Version 3.3
October 2002

# LONMARK® Application-Layer Interoperability Guidelines

Printed in the United States of America.



Echelon Corporation

www.echelon.com

# Table of Contents

# 1

# Introduction

With thousands of application developers and millions of devices installed worldwide, the LONWORKS platform is the leading open solution for building and home automation, industrial, transportation, and public utility control networks. A control network is any group of devices working in a peer-to-peer fashion to monitor sensors, control actuators, communicate reliably, manage network operation, and provide local and remote access to network data. A LONWORKS network uses the ANSI/EIA/CEA 709.1 Control Network Protocol to accomplish these tasks. The ANSI/EIA/CEA 709.1 protocol is implemented by the firmware provided with Neuron® Chips and Echelon Smart Transceivers; this implementation is known as the *LonTalk® protocol*.

The standard protocol provided by the LONWORKS platform makes it possible to design open control systems using products from multiple vendors. The *LONMARK Interoperability Guidelines* provide guidelines, detailed explanations, and technical insight on how to design interoperable products based on the LONWORKS platform. All products that carry the LONMARK logo (“”) are certified to comply with these guidelines. The LONMARK guidelines are presented in separate volumes for ISO OSI Reference Model layers 1–6 and for layer 7 of the ANSI/EIA/CEA 709.1 protocol. This document provides application-layer (layer 7) design guidelines, and also includes a glossary defining terms for both volumes. The *LONMARK Layer 1–6 Interoperability Guidelines* provides layer 1–6 guidelines.

# 1.1. Introduction to the LONWORKS Platform

A LONWORKS network consists of intelligent *devices*—such as sensors, actuators, and controllers—that communicate with each other using the ANSI/EIA/CEA 709.1 protocol over one or more *communications channels*. Network devices are sometimes also called *nodes*.

A device publishes information as instructed by the application that it is running. The applications on different devices are not synchronized, and it is possible that multiple devices may all try to communicate at the same time. Meaningful transfer of information between devices on a network, therefore, requires organization in the form of a set of rules and procedures. These rules and procedures are defined by the ANSI/EIA/CEA 709.1 protocol. The protocol defines the format of the messages being transmitted between devices and defines the actions expected when one device sends a message to another. The protocol implementation normally takes the form of embedded software or firmware code in each device on the network.

Applications in devices are divided into one or more *functional blocks*. A functional block performs a task by receiving configuration and operational data inputs, processing the data, and sending operational data outputs. A functional block may receive inputs from the network, from hardware attached to the device, or from other functional blocks on a device. A functional block may send outputs to the network, to hardware attached to the device, or to other functional blocks on the device.

The interface defined by the network inputs and outputs to the functional blocks on a device is called the *device interface* (it is also called the *application-layer interface* or the *external interface*). A network tool may upload the device interface definition from the device, or it may read the device interface definition from a standalone file called the *device interface (XIF) file*. In open multi-vendor networks, the design of the device interface is vital to providing interoperability and easy integration. Standardization of the device interfaces is an important element of designing for interoperability.

The following chapters provide detailed information on how products based on the LONWORKS platform should be designed so that the device interface will support easy interoperation across a LONWORKS network. The actual application software and hardware behind the interface is outside the scope of these guidelines. The purpose of the guidelines is to ensure interoperability, but not interchangeability of devices. A major benefit to end-users of interoperable devices is the freedom to choose among suppliers for the devices as well as for the maintenance of those devices. The ability to choose a specific device is provided by public device interfaces that describe the function of the device and how it exchanges information with other devices on a LONWORKS network. The ability to choose among suppliers for system maintenance is realized by ensuring that interoperable devices do not require any private information to be successfully commissioned.

Though interoperable devices may contain proprietary data that is known only to the device manufacturer and the manufacturer's agents, this proprietary data is outside the scope of these guidelines; however, the use of proprietary data cannot be required for the successful commissioning of an interoperable device.

## 1.2. Audience

The information contained in the *LONMARK Application-Layer Interoperability Guidelines* is particularly pertinent to original-equipment manufacturers (OEMs) who plan to design interoperable LONWORKS products, but is also of interest to end-users and specifiers of LONMARK products.

## 1.3. LONMARK Certification

Products may be submitted to the LONMARK Association for certification. A product that is certified by the LONMARK Association as complying with the application-layer and layer 1–6 guidelines may carry the LONMARK logo to indicate that it is capable of being part of an interoperable LONWORKS network. The LONMARK logo is an indication to manufacturers, end users, and network integrators that a product can be easily linked with other products in a multi-vendor network. One of the logos below must be used on the product documentation and/or product casing. If no casing is provided, the logo can be placed on a circuit board or equivalent. The logo cannot be used without at least the "3.3" designating this latest version of the LONMARK Interoperability Guidelines.


**Figure 1. LONMARK Logos**

Contact the LONMARK Association Principal Engineer at the following address for more information about LONMARK certification, or visit the LONMARK Web site for LONMARK certification details.

Principal Engineer - cert@lonmark.org
LONMARK Interoperability Association
550 Meridian Avenue
San Jose, CA 95126 USA
Tel: +1-408-938-5266, Fax: +1-408-790-3493

www.lonmark.org

## 1.4. Character Encoding

All characters referenced in this document as required by the device interface are single-byte, 7-bit ASCII characters unless noted otherwise.

## 1.5. Related Documentation

The following documents provide supplemental information to these guidelines. All documents listed here are available at www.lonmark.org unless noted otherwise.

- *ANSI/EIA/CEA 709.1 Control Network Protocol.* Specifies the services available at each of the seven layers of the ANSI/EIA/CEA 709.1 protocol. Copies of this document are available for purchase at www.global.ihs.com.
- *LONMARK Device Interface File Reference Guide.* Describes the content and structure of a device interface (XIF) file.
- *LONMARK Functional Profiles*. Provide detailed descriptions of all approved LONMARK functional blocks. Up-to-date documentation on all available LONMARK profiles is available on the LONMARK Web site at www.lonmark.org.
- *LONMARK Layer 1–6 Interoperability Guidelines.* Provides guidelines, detailed explanations, and technical insight on how to design and implement the ANSI/EIA/CEA 709 layer 1–6 interface for interoperable products based on the LONWORKS platform. These guidelines form the basis for obtaining the use of the LONMARK logo, which indicates that a product has been certified by the LONMARK Association.
- *LONMARK Program Overview.* Describes the organizational structure and membership options of the LONMARK Interoperability Association, and rules for use of the LONMARK Logo.
- *NodeBuilder® Resource Editor User's Guide.* Describes how to create, edit, and view resource files using the NodeBuilder Resource Editor. Included with the NodeBuilder Resource Editor download available from the LONMARK Web site at www.lonmark.org.

