# ECHELON®

# 2.0

# ShortStack FX ARM7
# Example Port User's Guide

# Welcome

Echelon's ShortStack® Micro Server enables any product that contains a microcontroller or microprocessor to quickly and inexpensively become a networked, Internet-accessible device. The ShortStack Micro Server provides a simple way to add LONWORKS® networking to new or existing smart devices.

This document describes the ShortStack FX ARM7 Example Port for an ARM7-family microprocessor, the Atmel® ARM® AT91SAM7S64. This example port includes ported example ShortStack applications, the host API, and a serial driver for the ARM7 processor. This example is available as a free download from the Echelon ShortStack Web site, www.echelon.com/shortstack.

# Audience

This document assumes that you have a good understanding of the LONWORKS platform, the ShortStack Micro Server, ShortStack LonTalk Compact API, general embedded system design methodologies, and the ARM7 family of embedded processors.

If you create a serial driver for communications between the host microprocessor and the ShortStack Micro Server, you need to be familiar with either the Serial Communications Interface (SCI) or Serial Peripheral Interface (SPI) interface standard.

# Related Documentation

The ShortStack FX Developer's Kit includes the *ShortStack FX User's Guide* (078-0365-01B), which describes how to develop applications for LONWORKS devices that use the ShortStack FX Micro Server. It also describes the architecture of a ShortStack device and how to develop one.

The ShortStack FX Developer's Kit also includes the following manuals:

- *Neuron C Programmer's Guide* (078-0002-02H). This manual describes the key concepts of programming using the Neuron C programming language and describes how to develop a LONWORKS application.

- *Neuron C Reference Guide* (078-0140-02F). This manual provides reference information for writing programs that use the Neuron C language.

- *Neuron Tools Errors Guide* (078-0402-01B). This manual describes error codes issued by the Neuron C compiler and related development tools.

After you install the ShortStack FX Developer's Kit, you can view any of these manuals from the Windows **Start** menu: select **Programs** → **Echelon ShortStack FX Developer's Kit** → **Documentation**.

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

- *FT 3120 / FT 3150 Smart Transceiver Data Book* (005-0139-01D)**.**  This manual provides detailed technical specifications on the electrical interfaces, mechanical interfaces, and operating environment characteristics for the FT 3120® and FT 3150® Smart Transceivers.

- *Introduction to the LONWORKS Platform* (078-0183-01B)*.*  This manual provides an introduction to the ISO/IEC 14908 (ANSI/CEA-709.1 and EN14908) Control Network Protocol, and provides a high-level introduction to LONWORKS networks and the tools and components that are used for developing, installing, operating, and maintaining them.

- *ISI Programmer's Guide* (078-0299-01F).  Describes how you can use the Interoperable Self-Installation (ISI) protocol to create networks of control devices that interoperate, without requiring the use of an installation tool.  Also describes how to use Echelon's ISI Library to develop devices that can be used in both self-installed as well as managed networks.

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

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

- *LonMaker User's Guide* (078-0333-01A).  This manual describes how to use the Turbo edition of the LonMaker® Integration Tool to design, commission, monitor and control, maintain, and manage a network.

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

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

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

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

All of the ShortStack documentation, and related product documentation, is available in Adobe® PDF format.  To view the PDF files, you must have a current version of the Adobe Reader®, which you can download from Adobe at: www.adobe.com/products/acrobat/readstep2.html.

As you create your serial driver for communications between your host processor and the ShortStack Micro Server, you will need to be familiar with either the SCI or SPI interface standard. You will find having an appropriate reference for the interface helpful.

# Table of Contents

# 1

# Introduction

This chapter introduces the ShortStack FX ARM7 Example Port and describes how to install the example software.

# Introduction

The ShortStack FX ARM7 Example Port includes ported example applications, host API, and a serial driver for the ARM7 microprocessor. The example applications include a simple example, a changeable-type example, and a self-installation example.

The simple example application is a very simple simulated analog actuator with a gain of two. This simulated device receives an input voltage value, multiplies the value by 2, and returns the new output value. The changeable-type example application includes the same functionality as the simple example application, but adds the ability to change the SNVT types for two of the network variables. The self-installation example demonstrates the basics of using the Interoperable Self-Installation (ISI) protocol for a ShortStack device.

This manual describes the example applications, including their design, how to build them, how to load them into the ARM7 microprocessor, and how to run them.

The source code for the example applications is installed in the following directories:

- [*ARM7Example*]\**Simple Example**

- [*ARM7Example*]\**Simple Changeable-type Example**

- [*ARM7Example*]\**Self-installation Example**

where [*ARM7Example*] is the directory in which you installed the ShortStack FX ARM7 Example Port, which by default is **C:\Program Files\LonWorks\ShortStack\Examples\ARM7**. See *Installing the ShortStack Example Software* on page 3 for information about creating this directory.

For more information about the example applications, see *Exploring the Example Applications* on page 29.

# Installing the Example Port

The ShortStack FX ARM7 Example Port requires:

- A development board with an Echelon 3120 or 3150 Smart Transceiver plus an ARM7 development board, such as the Echelon Pyxos™ FT EV Pilot Evaluation Board (see *Hardware Development Tools for the ARM7 Microprocessor* on page 6). The Pyxos FT EV Pilot EVB includes both an Echelon FT 3150 Smart Transceiver and an ARM7 host processor.

  Alternatively, you can use an Echelon FT 5000 EVB Evaluation Board with the Pyxos FT EV Pilot EVB or with a separate ARM7 development board, such as the Atmel AT91SAM7S-EK evaluation kit. However, this configuration requires some extra setup; see the *ShortStack FX User's Guide*.

- A software development environment for the ARM7 microprocessor.

- A hardware emulator and debugger that supports the Institute of Electrical and Electronics Engineers (IEEE) Standard Test Access Port

and Boundary-Scan Architecture (IEEE 1149.1-1990) of the Joint Test Action Group (JTAG).

- An In-System Programmer (ISP) to load program images into the ARM7 microprocessor.
- The ShortStack FX Developer's Kit software.
- The ShortStack FX ARM7 Example Port software.

To demonstrate ISI-specific behavior, the self-installation example also requires a separate ISI-enabled device, such as one of the following:

- An Echelon Mini EVK or Mini FX Evaluation Board, with the MiniGizmo board, running the MGDemo (or MGLight or MGSwitch) application.
- An Echelon FT 5000 EVB Evaluation Board, running the NcSimpleIsiExample application.

## Hardware Requirements

The ShortStack FX ARM7 Example Port does not have specific PC hardware requirements, other than those required for the ShortStack FX Developer's Kit and your software development environment.

However, you must have the following hardware for LONWORKS connectivity:

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

Other hardware requirements are described in *Hardware Development Tools for the ARM7 Microprocessor* on page 6 and *Loading Your Application into the ARM7 Microprocessor* on page 12.

## Software Requirements

The ShortStack FX ARM7 Example Port requires the ShortStack FX Developer's Kit to be installed.  See the *ShortStack FX User's Guide* for ShortStack software requirements.

The Example Port does not have additional PC software requirements beyond those of the ShortStack FX Developer's Kit and your ARM7 software development environment.

The following software is optional, depending on your requirements:

- Adobe Reader 9.1 or later
- NodeBuilder Resource Editor 4.00 or later (installed with the ShortStack FX Developer's Kit), if you need to create custom LONMARK® resource files and data type definitions

## Installing the ShortStack Example Software

To install the ShortStack FX ARM7 Example Port:

1. Download the ShortStack FX ARM7 Example Port from the Echelon ShortStack Web site, www.echelon.com/shortstack.

2. From Windows Explorer, double-click
   **ShortStack400ARM7ExamplePort.exe** to start the Echelon ShortStack
   FX ARM7 Example Port installer.

3. Follow the installation dialogs to install the example port onto your
   computer. The installer installs the example port into the **Examples**
   directory within your ShortStack FX directory, by default, **C:\Program
   Files\LonWorks\ShortStack\Examples\ARM7**.

If you plan to make substantial modifications to the example applications or to
the serial driver, you should make a backup copy of the
**\LonWorks\ShortStack\Examples\ARM7** directory.

# 2

# Overview of the Hardware Environment for the ARM7 Microprocessor

This chapter describes hardware development tools for the ARM7 microprocessor and the hardware interface for the ShortStack Micro Server.

# General Description of the Supported Hardware

The ShortStack FX ARM7 Example Port includes ported example applications, host API, and a serial driver for an ARM7-family microprocessor, the Atmel ARM AT91SAM7S64.

The AT91SAM7S64 microprocessor is a general-purpose microcontroller, featuring 64 KB of embedded high-speed flash memory, with sector lock capabilities and a security bit, and 16 KB of static random access memory (SRAM).

The AT91SAM7S64 microprocessor includes the following peripherals:

- A Universal Serial Bus (USB) 2.0 Full Speed Device Port

- Two universal synchronous/asynchronous receiver/transmitters (USARTs)

- Serial peripheral interface (SPI) Bus

- Synchronous Serial Controller (SSC)

- Two-wire interface (TWI)

- Power Management Controller (PMC)

- Advanced Interrupt Controller (AIC)

- An 8-channel, 10-bit analog-to-digital converter (ADC)

In addition, AT91SAM7S64 microprocessor's Peripheral DMA Controller channels eliminate processor bottlenecks during peripheral-to-memory transfers. And its System Controller manages interrupts, clocks, power, time, debug and reset, significantly reducing the external chip count and minimizing power consumption.

For more information about the Atmel ARM AT91SAM7S64 microprocessor, see www.atmel.com/dyn/products/product_card.asp?part_id=3521.

# Hardware Development Tools for the ARM7 Microprocessor

To work with the Atmel ARM AT91SAM7S64 microprocessor, you can use any of the many available tools that support the ARM7 family of microprocessors, such as the Atmel AT91SAM7S-EK evaluation kit.

The example applications that are included with the ShortStack FX ARM7 Example Port are built for the Echelon Pyxos FT EV Pilot Evaluation Board, as shown in **Figure 1** on page 7 and described in **Table 1** on page 7. In addition to providing a development platform for the Echelon Pyxos FT Chip, this evaluation board includes an Atmel ARM AT91SAM7S64 microprocessor and an Echelon FT 3150 Smart Transceiver. You do not need to work with the Pyxos FT Chip to work with the ShortStack FX ARM7 Example Port.
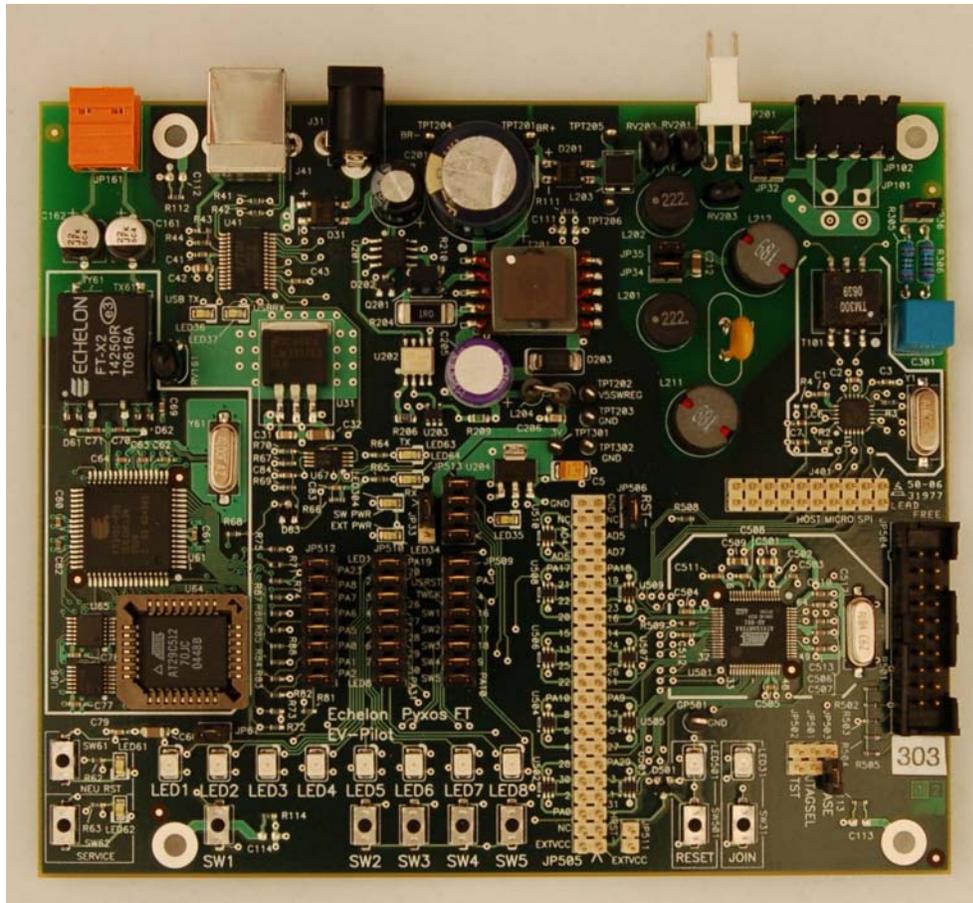
**Figure 1**. Echelon Pyxos FT EV Pilot Evaluation Board

**Table 1**. The Pyxos FT EV Pilot Evaluation Board

| **Pyxos FT EV Pilot Evaluation Board** |
|---|
| The Pyxos FT EV Pilot Evaluation Board is the controller for the Pyxos EVK network.  It contains an Atmel ARM AT91SAM7S64 host microcontroller that is connected to, and communicates with, a Pyxos FT Chip.  A Pyxos Pilot is responsible for configuring, maintaining, and communicating with the Pyxos Points in a Pyxos network.  The EV Pilot can also optionally communicate with LONWORKS devices using the included FT 3150 Smart Transceiver.  For more information about the Pyxos FT EV Pilot Evaluation Board, see www.echelon.com/pyxos. |

If you use the Pyxos FT EV Pilot Evaluation Board for the ShortStack FX ARM7 Example Port, you do not need any other development hardware, except a device programmer (such as the Atmel AT91SAM-ICE JTAG Emulator).  And although the Pyxos FT EV Pilot Evaluation Board includes a number of jumper settings to control the behavior of the board, you do not need to change any of the jumper settings to run the ShortStack example applications on the Pyxos FT EV Pilot Evaluation Board.

**Important**:  Because the Atmel AT29C512 flash memory part on the Pyxos FT EV Pilot Evaluation Board is factory pre-programmed with an older version of the ShortStack Micro Server, you must reprogram the flash part to load it with a

ShortStack FX Micro Server.  See *Preparing the ShortStack Micro Server* on page 13 for more information.

If you work with a separate ARM7 development board (such as the Atmel AT91SAM7S-EK evaluation kit), you must:

- Add a network interface with an Echelon Smart Transceiver, such as an Echelon FT 5000 EVB Evaluation Board, which is available separately and is included with the Echelon Mini FX Evaluation Kit or the NodeBuilder FX Development Tool

- Configure the Smart Transceiver to act as a ShortStack Micro Server (for example, by setting the appropriate jumpers on the FT 5000 EVB Evaluation Board; see the *ShortStack FX User's Guide* for the required jumper settings)

- Create or modify the ShortStack serial driver to work with the I/O that is available on the development board; see *Developing the ShortStack Driver* on page 15

- Run the LonTalk Interface Developer utility to generate application framework files for the Smart Transceiver

- Create your own software projects for the development environment

See the *ShortStack FX User's Guide* for more information about these tasks.

You can obtain the Pyxos FT EV Pilot Evaluation Board from Echelon.  It is available separately and as part of the Pyxos FT EVK Evaluation Kit.  For more information about the Pyxos FT EV Pilot Evaluation Board, see the *Pyxos FT EVK User's Guide*.

# Hardware Interface for the ShortStack Micro Server

The example applications use a ShortStack Micro Server with the characteristics listed in **Table 2**.  You can modify these characteristics using the LonTalk Interface Developer utility, which is a tool that generates the device interface data and the device interface file required to implement the device interface for your ShortStack device.

**Table 2**. ShortStack Micro Server Characteristics

| Device Characteristic | Simple Example | Changeable-Type Example | Self-Installation Example |
|---|---|---|---|
| Device type | FT 3150 Smart Transceiver | FT 3150 Smart Transceiver | FT 3150 Smart Transceiver |
| Channel type | TP/FT-10 | TP/FT-10 | TP/FT-10 |
| Clock speed | 10 MHz | 10 MHz | 10 MHz |
| Changeable interface | Disabled | Enabled | Disabled |

| Device Characteristic | Simple Example | Changeable-Type Example | Self-Installation Example |
|---|---|---|---|
| ISI API included | No | No | Yes |
| Device program ID | 9F:FF:FF:06:00:0A:04:01 | 9F:FF:FF:06:00:8A:04:02 | 9F:FF:FF:05:01:04:04:05 |
| Model File | [*ARM7Example*]\**Simple Example\ShortStack\Simple Example.nc** | [*ARM7Example*]\**Simple Changeable-type Example \ShortStack\Simple Changeable-type Example.nc** | [*ARM7Example*]\**Self-installation Example\ShortStack\ Self-installation Example.nc** |
| **Note**:  The changeable interface setting is specified in the LONMARK Standard Program ID Calculator, which is available from the Program ID Selection page of the LonTalk Interface Developer utility. | | | |

The ShortStack FX ARM7 Example Port implements communications between the ARM7 microprocessor and the ShortStack Micro Server using the SCI interface.  The [*ARM7Example*]\**Common\Driver\LdvSci.c** file contains the serial driver implementation; see *Developing the ShortStack Driver* on page 15 for more information.  To use an SPI interface, you must implement an SPI serial driver because the ShortStack FX ARM7 Example Port does not include one.  However, the Echelon Knowledge Base has an example SPI driver that you can download for an ARM7 microprocessor; go to www.echelon.com/support/kb/ and search for "KB635".  The SPI driver is compatible with either the ShortStack 2.1 ARM7 example or the ShortStack FX ARM7 example.

The pin connections between ARM7 microprocessor and the ShortStack Micro Server are defined in the **LdvSci.h** file.  **Table 3** shows the physical connections for the SCI interface on the Pyxos FT EV Pilot Evaluation Board.

