

If you choose not to support application messaging, you do not have to customize these functions. These functions are automatically included when your application declares at least one message tag.

Table 18. 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 a 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.
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/IP Compact API includes the optional network management query API callback handler functions listed in **Table 34**. These callbacks allow you to customize the application processing for responses to local network management commands (see **Table 29**). 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. Use IML **api** extension 1 to include these functions.

Table 19. 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.
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 35 lists the callback handler functions for the local utility functions described in *Local Utility Functions*.

You can include the local API functions and their callback handler functions with IML **api** extension 2.

Table 20. 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.
LonNvIsBoundReceived()	Indicates whether the specified network variable is bound.
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.

B

LonTalk/IP ISI API

This appendix describes the functions and callbacks included with the LonTalk/IP 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 LonTalk/IP ISI API, see the HTML documentation that is available in the **doc/api** directory within your local IzoT ShortStack SDK repository.

The LonTalk/IP ISI API

Table 36 lists the LonTalk/IP ISI API functions. When the host application calls one of the functions listed in **Table 36**, 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 21. LonTalk/IP 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. You can call this function 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. You cannot call this function 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. Use the **IsiExtendEnrollment()** function instead where supported, unless application requirements dictate otherwise. 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. You can call this function 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. You cannot call this function 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, the ISI engine creates a new connection. 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 can only be called from a domain address server.
IsiFetchDomain()	Starts or restarts the fetch domain process in a domain address server (DAS). This function can only be called 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 cannot 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.

Function	Description
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.
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 LonTalk/IP ISI Callback Handler Functions

Table 37 lists the ShortStack ISI callback handler functions.

In any ISI application, callback handlers provide application-specific details to the ISI engine. You can implement these callback handlers on your host processor or in a custom Micro Server for ShortStack ISI applications. 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 22. ShortStack ISI Callback Handler Functions

Function	Description
IsiApiComplete()	Indicates that the API function is complete and that the result has been received. The ISI engine calls this function when an API function completes. Do 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. The ISI engine calls this function prior to sending a CSMO message. You can implement this callback 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, you can use this function to send a message during one of the periodic message slots. If the application has no message to send, then this function must return **FALSE**. If it does have a message to send, then this function must return **TRUE**. To use this function, 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. You can implement this callback handler 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 a connection. The function returns **ISI_NO_ASSEMBLY** (0xFF) if no such assembly exists, or an application-defined assembly number (0 to 254). You can implement this callback 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 must return as soon as possible. If you override this function, you will typically 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. You can implement all three of these functions on the Micro Server for the best performance.
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. You can implement all three of these functions on the Micro Server for the best performance. 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). You can call this function after calling the **IsiGetAssembly()** function, unless **IsiGetAssembly()** returned **ISI_NO_ASSEMBLY**. You can implement this callback 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. You can implement this callback 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()**. You can implement this callback 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. You can implement this callback on the host, but it 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. You can only use this function 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. You can implement this callback on the Micro Server. By default, the default implementation is used. 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). You can implement this callback 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. You can implement this callback 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. You can implement this callback 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.

Function	Description
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.
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*. You can implement this callback handler 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 needs to 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 non-volatile 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 must 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. You can implement all three of these functions on the Micro Server for the best performance.
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 can 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* 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*]**\microserver\custom \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.

A callback handler function cannot 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*]**\microserver\custom\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 attempts 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 22**, *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.

C

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 a network management tool, such as the IzoT Commissioning Tool. However, you typically 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/IP Compact API nor the ShortStack Micro Server directly supports updating the host application over the network, you must do the following:

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 must support the following steps:

1. Prepare for application download.

   When the application download utility informs the current ShortStack host application to start an application download, the application must 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 can also inform the user of its state.

   During this stage, the ShortStack host application must 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 must 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 must 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 verifies the integrity of the image, and either:

   a. Accepts the image, and proceeds to the final steps below.

   b. Requests retransmission of some sections of the image.

   c. Rejects 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.

For the duration of the first three steps, the application must be running, the link-layer driver needs to 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/IP Compact API, and your application program.

Because both processor chips must 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/IP 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 may 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. Your host application download protocol can include an appropriate mechanism to determine and verify these requirements before passing control to the new application.