- *SNVT and SCPT Master List.* Documents the range, units, and resolution of all defined SNVTs and SCPTs, and defines all standard enumeration types.
- *Standard Program ID Reference (spidData.xml).* Specifies standard values for the Manufacturer, Device Class, Usage, and Channel Type fields of a standard program ID (SPID). This file is used by development and network tools to simplify construction of standard program IDs.
- *Standard Transceiver Reference (StdXcvr.xml).* Describes the transceiver and channel parameters and properties of all LONMARK channel types as well as some channel types that are not LONMARK compliant. This file is used by development and network tools for automatic validation and channel-type dependent calculations.

# 2

# Device Interfaces

This chapter describes the elements that comprise an interoperable device interface:

- Neuron ID,
- Standard program ID,
- Device channel ID,
- Device location field,
- Device self-documentation string,
- Device configuration properties,
- Functional blocks.

## 2.1. Device Interface Overview

The *device interface* is the network-visible interface to a device. Following is a summary of each of the elements that comprise an interoperable device interface:

- *Neuron ID.* A 48-bit unique identifier for a LONWORKS device.
- *Standard program ID (SPID).* A number that uniquely identifies the device interface for a device.
- *Device channel ID.* A number that optionally specifies the channel to which the device is attached.
- *Device location field*. A string or number that optionally specifies the device location.
- *Device self-documentation string*. A string that specifies the functional blocks on a device.
- *Device configuration properties*. Configuration data used to configure the device. Functional blocks may also have configuration properties.
- *Functional blocks*. Logical components implemented on the device.

With the exception of the Neuron ID, a network tool can read all of these elements directly from a device over the network, or from the device interface (XIF) file for the device as described in 2.9, *Device Interface (XIF) File*. The benefit of making this information available directly from the device itself is that a network tool can read all of the information needed to integrate and manage the device over the network, and no accompanying manufacturer documentation is required. The benefit of making this information available from a device information file is that the device may be designed into a network before physical access to the device is available. The latter method is typically used for engineered systems, but the former method is sometimes used when a device interface file is not available.

The device interface elements are described in the remainder of this chapter. Device configuration properties are described in 2.7.3, *Configuration Properties*.

## 2.2. Neuron ID

The *Neuron ID* is a 48-bit number within the read-only data structure of a device as defined by the ANSI/EIA/CEA 709.1 protocol. It is also called the *unique node ID*. The Neuron ID is a unique number written to a Neuron Chip or Smart Transceiver during manufacture, or to other processors during development or manufacturing. Network tools use the Neuron ID to send network installation messages to a device, prior to the device being assigned a network address as described in 4.1, *Network Addressing*.

| Guideline 2.2: | A certified device shall implement a standard Neuron ID as defined in 2.2, *Neuron ID*. |
|---|---|

Manufacturers may wish to provide two copies of the Neuron ID in a human- or machine-readable format, attached to the product. One copy should be removable so that an installer may place it on a system drawing, or similar plan. This can even be done using a barcode for ease and accuracy of Neuron ID input into a network tool. An example Neuron ID barcode label is shown in the following figure.


**Figure 2. Example Neuron ID Barcode Sticker**

While the LONMARK Association has not standardized on a bar-coding method, the CODE-39 format has been used by several manufacturers for compatibility with many off-the-shelf barcode readers.

## 2.3. Standard Program ID

The *standard program ID (SPID)* is an 8-byte number within the read-only data structure of a device as defined by the ANSI/EIA/CEA 709.1 protocol. It uniquely identifies the device interface for a device. It is used by network tools to associate a device with a device interface definition. This speeds up the commissioning process by allowing a network tool to obtain the device interface definition without uploading the entire definition from every device.

| Guideline 2.3: | A certified device shall implement a standard program ID as defined in 2.3, *Standard Program ID*. |
|---|---|

The 16 hex digits of the SPID are organized as 6 fields that identify the format (F), manufacturer (M), device class (C), usage (U), channel type (T), and model number (N) of the device. These 6 fields are organized as follows, and are described in the following sections:

```
FM:MM:MM:CC:CC:UU:TT:NN
```

The manufacturer, classification, channel type, and optionally the usage fields contain standard values defined in the spidData.xml file available from the LONMARK Web site at www.lonmark.org/spid. The spidData.xml file is a downloadable, extensible markup language (XML) file for use with any development or network tool. The NodeBuilder Resource Editor (available to LONMARK members from the LONMARK Web site) and Echelon's NodeBuilder 3 Development Tool use this file to simplify the generation of a standard program ID. Both of these tools include a SPID Calculator that automatically builds a standard program ID based on your selections in fields that correspond to the following sections.

### 2.3.1. Format Field

The Format field contains a four-bit value defining the structure of the program ID and device self-documentation strings. The format must be 8 or 9, where format 8 is reserved for devices that have completed certification by the LONMARK Interoperability Association, and format 9 is used for all other devices. Format 9 must be used for devices that will not be certified, devices that will be certified but are still in development, and for devices that have not yet completed the certification process. Device formats 0 – 2 and 10 – 15 (0xA – 0xF) are reserved by Echelon for future use. Device formats 3 – 7 are used by network interfaces and legacy non-interoperable devices and must not be used for other interoperable devices.

### 2.3.2. Manufacturer Field

The Manufacturer field contains a 20-bit *manufacturer ID* (MID). The MID uniquely identifies the device manufacturer. The most significant bit (msb) of the MID identifies a *permanent MID (*msb clear) or a *temporary MID* (msb set) as follows:

- Permanent MIDs are assigned to Partner and Sponsor members of the LONMARK Interoperability Association upon request. The LONMARK Association publishes the permanent MIDs in the spidData.xml file so that the device manufacturer of a certified device is easily identified. Permanent MIDs are never reused or reassigned, but the manufacturer name may change if requested by the manufacturer (as in the case of the manufacturer being acquired by another company).
- Temporary MIDs are available to anyone upon completing a simple form at www.lonmark.org/mid. They are not guaranteed to be unique, and they are not listed in the spidData.xml file.

### 2.3.3. Device Class Field

The Device Class field is a two-byte value identifying the primary function of the device. This value is drawn from a registry of pre-defined Device Class definitions.

A device may implement multiple functional blocks. One of these functional blocks may be designated as the *primary functional block*, and the definition of this functional block is called the *primary functional profile*. If the primary functional profile number is greater than 99 and less than 20 000, the device class may be set to the profile number.

Standard functional profiles are also given device classes equal to their functional profile number. If you choose to use a device class that is assigned to a standard functional profile, then the device containing that device class must contain a functional block implementation of that profile.

If an appropriate device class value is not available, the LONMARK Association will assign one, if appropriate, upon request from a LONMARK member. Please send your request for a device class to the certification email address (cert@lonmark.org).

## 2.3.4. Usage Field

The Usage field is a one-byte value describing the intended usage of the device. The Usage field consists of a one-bit Changeable-Interface flag, a one-bit Functional Profile-Specific flag, and a 6-bit usage ID. These subfields are described in the following sections.

### 2.3.4.1. Changeable-Interface Flag

The *Changeable-Interface flag* is the msb of the Usage field. It must be set if the device uses changeable network variable types or dynamic network variables as described in 2.7.2.2 and D.3. The flag must be clear if the device uses a static device interface.