**Table 3**. ARM7 to Micro Server Pin Connections for the SCI Interface

| AT91SAM7S64 Pin | | FT 3150 Smart Transceiver Pin | |
|---|---|---|---|
| **Number** | **Name** | **Number** | **Name** |
| 31 | PA8 | 2 | IO0 (CTS~) |
| 44 | PA2 | 3 | IO1 (HRDY~) |
| N/A | N/A | 5 | IO3 (SPI/SCI~) Tied to GND for SCI |
| 32 | PA7 | 10 | IO4 (RTS~) |
| 48 | PA0 | 11 | IO5 (SBRB0) |
| 47 | PA1 | 12 | IO6 (SBRB1) |

| AT91SAM7S64 Pin | | FT 3150 Smart Transceiver Pin | |
|---|---|---|---|
| Number | Name | Number | Name |
| 35 | PA5 | 14 | IO10 (TXD) |
| 34 | PA6 | 16 | IO8 (RXD) |
| 15 | PA23 | 6 | RESET~ |

If you use a development environment other than the Pyxos FT EV Pilot Evaluation Board, you must add 10 kΩ pull-up resistors to all communication lines between the ARM7 host microprocessor and the Smart Transceiver (the Pyxos FT EV Pilot Evaluation Board already includes them).  If you use a cable to make the pin connections, keep the total cable length to a maximum of 24 inches (0.6 meters).  Note that the Echelon FT 5000 EVB also already includes the appropriate pull-up resistors.  However, the Echelon Mini EVK or Mini FX EVBs do not include pull-up resistors for connecting to an external host processor.

Although the Pyxos FT EV Pilot Evaluation Board defines connections between the Micro Server and the ARM7 host processor for the IO5 and IO6 pins (SBRB0 and SBRB1), the SCI interface does not require that the host processor control these signals; in general, most devices would tie these signals to $V_{DD}$ or GND (as appropriate; see the *ShortStack FX User's Guide*) to specify the link-layer bit rate.  In addition, if you use an EVB other than the Pyxos FT EV Pilot Evaluation Board for the ShortStack Micro Server (such as the FT 5000 EVB), do not connect that EVB's IO5 and IO6 lines to the ARM7 processor's PA0 and PA1 pins.  By default, the ShortStack FX ARM7 example port's serial driver sets the SCI serial bit rate to 76800 bps for a 10 MHz FT 3150 Smart Transceiver (see the **LdvSci.h** file); connecting the EVB's IO5 and IO6 lines to the ARM7 processor's PA0 and PA1 pins can create a mismatch in the expected bit rate for the Micro Server if the Smart Transceiver runs at a different system clock rate or uses a different external crystal frequency.  Thus, you should use the jumpers on the EVB to set the Micro Server bit rate to match the driver's expected bit rate.

The example application code assumes that the AT91SAM7S64 microprocessor runs at 47.9232 MHz and that the ShortStack Micro Server runs at 10 MHz.  By default, the host microprocessor communicates with the ShortStack Micro Server through the SCI interface at the 76 800 bit rate.

You can change the communication interface through compilation-time configuration, as described in *Communications Configuration Options* on page 16.

# 3

# Developing ShortStack Applications for the ARM7 Microprocessor

This chapter describes software development tools for the ARM7 microprocessor, and tools for loading and debugging your ShortStack application.

# Supported Software and Tools Overview

The example applications for the ShortStack FX ARM7 Example Port require a C compiler (with its associated development environment), a hardware emulator and debugger (or other device programmer) to load the compiled program into the ARM7 microprocessor, and optionally an in-circuit debugger to debug and verify the program.

The example also requires that you load the appropriate ShortStack Micro Server firmware image into the FT Smart Transceiver.

The rest of this chapter describes the tools required for development of the example applications.

# Software Development Tools for the ARM7 Microprocessor

To develop your ShortStack application for an ARM7 microprocessor, you can use any tools that support the ARM7 family of microprocessors, such as those listed in **Table 4**.

Table 4. Software Development Tools for an ARM7 Microprocessor

| ARM RealView® Development Suite |
| --- |
| The RealView Development Suite is a set of development tools that support all ARM processors and ARM debug technology.  It includes a C/C++ compiler, assembler, linker, virtual platforms, text editor, and debugger.  It also allows you to choose an IDE for code development.  For more information about ARM RealView, see www.arm.com/products/DevTools/. |
| IAR Embedded Workbench® |
| The IAR Embedded Workbench is a set of development tools for programming embedded applications.  It includes a C/C++ compiler, assembler, linker, librarian, text editor, project manager, and debugger.  For more information about the IAR Embedded Workbench, see www.iar.com/ewarm. |

The example applications were built with the IAR Embedded Workbench for ARM 5.40.  The IAR Embedded Workbench project files for the example applications are included with the source code; see *Exploring the Example Applications* on page 29 for more information about the example applications.

**Important**:  The project files for the example applications require version 5.40 or later of the IAR Embedded Workbench; earlier versions cannot open the project files.

# Loading Your Application into the ARM7 Microprocessor

To load your application, the ShortStack LonTalk Compact API, and the serial driver into the ARM7 microprocessor, you can use a hardware emulator and debugger, such as those listed in **Table 5**.

**Table 5**. Hardware Emulator and Debugger for the ARM7 Microprocessor

| AT91SAM-ICE JTAG Emulator |
| --- |
| The SAM-ICE is a JTAG emulator designed for Atmel ARM processors. You can get software support for the SAM-ICE from www.segger.com (www.segger.com/cms/downloads.html; scroll to J-Link ARM).  For more information about the SAM-ICE, see www.atmel.com/dyn/products/tools_card.asp?tool_id=3892. |
| IAR J-Link |
| IAR J-Link is a small ARM JTAG hardware debug probe that connects to a Windows computer by a USB connection.  You can use IAR J-Link with the IAR Embedded Workbench.  For more information about the J-Link, see www.iar.com/jlinkarm. |

To load program images (such as the ones contained in the [*ARM7Example*]\**Images** directory), you can use an In-System Programmer (ISP) such as the one listed in **Table 6**.

**Table 6**. In-System Programmer (ISP) for the ARM7 Microprocessor

| Atmel SAM-PROG |
| --- |
| The SAM-PROG is included with the AT91 In-System Programmer (ISP), an open set of tools for programming the AT91SAM7 and AT91SAM9 ARM-based microcontrollers.  SAM-PROG allows you to directly program your application through a SAM-ICE or a J-Link JTAG Probe. For more information about the AT91ISP and SAM-PROG, see www.atmel.com/dyn/products/tools_card.asp?tool_id=3883. |

# Preparing the ShortStack Micro Server

The Atmel AT29C512 flash memory on the Pyxos FT EV Pilot Evaluation Board is factory pre-programmed with an earlier version of the ShortStack Micro Server that is not compatible with the ShortStack FX examples.  Therefore, you must load the ShortStack FX Micro Server firmware into the flash memory on the Pyxos FT EV Pilot Evaluation Board.

To load the ShortStack Micro Server using ex-circuit programming, use a PROM programmer to load an NEI file for the ShortStack FX Micro Server into the FT 3150 Smart Transceiver's off-chip memory.  The ShortStack FX ARM7 Example Port includes Micro Server NEI files for each of the three examples.  Using a PROM programmer for the initial load of the ShortStack FX Micro Server is the

recommended method for upgrading the flash memory part on the Pyxos FT EV Pilot Evaluation Board.

If your Pyxos FT EV Pilot Evaluation Board can communicate with a LONWORKS network (that is, you have a working ShortStack Micro Server, link-layer driver, and host application), you can reload the ShortStack image using in-circuit programming using a network management tool such as the NodeLoad utility.

For information about using these tools to load a ShortStack Micro Server, see the *ShortStack FX User's Guide*.

The firmware images (APB, NEI, and NDL files) for the example applications are located in the following directories:

- [*ARM7Example*]\**Simple Example\ShortStack**
- [*ARM7Example*]\**Simple Changeable-type Example\ShortStack**
- [*ARM7Example*]\**Self-installation Example\ShortStack**

If you use a different EVB for the ShortStack Micro Server (such as the FT 5000 EVB), see the *ShortStack FX User's Guide* for information about how to prepare the Micro Server.

# Debugging Your Application

To debug your application, you can use the debug tools supplied with the IAR Embedded Workbench (or other software development platform), along with the hardware emulator and debugger described in *Loading Your Application into the ARM7 Microprocessor* on page 12.

Debugging your application typically requires halting the application at breakpoints, single-stepping through code, inspecting and editing of variables, and similar tasks. Such debugging tools significantly affect the application's performance. You can debug your application using these tools, but depending on the code that you debug, the overall system might not work as designed while you are debugging your application.

For example, halting the application while the application gathers data for the response to a network variable poll request can cause the poll request to time out and fail. Likewise, debugging the serial driver code, or the code that drives the initialization and application registration phase, can disrupt the link-layer timing requirements, and thus lead to an unsuccessful completion.

# 4

# Developing the ShortStack Driver

This chapter describes the communications configuration options for the ShortStack application and the ShortStack serial driver implementation.

# Communications Configuration Options

You can set the serial interface type and the interface bit rate for the example applications.

## *Setting the Serial Interface Type*

The example applications for the ShortStack FX ARM7 Example Port use the SCI asynchronous interface.  The ShortStack FX ARM7 Example Port does not include an SPI driver.  If you want to use the SPI synchronous interface for communications between the ARM7 microprocessor and the ShortStack Micro Server, you must implement a lower-layer SPI serial driver for the ShortStack Micro Server.  However, the Echelon Knowledge Base has an example SPI driver that you can download for an ARM7 microprocessor; go to www.echelon.com/support/kb/ and search for "KB635".  The SPI driver is compatible with either the ShortStack 2.1 ARM7 example or the ShortStack FX ARM7 example.

## *Setting the Interface Bit Rate*

The interface bit rate for the example application is defined in the ShortStack serial driver.  If you want to change the application bit rate, add or modify the following statement to the [*ARM7Example*]\**Common**\**Driver**\**LdvSci.h** file:

```
#define SS_BAUD_RATE    0
```

The valid values for the **SS_BAUD_RATE** literal are defined in a set of comments in the **LdvSci.h** file.  For a Micro Server running on a 10 MHz FT 3150 Smart Transceiver, a value of 0 represents a bit rate of 76 800 bps for the SCI interface and a value of 1 represents a bit rate of 9600 bps.  To define other bit rates, modify the definition of the **USART_BAUD_RATE** symbol in the **LdvSci.c** file.

The selected serial interface type and interface bit rate must match the configuration of the ShortStack Micro Server to avoid communications problems.  Because the example serial driver sets the interface bit rate (in the **LdvInit()** function), you do not need to modify the jumper settings on the Pyxos FT EV Pilot Evaluation Board.

In addition, if you use an EVB other than the Pyxos FT EV Pilot Evaluation Board for the ShortStack Micro Server (such as the FT 5000 EVB), do not connect that EVB's IO5 and IO6 lines to the ARM7 processor's PA0 and PA1 pins.  Connecting the EVB's IO5 and IO6 lines to the ARM7 processor's PA0 and PA1 pins can create a mismatch in the expected bit rate for the Micro Server if the Smart Transceiver runs at a different system clock rate or uses a different external crystal frequency.  Thus, you should use the jumpers on the EVB to set the Micro Server bit rate to match the driver's expected bit rate.

# Serial Driver Implementation

The example serial driver for the ShortStack FX ARM7 Example Port example applications implements two interface layers:

- An upper-layer interface to the ShortStack LonTalk Compact API

- A lower-layer interface to the ARM7 and Micro Server hardware

The source files for the serial-driver implementation are in the
[*ARM7Example*]\**Common\Driver** directory.

You can use these files as templates for developing ShortStack serial drivers for
any microcontroller or microprocessor.  The example SCI driver supports the
interface functions that the ShortStack LonTalk Compact API requires.

The upper-layer driver interface includes the functions that the ShortStack
LonTalk Compact API calls on behalf of the application to initialize the interface
(**LdvInit()**), send messages (**LdvPutMsg()**), receive messages (**LdvGetMsg()**), and
so on.  These functions are listed in *Upper-Layer Serial Driver Implementation*
on page 27.

The lower-layer driver interface includes the functions that manage the interface
with the ARM7 microprocessor and handle the handshake protocol for
exchanging messages between the Micro Server and the ARM7 microprocessor.
These functions are listed in *Lower-Layer SCI Serial Driver Implementation* on
page 27.

The main functionality of the lower-layer serial driver is interrupt driven.  The
ARM7 microprocessor provides hardware interrupts for its programmed I/O lines,
and the lower-level serial driver uses these interrupts for the CTS~, RXD, and
TXD signals.

## *Transmitting Data to the Micro Server*

The lower-layer driver controls the handshaking protocol with the ShortStack
Micro Server to transmit and receive data.

Receiving data from the Micro Server is relatively straightforward because as
long as the HRDY~ line is asserted, the ARM7 microprocessor is ready to receive
data.  When it receives data, if the driver has sufficient buffer space available, it
processes the data, otherwise it ignores the data.

Transmitting data to the Micro Server is more complex because the protocol
expects a handshake for the header data, another handshake for the extended
header (if the command refers to a network variable with index greater than 62),
and another handshake for the payload data (if any).  The driver relies on two
interrupt service routines (ISRs) to manage the handshake protocol and manage
the transmission of the header and payload:

- The CTS ISR is triggered whenever the state of the CTS~ line changes.
  The Micro Server asserts and deasserts the CTS~ line as part of the
  handshake protocol.

- The Transmit ISR is triggered continuously while the Transmit Interrupt
  is enabled.  The driver enables this interrupt only when there is data to
  send to the Micro Server.  Because the ARM7 microprocessor has a one-
  byte buffer for sending data, this ISR is triggered for every byte of a data
  transmission (header and payload) to the Micro Server.

**Figure 2** on page 19 shows the basic control flow for transmitting data from the
serial driver to the Micro Server.  The application signals that there is data to
send to the Micro Server by calling the **LdvPutMsg()** function, followed by the
**LdvFlushMsgs()** function, which begins the flow shown in the figure.

For network variable updates or poll requests where the network variable index is greater than 62, the command header includes a special network variable index value (0x3F, decimal 63) to indicate that the Micro Server should expect an extended two-byte header to be sent before the payload (if any). The Micro Server also expects another handshake for receiving these extended header bytes. **Figure 3** on page 21 shows the control flow for transmitting data from the serial driver to the Micro Server for network variable updates or poll requests where the network variable index is greater than 62.

See the *ShortStack FX User's Guide* for information about how the Micro Server handles the SCI downlink transfer.

**Micro Server**          **Serial Driver**

**LdvFlushMsgs() Function**

Check CTS~
If CTS~ asserted
        Return
Else
        Check transmit state
        If state == IDLE
                Assert RTS~
                Enable interrupts

← Assert RTS~

Wait

Assert CTS~   → Assert CTS~

**CTS ISR**

Enable Transmit Interrupt
Deassert RTS~

← Deassert RTS~

**Transmit ISR**

Check transmit state
If state == IDLE
        Send Length byte
        Move to COMMAND state

← Send Length Byte

**Transmit ISR**

Check transmit state
If state == COMMAND
        Send Command byte
        If Command == (NV update or NV poll) and NV index == 63
                Disable Transmit Interrupt
                Move to HANDSHAKE state
                Set NextState to INFO
        Else
                Check payload length
                        Disable Transmit Interrupt
                        Move to HANDSHAKE state
                        Set NextState to PAYLOAD
                Else
                        Move to DONE state

← Send Command Byte

Deassert CTS~   → Deassert CTS~

**CTS ISR**

Check CTS~
If CTS~ asserted
        Enable Transmit Interrupt
        Deassert RTS~
Else
        Check transmit state
        If state == HANDSHAKE
                Assert RTS~
                If NextState == INFO
                        Move to INFO state
                Else
                        Move to PAYLOAD state

← Assert RTS~

Wait

Assert CTS~   → Assert CTS~

**CTS ISR**

Enable Transmit Interrupt
Deassert RTS~

← Deassert RTS~

**Transmit ISR**

Check transmit state
If state == PAYLOAD
        Send Payload byte
        If payload length = 0
                Move to DONE state

← Send Payload Byte

Transmit ISR is
called repeatedly
until Transmit
Interrupt is
disabled

**Transmit ISR**

Check transmit state
If state == DONE
        Disable Transmit Interrupt
        Move to IDLE state

**Figure 2**. Control Flow for Transmitting Data to the Micro Server

The figure shows the interaction of the two ISRs. It also shows the use of a state machine within the driver to control the transmission of data. These states are described in *Transmit States* on page 23.

**Figure 3** on page 21 shows the same control flow as **Figure 2**, but includes the extra handshake and header transmission for network variable updates or poll requests where the network variable index is greater than 62. For a network variable update or poll request, the message includes:

- The length byte

- The command byte

- The two extended header bytes

- The payload

The extended header contains the actual network variable index for the command.