In some cases, your host application download may 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 cannot 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 be postponed. For example, the new host application may 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 reads the application image to be loaded, and runs the application download protocol described in *Custom Host Application Download Protocol*. You can write the utility as an IzoT Net plug-in or as any type of network-aware application.

Download Capability within the Application

Your application implements the custom application download protocol, and provides non-volatile storage for the new application image. The application must also tolerate time consuming writes to flash during the transfer. At a minimum, the ShortStack host application must reserve enough RAM to buffer two flash sectors. When one sector has been completely received, the application writes it to flash in a background process. If the write is not complete when the second buffer is filled, the ShortStack host application tells 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 performs all necessary verification tasks, and prepares 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 may 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.

D

Glossary

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

B

block

A block, also known as a functional block, is a network visible component of the software application on a LonTalk/IP or LON device. A block encapsulates the datapoints and properties required for a task performed by the device application.

For example, an LED controller device may provide functionality to independently control the color of multiple LED lamps, and also to monitor the power consumption and energy usage of the lamps. The LED controller application may expose this functionality as an independent load control block for controlling each lamp, as well as independent analog sensor blocks for monitoring instantaneous power and energy consumption for each lamp.

Each block is defined by a *profile* that defines the datapoint and property members that can be implemented by the block. A profile defines mandatory and optional members. A block always implements the mandatory members, and may also implement any of the optional members from the profile.

D

datapoint

A datapoint is a data value or structured set of values where each value has specified encoding, units, range, and scaling. A datapoint may be published or subscribed to by a LonTalk/IP or LON device, or it may be published by an IzoT Server via the IzoT REST API. A datapoint published or subscribed to by a device is called a *device datapoint*, and is also called a *network variable*. A datapoint published by an IzoT Server is called a *server datapoint*. Both

types of datapoints are just called "datapoint" when used in the context of a device or an IzoT Server.

A device datapoint is a generalization of a network variable. Most device datapoints implement a network variable (and are often the same as a network variable), but a device datapoint can also implement other forms of network data objects, or can hold additional data or meta-data.

Device datapoints may be shared among multiple LonTalk/IP and LON devices. Each device datapoint represents a single scalar value or a structure or union of multiple values containing 1 to 225 bytes. A device may have multiple datapoints, and each datapoint may be shared with one or more datapoints on any device or group of devices within a network.

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 or microprocessor that is attached to a ShortStack Micro Server and runs a LonTalk/IP or LON application.

I

IzoT Interface Interpreter

A utility that generates the framework for your application and produces device interface files.

The IzoT Interface Interpreter supersedes the LonTalk Interface Developer utility, which was included with the ShortStack FX SDK.

IzoT ShortStack SDK

Software required to develop LonTalk/IP or LON applications for any microcontroller or microprocessor. The kit includes software tools, examples, documentation, plus the ShortStack LonTalk/IP Compact API and ShortStack firmware.

L

link layer

A protocol and interface definition for communication between a host processor and a 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/IP API

A C language interface that can be used by a LonTalk/IP or LON application to send and receive network variable updates and application messages. There is a full featured version shipped with the IzoT SDK and a smaller version called the ShortStack LonTalk/IP Compact API shipped with the IzoT ShortStack SDK.

LonTalk application framework

Application code and device interface data structures created by the IzoT Interface Interpreter supporting an as-if method of programming using expressions in annotated standard C source code or by the LonTalk Interface Developer based on a model file.

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 the ShortStack FX SDK.
The IzoT ShortStack SDK replaces the LonTalk Interface Developer with the IzoT Interface Interpreter.

M

model file

A Neuron C application that is used to define the network interface for a ShortStack FX SDK application.

The IzoT ShortStack SDK uses the IzoT Interface Interpreter, which does not require or support model files.

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 device

A LONWORKS device based on the ShortStack LonTalk/IP 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 or Neuron Processor that enables the Smart Transceiver to be used as a network interface by a ShortStack host processor.

ShortStack host processor

A microprocessor or microcontroller that is integrated with the ShortStack LonTalk/IP Compact API, ShortStack Driver API, and a ShortStack Micro Server to create an IzoT 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 LonTalk/IP Compact API

A compact version of the LonTalk/IP API for ShortStack devices with support for up to 254 network variables.

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

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X