### 2.3.4.2. Functional Profile-Specific Flag

The *Functional Profile-Specific flag* is the second-msb of the Usage field. It must be set if the usage ID value is defined by the primary functional profile for the device. The flag must be clear if the usage ID value is defined by the standard usage ID values in the spidData.xml file or if the Device Class field does not identify the functional profile number of the primary functional profile for the device.

### 2.3.4.3. Usage ID

The *usage ID* is a 6-bit value in the least-significant portion of the Usage field that identifies the primary intended usage of the device. Based on the setting of the Functional Profile-Specific flag, the usage ID is defined by one of the following:

- If the Functional Profile-Specific flag is clear, the usage ID must be set to one of the standard usage ID values in the spidData.xml file.
- If the Functional Profile-Specific flag is set, the usage ID must be set to one of the usage ID values specified by the primary functional profile for the device, as determined by the Device Class field.

## 2.3.5. Channel Type Field

The Channel Type field is a one-byte value identifying the communications channel type supported by the device's LONWORKS network transceiver. The standard channel-type values are drawn from a registry of pre-defined channel-type definitions. This field must be set to the Custom channel type if the device does not use one of the standard channel types listed in the spidData.xml file (such a device cannot be certified). This file includes channel types that are approved for use in LONMARK devices, as well as channel types that have not been approved for use in LONMARK devices. To be certified, a device must be compatible with a channel type that has been approved for use in LONMARK devices.

### 2.3.6. Model Number Field

The Model Number field is a one-byte value identifying the specific product model of the device. Model numbers are assigned by the product manufacturer and must be unique within the device class, usage ID, and channel type for a manufacturer ID. The same hardware may be used for multiple model numbers depending on the program that is loaded into the hardware. The model number within the SPID does not have to conform to the manufacturer's marketing or engineering model numbers. It can be used as a decimal reference, hexadecimal reference, or any other method of convenience.

**EXAMPLES**

| | |
|---|---|
| ***Decimal Model Numbers*** | 0; 1; 2; 3;…9; 10; 11… = sequential by 1<br>10; 20; 30;…90; 100; 110… = incremental by 10s' place |
| ***Hexadecimal Model Numbers*** | 01; 02; 03;…09; 0A; 0B… = sequential by 1<br>10; 20; 30;…90; A0; B0… = incremental by nibbles' place |
| ***ASCII-Character Model Numbers*** | "A"; "B"; "C"; … = sequential by 1 using ASCII values for the representation of characters (0x41; 0x42; 0x43;…) |

## 2.4. Device Channel ID

The *device channel ID* is a 2-byte unsigned-long field within the configuration structure of a device as defined by the ANSI/EIA/CEA 709.1 protocol. Network tools may use the device channel ID to track the channel to which a device is attached. A value of zero indicates that the device's channel ID is unassigned. The device application must not require specific values in this field since a network tool may change the value as needed.

| |
|---|
| Guideline 2.4A: A certified device shall be manufactured with a zero channel ID. |

| Guideline 2.4B: | A certified device shall not modify its channel ID field. |
|---|---|

## 2.5. Device Location Field

The *device location field* is a 6-byte field within the configuration structure of a device as defined by the ANSI/EIA/CEA 709.1 protocol. Integrators, network tools, or the device itself may use this field to document the physical location of a device. This field may be read and written over the network using the Read Memory and Write Memory network-management messages defined by the ANSI/EIA/CEA 709.1 protocol. Some devices can determine their physical location by reading external physical inputs such as DIP switches, keyed connectors, or card-cage slot numbers. Such devices may use the device location field to communicate their physical location information to a network tool that can use this information to identify the physical location of the device.

Use of this field is optional, but if used must conform to the following guideline. A device may optionally implement a SCPTlocation configuration property (see 2.7.3, *Configuration Properties*) that can be used to provide a more complete location description than is possible in the 6-byte location field. The SCPTlocation value is a string of up to 31 characters. When used for device location, the SCPTlocation configuration property must apply to the Node Object functional block of the device if the device has a Node Object functional block, otherwise it must apply to the entire device. If a device has multiple locations, such as a device with multiple remote sensors, each of the functional blocks on the device may also implement a SCPTlocation configuration property to identify the location of each of the remote components. The SCPTlocation configuration property associate with the Node Object identifies the location of the device itself, whereas the other SCPT location configuration properties identify the locations of their respective hardware components. The SCPT Master List provides additional guidelines for use of the SCPTlocation configuration property.

| | |
|---|---|
| Guideline 2.5A: | A certified device's application program that wishes to communicate its physical location or ID assignment to a network tool can write this information into the location ID field of its configuration structure when the device is reset. If the most-significant bit of the first byte is one, the information is encoded as a 15-bit unsigned integer in the range 0 to 32,767, with the most-significant 7-bits in the lower 7-bits of the first byte (location[0]). If the most-significant bit of the first byte is zero, the information is encoded as a 0- to 6-character ASCII string. If the string is shorter than six characters, it must be nul-terminated (0x00). |

| | |
|---|---|
| Guideline 2.5B: | A certified device that implements a SCPTlocation configuration property to represent the device location shall apply the CP to the Node Object functional block if the device has a Node Object functional block, and shall otherwise apply the CP to the entire device. |

## 2.6. Device Self-Documentation String

The *device self-documentation string* is a string of up to 1024 bytes (subject to device memory limits) within the self-identification structure of a device as defined by the ANSI/EIA/CEA 709.1 protocol. This string specifies the self-documentation string structure, the functional blocks, and optionally describes the function of a device. The Neuron C Version 2 Compiler automatically creates this string. Developers using other tools can manually create the device self-documentation string as described in Appendix C.

| | |
|---|---|
| Guideline 2.6A: | A certified device shall contain a device self-documentation string that specifies the self-documentation string structure and the functional profiles implemented by each functional block on the device as described in 2.6, *Device Self-Documentation String*. |

| | |
|---|---|
| Guideline 2.6B: | A certified device shall store the device self-documentation string in the application image as described in the ANSI/EIA/CEA 709.1 protocol. |

## 2.7. Functional Blocks

A device application is divided into one or more *functional blocks*. A functional block is a portion of a device's application that performs a task by receiving configuration and operational data inputs, processing the data, and sending operational data outputs. A functional block may receive inputs from the network, hardware attached to the device, or from other functional blocks on a device. A functional block may send outputs to the network, to hardware attached to the device, or to other functional blocks on the device.

The device application must implement a functional block for each function on the device to which other devices should communicate, or that requires configuration for particular application behavior. Each functional block must be defined by a *functional profile* as described in Chapter 3, *Resource Files*. Functional profiles are templates for functional blocks, and each functional block is an implementation of a functional profile.

The network inputs and outputs of a functional block, if any, are provided by *network variables* and *configuration properties*. A network variable is an operational data input or output for a functional block. A configuration property is a data value used for configuring the behavior of a network variable, functional block, or the entire device. Configuration properties used to configure an entire device are not part of any functional block—they are instead associated with the device itself.