Micro Server          Serial Driver

LdvFlushMsgs() Function

Check CTS~
If CTS~ asserted
    Return
Else
    Check transmit state
    If state == IDLE
        Assert RTS~
        Enable interrupts

← Assert RTS~

Wait

Assert CTS~    Assert CTS~ →

CTS ISR

Enable Transmit Interrupt
Deassert RTS~

← Deassert RTS~

Transmit ISR

Check transmit state
If state == IDLE
    Send Length byte
    Move to COMMAND state

← Send Length Byte

Transmit ISR

Check transmit state
If state == COMMAND
    Send Command byte
    If Command == (NV update or NV poll) and NV index == 63
        Disable Transmit Interrupt
        Move to HANDSHAKE state
        Set NextState to INFO
    Else
        Check payload length
        If length > 0
            Disable Transmit Interrupt
            Move to HANDSHAKE state
            Set NextState to PAYLOAD
        Else
            Move to DONE state

← Send Command Byte

Deassert CTS~    Deassert CTS~ →

CTS ISR

Check CTS~
If CTS~ asserted
    Enable Transmit Interrupt
    Deassert RTS~
Else
    Check transmit state
    If state == HANDSHAKE
        Assert RTS~
        If NextState == INFO
            Move to INFO state
        Else
            Move to PAYLOAD state

← Assert RTS~

These actions occur only
for NV updates or polls
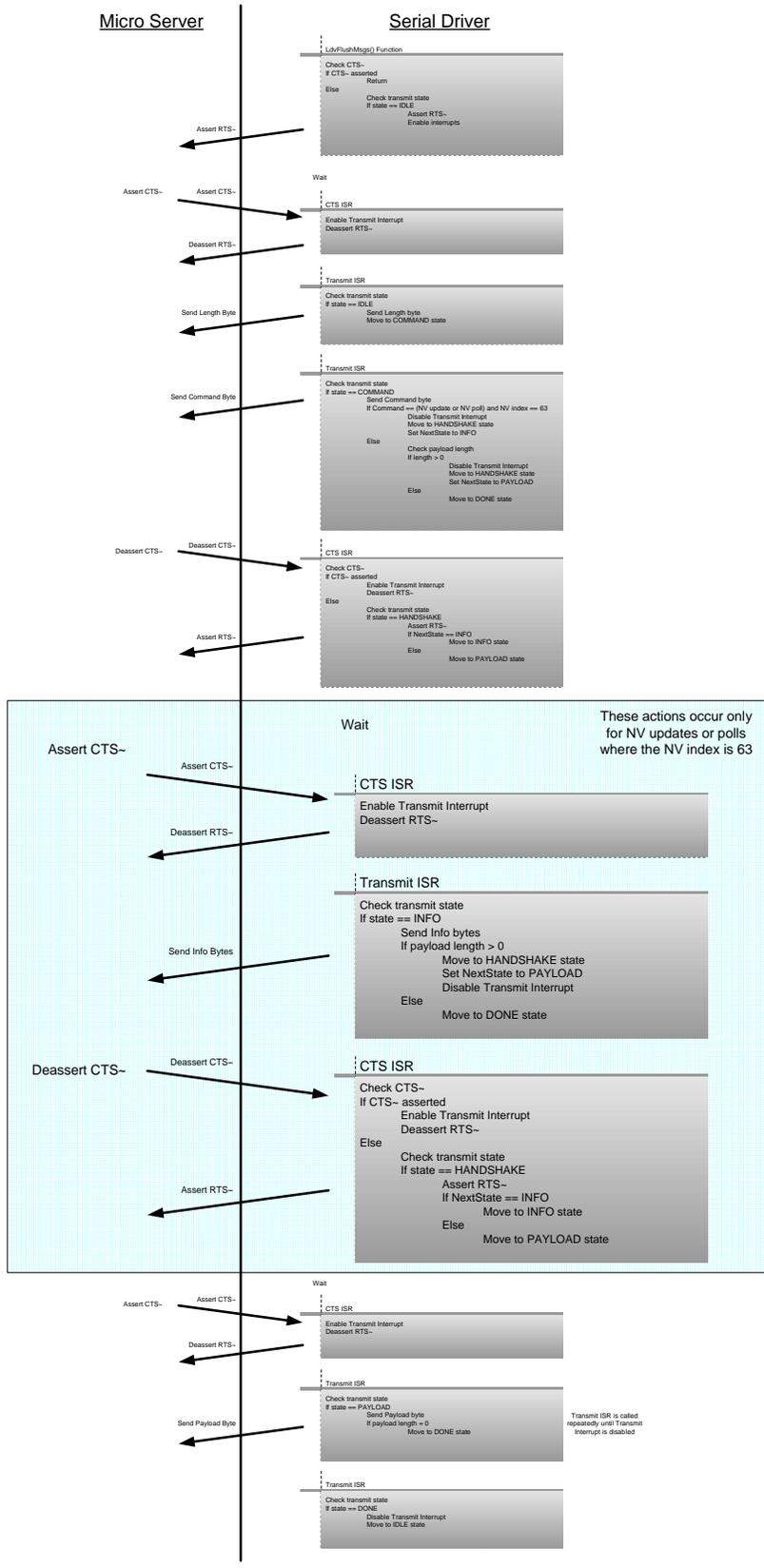where the NV index is 63

Wait

Assert CTS~    Assert CTS~ →

CTS ISR

Enable Transmit Interrupt
Deassert RTS~

← Deassert RTS~

Transmit ISR

Check transmit state
If state == INFO
    Send Info bytes
    If payload length > 0
        Move to HANDSHAKE state
        Set NextState to PAYLOAD
        Disable Transmit Interrupt
    Else
        Move to DONE state

← Send Info Bytes

Deassert CTS~    Deassert CTS~ →

CTS ISR

Check CTS~
If CTS~ asserted
    Enable Transmit Interrupt
    Deassert RTS~
Else
    Check transmit state
    If state == HANDSHAKE
        Assert RTS~
        If NextState == INFO
            Move to INFO state
        Else
            Move to PAYLOAD state

← Assert RTS~

Wait

Assert CTS~    Assert CTS~ →

CTS ISR

Enable Transmit Interrupt
Deassert RTS~

← Deassert RTS~

Transmit ISR

Check transmit state
If state == PAYLOAD
    Send Payload byte
    If payload length = 0
        Move to DONE state

Transmit ISR is called
repeatedly until Transmit
Interrupt is disabled

← Send Payload Byte

Transmit ISR

Check transmit state
If state == DONE
    Disable Transmit Interrupt
    Move to IDLE state

**Figure 3**. Control Flow for Transmitting the Extended Header to the Micro Server

ShortStack FX ARM7 Example Port User's Guide          21

## *Serial Driver State Machines*

The lower-layer driver implements a set of state machines for managing the driver and the SCI receive and transmit operations. The driver also implements state machines for the receive and transmit buffers. The **LdvSci.c** file implements these state machines. You can use this driver implementation as the basis for your own SCI serial driver.

The states described in this section represent a minimal and complete set of states required to implement an SCI serial driver. Although it is not required that your serial driver implementation be based on state machines, this approach simplifies the implementation of the required protocol.

# Driver States

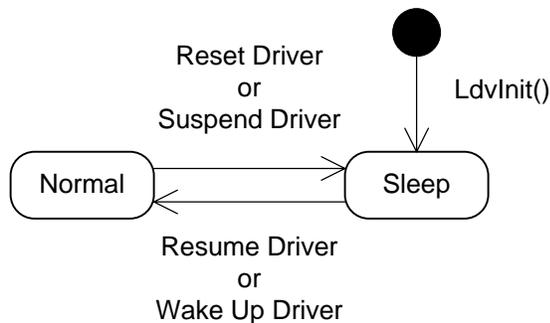**Figure 4** shows the states and state transitions for the driver.



**Figure 4**. State Machine for the Driver

The driver implements a state machine with two states to manage its overall operations:

- *Sleep state*: The initial state set by the **LdvInit()** function. The driver also enters this state whenever the driver is reset or suspended. In the Sleep state, the driver is non-operational.

- *Normal state*: The driver enters this state whenever the driver resumes operations (after having been suspended) or is woken up (after entering the Sleep state). All important driver operations occur while the driver is in this state.

# Receive States

**Figure 5** on page 23 shows the states and state transitions for receiving data.
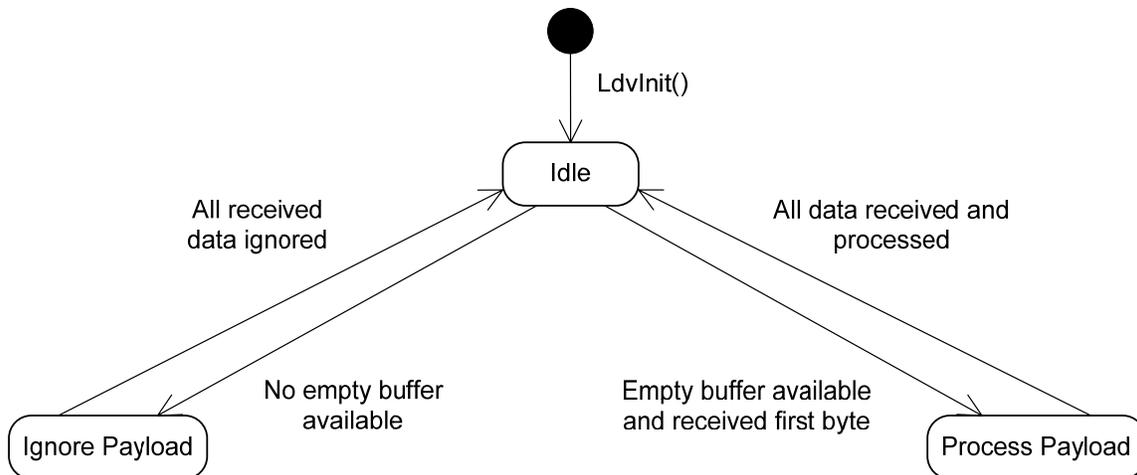
Developing the ShortStack Driver

**Figure 5**. State Machine for Receiving Data

The driver implements three states to manage receiving data from the Micro Server:

- *Idle state*:  The initial state set by the **LdvInit()** function.  The driver also enters this state after receiving and processing (or ignoring) all data.

- *Ignore state*:  The driver enters this state when it receives the first byte of data but has no available buffers to store the data.  The driver remains in this state until all bytes for the message are received.

- *Payload state*:  The driver enters this state when it receives the first byte of data and has at least one available buffer to store the data.  The driver remains in this state until all bytes for the message are received.

# Transmit States

**Figure 6** on page 24 shows the states and state transitions for transmitting data.
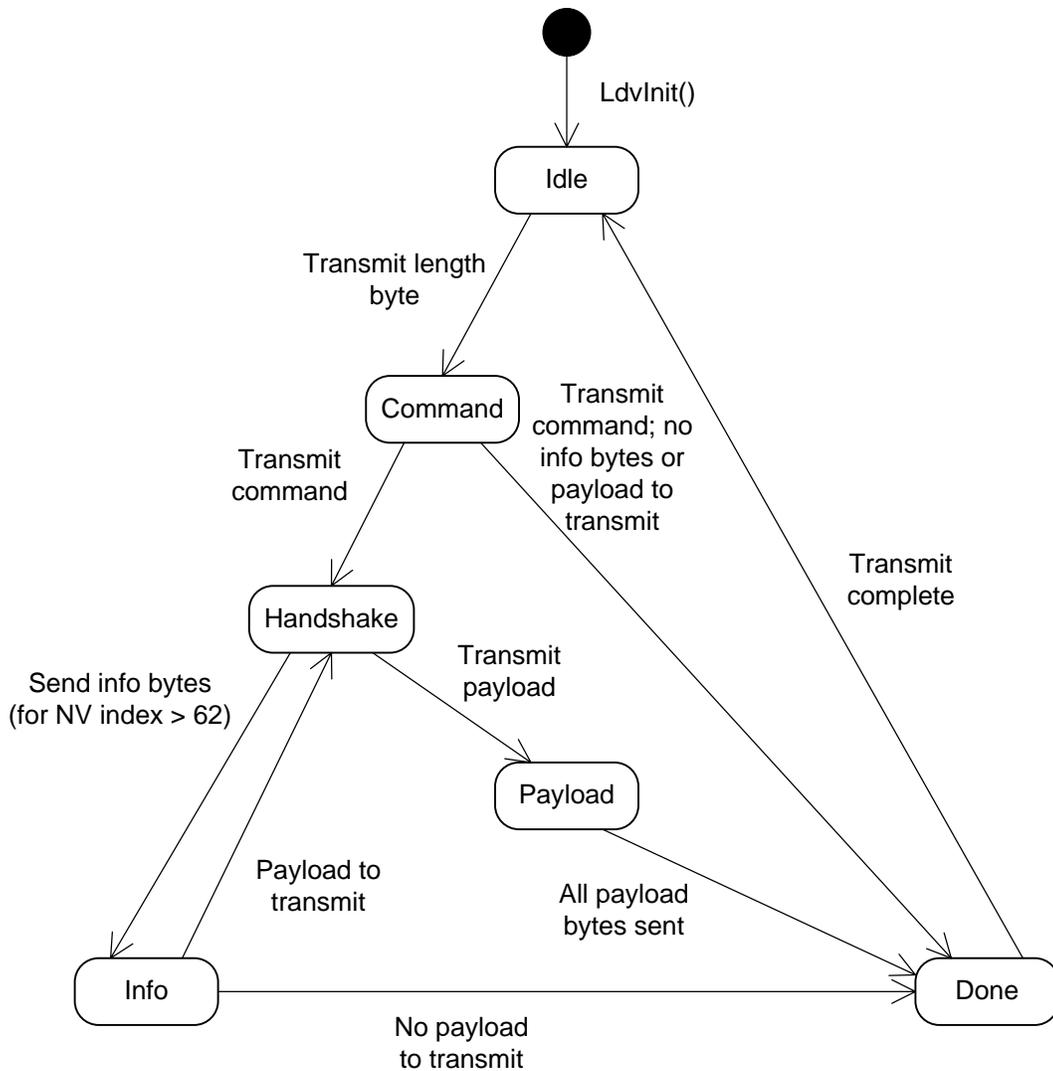
**Figure 6**. State Machine for Transmitting Data

The driver implements six states to manage transmitting data to the network:

- *Idle state*: The initial state set by the **LdvInit()** function. The driver also enters this state after transmitting all data.

- *Command state*: The driver enters this state after it completes the downlink handshake and sends the length byte of a message header to the Micro Server. The driver then sends the command byte of the message to the Micro Server. If there are no info bytes or message payload to process, the driver enters the Done state, otherwise it enters the Handshake state.

- *Handshake state*: The driver enters this state after it sends the command byte of a message header. After the downlink handshake is complete (the driver receives the CTS~ interrupt), the driver then checks the command byte to determine if the command is a network variable update or poll command for a network variable index greater than 62. For these network variable commands, the driver enters the Info state. For other commands, the driver enters the Payload state.

Developing the ShortStack Driver

- *Info state*: Although **Figure 6** shows only one Info state, the driver implements two Info states: Info1 and Info2. The driver enters the Info1 state to send the first byte of the extended header for a network variable update or poll command for a network variable index greater than 62. The driver then enters the Info2 state to send the second byte of the extended header. There is no handshake required for sending the second info byte. If the command byte indicated a payload for the message, the driver re-enters the Handshake state to process the handshake for sending the payload. If there is no payload, the driver enters the Done state.

- *Payload state*: The driver enters this state after the handshake is complete (the CTS~ line is asserted) if there is payload data to transmit. The driver remains in this state until all bytes for the message are transmitted. Then, it enters the Done state.

- *Done state*: The driver enters this state either after all payload bytes are sent or if there is no payload for the message. From this state, the driver returns to the Idle state.

# Receive Buffer States

In addition to the states managed by the driver, the driver also manages a set of states for the buffers. **Figure 7** shows the states and state transitions for the receive buffers.
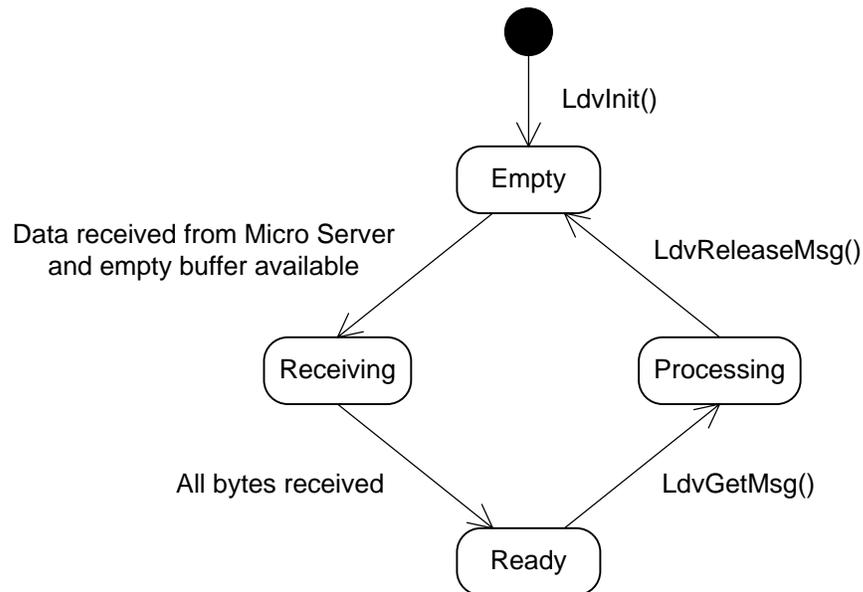


**Figure 7**. State Machine for Receive Buffers

The receive buffers for the driver implement four states:

- *Empty state*: The initial state set by the **LdvInit()** function. A buffer also enters this state as a result of the **LdvReleaseMsg()** call. This call indicates that the ShortStack LonTalk Compact API is done using the buffer (it has copied the contents of the buffer to its own local memory) and that the buffer can be released to the buffer pool.

- *Receiving state*:  When the driver receives data from the Micro Server and there is a buffer available for the message, the driver reserves a buffer and the buffer enters this state.  The buffer remains in this state until all bytes have been received from the Micro Server.

- *Ready state*:  A buffer enters this state after all bytes are received for the message.  This state informs the API that the buffer is available for processing, and that the API will receive this buffer through a call in its event handler.

- *Processing state*:  A buffer enters this state as a result of the **LdvGetMsg()** call.  This state means that the API has access to the buffer's contents and that those contents should not be overwritten.  After the API is done using the buffer, it calls the **LdvReleaseMsg()** function to return the buffer to the driver.

# Transmit Buffer States

**Figure 8** shows the states and state transitions for the transmit buffers.
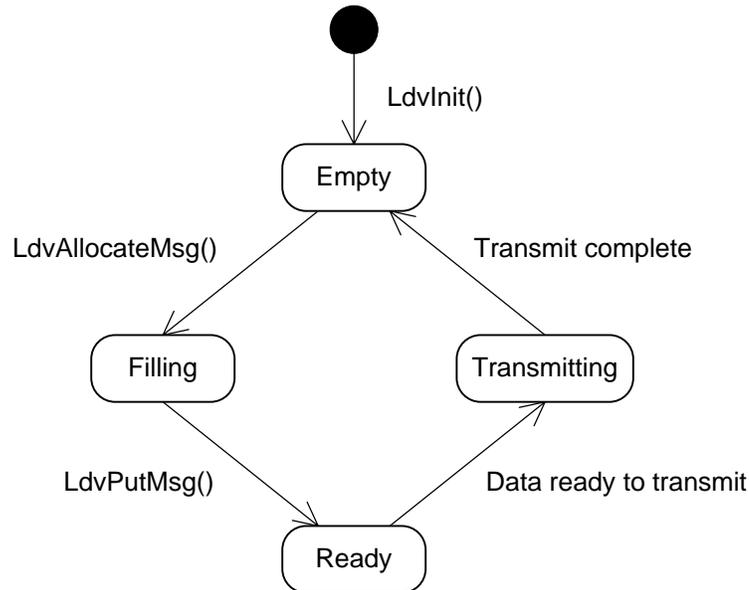


**Figure 8**. State Machine for Transmit Buffers

Similar to the receive buffers, the transmit buffers for the driver implement four states:

- *Empty state*:  The initial state set by the **LdvInit()** function.  A buffer also enters this state after data transmission is complete.

- *Filling state*:  A buffer enters this state as a result of the **LdvAllocateMsg()** call.  This state indicates that the API has access to the buffer and that the API is in the process of filling the buffer with data.

- *Ready state*:  A buffer enters this state as a result of the **LdvPutMsg()** call.  This state indicates that the API has completely filled the buffer with data and that the buffer is ready to be sent to the Micro Server.  By calling the **LdvPutMsg()** function, the API gives the control of the buffer to the driver.

- *Transmitting state*:  A buffer enters this state when the driver begins transmitting its data to the Micro Server.  The buffer remains in this state until all of the data has been transmitted.

# Upper-Layer Serial Driver Implementation

The upper-layer serial driver represents the interface between the ShortStack LonTalk Compact API callback handler functions and the serial driver.  The upper-layer serial driver includes the following functions:

- **LdvInit()**:  Initializes the serial driver, including the hardware interface between the ARM7 microprocessor and the ShortStack Micro Server.

- **LdvGetMsg()**:  Determines if there are messages waiting in a receive buffer, and provides a pointer to the first available data message.

- **LdvReleaseMsg()**:  Releases a receive buffer to the serial driver.

- **LdvAllocateMsg()**:  Determines if there is an available transmit buffer, and provides a pointer to the allocated buffer.

- **LdvPutMsg()**:  Instructs the driver to send the message downlink.  The **RxTxInterruptHandler()** function of the lower-layer serial driver manages the handshake signals and sends the message downlink.

- **LdvPutMsgBlocking()**:  Copies a message into the driver, calls the **LdvFlushMsgs()** function, and waits for the message to be sent downlink.  Use this function to send messages downlink synchronously, or for messages that are larger than a single transmit buffer.

- **LdvFlushMsgs()**:  Completes all pending transactions, including transmitting messages in the buffers or in the driver.  The **RxTxInterruptHandler()** function of the lower-layer serial driver manages the handshake signals and sends any pending messages downlink.

- **LdvReset()**:  Resets the serial driver when it receives an uplink reset message from the Micro Server.

# Lower-Layer SCI Serial Driver Implementation

The lower-layer serial driver represents the interface between the upper-layer serial driver and the hardware.  The lower-layer serial driver includes the following functions:

- **ResetMicroServer()**:  This function resets the ShortStack Micro Server.

- **SuspendSci()**:  This function suspends the operation of the SCI driver.

- **ResumeSci()**:  This function resumes the operation of the SCI driver.

- **CyclicIncrement()**:  This is a driver-internal function that ensures that the indices for the buffers are incremented properly.

- **PrintData()**:  This is a driver-internal function that prints data to the STDOUT device.  This function is included only if the **PRINT_LINK_LAYER** macro is defined.

- **CtsInterruptHandler()**:  The SCI interrupt handler for the CTS~ signal. The ARM7 microprocessor generates an interrupt when the CTS~ signal changes.

- **RxTxInterruptHandler()**:  The SCI interrupt handler for both the RXD and TXD signals.  The ARM7 microprocessor generates an interrupt whenever the receive buffer is full or the transmit buffer is empty. Changes to these buffers represent a need to process uplink or downlink data.  This interrupt handler determines which buffer caused the interrupt, and takes the appropriate action.

- **PeriodicIntervalTimerHandler()**:  The SCI interrupt handler for the periodic interval timer.  The ARM7 microprocessor generates an interrupt when the periodic interval timer expires.  The default value of this timer is 1 millisecond.  This interrupt handler updates two utility timers, and then checks whether the driver's receive timer, sleep timer, keep-alive timer, or message-blocking timer has expired, and takes the appropriate action.

In addition to the lower-layer driver functions, the **LdvSci.h** file implements the following macros to manage the handshake protocol:

- CHECK_CTS_DEASSERTED()
- CHECK_CTS_ASSERTED()
- DEASSERT_RTS()
- ASSERT_RTS()
- CHECK_RTS_DEASSERTED()
- DEASSERT_HRDY()
- ASSERT_HRDY()
- ENABLE_RX_INT()
- DISABLE_RX_INT()
- ENABLE_TX_INT()
- DISABLE_TX_INT()

# 5

# Exploring the Example Applications

This chapter describes the example applications that are included with the ShortStack FX ARM7 Example Port. This chapter describes each application's design, I/O, **main()** function, callback handler functions, application-specific utility functions, and model file. It also describes how to build and load the application images and run the example applications.

# Overview

The ShortStack FX ARM7 Example Port includes three example applications: a simple example, a changeable-type example, and a self-installation example:

- The simple example application is a very simple application that simulates an analog actuator with a gain of two. This simulated device receives an input voltage value, multiplies the value by 2, and returns the new output value.

- The changeable-type example application includes the same functionality as the simple example application, but adds the ability to change the SNVT types for two of the network variables.

- The self-installation example demonstrates the basics of using the Interoperable Self-Installation (ISI) protocol for a ShortStack device. The application is similar to the NcSimpleIsiExample example application that is included with the Echelon NodeBuilder FX/FT Development Tool and the Echelon Mini FX/FT Evaluation Kit, and is similar to the MGDemo example application that is included with the Echelon Mini FX/PL Evaluation Kit.

The following sections describe the example applications, including their design, how to build them in the IAR Embedded Workbench, how to load them into the ARM7 microprocessor on the Pyxos FT EV Pilot Evaluation Board, and how to run them.

# The Simple Example

The simple example application is a very simple application that simulates an analog actuator device that has a gain of two. This simulated device receives an input voltage value, multiplies the value by 2, and returns the new output value.

The model file for this example includes a single **SFPTclosedLoopActuator** functional block for the two network variables. It does not include a Node Object functional block.

The design of the example application is very simple. It includes a single C source file (**main.c**) and the ShortStack LonTalk Compact API files that are generated by the LonTalk Interface Developer utility.

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

## *Application I/O*

The simple example application provides a status LED on the Pyxos FT EV Pilot Evaluation Board. This status LED indicates the current status of the application:

- Before and during initialization: the LED is solid on

- After a successful initialization: the LED is solid off

- After a failed initialization: the LED blinks rapidly (100 ms interval)

**LED1** on the Pyxos FT EV Pilot Evaluation Board is the status LED, as shown in **Figure 9**.
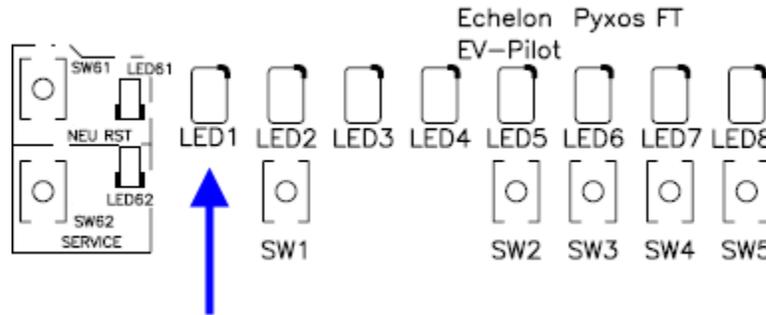


**Figure 9**. Status LED on the Pyxos FT EV Pilot Evaluation Board

The simple example application does not use any other hardware input I/O from the Pyxos FT EV Pilot Evaluation Board.

## Main Function

The **main()** function is in the **main.c** file, which is in the [*ARM7Example*]\**Simple Example** directory.

The **main()** function performs the following tasks:

1. Initializes the ARM7 microprocessor.

2. Illuminates the application's status LED (**LED1** on the EV Pilot Evaluation Board).

3. Calls the **LonInit()** ShortStack LonTalk Compact API function to initialize the ShortStack LonTalk Compact API, the ShortStack serial driver, and the ShortStack Micro Server.

4. If the call to the **LonInit()** function is successful:

    a. Turns off the application's status LED.

    b. Runs an infinite loop to repeatedly call the **LonEventHandler()** API function to handle LONWORKS events. This loop checks whether the Micro Server needs to be re-initialized (such as after the Micro Server image changes, for example, if you load a new Micro Server image), and calls **LonInit()** if necessary.

5. If the call to the **LonInit()** function is not successful, it causes the application's status LED to blink.

Although the **main()** function for this application is an example, you can use the same basic algorithmic approach for a production-level application.

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

```
void main(void) {
  /* Initialize the ARM7 processor */
  ProcessorInit();

  StatusLedInit();

  /* Initialize ShortStack API, serial driver, and
   * ShortStack Micro Server */
```

```
      if (LonInit() != LonApiNoError) {
        /* Initialization failed. Take some defensive action */
        bError = TRUE;
      }
      else {
        /* Assume initialization succeeded */
        bInitialized = TRUE;
        StatusLedOutput(FALSE);

        /* This is the main control loop, which runs forever.*/
        while (!bError) {
          if (!bInitialized) {
            /* The Micro Server might have been updated and
             * needs re-initializing */
            if (LonInit() != LonApiNoError) {
              /* Initialization failed. Take some defensive
               * action */
              bError = TRUE;
            }
            else {
              /* Assume initialization succeeded */
              bInitialized = TRUE;
            }
          }
          /* Update the watch dog timer each time through the
           * control loop */
          UpdateWatchDogTimer();

          /* Handle LonWorks Events */
          LonEventHandler();
        }
      }
      while (bError) {
        StatusLedSignalError();
      }
    }
```

# *Callback Handler Functions*

To signal to the main application the occurrence of certain types of events, the ShortStack LonTalk Compact API calls specific callback handler functions.  For the simple example application, two of the API's callback handler functions have been modified to provide basic LONWORKS networking capability:

- **LonNvUpdateOccurred()**

- **LonResetOccurred()**

The **ShortStackHandlers.c** file (in the [*ARM7Example*]\**Simple Example\ShortStack** directory) contains the modified functions.

Within the **ShortStackHandlers.c** file, each modified function calls a corresponding function in the **main.c** file that provides the application-specific behavior.  This functional-separation approach keeps changes to the LonTalk Interface Developer utility-generated files to a minimum.  For a production-level application, you can place application-specific code wherever your application design requires it.

# LonNvUpdateOccurred()

The modified **LonNvUpdateOccurred()** function is called when the host processor receives a network-variable update.  This function simply calls the **myNvUpdateOccurred()** function in the **main.c** file that provides the application-specific behavior.

The **myNvUpdateOccurred()** function contains a C **switch** statement, which contains a single **case** statement because the **VoltActuator** functional block includes only a single input network variable, **nviVolt**.

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

- Performs bounds checking for the network variable

- Sets the output network variable to double the value of the input network variable

- Propagates the output feedback network variable to the network

The two network variables are defined in the model file, which is described in *Model File* on page 34.

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

```
void myNvUpdateOccurred(const LonByte nvIndex,
        const LonReceiveAddress* const pNvInAddr) {
  switch (nvIndex) {
    case LonNvIndexNviVolt:
    {
      /* Whenever nviVolt is updated, set nvoVoltFb to
       * twice the value of nviVolt.
       */
      /* Copy the input value to another variable as
       * nviVolt is volatile */
      SNVT_volt nviVoltLocal = nviVolt;
      LonBits16 value = LON_GET_SIGNED_WORD(nviVoltLocal);
      if (value > MAX_VOLT) {
        /* Input value is out of range.  Set it to the
         * maximum */
        value = MAX_VOLT;
        LON_SET_SIGNED_WORD(nviVolt, value);
      }
      else if (value < MIN_VOLT) {
        /* Input value is out of range.  Set it to the
         * minimum */
        value = MIN_VOLT;
        LON_SET_SIGNED_WORD(nviVolt, value);
      }
      /* Set nvoVoltFb to 2*nviVolt to simulate a built-in
       * gain of 2.  In a real actuator, nviVolt would be
       * used to set a physical output to adjust the
       * voltage level.  The resulting voltage output would
       * then be read and reported using the nvoVoltFb.
       */
      LON_SET_SIGNED_WORD(nvoVoltFb, value * 2);

      /* Propagate the NV onto the network. */
```

```
            if (LonPropagateNv(LonNvIndexNvoVoltFb) !=
                  LonApiNoError) {
               /* Handle error here, if desired. */
            }
            break;
         }
         /* Add more input NVs here, if any */

         default:
            break;
      }
   }
}
```

# LonResetOccurred()

The modified **LonResetOccurred()** function is called when the Micro Server completes a reset. This function simply calls the **myResetOccurred()** function in the **main.c** file that provides the application-specific behavior, which is to check that the link-layer protocol is the correct version and check whether the Micro Server is properly initialized.

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

```
void myResetOccurred(const LonResetNotification*
      const pResetNotification) {
   if (pResetNotification->Version !=
         LON_LINK_LAYER_PROTOCOL_VERSION)
      bError = TRUE;
   else if (!LON_GET_ATTRIBUTE((*pResetNotification),
         LON_RESET_INITIALIZED))
      bInitialized = FALSE;
      /* This will result in a re-initialization of the
       * Micro Server */
}
```

## *Application-Specific Utility Functions*

The simple example application includes several application-specific utility functions for handling the I/O on the Pyxos FT EV Pilot Evaluation Board.

The I/O functions are:

- **StatusLedInit()** to initialize the status LED

- **StatusLedOutput()** to set the output value of the status LED

- **StatusLedSignalError()** to blink the status LED to signal an error

These functions are defined in the **main.c** file.

## *Model File*

The model file, **Simple Example.nc,** defines the LONWORKS interface for the example ShortStack device. This file is in the [*ARM7Example*]\**Simple Example\ShortStack** directory.

The model file defines one functional block, **VoltActuator**. The **VoltActuator** functional block includes two network variables, **nviVolt** and **nvoVoltFb**. The

functionality for these network variables is implemented in the **myNvUpdateOccurred()** function described in *Callback Handler Functions* on page 32.

The model file is shown below.

```
#pragma enable_sd_nv_names

network input SNVT_volt nviVolt;

network output SNVT_volt bind_info(unackd) nvoVoltFb;

fblock SFPTclosedLoopActuator
{
    nviVolt    implements nviValue;
    nvoVoltFb implements nvoValueFb;
} VoltActuator
external_name("VoltActuator");
```

For more information about creating and using a model file, see the *ShortStack FX User's Guide*.

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

1. Define the interface in the **Simple Example.nc** model file.

2. Run the LonTalk Interface Developer utility to generate an updated application framework. See *Application Framework Data* for information about some of this generated framework.

3. Make appropriate changes to the callback handler functions in the **ShortStackHandlers.c** file or the **main.c** file.

4. Rebuild the project.

5. Load the new executable file into the ARM7 microprocessor.

# *Application Framework Data*

The LonTalk Interface Developer utility generates several blocks of data as part of the application framework for your ShortStack application. The **ShortStackDev.c** file contains the generated framework, which for the simple example includes the following data:

- The C declarations for the two network variables

- The self-identification data for the application

- The application initialization data for the application

- The network variable table

- Functions to get any of the data defined as part of the framework

**Important**: Do not edit or modify the self-identification data or the application initialization data for the application.

You do not need to know the internal structure of the application framework to develop a ShortStack application because this data is generated by the LonTalk Interface Developer utility. However, certain network debugging tasks can be simpler if you are familiar with this data.

For example, the simple example's self-identification data and the application initialization data are defined as shown below.

```
/*
 *  Self-identification data
 *  DO NOT EDIT
 */
static const LonByte siData[] =
{
    0x00, 0x3D, 0x02, 0x00, 0x00, 0x8E, 0x2C, 0x8E,
    0x2C, 0x26, 0x33, 0x2E, 0x34, 0x40, 0x34, 0x56,
    0x6F, 0x6C, 0x74, 0x41, 0x63, 0x74, 0x75, 0x61,
    0x74, 0x6F, 0x72, 0x00, 0x30, 0x6E, 0x76, 0x69,
    0x56, 0x6F, 0x6C, 0x74, 0x00, 0x40, 0x30, 0x7C,
    0x31, 0x00, 0x30, 0x6E, 0x76, 0x6F, 0x56, 0x6F,
    0x6C, 0x74, 0x46, 0x62, 0x00, 0x40, 0x30, 0x7C,
    0x32, 0x00, 0x7F, 0x00, 0x7F
};


/*
 *  Application initialization data
 *  DO NOT EDIT
 */
static const LonByte appInitData[] =
{
    /* 16-bit application signature: */
    LON_APP_SIGNATURE%256u, LON_APP_SIGNATURE/256u,
    /* program Id: */
    0x9F, 0xFF, 0xFF, 0x06, 0x00, 0x0A, 0x04, 0x01,
    /* communication parameters: */
    0x25, 0x2E, 0x08, 0x05, 0x0C, 0x0E, 0x0F, 0x00,
    0x04, 0x00, 0xA4, 0x00, 0x00, 0x00, 0x00, 0x00,
    /* preferences */
    LON_EXPLICIT_ADDRESSING | LON_SERVICE_PIN_TIMER,
    /* number of static network variables: */
    0x02,
    /* one configuration byte per network variable: */
    0x00, 0x60
};
```

The **appInitData[]** array is a ShortStack-specific data block, and is described in the *ShortStack FX User's Guide*. The **siData[]** array is a LONWORKS defined data block, and is described in the *Control Network Protocol Specification*, ISO/IEC 14908. The **siData[]** array contains such information as:

- The number of network variables
- The number of message tags
- Descriptor records for each network variable
- The device self-documentation string
- Extension records for each network variable
- Alias field structures

Your application's self-identification data could include additional fields or different fields.

# The Simple Changeable-Type Example

The simple changeable-type example application demonstrates the basics of using the LONMARK changeable-type protocol in a ShortStack application. The example is not a complete implementation of the protocol, but is meant to serve a starting point for writing your own application.

To demonstrate changeable-type network variables, the example application has a configuration network variable (CPNV) named **nciNvType**, which maintains the current type of the **nviVolt** network variable. The application supports changing the network variable type for this network variable between the **SNVT_volt** type and the **SNVT_volt_mil** type. Any attempt to change the NV to an unsupported type causes the device to reject the change and to revert the **nciNvType** CPNV to its last-known good value. It also disables the functional block.

The model file for this example includes a single **SFPTclosedLoopActuator** functional block for the two network variables. It also includes an **SFPTnodeObject** functional block to manage the device interface.

The design of the example application is relatively simple. It includes a single C source file (**main.c**) and the ShortStack LonTalk Compact API files that are generated by the LonTalk Interface Developer utility.

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

## *Application I/O*

The changeable-type example application provides a status LED on the Pyxos FT EV Pilot Evaluation Board. This status LED indicates the current status of the application:

- Before and during initialization: the LED is solid on
- After a successful initialization: the LED is solid off
- After a failed initialization: the LED blinks rapidly (100 ms interval)

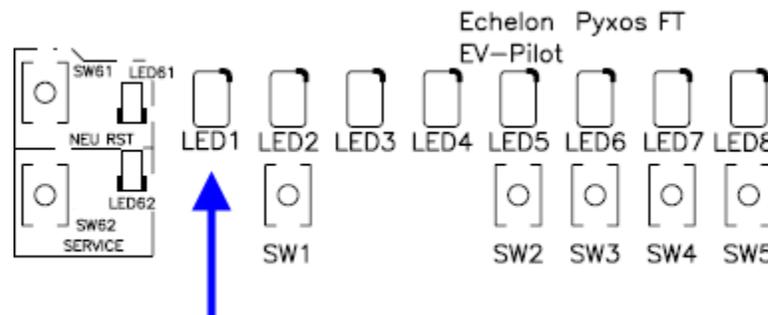**LED1** on the Pyxos FT EV Pilot Evaluation Board is the status LED, as shown in **Figure 10**.