A special type of functional block is called the *Node Object* functional block. Network tools use the Node Object functional block to test and manage the other functional blocks on a device. The Node Object functional block is also used to report alarms generated by the device. In the case of a device with only a single functional block, other mechanisms may be available for the test and management functions. In such a case, the Node Object functional block may be omitted, provided that both of the following conditions apply:

- The application program does not need to continue operating when the functional block is disabled. In this case, setting the device off-line will disable the functional block.
- The device does not implement alarms, self-test, range checking, fault detection, file transfer, or other functions belonging to the Node Object functional block.

| Guideline 2.7: | A certified device that supports more than one functional block shall include a Node Object functional block to allow monitoring and control of the functional blocks within the device. A certified device created with a single functional block shall also include a Node Object functional block if the device implements alarms, self-test, range checking, fault detection, file transfer, or other functions belonging to the Node Object functional block; or if the application program must continue to operate when the functional block is disabled. |
|---|---|

Figure 3 illustrates the relationship between the Node Object functional block, other functional blocks on a device, network variables, and configuration properties. Sections 2.7.2 and 2.7.3 describe network variables and configuration properties.

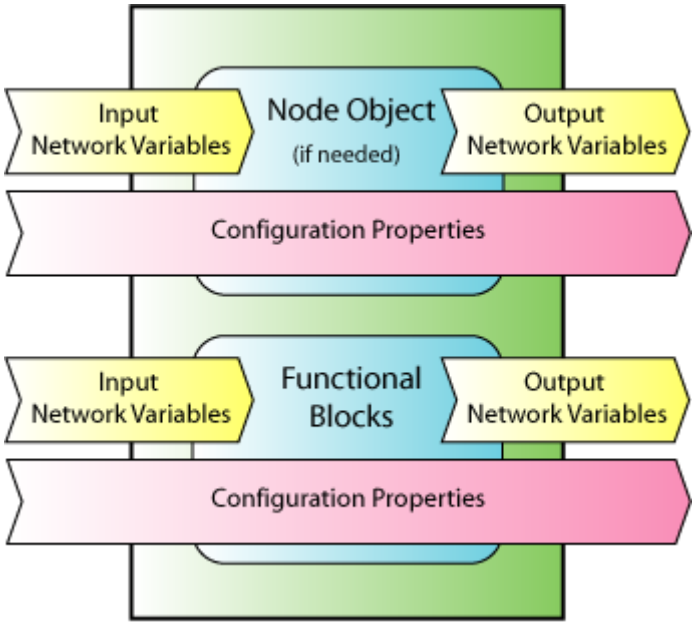


**Figure 3. Functional Block Interfaces**

## 2.7.1. Implementing a Functional Block

The device self-documentation string contains a list of the functional blocks on a device. This list identifies the functional profile number implemented by each functional block, and assigns a unique functional block index number (also called the *global index*) to each functional block on the device, starting with zero. Each network variable and configuration property on a device includes self-documentation or configuration data that associate the network variable or configuration property with a functional block on the device using the functional block index number. The

size of this self-documentation and configuration data is also subject to device memory limits, which is an especially important consideration for devices based on the Neuron 3120 Chip.

To implement a functional block on a device, you can create the self-documentation and configuration data manually, as described in Appendix C. Alternatively, if you are developing an application using the Neuron C Version 2, or newer, programming language, you can implement a functional block as described in this section.

To create a functional block, use the Neuron C **fblock** construct to declare the functional block and to identify the network variable and configuration property members of the functional block. The relationship between the **fblock** keyword and the network variable and configuration property members of the functional block is shown in the following figure.

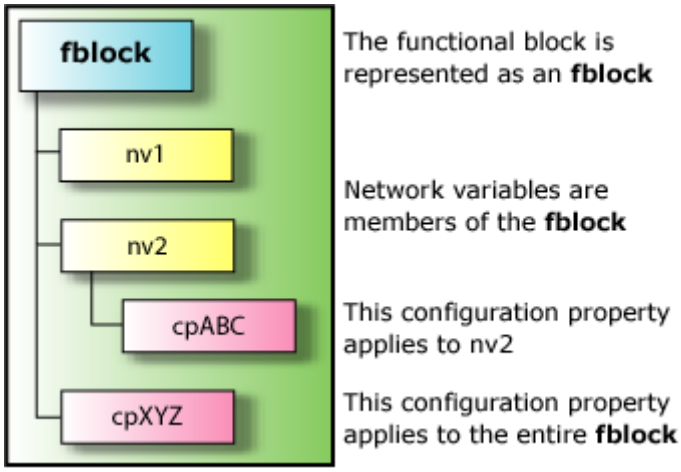


**Figure 4. NV/CP Relation to Functional Block**

The **fblock** declaration references a functional profile name as described in Chapter 3, *Resource Files*.

The syntax for declaring a functional block is the following (see the *Neuron C Reference Guide* for a complete description of the syntax):

**fblock** *fp-identifier* **{** *fblock-body* **}** *identifier* [**[***array-bound***]**] [*ext-name*] [*fb-prop-list*] **;**

The following code fragment shows an example **fblock** declaration that creates a functional block with a programmatic name of fbTempSensor that is based on the SFPThvacTempSensor functional profile.

```
fblock SFPThvacTempSensor {
      …
} fbTempSensor;
```

Sections 2.7.2 and 2.7.3 describe how to implement the network variable and configuration property members of the functional block.

## 2.7.2. Network Variables

The ANSI/EIA/CEA 709.1 protocol employs a data-oriented application layer that supports the sharing of data between devices, rather than simply the sending of commands between devices. In this approach, application data such as temperatures, pressures, states, and text strings can be sent to multiple devices—each of which may have a different application for each type of data.

Applications exchange data with other LONWORKS devices using *network variables*. Every network variable has a *direction*, *type*, and *length*. The network variable direction can be either input or output, depending on whether the network variable is used to receive or send data. The network variable type determines the format of the data. The standard resource file set described in Chapter 3, *Resource Files*, defines a set of standard types for network variables; these are called *standard network variable types* (SNVTs). Device manufacturers may also create custom *network variable* types as described in Chapter 3. These are called *user network variable types* (UNVTs).

Network variables of identical type and length but opposite directions can be connected to allow the devices to share information. For example, an application on a lighting device could have an input network variable that was of the switch type, while an application on a dimmer-switch device could have an output network variable of the same type. A network tool could be used to connect these two devices, allowing the dimmer switch to control the lighting device, as shown in Figure 5.


**Figure 5. Network Variable Point-to-Point Connection**

The direction indicated by the triangle in the above figure indicates the direction of the network variable. A single network variable may be connected to multiple network variables of the same type but opposite direction. A single *network variable* output connected to multiple inputs is called a *fan-out connection*. A single *network variable* input that receives inputs from multiple *network variable* outputs is called a *fan-in connection*. Figure 6 shows the same dimmer switch being used to control three lights using a fan-out connection:

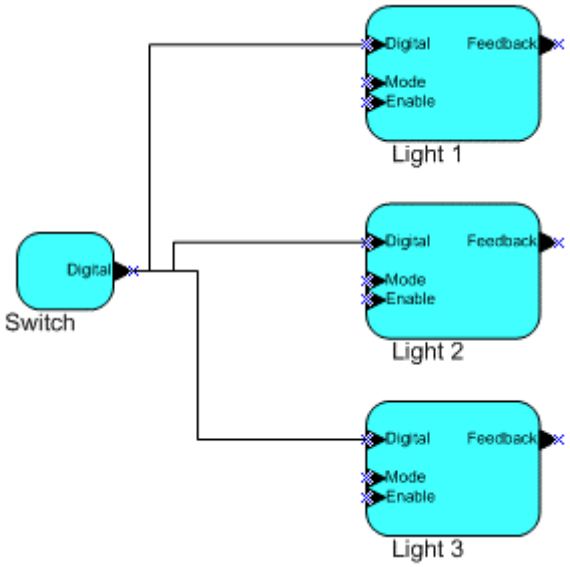


**Figure 6: Network Variable Fan-out Connection**

The application program in a device does not need to know where input network variable values come from or where output network variable values go. When the application program has a changed value for an output network variable, it simply passes the new value to the device firmware. Through a process called *binding* that takes place during network design and installation, the device firmware is configured to know the logical address of the other device or group of devices in the network expecting that network variable's values. It assembles and sends the appropriate packets to these devices. Similarly, when the device firmware receives an updated value for an input network variable required by its application program, it passes the data to the application program. The binding process thus creates logical *connections* between an output network variable in one device and an input network variable in another device or group of devices. Connections may be thought of as "virtual wires." For example, the dimmer-switch device in the dimmer-switch-light example could be replaced with an occupancy sensor, without making any changes to the lighting device.

### 2.7.2.1. Implementing a Network Variable

Each network variable that is part of the interoperable interface for a device must be associated with a functional block. A single network variable can only be associated with a single functional block. To implement a network variable on a device and associate it with a functional block, you can create the self-documentation data manually as described in Appendix C. Alternatively, if you are developing an application using the Neuron C Version 2 programming language, you can implement a network variable as described in this section. The Neuron C Version 2 compiler automatically generates self-documentation strings as described in Appendix C. Using either approach, the self-documentation strings must be stored

in non-volatile memory of the device to ensure that they are available after a power cycle. In addition, the self-identification information for changeable-type network variables must be stored in writeable non-volatile memory so that a network tool can modify the SNVT index stored within the device when the type is changed.

> Guideline 2.7.2.1A: A certified device shall include network variable self-documentation strings that map network variables to the functional blocks declared on the device. These strings may be created by the Neuron C Version 2 (or newer) compiler as described in 2.7.2.1, *Changeable-Type Network Variables*, or may be created manually as described in Appendix C.

> Guideline 2.7.2.1B: The network variable and configuration property self-documentation strings shall be stored in non-volatile memory of a certified device.

To implement a network variable using Neuron C Version 2, you must first declare the network variable, and then identify the network variable as a member of a functional block. The syntax for declaring a network variable is shown below (see the *Neuron C Reference Guide* for a complete description of the syntax).

**network input** | **output** [*netvar-modifier*] [*class*] *type*
[*connection-info*] *identifier* [**[***array-bound***]**]
[ = *initializer-list*] [*nv-property-list*] **;**

You can declare an array of network variables using the optional *array-bound* enclosed in square brackets ("**[**" and "**]**"). You can declare a changeable-type network variable using the **changeable_type** *netvar-modifier*.

To identify a network variable as a member of a functional block, include an *nv-reference* **implements** *member-name* clause in the **fblock** declaration. The *nv-reference* is the name supplied as the *identifier* in the network variable declaration. The *member-name* is the name of the network variable member of the functional profile as defined in the functional profile.

Figure 7 illustrates two example **implements** clauses. The first associates the nvoTemperature network variable with the nvoHVACTemp member of the SFPThvacTempSensor functional profile. The second associates the nvoTempFloat network variable with the nvoFloatTemp member of the functional profile.

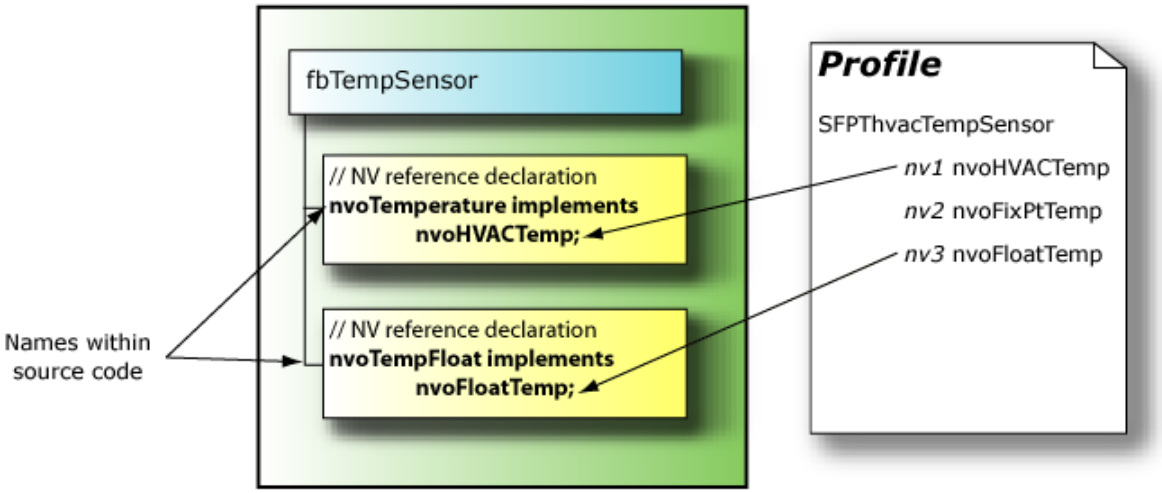


**Figure 7. fblock Network Variable Member References**

The following Neuron C Version 2 code implements the declarations in Figure 7:

```
// Network variable declarations
SNVT_temp_p output nvoTemperature;
SNVT_temp_f output nvoTempFloat;

// fblock declaration
fblock SFPThvacTempSensor {
      nvoTemperature implements nvoHVACTemp;
      nvoTempFloat implements nvoFloatTemp;
} fbTempSensor;
```

### 2.7.2.2. Changeable-Type Network Variables

Network variables may be of *changeable type*. Such network variables are used when the device developer cannot know the correct type of the network variable in advance. For example, a changeable-type output would be used for a generic sensor device that can attach to any standard sensor and report any sensed value. The actual type of the network variable can be changed to meet the physical units measured, however, the developer must still declare an initial type for the network variable.