**Figure 10**. Status LED on the Pyxos FT EV Pilot Evaluation Board

The changeable-type example application does not use any other hardware input I/O from the Pyxos FT EV Pilot Evaluation Board.

# Main Function

The **main()** function is in the **main.c** file, which is in the [*ARM7Example*]\\**Simple Changeable-type Example** directory.

The **main()** function performs the following tasks:

1. Initializes the ARM7 microprocessor.

2. Illuminates the application's status LED (**LED1** on the EV Pilot Evaluation Board).

3. Reads the **nciNvType** configuration property network variable in non-volatile data to set its type to the last known good value.

4. Calls the **LonInit()** ShortStack LonTalk Compact API function to initialize the ShortStack LonTalk Compact API, the ShortStack serial driver, and the ShortStack Micro Server.

5. If the call to the **LonInit()** function is successful:

   a. Turns off the application's status LED.

   b. Runs an infinite loop to repeatedly call the **LonEventHandler()** API function to handle LONWORKS events. This loop checks whether the Micro Server needs to be re-initialized (such as after the Micro Server image changes, for example, if you load a new Micro Server image), and calls **LonInit()** if necessary. Within this loop, the application also updates **nciNvType** configuration property network variable in non-volatile data, if necessary.

6. If the call to the **LonInit()** function is not successful, causes the application's status LED to blink.

Although the **main()** function for this application is an example, you can use the same basic algorithmic approach for a production-level application.

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

```
void main(void) {
  unsigned fbIndex;

  /* Initialize the ARM7 processor */
  ProcessorInit();

  StatusLedInit();

  ReadNonVolatileData();

  memset(FbStatus, 0, sizeof(FbStatus));
  for (fbIndex = 0; fbIndex < FBIDX_count; fbIndex++) {
    LON_SET_UNSIGNED_WORD(FbStatus[fbIndex].object_id,
        fbIndex);
  }

  /* Initialize ShortStack API, serial driver, and
   * ShortStack Micro Server */
  if (LonInit() != LonApiNoError) {
    /* Initialization failed. Take some defensive action */
    bError = TRUE;
```

```
    }
    else {
      /* Assume initialization succeeded */
      bInitialized = TRUE;
      StatusLedOutput(FALSE);

      /* This is the main control loop, which runs forever.*/
      while (!bError) {
        if (!bInitialized) {
          /* The Micro Server might have been updated and
           * needs re-initializing */
          if (LonInit() != LonApiNoError) {
            /* Initialization failed. Take some defensive
             * action */
            bError = TRUE;
          }
          else {
            /* Assume initialization succeeded */
            bInitialized = TRUE;
          }
        }
        /* Update the watch dog timer each time through the
         * control loop */
        UpdateWatchDogTimer();

        /* Handle Events */
        LonEventHandler();

        /* Write to the non-volatile data (if required) */
        if (bWriteNonVolatileData)
          WriteNonVolatileData();
      }
    }
    while (bError) {
      StatusLedSignalError();
    }
}
```

# *Callback Handler Functions*

To signal to the main application the occurrence of certain types of events, the ShortStack LonTalk Compact API calls specific callback handler functions. For the changeable-type example application, the following ShortStack LonTalk Compact API callback handler functions have been modified to provide basic LONWORKS networking capability:

- **LonNvUpdateOccurred()**

- **LonOnline()**

- **LonResetOccurred()**

Another important callback handler function is the **LonGetCurrentNvSize()** function. The changeable-type example application uses the default implementation for this callback handler function, which is generated by the LonTalk Interface Developer utility.

The **ShortStackHandlers.c** file (in the [*ARM7Example*]\**Simple Changeable-type Example**\**ShortStack** directory) contains the modified functions and the default implementation for the **LonGetCurrentNvSize()** function.

Within the **ShortStackHandlers.c** file, each modified function calls a corresponding function in the **main.c** file that provides the application-specific behavior.  This functional-separation approach keeps changes to the LonTalk Interface Developer utility-generated files to a minimum.  For a production-level application, you can place application-specific code wherever your application design requires it.

# LonOnline()

The modified **LonOnline()** function is called when the ShortStack device goes online.  This function simply calls the **myOnline()** function in the **main.c** file that provides the application-specific behavior, which is to call the **ProcessTypeChange()** utility function.  That function accepts or rejects a requested type change for the **nviVolt** network variable.  See *ProcessTypeChange()* on page 45 for more information about this function.

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

```
void myOnline(void) {
   /* Type changes may occur while offline.  In that case
    * the value of the nciNvType will be updated but not
    * delivered to the application.  Check for changes every
    * time the device goes online or is reset.
    */
   ProcessTypeChange();
}
```

# LonNvUpdateOccurred()

The modified **LonNvUpdateOccurred()** function is called when the host processor receives a network-variable update.  This function simply calls the **myNvUpdateOccurred()** function in the **main.c** file that provides the application-specific behavior.

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

- A change to the **nciNvType** configuration network variable (CPNV), which controls the type of the **nviVolt** and **nvoVoltFb** network variables.

- A change to the node object's **nviRequest** network variable, which controls the status of the ShortStack device's functional blocks.

- A change to the voltage amplifier's **nviVolt** network variable, which controls the input to the actuator.

The **case** statement for the **nciNvType** CPNV (specified by the **LonNvIndexNciNvType** network variable index) checks the status of the functional block.  If the functional block is disabled, it does nothing.  If the functional block is not disabled, it calls the **ProcessTypeChange()** utility function. See *ProcessTypeChange()* on page 45 for more information about this function.

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

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

The **case** statement for the **nviVolt** network variable (specified by the **LonNvIndexNviVolt** network variable index) checks the status of the functional block. If the functional block is disabled, it does nothing. If it is not disabled, it calls the **UpdateOutputNv()** utility function. See *UpdateOutputNv()* on page 47 for more information about this function.

The functional blocks and network variables are defined in the model file, which is described in *Model File* on page 48.

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

```
void myNvUpdateOccurred(const LonByte nvIndex,
        const LonReceiveAddress* const pNvInAddr) {
  switch (nvIndex) {
    case LonNvIndexNciNvType:
    {
      if (!LON_GET_ATTRIBUTE(FbStatus[FBIDX_VoltActuator],
            LON_DISABLED)) {
        /* The nciNvType has been updated, which controls
         * the type of nviVolt and nvoVoltFb. Validate the
         * change to determine whether to accept the update
         * or not.
         */
        ProcessTypeChange();
      }
      break;
    }

    case LonNvIndexNviRequest:
    {
      /* Copy the input nv locally as it is declared as
       * volatile */
      SNVT_obj_request nviRequestLocal = nviRequest;
      LonUbits16 index =
          LON_GET_UNSIGNED_WORD(nviRequestLocal.object_id);
      memset((void *)&nvoStatus, 0, sizeof(nvoStatus));
      LON_SET_UNSIGNED_WORD(nvoStatus.object_id, index);

      if (index >= FBIDX_count) {
        /* We don't support this object - flag it as an
         * invalid ID. */
```

```
              LON_SET_ATTRIBUTE(nvoStatus, LON_INVALIDID, 1);
           }
           else {
              /* If reportStatus is TRUE, we will set the
               * objectStatus to the status of the specified
               * function block.
               */
              LonBool reportStatus = TRUE;

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

              switch (nviRequestLocal.object_request) {
                 int i;
                 case RQ_NORMAL:
                    /* Set the object (or all objects) to normal by
                     * clearing the disabled and in_override flags.
                     */
                    for (i = start; i <= limit; i++) {
                       LON_SET_ATTRIBUTE(FbStatus[i], LON_DISABLED,
                             0);
                       LON_SET_ATTRIBUTE(FbStatus[i],
                             LON_INOVERRIDE, 0);
                    }

                    /* If the VoltActuator FB got enabled, sync the
                     * output */
                    if (!LON_GET_ATTRIBUTE(
                        FbStatus[FBIDX_VoltActuator],LON_DISABLED))
                    {
                       UpdateOutputNv();
                    }
                    break;

                 case RQ_UPDATE_STATUS:
                    /* Update the status.  If the object is not the
                     * node object, just return the current status
                     * of the object.  Special processing below for
                     * node object only.
                     */
                    if (index == FBIDX_NodeObject) {
                       /* When requesting the status of the node
                        * object, return a status that represents
```

Exploring the Example Applications

```
                 * the OR of the statuses of all function
                 * blocks.
                 * Don't report the status of the node object
                 * - use the summary below.
                 */
               reportStatus = FALSE;

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

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

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

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

             break;

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

         case RQ_ENABLE:
             /* Enable the object or all objects */
             for (i = start; i <= limit; i++) {
               LON_SET_ATTRIBUTE(FbStatus[i], LON_DISABLED,
                     0);
             }

             /* If the VoltActuator FB got enabled, sync the
              * output */
             if (!LON_GET_ATTRIBUTE(
                 FbStatus[FBIDX_VoltActuator], LON_DISABLED))
             {
               UpdateOutputNv();
```

```
          }
          break;

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

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

    case LonNvIndexNviVolt:
    {
      if (!LON_GET_ATTRIBUTE(
          FbStatus[FBIDX_VoltActuator], LON_DISABLED)) {
        /* Whenever nviVolt is updated, set nvoVoltFb to
         * twice the value of nviVolt.
         */
        UpdateOutputNv();
      }
      break;
    }
    /* Add more input NVs here, if any */

    default:
      break;
  }
}
```

# LonResetOccurred()

The modified **LonResetOccurred()** function is called when the Micro Server completes a reset.  This function simply calls the **myResetOccurred()** function in the **main.c** file that provides the application-specific behavior, which is to check that the link-layer protocol is the correct version and check whether the Micro Server is properly initialized.  If it is the correct version, the function then reads the non-volatile data for the value of the **nciNvType** configuration network variable, and then calls the **ProcessTypeChange()** utility function.  See *ProcessTypeChange()* on page 45 for more information about this function.

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

```
    void myResetOccurred(const LonResetNotification*
        const pResetNotification) {
```

```
        if (pResetNotification->Version !=
            LON_LINK_LAYER_PROTOCOL_VERSION)
          bError = TRUE;
        else if (!LON_GET_ATTRIBUTE((*pResetNotification),
            LON_RESET_INITIALIZED)) {
          bInitialized = FALSE;
          /* This will result in a re-initialization of the Micro
           * Server */
        }
        else {
          ReadNonVolatileData();

          /* Type changes may occur while offline.  In that case
           * the value of the nciNvType will be updated but not
           * delivered to the application.  Check for changes
           * every time the device goes online or is reset.
           */
          ProcessTypeChange();
        }
      }
```

# Application-Specific Utility Functions

The changeable-type example application includes several application-specific utility functions for handling the I/O on the Pyxos FT EV Pilot Evaluation Board and for managing non-volatile data in the ARM7 microprocessor's flash memory.

The I/O functions are:

- **StatusLedInit()** to initialize the status LED
- **StatusLedOutput()** to set the output value of the status LED
- **StatusLedSignalError()** to blink the status LED to signal an error

The non-volatile data functions are:

- **ReadNonVolatileData()** to read non-volatile data
- **WriteNonVolatileData()** to write non-volatile data

In addition, the changeable-type example application includes two functions that manage some of the behavior for the network variables:

- **ProcessTypeChange()** manages changes to the **nviVolt** network variable
- **UpdateOutputNv()** updates the **nvoVolt** network variable to be twice the current value of the **nviVolt** network variable

All of these functions are defined in the **main.c** file.

# ProcessTypeChange()

For each type change to the **nviVolt** or **nvoVoltFb** network variables, this function processes the type change.  If the type change is valid (from **SNVT_volt** to **SNVT_volt_mil** or from **SNVT_volt_mil** to **SNVT_volt**), this function converts the network variable's value to match the new type, stores the type and value in the **nciNvType** configuration property network variable, and specifies that the value should be stored in non-volatile memory.

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

```
void ProcessTypeChange(void) {
  /* Accept the CP only if it has changed and the new value
   * is valid. */
  LonBool bAcceptChange = FALSE;
  unsigned newTypeIndex = INDEX_SNVT_VOLT;

  /* Copy the input NV locally as it is declared as
   * volatile */
  SCPTnvType nciNvTypeLocal = nciNvType;

  if (memcmp((void *)&nciNvTypeLocal,
      &nciNvTypeLastKnownGoodValue, sizeof(nciNvType)) != 0)
  {
    /* A change has been requested. See if it is legal. */
    if (nciNvTypeLocal.type_category == 0) {
      newTypeIndex = INDEX_SNVT_VOLT;
      bAcceptChange = TRUE;
    }
    else if (nciNvTypeLocal.type_category >= 1  &&
                nciNvTypeLocal.type_category <= 6
                && nciNvTypeLocal.type_scope == 0) {
      newTypeIndex =
          LON_GET_UNSIGNED_WORD(nciNvTypeLocal.type_index);
      switch (newTypeIndex) {
        case INDEX_SNVT_VOLT:
        case INDEX_SNVT_VOLT_MIL:
          bAcceptChange = TRUE;
          break;
        default:
          break;
      }
    }

    if (bAcceptChange){
      LON_SET_ATTRIBUTE(nvoStatus, LON_INVALIDREQUEST,
            0);
    }
    else {
      /* Invalid type.  Set the CP back to the last known
       * good value. */
      memcpy((void*)&nciNvTypeLocal,
            &nciNvTypeLastKnownGoodValue,
            sizeof(SCPTnvType));
      nciNvType = nciNvTypeLocal;
      /* Reject the unsupported type change */
      LON_SET_ATTRIBUTE(FbStatus[FBIDX_VoltActuator],
            LON_DISABLED, 1);
      LON_SET_ATTRIBUTE(nvoStatus, LON_INVALIDREQUEST, 1);
    }

    if (LonPropagateNv(LonNvIndexNvoStatus) !=
          LonApiNoError) {
      /* Handle error here, if desired. */
    }
  }
```

```
        if (bAcceptChange) {
          SNVT_volt nviVoltLocal = nviVolt;
          LonBits16 value = LON_GET_SIGNED_WORD(nviVoltLocal);
          /* Change the values */
          if (LON_GET_UNSIGNED_WORD(
                      nciNvTypeLastKnownGoodValue.type_index) ==
                      INDEX_SNVT_VOLT && newTypeIndex ==
                      INDEX_SNVT_VOLT_MIL) {
            /* Change from SNVT_volt to SNVT_volt_mil */
            LON_SET_SIGNED_WORD(nviVolt, value * 1000);
            LON_SET_SIGNED_WORD(nvoVoltFb, (value * 2) * 1000);
          }
          else if (LON_GET_UNSIGNED_WORD(
                  nciNvTypeLastKnownGoodValue.type_index) ==
                  INDEX_SNVT_VOLT_MIL && newTypeIndex ==
                  INDEX_SNVT_VOLT) {
            /* Change from SNVT_volt_mil to SNVT_volt */
            LON_SET_SIGNED_WORD(nviVolt, value / 1000);
            LON_SET_SIGNED_WORD(nvoVoltFb, (value * 2) / 1000);
          }

          /* Store the new cp information */
          memcpy((void*) &nciNvTypeLastKnownGoodValue, (void*)
                  &nciNvTypeLocal, sizeof(SCPTnvType));
          LON_SET_UNSIGNED_WORD(
                  nciNvTypeLastKnownGoodValue.type_index,
                  newTypeIndex);
          /* Mark the non-volatile data to be written to flash */
          bWriteNonVolatileData = TRUE;
        }
      }
```

# UpdateOutputNv()

Whenever the input **nviVolt** network variable changes, the application updates
the **nvoVolt** network variable to be twice the current value of the **nviVolt** network
variable.  This function ensures that the calculation returns the correct value
based on the current type specified for the **nviVolt** network variable.

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

```
    void UpdateOutputNv(void) {
      /* Copy the input nv locally as it is declared as
       * volatile */
      SNVT_volt nviVoltLocal = nviVolt;
      LonBits16 value = LON_GET_SIGNED_WORD(nviVoltLocal);
      int divider = 1;
      if (LON_GET_UNSIGNED_WORD(
              nciNvTypeLastKnownGoodValue.type_index) ==
              INDEX_SNVT_VOLT)
        divider = 1000;
      if (value > MAX_VOLT_MIL / divider) {
        value = MAX_VOLT_MIL / divider;
        LON_SET_SIGNED_WORD(nviVolt, value);
      }
      else if (value < MIN_VOLT_MIL / divider) {
```

```
      value = MIN_VOLT_MIL / divider;
      LON_SET_SIGNED_WORD(nviVolt, value);
    }
    LON_SET_SIGNED_WORD(nvoVoltFb, value * 2);

    if (LonPropagateNv(LonNvIndexNvoVoltFb) != LonApiNoError)
    {
      /* Handle error here, if desired. */
    }
}
```

# Model File

The model file, **Simple Changeable-type Example.nc,** defines the LONWORKS interface for the example ShortStack device. This file is in the [*ARM7Example*]\**Simple Changeable-type Example\ShortStack** directory.

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

The **VoltActuator** functional block includes two network variables, **nviVolt** and **nvoVoltFb**. The functionality for these network variables is implemented in the **myNvUpdateOccurred()** function described in *Callback Handler Functions* on page 39.

The two network variables for the **VoltActuator** functional block include references to a configuration network variable (CPNV), **nciNvType**. These references allow the application to be informed when a network management tool modifies the network variable type. The references also allow the **nviVolt** and **nvoVoltFb** network variables to maintain type changes in non-volatile memory, and thus be preserved across device resets.