If a device supports any changeable-type network variables, it must set the Changeable-Interface flag in the program ID as described in 2.3.4.1, *Changeable-Interface Flag*. It must also declare a **SCPTnvType** configuration property that applies to the changeable-type network variable. Network tools use this configuration property to notify the device application of changes to the network variable type. The device application will require notification of changes to this

configuration property. Notification can be provided using one of the methods described in the following procedure.

> Guideline 2.7.2.2: A changeable-type network variable on a certified device shall implement a SCPTnvType configuration property that applies to the changeable-type network variable.

A configuration property may inherit its type from a changeable-type network variable. If the configuration property applies to multiple changeable-type network variables, all of the network variables must share the same SCPTnvType configuration property—and the same SCPTmaxNVLength configuration property if implemented.

To create a changeable-type network variable using Neuron C Version 2, follow these steps:

1. Declare the network variable with the **changeable_type** keyword. This keyword results in information being provided in the device interface description. This information tells that the variable's implementation permits the type of the network variable to be changed by a network tool. You must declare an initial type for the network variable, and the size of the initial type must be equal to the largest network variable size that your application supports.

   For example, the following declaration declares a changeable-type output network variable, with an initial type of **SNVT_volt_f**. This type is a 4-byte floating-point value, so this network variable can support changes to any network variable type of 4 or less bytes.

   ```
   network output changeable_type SNVT_volt_f  . . .
   ```

2. Set the changeable-interface bit in the program ID.

3. Declare a **SCPTnvType** configuration property that applies to the changeable-type network variable.

   Your application will require notification of changes to this configuration property. You can provide this notification by declaring the configuration property with the **reset_required** modifier and checking the **SCPTnvType** value in the reset director function, implementing configuration property access via FTP and checking in the **stop_transfer()** function whether the **SCPTnvType** value has been modified, and/or by implementing the **SCPTnvType** configuration property as a configuration network variable and checking the current type in the task for the **nv_update_occurs(***nv-name***)** event.

   For example, the following code declares a changeable-type output network variable with its **SCPTnvType** configuration property:

```
SCPTnvType cp_family cp_info(reset_required) nvType;
network output changeable_type SNVT_volt_f nvo1
      nv_properties {nvType};
```

4 You can optionally declare a **SCPTmaxNVLength** configuration property that applies to the changeable-type network variable. This configuration property can be used to inform network tools of the maximum type length supported by the changeable-type network variable. This value is a constant, so declare this configuration property with the **const** modifier. For example, the following code adds a **SCPTmaxNVLength** configuration property to the example in the previous step:

```
SCPTnvType cp_family cp_info(reset_required) nvType;
const SCPTmaxNVLength cp_family nvMaxLength;
network output changeable_type SNVT_volt_f nvo1
      nv_properties {nvType,
                     nvMaxLength=sizeof(SNVT_volt_f)};
```

5 Implement code in your Neuron C application to process changes to the **SCPTnvType** value. The required code is described in the *Neuron C Programmer's Guide*.

### 2.7.2.3. Network Variable Naming Conventions

The programmatic name of a network variable may be prefixed with its storage class, as defined below. For compactness, underscores are typically not used and all characters are typically lowercase, except the first character of a word. The following conventions are used, but not required:

- network variable input: nviXxxxxxxxxxxx
- network variable output: nvoXxxxxxxxxxxx
- network variable output (ROM): nroXxxxxxxxxxxx
- configuration network variable input: nciXxxxxxxxxxxx

Due to the limitation of 16 characters for names of the network variables and configuration properties, there is a convention for abbreviations. The following list represents some typical abbreviations, but it is not meant to be all-inclusive:

| | |
|---|---|
| Actual | Act |
| Calendar | Cal |
| Clear | Clr |
| Continuous | Cont |
| Delay | Dly |
| Device | Dev |
| Discrete | Disc |
| Electric | Elec |
| Feedback | Fb |
| Floating-point | f |
| Frequency | Freq |
| Hardware | Hw |
| Increment | Inc |

| | |
|---|---|
| Inhibit | Inh |
| Input | In |
| Level | Lev |
| Maximum | Max |
| Micrometer | Micr |
| Minimum | Min |
| Parts-per-million | Ppm |
| Object | Obj |
| Output | Out |
| Position | Pos |
| Range | Rnge |
| Request | Req |
| Rate | Rt |
| Resistance | Res |
| Source | Src |
| Standby | Stby |
| String | Str |
| Table | Tbl |
| Time | T |
| Translation | Trans |
| Volume | Vol |
| Watt-hour | Whr |

## 2.7.3. Configuration Properties

A *configuration property (CP)* is a data item that, like a network variable, is part of the device interface for a device. Configuration properties characterize the behavior of a device in the system. Network tools manage this attribute and keep a copy of its value in a database to support maintenance operations. If a device fails and needs to be replaced, the configuration property data stored in the database is downloaded into the replacement device to restore the behavior of the replaced device in the system.

Configuration properties facilitate interoperable installation and configuration tools by providing a well-defined and standardized interface for configuration data. Each configuration property type is defined in a resource file that specifies the data encoding, scaling, units, default value, range, and behavior for configuration properties based on the type. A rich variety of *standard configuration property types (SCPTs)* are defined in the standard resource file set described in Chapter 3, *Resource Files*. SCPTs provide standard type definitions for commonly used configuration properties such as dead-bands, hysteresis thresholds, and message heartbeat rates. You can also create your own *user configuration property types (UCPTs)* that are defined in resource files you create.

A configuration property must apply to one of the following items:

- A single network variable,
- A single functional block,
- A series of network variables,

- A series of functional blocks,
- A compilation of network variables,
- A compilation of functional blocks,
- The entire device.

The item to which the configuration property applies is known as the *application set* of the configuration property. The application set cannot contain more than one of the items in the list above.

Each configuration property within its specified application set must be based on a unique configuration property type (SCPT or UCPT). For example, if a functional block has three output network variables, each requiring an independent maximum send-time configuration property, then the maximum send-time properties must be declared to apply to the network variables within the functional block, as shown in Example B of Figure 8. In Example A, declaring them to apply to the functional block would be ambiguous because it would be impossible for a network tool to know in an interoperable way which maximum send-time value controlled which network variable within the functional block. As described in the next section, configuration properties with compatible application sets may be shared, so a single SCPTmaxSendTime configuration property may be shared among nv1, nv2, and nv3 in Example B.