The model file is shown below.

```
#pragma enable_sd_nv_names

network input cp SCPTnvType nciNvType;

network input SNVT_obj_request nviRequest;

network output sync SNVT_obj_status nvoStatus;

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

network input changeable_type SNVT_volt nviVolt
nv_properties
{
    global nciNvType
};
```

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

fblock SFPTclosedLoopActuator
{
    nviVolt    implements nviValue;
    nvoVoltFb implements nvoValueFb;
} VoltActuator
external_name("VoltActuator");
```

For more information about creating and using a model file, see the *ShortStack FX User's Guide*.

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

1. Define the interface in the **Simple Changeable-type Example.nc** model file.

2. Run the LonTalk Interface Developer utility to generate an updated application framework.

3. Make appropriate changes to the callback handler functions in the **ShortStackHandlers.c** file or the **main.c** file.

4. Rebuild the project.

5. Load the new executable file into the ARM7 microprocessor.

# The Self-Installation Example

The self-installation example application demonstrates the basics of using the Interoperable Self-Installation (ISI) protocol for a ShortStack device.  The example does not include a complete implementation of the protocol, but is meant as a starting point for writing your own application.

The self-installation example application is similar to the NcSimpleIsiExample example application that is included with the Echelon NodeBuilder FX/FT Development Tool and the Echelon Mini FX/FT Evaluation Kit, except that the ShortStack example implements two switch and light pair sets rather than one. It is also similar to the MGDemo example application that is included with the Echelon Mini FX/PL Evaluation Kit, except that the ShortStack example implements only two of the four switch and light pair sets, and does not include the temperature sensor or piezo buzzer.

As described in *Application I/O* on page 50, the example implements two sets of switch and light pairs.  For each set, one switch and light pair implements a switch that is hard-wired to a local light, and the other switch and light pair controls the ISI connection for the first pair of the set.  Each switch is implemented with an **SFPTclosedLoopSensor** functional block, and each light is implemented with an **SFPTclosedLoopActuator** functional block.

When you use the self-installation example application in a self-installed environment, you cannot connect a switch without its corresponding light.  For

example, the **LED5** light emulates a light bulb that is physically connected to the corresponding **SW2** switch.

When you use the self-installation example application in a managed environment, you can independently connect each of the switch and light functional blocks to demonstrate the additional flexibility provided by managed networks.  For example, you can use the self-installation example application with the LonMaker tool and connect each of the four individual functional blocks independently.  However, in the managed environment, the application stops the ISI engine in the Micro Server.

## *Application I/O*

The self-installation example application provides a status LED on the Pyxos FT EV Pilot Evaluation Board.  This status LED indicates the current status of the application:

- Before and during initialization:  the LED is solid on

- After a successful initialization:  the LED is solid off

- After a failed initialization:  the LED blinks rapidly (100 ms interval)

**LED1** on the Pyxos FT EV Pilot Evaluation Board is the status LED.

In addition, the self-installation example application provides four pushbuttons and four LEDs:

- **SW2** and **LED5** are switch and light pair 1

- **SW3** and **LED6** are the ISI connect switch and light for pair 1

- **SW4** and **LED7** are switch and light pair 2

- **SW5** and **LED8** are the ISI connect switch and light for pair 2

The self-installation example application does not use any other hardware input I/O from the Pyxos FT EV Pilot Evaluation Board.

The LEDs and pushbuttons for the self-installation example are shown in **Figure 11** on page 51.
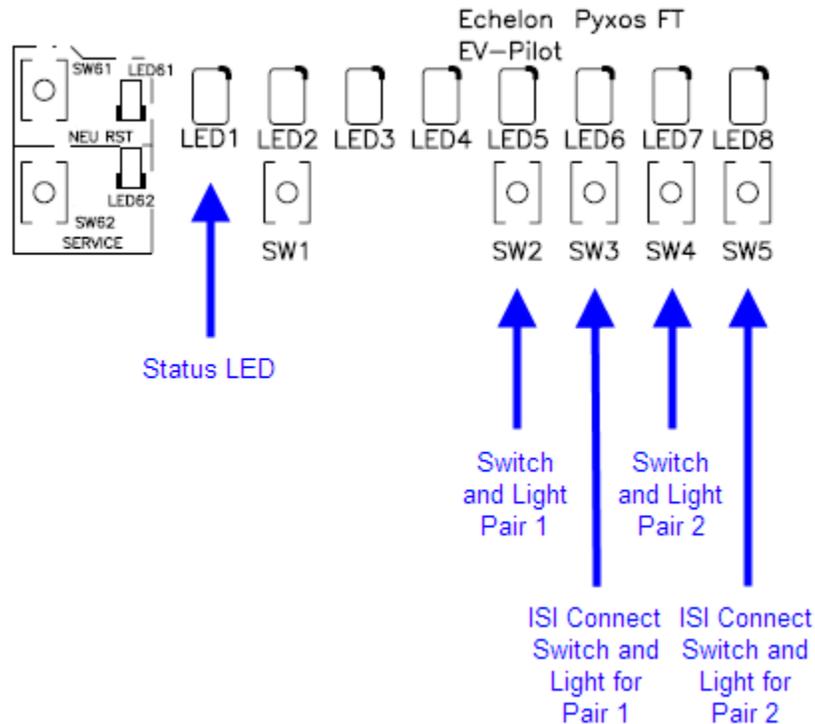
Exploring the Example Applications

**Figure 11**. ISI Example I/O on the Pyxos FT EV Pilot Evaluation Board

Pressing the **SW2** button toggles the state of **LED5**. Likewise, pressing the **SW4** button toggles the state of **LED7**.

When the device operates in ISI mode, **SW3** has the following behavior:

- Pressing the **SW3** button initiates an ISI connection for **SW2**; the Connect LED for pair 1 (**LED6**) blinks. While the device is in this state, other ISI devices can join this connection. Press the **SW3** button again to complete the connection.

- The Connect LED for pair 1 (**LED6**) also blinks if the Connect switch on another ISI device is pressed. Press the Connect switch for pair 1 (**SW3**) to make the pair 1 switch and light (**SW2** and **LED5**) a part of that connection. Then, press the other ISI device's Connect switch to complete the connection.

Similarly, when the device operates in ISI mode, **SW5** has the following behavior:

- Pressing the **SW5** button initiates an ISI connection for **SW4**; the Connect LED for pair 2 (**LED8**) blinks. While the device is in this state, other ISI devices can join this connection. Press the **SW5** button again to complete the connection.

- The Connect LED for pair 2 (**LED8**) also blinks if the Connect switch on another ISI device is pressed. Press the Connect switch for pair 2 (**SW5**) to make the pair 2 switch and light (**SW4** and **LED7**) a part of that connection. Then, press the other ISI device's Connect switch to complete the connection.

# Main Function

The **main()** function is in the **main.c** file, which is in the [*ARM7Example*]\**Self-installation Example** directory.

The **main()** function performs the following tasks:

1. Initializes the ARM7 microprocessor.

2. Initializes the LED and pushbutton I/O on the Pyxos FT EV Pilot Evaluation Board.

3. Illuminates the application's status LED (**LED1** on the Pyxos FT EV Pilot Evaluation Board).

4. Reads the **nciNetConfig** configuration network variable in non-volatile data to set the device configuration to its last valid state.

5. Calls the **LonInit()** ShortStack LonTalk Compact API function to initialize the ShortStack LonTalk Compact API, the ShortStack serial driver, and the ShortStack Micro Server.

6. If the call to the **LonInit()** function is successful:

   a. Turns off the application's status LED.

   b. Runs an infinite loop to repeatedly call the **LonEventHandler()** API function to handle LONWORKS events.  If the Micro Server image changes (for example, if you load a new Micro Server image), this loop checks if the Micro Server needs to be re-initialized and calls **LonInit()**.  Within this loop, the application also:

      - Calls the **HandleSwitchUpdates()** function to process pushbutton activity and toggle the corresponding LEDs.

      - Calls the **UpdateConnectionLeds()** function to set the ISI-related LEDs (**LED6** and **LED8** on the Pyxos FT EV Pilot Evaluation Board) based on the current state of the ISI assemblies.

      - Updates the **nciNetConfig** configuration network variable in non-volatile data, if necessary.

7. If the call to the **LonInit()** function is not successful, causes the application's status LED to blink.

Although the **main()** function for this application is an example, you can use the same basic algorithmic approach for a production-level application.

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

```
void main(void) {
  unsigned fbIndex;

  /* Initialize the ARM7 processor */
  ProcessorInit();

  IoInit();
  SetDigitalOutput(LED_STATUS, TRUE);

  ReadNonVolatileData();
```

Exploring the Example Applications

```
memset(FbStatus, 0, sizeof(FbStatus));
for (fbIndex = 0; fbIndex < FBIDX_count; fbIndex++) {
  LON_SET_UNSIGNED_WORD(FbStatus[fbIndex].object_id,
        fbIndex);
}

/* Initialize ShortStack API, serial driver, and */
/* ShortStack Micro Server */
if (LonInit() != LonApiNoError) {
  /* Initialization failed. Take some defensive action */
  bError = TRUE;
}
else {
  /* Assume initialization succeeded */
  bInitialized = TRUE;
  SetDigitalOutput(LED_STATUS, FALSE);

  /* This is the main control loop, which runs forever.*/
  while (!bError) {
    if (!bInitialized) {
      /* The Micro Server might have been updated and
       * needs re-initializing */
      if (LonInit() != LonApiNoError) {
        /* Initialization failed. Take some defensive
         * action */
        bError = TRUE;
      }
      else {
        /* Assume initialization succeeded */
        bInitialized = TRUE;
      }
    }
    /* Update the watch dog timer each time through */
    /* the control loop */
    UpdateWatchDogTimer();

    /* Handle Events */
    LonEventHandler();

    HandleSwitchUpdates();

    UpdateConnectionLeds();

    /* Write to the non-volatile data (if required) */
    if (bWriteNonVolatileData)
      WriteNonVolatileData();
  }
}
while (bError) {
    SignalError();
}
}
```

## Callback Handler Functions

To signal to the main application the occurrence of certain types of events, the ShortStack LonTalk Compact API calls specific callback handler functions. For the self-installation example application, the following ShortStack LonTalk Compact API callback handler functions have been modified to provide basic LONWORKS networking capability:

- **LonNvUpdateOccurred()**
- **LonResetOccurred()**
- **LonServicePinHeld()**

The **ShortStackHandlers.c** file (in the [*ARM7Example*]\**Self-installation Example\ShortStack** directory) contains the modified functions.

In addition, the following ShortStack ISI API callback handler functions have been modified to provide self-installation capability:

- **IsiCreateCsmo()**
- **IsiGetAssembly()**
- **IsiGetNextAssembly()**
- **IsiGetNextNvIndex()**
- **IsiGetNvIndex()**
- **IsiGetPrimaryGroup()**
- **IsiGetWidth()**
- **IsiUpdateUserInterface()**

The **ShortStackIsiHandlers.c** file (in the [*ARM7Example*]\**Self-installation Example\ShortStack** directory) contains the modified functions.

Within the **ShortStackHandlers.c** and **ShortStackIsiHandlers.c** files, each modified function calls a corresponding function in the **main.c** file that provides the application-specific behavior. This functional-separation approach keeps changes to the LonTalk Interface Developer utility-generated files to a minimum. For a production-level application, you can place application-specific code wherever your application design requires it.

# Functions in ShortStackHandlers.c

This section describes the modified functions in the **ShortStackHandlers.c** file.

## LonNvUpdateOccurred()

The modified **LonNvUpdateOccurred()** function is called when the host processor receives a network-variable update. This function simply calls the **myNvUpdateOccurred()** function in the **main.c** file that provides the application-specific behavior.

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

- A change to the **nciNetConfig** configuration network variable (CPNV), which contains the current configuration of the device.

- A change to the node object's **nviRequest** network variable, which controls the status of the ShortStack device's functional blocks.

- A change to the **nviLight** network variable for the first light and switch pair (**LED5**), which represents a change in the state of that pair.

- A change to the **nviLight** network variable for the second light and switch pair (**LED7**), which represents a change in the state of that pair.

The **case** statement for the **nciNetConfig** CPNV, specified by the **LonNvIndexNciNetConfig** network variable index, performs the following tasks:

- Checks whether the device's current configuration and the last-known good value for its configuration:

  - If the current configuration is set to external, and the last-known configuration is not set to external, a network management tool is controlling the device. The function stops the ISI engine.

  - If the current configuration is set to local, and the last-known configuration is not set to local, the network management tool is no longer controlling the device. The function starts the ISI engine.

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

- Checks whether the object index represents a supported object:

  - For non-supported objects, sets the object status to invalid.

  - For supported objects:

    - Checks whether the command applies to all objects or to a specified functional block

    - Performs the specified command (implemented in a **switch** statement based on the **nviRequest.object_request** variable)

    - Checks whether it should report the status for the object

- Propagates the change to the network

The **case** statement for the **nviLight[0]** network variable for the first light and switch pair (**LED5**), specified by the **LonNvIndexNviLight__1** network variable index, performs the following tasks:

- Checks whether the functional block is disabled. If it is disabled, it does nothing. If it is not disabled:

  - Sets the value for **LED5** to match the value of the **nviLight[0]** network variable

  - Sets the value of the **nvoLightFb[0]** output network variable to the value of the **nviLight[0]** network variable

  - Sets the value of the **nvoSwitch[0]** output network variable to the value of the **nviLight[0]** network variable

> o   Sets the value of the **nviSwitchFb[0]** input network variable to the value of the **nviLight[0]** network variable

The **case** statement for the **nviLight[1]** network variable for the second light and switch pair (**LED7**), specified by the **LonNvIndexNviLight__2** network variable index, performs the following tasks:

- Checks whether the functional block is disabled.  If it is disabled, it does nothing.  If it is not disabled:

  > o   Sets the value for **LED7** to match the value of the **nviLight[1]** network variable

  > o   Sets the value of the **nvoLightFb[1]** output network variable to the value of the **nviLight[1]** network variable

  > o   Sets the value of the **nvoSwitch[1]** output network variable to the value of the **nviLight[1]** network variable

  > o   Sets the value of the **nviSwitchFb[1]** input network variable to the value of the **nviLight[1]** network variable

The functional blocks and network variables are defined in the model file, which is described in *Model File* on page 66.

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

```
void myNvUpdateOccurred(const LonByte nvIndex,
        const LonReceiveAddress* const pNvInAddr) {
  switch (nvIndex) {
    case LonNvIndexNciNetConfig:
    {
      if (nciNetConfig == CFG_EXTERNAL  &&
            nciNetConfigLastKnownGoodValue != CFG_EXTERNAL)
      {
        /* Some network tool is now managing this device.
            It's not an ISI device anymore. */
        nciNetConfigLastKnownGoodValue = nciNetConfig;
        bWriteNonVolatileData = TRUE;
        IsiStop();
      }
      else if (nciNetConfig == CFG_LOCAL  &&
              nciNetConfigLastKnownGoodValue != CFG_LOCAL)
      {
        /* The external tool has stopped managing the
            device.  Go back to the ISI mode. */
        nciNetConfigLastKnownGoodValue = nciNetConfig;
        bWriteNonVolatileData = TRUE;
        IsiStart(IsiTypeS, IsiFlagExtended);
      }
      break;
    }

    case LonNvIndexNviRequest:
    {
      /* Copy the input NV locally as it is declared as
       * volatile */
      SNVT_obj_request nviRequestLocal = nviRequest;
      LonUbits16 index =
          LON_GET_UNSIGNED_WORD(nviRequestLocal.object_id);
```

```
                memset((void *)&nvoStatus, 0, sizeof(nvoStatus));
                LON_SET_UNSIGNED_WORD(nvoStatus.object_id, index);

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

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

                  switch (nviRequest.object_request) {
                    int i;
                    case RQ_NORMAL:
                      /* Set the object (or all objects) to normal by
                       * clearing the disabled and in_override flags.
                       */
                      for (i = start; i <= limit; i++) {
                        LON_SET_ATTRIBUTE(FbStatus[i], LON_DISABLED,
                            0);
                        LON_SET_ATTRIBUTE(FbStatus[i],
                            LON_INOVERRIDE, 0);
                      }
                      break;

                    case RQ_UPDATE_STATUS:
                      /* Update the status.  If the object is not the
                       * node object, just return the current status
                       * of the object.  Special processing below for
                       * node object only.
                       */
                      if (index == FBIDX_NodeObject) {
                        /* When requesting the status of the node
                         * object, return a status that represents
                         * the OR of the statuses of all functional
                         * blocks.
```

```
                          * Don't report the status of the node object
                          * - use the summary below.
                          */
                        reportStatus = FALSE;

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

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

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

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

                    break;

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

                  case RQ_ENABLE:
                    /* Enable the object or all objects */
                    for (i = start; i <= limit; i++) {
                      LON_SET_ATTRIBUTE(FbStatus[i], LON_DISABLED,
                          0);
                    }
                    break;

                  default:
                    /* Mark all other requests as invalid */
                    LON_SET_ATTRIBUTE(nvoStatus,
                        LON_INVALIDREQUEST, 0);
                    reportStatus = FALSE;
                }
                if (reportStatus) {
```

```
                    /* Report the current status of the function
                     * block */
                    nvoStatus = FbStatus[index];
                }
            }

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

        case LonNvIndexNviLight__1:
        {
            if (!LON_GET_ATTRIBUTE(FbStatus[FBIDX_FbLight_1],
                    LON_DISABLED)) {
                SetDigitalOutput(LED_1, nviLight[0].state);
                nvoLightFb[0] = nviLight[0];
                nvoSwitch[0] = nviLight[0];
                nviSwitchFb[0] = nviLight[0];
            }
            break;
        }

        case LonNvIndexNviLight__2:
        {
            if (!LON_GET_ATTRIBUTE(FbStatus[FBIDX_FbLight_2],
                    LON_DISABLED)) {
                SetDigitalOutput(LED_2, nviLight[1].state);
                nvoLightFb[1] = nviLight[1];
                nvoSwitch[1] = nviLight[1];
                nviSwitchFb[1] = nviLight[1];
            }
            break;
        }
        /* Add more input NVs here, if any */

    default:
        break;
    }
}
```

## LonResetOccurred()

The modified **LonResetOccurred()** function is called when the Micro Server completes a reset. This function simply calls the **myResetOccurred()** function in the **main.c** file that provides the application-specific behavior.

The **myResetOccurred()** function checks that the link-layer protocol is the correct version, checks whether the Micro Server is standard or custom, and checks that the Micro Server supports ISI. For a custom Micro Server, you must ensure that it includes ISI support. The function then reads the non-volatile data to retrieve the configuration data for the device. Based on the configuration data, the function then determines which ISI mode the device is in, and starts the ISI engine.

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