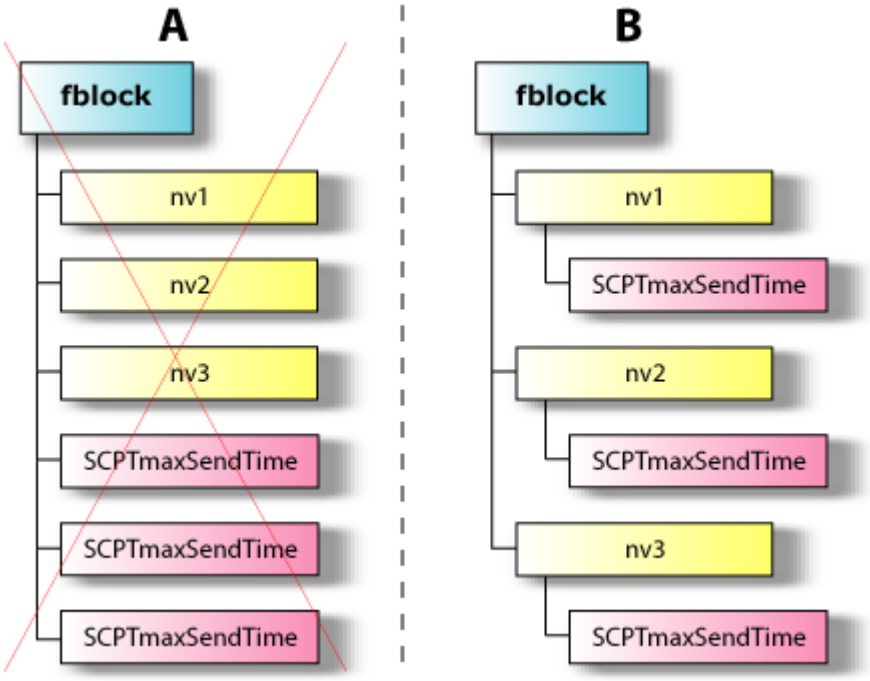


**Figure 8: CP Application to NVs vs. FBs**

Manufacturer-defined user configuration properties are permitted because the configuration data for a given functional block is often implementation-specific. It is

also permitted that modification of these user configuration data be necessary for the successful commissioning of the device. Any user configuration properties necessary to successfully commission the device must either be defined with documentation and machine-readable LONMARK resource files as described in Chapter 3, or must be modifiable with a passive configuration tool as defined in 4.2, *Passive Configuration Tools*.

Configuration properties that are members of functional blocks may be implemented as arrays. For example, a configuration property array may be used to create an event schedule or a translation table for the linearization of sensor data. The functional profile contains information for each member configuration property, whether implementation as an array is permitted, required, or disallowed. The functional profile also defines the valid array boundaries for each member configuration property that permits array implementation.

Guideline 2.7.3A: If user configuration properties must be modified for successful commissioning of a certified device, those configuration properties must be defined with LONMARK resource files and user documentation as described in Chapter 3, or a passive configuration tool must be provided as described in 4.2, *Passive Configuration Tools*. The documentation and resource files, if any, must be supplied at the time the device is certified. The passive configuration tool, if any, must be available at the time the device is certified.

Guideline 2.7.3B: If a passive configuration tool is not required for the successful commissioning of a certified device, then any and all configuration information required to be modified to successfully commission the device shall be implemented and exposed via LONMARK configuration properties with user documentation.

### 2.7.3.1. Configuration Property Distribution Methods

When a configuration property or configuration property array applies to multiple functional blocks or multiple network variables, there are two distribution methods that determine how the configuration property or elements of the array are distributed among the applicable functional blocks or network variables.

The first method is called *CP sharing*. Using this method, the entire configuration property or CP array applies to all the specified functional blocks or network

variables. For example, given a configuration property array, cpMyCps[4], here is the disbursement of the entire array to four functional blocks using CP array sharing:

cpMyCps[0] → functional block 0
cpMyCps[1] → functional block 0
cpMyCps[2] → functional block 0
cpMyCps[3] → functional block 0

cpMyCps[0] → functional block 1
cpMyCps[1] → functional block 1
cpMyCps[2] → functional block 1
cpMyCps[3] → functional block 1

cpMyCps[0] → functional block 2
cpMyCps[1] → functional block 2
cpMyCps[2] → functional block 2
cpMyCps[3] → functional block 2

cpMyCps[0] → functional block 3
cpMyCps[1] → functional block 3
cpMyCps[2] → functional block 3
cpMyCps[3] → functional block 3

The second method is called *CP dividing*. CP dividing only applies to CP arrays; a singular CP is atomic and may not be divided. Using this method, each element of a CP array is applied to a corresponding element of the applicable functional blocks or network variables. For example, given a configuration property array, cpMyCps[4], here is the disbursement of the individual elements of the array to four functional blocks using CP array dividing:

cpMyCps[0] → functional block 0

cpMyCps[1] → functional block 1

cpMyCps[2] → functional block 2

cpMyCps[3] → functional block 3

This avoids having a separate self-documentation string for every element in the array. This technique can also result in substantial memory and code-space savings, though commissioning time is typically increased due to a potentially less efficient memory layout. CP array dividing cannot be used unless the CP array has exactly the same number of elements as the number of functional blocks or network variables to which the CP applies.

CP sharing can be used with both a series of network variables or functional blocks and a compilation of network variables or functional blocks. CP dividing cannot be used with compilations.

The implementation of multiple members of a functional profile may share the same configuration property unless the functional profile documentation specifies that

they cannot be shared. For example, a profile may define nvo1, nvo2, and nvo3 outputs, each with a mandatory SCPTmaxSendTime CP member that applies to it called cpMaxSendTime1, cpMaxSendTime2, and cpMaxSendTime3. A functional block implementation of this profile may implement a single SCPTmaxSendTime CP that is shared among nvo1, nvo2, and nvo3, and still fulfill the requirement to implement all mandatory members of the profile.

### 2.7.3.2. Configuration Property Implementation Methods

You can implement a configuration property using one of two different methods. The first, called a *configuration network variable*, uses a network variable to implement the configuration property. This has the advantage of enabling the configuration property to be modified by another LONWORKS device, just like any other network variable. It also has the advantage of having the network variable event mechanism available to provide notification of configuration property updates to the application.

The disadvantages of configuration network variables are that they are limited to a maximum of 31 bytes each, and a Neuron Chip or Smart Transceiver hosted device is limited to a maximum of 62 network variables.

The second method of implementing configuration properties uses *configuration files* to implement the configuration properties for a device. Rather than being separate externally-exposed elements of data, all configuration properties implemented within configuration files are combined into one or two blocks of data called *value files*. A value file consists of configuration property records of varying length concatenated together. Each value file must fit as contiguous bytes into the memory space of the device that is accessible by the application. When there are two value files, one contains writeable configuration properties and the second contains read-only data. To permit a network tool to access individual elements of data in the value file, there is also a *template file* consisting of an array of text characters that describes the elements in the value files.

The advantages of implementing configuration properties as configuration files are that there are no limits on configuration property size or the number of configuration properties—except as constrained by the available memory space on the device and the maximum file size for the LONWORKS File Transfer Protocol (FTP). The disadvantages are that other devices cannot connect to or poll a configuration property implemented as a configuration file—typically requiring a network tool to modify a configuration property implemented within a configuration file—and no events are automatically generated when a configuration property implemented within a configuration file is updated. The application can force notification of updates by requiring network tools to reset the device, disable the functional block, or take the device offline when a configuration property is updated. Alternatively, the application can also force notification by implementing

configuration-file access via the LONWORKS FTP and monitoring the transfer. This option requires additional code space for the FTP-server code.