```c
void myResetOccurred(const LonResetNotification*
    const pResetNotification) {
  if (pResetNotification->Version !=
      LON_LINK_LAYER_PROTOCOL_VERSION)
    bError = TRUE;
  else if (!LON_GET_ATTRIBUTE((*pResetNotification),
      LON_RESET_INITIALIZED)) {
    bInitialized = FALSE;  /* This will result in a
        re-initialization of the Micro Server */
  }
  else if (!(LON_GET_UNSIGNED_WORD(pResetNotification->Key)
                    & 0x8000)) {
    /* This is a standard Micro Server. Make sure that it
     * supports ISI.  If you are using a custom Micro
     * Server in this device, make sure that you include
     * the appropriate ISI library in that Micro Server. */
    if (!(LON_GET_UNSIGNED_WORD(pResetNotification->Key)
              & 0x0008))
      bError = TRUE; /* Micro Server doesn't support ISI */
  }
  else {
    SCPTnwrkCnfg nciNetConfigLocal;
    ReadNonVolatileData();

    nciNetConfigLocal = nciNetConfigLastKnownGoodValue;

    if (nciNetConfigLocal == CFG_NUL) {
      /* For the first application start, set */
      /* nciNetConfig to CFG_LOCAL, thus allow the ISI */
      /* engine to run by default */
      nciNetConfig = CFG_LOCAL;
      bWriteNonVolatileData = TRUE;
    }

    nciNetConfigLastKnownGoodValue = nciNetConfig;

    if (nciNetConfig == CFG_LOCAL) {
      /* We are in self-installed mode */
      if (nciNetConfigLocal == CFG_EXTERNAL) {
        /* The application has just returned into the */
        /* self-installed mode.  Make sure to */
        /* re-initialize the entire ISI engine */
        IsiReturnToFactoryDefaults();
        /* Call NEVER returns! (resets the device) */
      }
      /* Initialize the arrays */
      /* Set to IsiNormal*/
      memset(IsiAssemblyState, 0,
          sizeof (IsiAssemblyState));
      /* Set to FALSE */
      memset(IsiLedState, 0, sizeof(IsiLedState));
      /* Start the ISI engine */
      IsiStart(IsiTypeS, IsiFlagExtended);
    }
  }
```

```
    }
```

## *LonServicePinHeld()*

The modified **LonServicePinHeld()** function is called when the service pin on the device is pressed (held) for a specified amount of time.  From the System Preferences page of the LonTalk Interface Developer utility, you can specify how long the device's service pin must be held before the ShortStack LonTalk Compact API calls this function.  This function simply calls the **myServicePinHeld()** function in the **main.c** file that provides the application-specific behavior.

The **myServicePinHeld()** function sets the value for the **nciNetConfig** configuration network variable (CPNV), sets a flag so that this value is written to the device's non-volatile data, and then calls the **IsiReturnToFactoryDefaults()** function to reset the ISI engine.

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

```
void myServicePinHeld(void) {
   nciNetConfigLastKnownGoodValue = CFG_LOCAL;
   nciNetConfig = CFG_LOCAL;
   bWriteNonVolatileData = TRUE;
   IsiReturnToFactoryDefaults();    /* Never returns! */
}
```

# Functions in ShortStackIsiHandlers.c

This section describes the modified functions in the **ShortStackIsiHandlers.c** file.

## *IsiCreateCsmo()*

The ISI engine calls the modified **IsiCreateCsmo()** function to define the open enrollment invitation message (the Connection Status Message Open [CSMO] message) for the automatic ISI network variable connection offered by the device. This function simply calls the **myCreateCsmo()** function in the **main.c** file that provides the application-specific behavior.

The **myCreateCsmo()** function checks that the assembly number is within the range defined for the application, and then copies the CSMO data to the address provided in the function call.

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

```
void myCreateCsmo(unsigned assembly, IsiCsmoData** ppCsmo)
{
   if (assembly <= ASSEMBLY_LAST_SWITCHLIGHTPAIR)
       memcpy(*ppCsmo, &CsmoData, sizeof(IsiCsmoData));
}
```

## *IsiGetAssembly()*

The modified **IsiGetAssembly()** function returns the number of the first assembly that can join the enrollment.  This function simply calls the **myGetAssembly()** function in the **main.c** file that provides the application-specific behavior.

The **myGetAssembly()** performs the following tasks:

- Checks whether the CSMO requires the acknowledged service or polling. Because the host application does not support either of these features, this function returns the **ISI_NO_ASSEMBLY** value.

- Checks whether the CSMO requires automatic connections. Because the host application does not support this feature, this function returns the **ISI_NO_ASSEMBLY** value.

- Checks all existing assemblies to determine if one of them initiated the connection; in which case, the function returns the **ISI_NO_ASSEMBLY** value.

- Checks for acceptable connections by verifying the CSMO data, the ISI scope, the extended attributes, the profile, and the variant; for acceptable connections, the function returns the assembly number of the first light and switch pair, otherwise it returns the **ISI_NO_ASSEMBLY** value.

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

```
unsigned myGetAssembly(const IsiCsmoData* pCsmo,
                       LonBool automatic) {
  unsigned assembly;

  /* This application does not accept connections requiring
   * acknowledged service or polling */
  if (!LON_GET_ATTRIBUTE(pCsmo->Extended, ISI_CSMO_ACK) &&
      !LON_GET_ATTRIBUTE(pCsmo->Extended, ISI_CSMO_POLL)) {
    if (!automatic) {
      /* Suppress turn-around connections by looking at
       * existing assemblies. If one of them initiated the
       * connection return no assembly */
      for (assembly = ASSEMBLY_FIRST_SWITCHLIGHTPAIR;
           assembly <= ASSEMBLY_LAST_SWITCHLIGHTPAIR;
           ++ assembly) {
        if(IsiAssemblyState[assembly] == IsiPendingHost ||
           IsiAssemblyState[assembly] == IsiApprovedHost) {
          assembly = ISI_NO_ASSEMBLY;
          break;
        }
      }
    }

    if (assembly != ISI_NO_ASSEMBLY) {
      /* Now test for the different acceptable
       * connections */
      if ((memcmp(pCsmo, &CsmoData, sizeof(IsiCsmoData))
           == 0)
          || ((LON_GET_ATTRIBUTE(pCsmo->Extended,
               ISI_CSMO_SCOPE) == IsiScopeStandard
               && pCsmo->Extended.Member == 1
               && LON_GET_ATTRIBUTE((*pCsmo),
                  ISI_CSMO_WIDTH) == 2
               && pCsmo->NvType == 95u)
               &&
               (((LON_GET_UNSIGNED_WORD(pCsmo->Profile)
                  == 5 && pCsmo->Variant == 128u)
               || (LON_GET_UNSIGNED_WORD(pCsmo->Profile)
                  == 3 && pCsmo->Variant == 0)))
```

```
                        )
                     )
                  {
                     assembly = ASSEMBLY_FIRST_SWITCHLIGHTPAIR;
                  }
                  else
                     assembly = ISI_NO_ASSEMBLY;
               }
            }
            else
               assembly = ISI_NO_ASSEMBLY;
         }
         else
            assembly = ISI_NO_ASSEMBLY;
         return assembly;
      }
```

## IsiGetNextAssembly()

The modified **IsiGetNextAssembly()** function returns the next applicable
assembly for an incoming CSMO message that follows the specified assembly.
This function is called after calling the **IsiGetAssembly()** function, unless
**IsiGetAssembly()** returned **ISI_NO_ASSEMBLY**.  This function simply calls the
**myGetNextAssembly()** function in the **main.c** file that provides the application-
specific behavior.

The **myGetNextAssembly()** function checks whether there are more assemblies
available within the range defined for the application, and returns the next
available assembly number.

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

```
      unsigned myGetNextAssembly(const IsiCsmoData* pCsmo,
            LonBool automatic, unsigned prevAssembly) {
         unsigned assembly = ISI_NO_ASSEMBLY;
         if (prevAssembly < ASSEMBLY_LAST_SWITCHLIGHTPAIR) {
            /* We support a set of multiple similar assemblies
             * ASSEMBLY_FIRST_SWITCHLIGHTPAIR ..
             * ASSEMBLY_LAST_SWITCHLIGHTPAIR */
            assembly = prevAssembly + 1;
         }
         return assembly;
      }
```

## IsiGetNextNvIndex()

The modified **IsiGetNextNvIndex()** function returns the network variable index of
the network variable at the specified offset within the specified assembly,
following the specified network variable.  The function returns the
**ISI_NO_INDEX** value if there are no more network variables.  This function
simply calls the **myGetNextNvIndex()** function in the **main.c** file that provides
the application-specific behavior.

The **myGetNextNvIndex()** function determines whether the assembly number
corresponds to the first or the second switch and light pair, then determines
which network variable index precedes the one passed to the function, and sets
the network variable index to the one that follows.

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

```
unsigned myGetNextNvIndex(unsigned assembly,
              unsigned offset, unsigned prevIndex) {
  unsigned nvIndex = ISI_NO_INDEX;
  if (assembly == ASSEMBLY_FIRST_SWITCHLIGHTPAIR) {
    if (prevIndex == LonNvIndexNvoSwitch__1) {
      nvIndex = LonNvIndexNviLight__1;
    }
    else if (prevIndex == LonNvIndexNvoLightFb__1) {
      nvIndex = LonNvIndexNviSwitchFb__1;
    }
  }
  else if (assembly == ASSEMBLY_LAST_SWITCHLIGHTPAIR) {
    if (prevIndex == LonNvIndexNvoSwitch__2) {
      nvIndex = LonNvIndexNviLight__2;
    }
    else if (prevIndex == LonNvIndexNvoLightFb__2) {
      nvIndex = LonNvIndexNviSwitchFb__2;
    }
  }
  return nvIndex;
}
```

## IsiGetNvIndex()

The modified **IsiGetNvIndex()** function returns the network variable index of the
network variable at the specified offset within the specified assembly; it returns
the **ISI_NO_INDEX** value if no such network variable exists. This function
simply calls the **myGetNvIndex()** function in the **main.c** file that provides the
application-specific behavior.

The **myGetNvIndex()** function determines whether the assembly number
corresponds to the first or the second switch and light pair, and sets the network
variable index value to either the switch or the light, based on the offset value.

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

```
unsigned myGetNvIndex(unsigned assembly, unsigned offset)
{
  unsigned nvIndex = ISI_NO_INDEX;
  if (assembly == ASSEMBLY_FIRST_SWITCHLIGHTPAIR) {
    nvIndex = offset ? LonNvIndexNvoSwitch__1 :
              LonNvIndexNvoLightFb__1;
  }
  else if (assembly == ASSEMBLY_LAST_SWITCHLIGHTPAIR) {
    nvIndex = offset ? LonNvIndexNvoSwitch__2 :
              LonNvIndexNvoLightFb__2;
  }
  return nvIndex;
}
```

## IsiGetPrimaryGroup()

The modified **IsiGetPrimaryGroup()** function returns the group ID for the
specified assembly. This function simply calls the **myGetPrimaryGroup()**
function in the **main.c** file that provides the application-specific behavior.

The **myGetPrimaryGroup()** function returns the default ISI group number (128, defined in **ShortStackIsiTypes.h**).

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

```
unsigned myGetPrimaryGroup(unsigned assembly)
{
    return ISI_DEFAULT_GROUP;
}
```

## IsiGetWidth()

The modified **IsiGetWidth()** function returns the width of the specified assembly. The width is equal to the number of network variable selectors associated with the assembly. This function simply calls the **myGetWidth()** function in the **main.c** file that provides the application-specific behavior.

The **myGetWidth()** function returns a width of 2 for all assemblies because each switch and light pair defined in the model file contains two network variables for each functional block.

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

```
unsigned myGetWidth(unsigned assembly) {
   return 2;
}
```

## IsiUpdateUserInterface()

The modified **IsiUpdateUserInterface()** function is called to synchronize the device's user interface with the ISI engine. This function simply calls the **myUpdateUserInterface()** function in the **main.c** file that provides the application-specific behavior.

The **myUpdateUserInterface()** function checks the type of ISI event that caused the ISI engine to call this function, and checks the number of assemblies defined for the device. If there are no assemblies and the ISI event is either normal or cancelled, the function resets the assembly state and the LED state. If there are assemblies, and the ISI event is the start of the ISI engine, the function sets a flag; otherwise the function stores the event in the assembly state array.

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

```
void myUpdateUserInterface(IsiEvent event,
        unsigned parameter) {
  if (parameter == ISI_NO_ASSEMBLY && (event == IsiNormal
        || event == IsiCancelled)) {
    memset(IsiAssemblyState, 0, sizeof (IsiAssemblyState));
    memset(IsiLedState, 0, sizeof(IsiLedState));
  }
  else if (event == IsiRun) {
    bIsiEngineRunning = parameter ? TRUE : FALSE;
  }
  else if (parameter < sizeof(IsiAssemblyState)) {
    IsiAssemblyState[parameter] = event;
  }
}
```

# Application-Specific Utility Functions

The self-installation example application includes a number of application-specific utility functions for handling the I/O on the Pyxos FT EV Pilot Evaluation Board and for managing non-volatile data in the ARM7 microprocessor's flash memory.

The I/O functions are:

- **IoInit()** to initialize the I/O

- **GetDigitalInput()** to get the value of a specified digital input

- **SetDigitalOutput()** to set the specified digital output (on or off)

- **SignalError()** to blink the status LED to signal an error

- **HandleSwitchUpdates()** to check for updates to the state of any of the switches

- **UpdateConnectionLeds()** to set the state of an LED based on assembly state

- **ProcessNormalSwitch()** to update the network variable values for a specified switch

- **ProcessConnectionSwitch()** to handle ISI enrollment for a specified switch

The non-volatile data functions are:

- **ReadNonVolatileData()** to read non-volatile data

- **WriteNonVolatileData()** to write non-volatile data

All of these functions are defined in the **main.c** file.

# Model File

The model file, **Self-installation Example.nc,** defines the LONWORKS interface for the example ShortStack device. This file is in the [*ARM7Example*]\**Self-installation Example\ShortStack** directory.

The model file defines three functional blocks:  **NodeObject**, **FbLight**, and **FbSwitch**.  The **NodeObject** functional block allows a network management tool to enable or disable the functional blocks for the ShortStack device.  The **FbLight** and **FbSwitch** functional blocks define the interface for the application.

The **NodeObject** functional block, in addition to including the two required **SNVT_obj_status** network variables, includes the **nciNetConfig** configuration network variable (CPNV) within its **fb_properties** clause.  This CPNV allows the device to maintain its current configuration state in non-volatile memory, and thus be preserved across device resets.

The **FbLight** functional block is a functional-block array of two elements.  Each array element includes two network variables, **nviLight** and **nvoLightFb**:

- The **nviLight** network variable array defines the LEDs (**LED5** and **LED7**) of the switch and light pairs.

- The **nvoLightFb** network variable array defines the Connect LEDs (**LED6** and **LED8**) that illuminate to show the ISI connection state for the switch and light pairs.

The **FbSwitch** functional block is also a functional-block array of two elements. Each array element includes two network variables, **nvoSwitch** and **nviSwitchFb**:

- The **nvoSwitch** network variable array defines the pushbuttons (**SW2** and **SW4**) of the switch and light pairs.

- The **nviSwitchFb** network variable array defines the Connect switches (**SW3** and **SW5**) that control the ISI connection state for the switch and light pairs.

See *Application I/O* on page 50 for a description of the LED and pushbutton I/O for the application.

The functionality for the network variables within both functional blocks is implemented in the **myNvUpdateOccurred()** function described in *Callback Handler Functions* on page 54.

The model file is shown below.

```
#include "SNVT_CFG.h"

#pragma enable_sd_nv_names

network input cp SCPTnwrkCnfg nciNetConfig = CFG_EXTERNAL;

network input SNVT_obj_request nviRequest;

network output sync SNVT_obj_status nvoStatus;

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

network input  SNVT_switch nviLight[2];
network output bind_info(unackd) SNVT_switch nvoLightFb[2];
fblock SFPTclosedLoopActuator
{
    nviLight[0]    implements nviValue;
    nvoLightFb[0] implements nvoValueFb;
} FbLight[2] external_name("FbLight");

network output bind_info(unackd_rpt) SNVT_switch
nvoSwitch[2];
network input  SNVT_switch nviSwitchFb[2];
fblock SFPTclosedLoopSensor
{
    nvoSwitch[0]    implements nvoValue;
    nviSwitchFb[0] implements nviValueFb;
} FbSwitch[2] external_name("FbSwitch");
```

For more information about creating and using a model file, see the *ShortStack FX User's Guide*.

# Building the Application Image

To build the software image for any of the example applications included with the ShortStack FX ARM7 Example Port:

1. Start the IAR Embedded Workbench.

2. Select **File → Open → Workspace** to open the Open Workspace dialog.

3. In the Open Workspace dialog, select one of the following workspace files, and click **Open**:

   - [*ARM7Example*]\**Simple Example\Simple Example.eww**

   - [*ARM7Example*]\**Simple Changeable-type Example\Simple Changeable-type Example.eww**

   - [*ARM7Example*]\**Self-installation Example\Self-installation Example.eww**

4. From the Workspace utility window, select **RAM Debug**, **Flash Debug**, or **Flash Release** from the dropdown list box at the top of the window.  This selection allows you to work with one of the predefined project configurations:

   - The RAM Debug configuration loads the application into RAM and allows you to debug it from the IAR Embedded Workbench.  This configuration is generally larger than the Flash Release configuration because of the added debug support.

   - The Flash Debug configuration loads the application into flash memory and allows you to debug it from the IAR Embedded Workbench.  This configuration is generally larger than the Flash Release configuration because of the added debug support.

   - The Flash Release configuration loads the application into flash memory without debug support; see *Loading the Application Image*.

5. Select **Project → Clean** to clean the project.

6. Select **Project → Rebuild All** to rebuild the project files.

After you build the project, you can run it, as described in *Running the Application* on page 69.

# Loading the Application Image

To load and run either the RAM Debug project configuration or the Flash Debug project configuration, simply run the application from the IAR Embedded Workbench, as described in *Running the Application* on page 69.  However, to load the Flash Release project configuration image into the ARM7 microprocessor, you must use an In-System Programmer, such as the SAM-PROG programmer.

To load the Flash Release project configuration software image into the ARM7 microprocessor:

1. Ensure that the Pyxos FT EV Pilot Evaluation Board is powered on and that a hardware emulator and debugger, such as the AT91SAM-ICE

Exploring the Example Applications

JTAG Emulator, is connected to the evaluation board's JTAG header connector (**JP504**).

2. Build the Flash Release configuration for the project. See *Building the Application Image* on page 68.

3. Start the SAM-PROG programmer.

4. Click **Browse** to open the Open dialog.

5. In the Open dialog, select one of the following executable files, and click **Open**:

   - [*ARM7Example*]**\Simple Example\Flash Release\Exe\Simple Example.bin**

   - [*ARM7Example*]**\Simple Changeable-type Example\Flash Release\Exe\Simple Changeable-type Example.bin**

   - [*ARM7Example*]**\Self-installation Example\Flash Release\Exe\Self-installation Example.bin**

6. Click the Target Connected? **Yes** button to establish communications between the Pyxos FT EV Pilot Evaluation Board and the hardware emulator and debugger.

7. Click **Write Flash** to write the program image to flash memory.

8. After the software is loaded, perform a reset by pressing the **RESET** button on the Pyxos FT EV Pilot Evaluation Board.

9. Click **EXIT** to close the SAM-PROG programmer.

The ARM7 microprocessor runs the loaded software as soon as the processor completes restart processing.

# Running the Application

Before you run any of the applications, load the appropriate Micro Server image into the Atmel AT29C512 flash memory on the Pyxos FT EV Pilot Evaluation Board:

- For the simple example, load [*ARM7Example*]**\Simple Example\ShortStack\Simple Example.nei**

- For the changeable-type example, load [*ARM7Example*]**\Simple Changeable-type Example\ShortStack\Simple Changeable-type Example.nei**

- For the self-installation example, load [*ARM7Example*]**\Self-installation Example\ShortStack\Self-installation Example.nei**

**Recommendation**:  To be able to run all three examples with a single Micro Server image, load the **Self-installation Example.nei** image into the Atmel AT29C512 flash memory; an ISI-enabled Micro Server can run with either an ISI application or a non-ISI application.

See the *ShortStack FX User's Guide* for information about loading a Micro Server image into a Smart Transceiver.

If you loaded the Flash Debug or Flash Release project configuration software image into the ARM7 microprocessor, the application runs automatically as soon

as the ShortStack Micro Server and the ARM7 microprocessor are properly programmed and reset.

You can also run either the RAM Debug or Flash Debug project configuration software image from the IAR Embedded Workbench:

1. Ensure that the Pyxos FT EV Pilot Evaluation Board is powered on and that a hardware emulator and debugger, such as the AT91SAM-ICE JTAG Emulator, is connected to the evaluation board's JTAG header connector (**JP504**).

2. Start the IAR Embedded Workbench.

3. Select **File** → **Open** → **Workspace** to open the Open Workspace dialog.

4. In the Open Workspace dialog, select one of the following workspace files, and click **Open**:

   - [*ARM7Example*]\**Simple Example\Simple Example.eww**

   - [*ARM7Example*]\**Simple Changeable-type Example\Simple Changeable-type Example.eww**

   - [*ARM7Example*]\**Self-installation Example\Self-installation Example.eww**

5. From the Workspace utility window, select **Flash Debug** from the dropdown list box at the top of the window.

6. Select **Project** → **Rebuild All** to rebuild the project files.

7. Select **Project** → **Download and Debug** to load the application image into the ARM7 microprocessor and begin a debugging session.

8. To run the application, select **Debug** → **Go**. To step through the application program, select one of the step options from the **Debug** menu.

## *Running the Simple Example*

To verify that the application runs as expected, connect the Pyxos FT EV Pilot Evaluation Board to a LONWORKS TP/FT-10 compatible network interface, such as a U10 USB Network Interface or an *i*.LON SmartServer, and connect that interface to a network management tool, such as the LonMaker Integration tool. From the tool, modify the value for the **nviVolt** network variable and confirm that the value for the **nvoVoltFb** network variable is double that value:

1. Open the LonMaker drawing for the ShortStack device. See the *LonMaker User's Guide* for more information about working with LonMaker drawings.

2. Ensure that the ShortStack device is properly configured and commissioned.

3. Right-click the ShortStack device on the LonMaker drawing and select **Browse** to open the LonMaker Browser window.

4. Within the LonMaker Browser window, select the row for the **nviVolt** network variable.

5. Enter a value for the network variable in the **Value** field at the top of the window. Click the **Set value** button to set the network variable's value.

6. Select the row for the **nvoVoltFb** network variable, and click the **Get value** button to see its current value. That value should be twice the **nviVolt** value, as shown in **Figure 12**.

| Subsystem | Device | Functional Block | Network Variable | Config Prop | Mon | Value |
|-----------|--------|------------------|------------------|-------------|-----|-------|
| Subsystem 1 | Pilot | VoltActuator | nviVolt | | N | 48.0 |
| Subsystem 1 | Pilot | VoltActuator | nvoVoltFb | | N | 96.0 |

**Figure 12**. LonMaker Browser View for the Two Network Variables

## Running the Changeable-Type Example

To verify that the application runs as expected, connect the Pyxos FT EV Pilot Evaluation Board to a LONWORKS TP/FT-10 compatible network interface, such as a U10 USB Network Interface or an *i*.LON SmartServer, and connect that interface to a network management tool, such as the LonMaker Integration tool. From the tool, modify the value for the **nviVolt** network variable and confirm that the value for the **nvoVoltFb** network variable is double that value, as described for the simple example.

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

1. Open the LonMaker drawing for the ShortStack device. See the *LonMaker User's Guide* for more information about working with LonMaker drawings.

2. Ensure that the ShortStack device is properly configured and commissioned.

3. Right-click the ShortStack device on the LonMaker drawing and select **Browse** to open the LonMaker Browser window.

4. Select the row for either the **nviVolt** or **nvoVoltFb** network variable. Be sure *not* to select the row for the corresponding configuration property, if any (displayed in green, with the value **SCPTnvType** in the Config Prop column).

5. Right-click the **nviVolt** or **nvoVoltFb** network variable and select **Change Type** to open the Select Network Variable Type dialog.

6. In the Select Network Variable Type dialog, select either **SNVT_volt** or **SNVT_volt_mil** (if necessary, expand the tree view for **\LonWorks\Types\STANDARD.FMT**), as shown in **Figure 13** on page 72. Click **OK** to change the type and close the Select Network Variable Type dialog.

   **Important**: You can only change the type for a network variable if it is not connected to another network variable. LonMaker implicitly binds network variables when you enable monitoring from the LonMaker drawing (from the drawing itself or from any connections to the functional blocks) or the LonMaker Browser window. Thus, you must disable monitoring for both the **nviVolt** and **nvoVoltFb** network variables before you change their type.
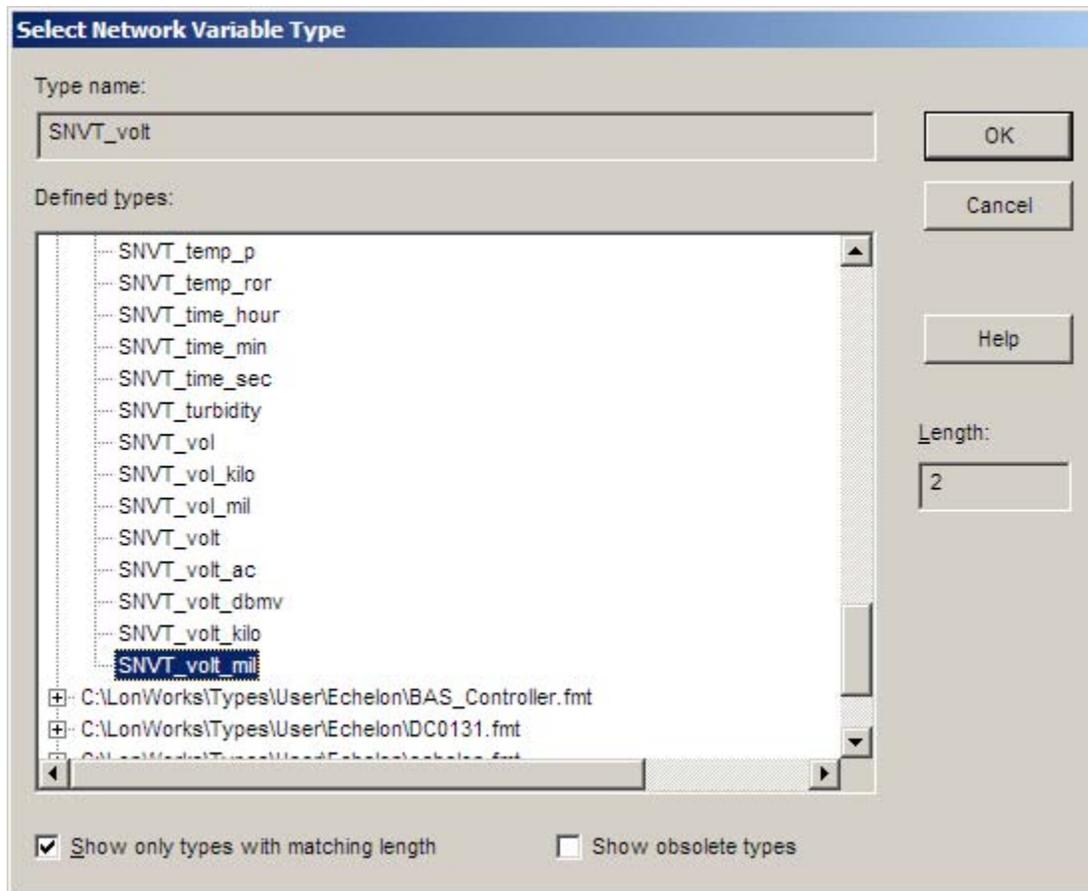
**Figure 13**. The Select Network Variable Type Dialog

To verify that the type changed successfully:

1. Within the LonMaker Browser window, select the row for the **nviVolt** network variable.

2. Right-click the **nviVolt** network variable and select **Properties** to open the Network Variable Properties dialog.

3. On the Description page of the Network Variable Properties dialog, the **Type name** field displays the current type for the network variable.  The current type should be either **SNVT_volt** or **SNVT_volt_mil**, as shown in **Figure 14** on page 73.
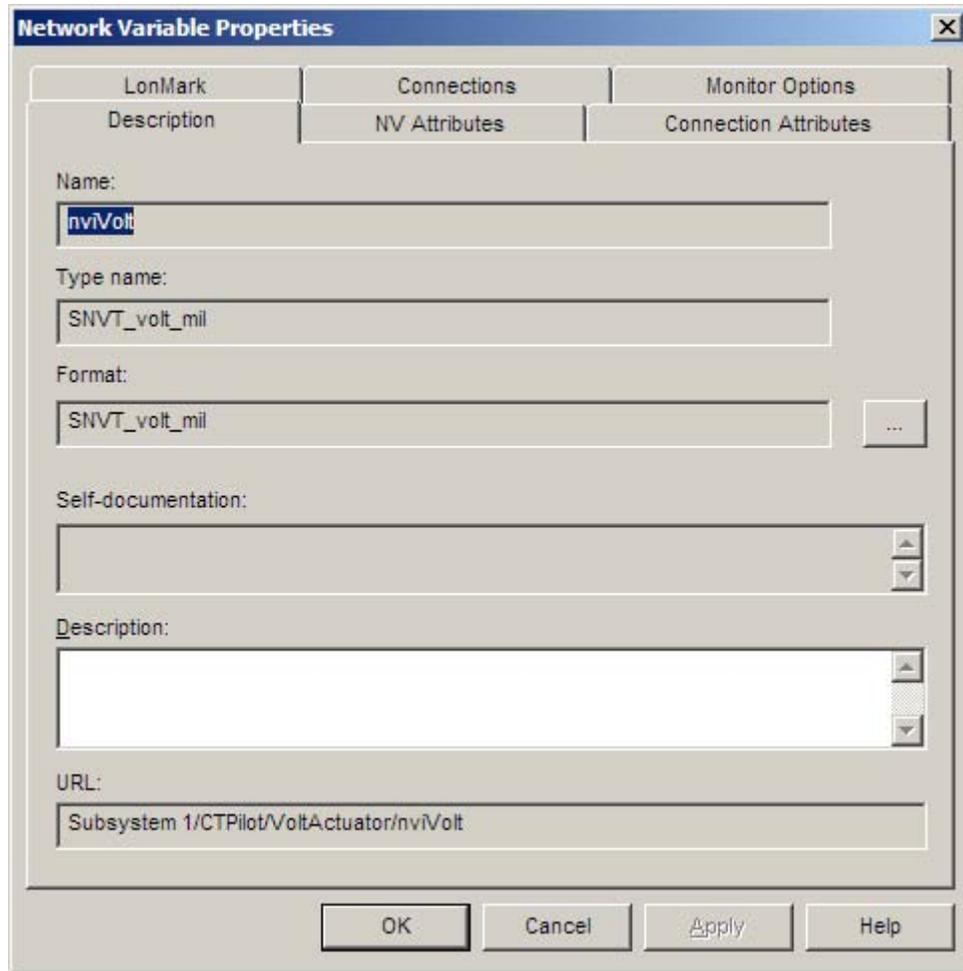
**Figure 14**. The Network Variable Properties Dialog

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

- For **nviVolt** as **SNVT_volt**:  ± 1.6 V

- For **nvoVoltFb** as **SNVT_volt**:  ± 3.2 V

- For **nviVolt** as **SNVT_volt_mil**:   -1638.4 mV to +1638.3 mV

- For **nviVolt** as **SNVT_volt_mil**:  -3276.8 mV to +3276.6 mV

**Important**:  Be sure to change the type to either **SNVT_volt** or **SNVT_volt_mil**, not to **SNVT_vol**, **SNVT_vol_mil**, or any other type.  If you change the type to an invalid type, the changeable-type example application rejects the change and disables the **VoltActuator** functional block.  However, the LonMaker Browser window does not indicate that the application rejected the change.  In this case, you must re-enable the functional block:

1. Change the type for either the **nviVolt** or **nvoVoltFb** network variable to a valid type, **SNVT_volt** or **SNVT_volt_mil**.

2. Within the LonMaker drawing, right-click the **VoltActuator** functional block and select **Manage** to open the LonMaker Device Manager dialog.

3. From the Functional Blocks tab of the LonMaker Device Manager dialog, click **Enable** to re-enable the functional block. Click **Test** to verify that the output for the functional block displays "Disabled: 0", which signifies that the functional block is not disabled.

## *Running the Self-Installation Example*

You can verify the non-ISI part of the application directly from the Pyxos FT EV Pilot Evaluation Board, but to verify that the application is able to make ISI connections, you need to connect the Pyxos FT EV Pilot Evaluation Board to an ISI-capable device through a LONWORKS TP/FT-10 compatible network cable. For example, you can connect the Pyxos FT EV Pilot Evaluation Board to an Echelon FT 5000 EVB Evaluation Board running the NcSimpleIsiExample application or to a Mini FX/PL Evaluation Board running the MGDemo example application.

To test the basic, non-ISI, behavior of the application, press the **SW2** button Pyxos FT EV Pilot Evaluation Board; **LED5** should illuminate. Press the **SW2** button again to turn **LED5** off. Similarly, you can press the **SW4** button to toggle the state of **LED7**.

To test the ISI behavior of the application for the first switch and light pair when the Pyxos FT EV Pilot Evaluation Board is connected to a Mini EVB with an attached MiniGizmo I/O board, perform the steps listed in **Table 7**. The steps in the table begin with the Pyxos FT EV Pilot Evaluation Board.

**Table 7**. Steps for Verifying the ISI Behavior for the First Switch and Light Pair

| Step | On the Pyxos FT EV Pilot Evaluation Board | On the Mini EVB Evaluation Board's MiniGizmo I/O Board |
|------|-------------------------------------------|--------------------------------------------------------|
| 1 | Press the **SW3** button to open enrollment for the first switch and light pair | |
| 2 | **LED6** blinks to indicate that the application is waiting for an ISI device to join the connection | **LED5** through **LED8** all blink to indicate that they are ready to join a connection |
| 3 | | Press the **SW5** button to join the connection |
| 4 | **LED6** stops blinking and stays on | **LED5** stops blinking and stays on to indicate that the connection is pending |
| 5 | Press the **SW3** button to approve the connection and create the enrollment for the first switch and light pair | |
| 6 | **LED6** turns off | **LED5** turns off |
| 7 | Press the **SW2** button | |
| 8 | **LED5** turns on | **LED1** turns on |

| Step | On the Pyxos FT EV Pilot Evaluation Board | On the Mini EVB Evaluation Board's MiniGizmo I/O Board |
|------|-------------------------------------------|-------------------------------------------------------|
| 9 | Press the **SW2** button again | |
| 10 | **LED5** turns off | **LED1** turns off |
| 11 | | Press the **SW1** button |
| 12 | **LED5** turns on | **LED1** turns on |
| 13 | | Press the **SW1** button again |
| 14 | **LED5** turns off | **LED1** turns off |

**Notes:**

- For step 1, you can press either **SW3** or **SW5**. If you press **SW5**, you open enrollment for the second switch and light pair, and **LED8** blinks in step 2.

- For step 3, you can press any of the buttons, **SW5**, **SW6**, **SW7**, or **SW8**. Which button you press determines which of the switch and light pairs on the MiniGizmo join the connection:

    o **SW5** controls **SW1** and **LED1**

    o **SW6** controls **SW2** and **LED2**

    o **SW7** controls **SW3** and **LED3**

    o **SW8** controls **SW4** and **LED4**

To cancel enrollment after step 1 and before step 5, press and hold the Connect button (**SW3** in **Table 7**) for eight seconds. After you release the button, all blinking LEDs turn off.

Instead of opening enrollment from the Pyxos FT EV Pilot Evaluation Board, you can open enrollment from the MiniGizmo board. In this case, press one of the buttons **SW5** through **SW8** to open enrollment for one of the switch and light pairs. Then, **LED6** and **LED8** on the Pyxos FT EV Pilot Evaluation Board blink to indicate that they are ready to join a connection. The remaining steps are similar to those of **Table 7**.

You could also use the FT 5000 Evaluation Board with the Pyxos FT EV Pilot Evaluation Board instead of the Mini EVB and MiniGizmo board. For the FT 5000 EVB, **LED2** blinks in step 2 in **Table 7**; press **SW2** in step 3. Press **SW1** in step 11; **LED1** turns on in step 12.

The self-installation example is limited to one-to-one connections between light and switch pairs from each device. That is, after you open enrollment for a switch and light pair (for example, by pressing the Pilot's **SW3** button, as in **Table 7** step 1), when you press a button to join the connection (for example, by pressing the MiniGizmo's **SW5** button, as in **Table 7** step 3), the connection is pending, and thus you cannot then press another button (such as **SW4** on the MiniGizmo) to have it also join the connection. Likewise, if you connect both a MiniGizmo and an FT 5000 EVB to the network with the Pyxos FT EV Pilot

Evaluation Board, after you initiate enrollment, the first button that you press from either board becomes the light and switch pair that is connected.

See the *Mini FX/PL Hardware Guide* for more information about the Mini EVB Evaluation Board and the MiniGizmo board. See the *Mini FX/PL Examples Guide* for more information about the MGDemo application. See the *FT 5000 EVB Hardware Guide* for information about the FT 5000 EVB. See the *FT 5000 EVB Examples Guide* for information about the NcSimpleIsiExample application.

**ECHELON®**

www.echelon.com