Table 1 summarizes the advantages and disadvantages of the two implementation methods. For a given configuration property, the developer must choose one of these methods. However, both methods can be implemented on a device, just not for the same configuration property.

**Table 1. CP Implementation Trade-offs**

| *Configuration NV* | *CP Within a Configuration File* |
|---|---|
| ▪ Limited to 31 bytes in length | ▪ May be of any length |
| ▪ Uses one network variable for each configuration property declared | ▪ Does not require a network variable |
| ▪ Application notified of updates via NV update event | ▪ Requires alternate update notification method |

Guideline 2.7.3.2: A certified device shall implement a given configuration property as either a configuration network variable or as an element of a configuration file.

## 2.7.3.3. Configuration-File Access Methods

When configuration properties are implemented within configuration files, you must provide a method for a network tool to access the configuration file. You may provide one of the following three access methods:

- Direct memory read/write,
- FTP with random and sequential access,
- FTP with sequential access.

The *direct memory read/write* access method enables a network tool to read and write configuration files using ANSI/EIA/CEA 709.1 Read Memory and Write Memory network-management messages. This method does not require file transfer code on the device, and is therefore the preferred method for applications running on Neuron Chips or Smart Transceivers—as long as the configuration file fits entirely within the standard memory space of the Neuron Chip or Smart Transceiver.

The *File Transfer Protocol (FTP) with random and sequential access method* enables a network tool to read and write configuration files using the LONWORKS File Transfer Protocol, with both random and sequential access to any blocks within the

configuration file. This method requires file transfer code on the device, but can be used with any host processor and can also be used for Neuron Chip or Smart Transceiver hosted devices when the configuration file does not fit in the processor's standard memory space.

The *File Transfer Protocol (FTP) with sequential access method* is identical to the FTP with random and sequential access method, with the exception that random access to blocks within the configuration file is not provided. This method is used when an FTP access method is required or preferred, and the host processor does not have sufficient application memory space to implement random access. It can be considerably more inefficient than the other two access methods and should only be used if none of the other two access methods can be provided.

Network tools determine which access method to use based on interfaces provided in the Node Object functional block. See the *Node Object Functional Profile* for details on these interfaces. See Echelon's *LONWORKS File Transfer Protocol Engineering Bulletin* (*005-0025-01*) for details on implementation of the File Transfer Protocol.

> Guideline 2.7.3.3: If a certified device implements any configuration properties within configuration files, then one and only one of the following access methods shall be provided as described in 2.7.3.3, *Configuration-File Access Methods*: direct memory read/write, FTP with random and sequential access, or FTP with sequential access.

### 2.7.3.4. Configuration Property Flags

Each configuration property on a device may specify optional flags that are used to notify a network tool of whether or not a configuration property can be modified, and if so, when it can be modified. These flags are optional. If a configuration property is declared without any flags, a network tool may assume that the configuration property can be modified at any time.

| | |
|---|---|
| *Constant* | Specifies a configuration property that can never be changed by a network tool. However, network tools may write such configuration properties when they reside in a writeable value file as long as the value is not changed. A network tool may do this as part of an update to another configuration property adjacent to the constant value. Configuration properties with the Constant flag but without the Device-Specific flag can be assumed to have the same value on all devices using the same standard program ID. |

| | |
|---|---|
| *Device-Offline* | Specifies that a network tool must take this device offline, or ensure the device is already offline, before modifying the configuration property. This flag or the FB-Disabled flag is recommended for a configuration property implemented within a configuration file with direct memory read/write access if the application requires update notification, or if the application cannot tolerate updates from the network at the same time the application is reading the configuration property.<br><br>This flag should not be used for configuration properties implemented within configuration files that are accessed via FTP, and network tools should ignore this flag for such configuration properties. This is because a device cannot transfer configuration property values via FTP while offline. In fact, an offline application must be placed into the online state for the duration of any FTP configuration property operations. |
| *Device-Specific* | Specifies a configuration property that must always be read from the device instead of relying upon the value in the device interface file or a value stored in a network database. Network tools must never change this property's value except as a side effect of a new program download. This is used for configuration properties that must be exclusively managed by the device, such as a setpoint that is updated by a local operator interface on the device. This flag is not used for configuration properties that are updated both by the device application as well as by network tools. Such configuration properties must provide an alternate means to determine which copy of the configuration property (the device copy or the network database copy) is current, or an alternate means to keep both copies synchronized. |
| *FB-Disabled* | Specifies that a network tool must disable the functional block containing the configuration property, take the device offline, or ensure that the functional block is already disabled or the device is already offline, before modifying the configuration property. This flag or the Device-Offline flag is recommended for a configuration property implemented within a configuration file with direct memory read/write access if the application requires update notification, or if the application cannot tolerate updates from the network at the same time the application |

| | |
|---|---|
| | is reading the configuration property.<br><br>A network tool may elect not to disable a functional block before modifying a configuration property with the FB-Disabled flag if that device is already offline and can be updated while offline. This is allowed because an offline device has all its functional blocks implicitly disabled, and because a functional block cannot be directly disabled when the device is already offline. |
| *Manufacturing-Only* | Specifies a factory setting that may be read or written when the device is manufactured, but is not normally (or ever) modified in the field. In this way, a standard installation tool may be used during manufacture to calibrate a device, while a field installation tool would observe the flag in the field and prevent modification of the value, or optionally require a password to modify the value. |
| *Reset-Required* | Specifies that a network tool must reset the device after changing the value of the configuration property. If multiple modifications of configuration properties are being made at the same time, then one reset can be completed in lieu of having to reset the device the same number of times as Reset-flagged configuration properties were modified. |

### 2.7.3.5. Implementing a Configuration Property

To implement a configuration property on a device and associate it with a network variable, a functional block, or the device itself, you can create the self-documentation data or configuration files manually as described in Appendix C. Alternatively, if you are developing an application using the Neuron C Version 2 programming language, you can implement a configuration property as described in this section.

> Guideline 2.7.3.5: A certified device that includes configuration properties within the interoperable interface shall include configuration property documentation strings that declare the configuration properties and map them to the entire device; one or more network variables; or, one or more functional blocks declared on the device. These strings may be created by the Neuron C Version 2 (or newer) compiler as described in 2.7.3.5, *Implementing a Configuration Property*, or may be created manually as described in Appendix C.

To implement a configuration property using Neuron C Version 2, you must first declare the configuration property, and then associate the configuration property with a network variable, functional block, or the device.

The syntax for declaring a configuration network variable is identical to the syntax for declaring an input network variable, with the addition of a **config_prop** keyword. The **config_prop** keyword may be abbreviated to **cp**. The complete syntax is shown below (see the *Neuron C Reference Guide* for a complete description of the syntax).

**network input** [*netvar-modifier*] [*class*] *type* **config_prop** [*cp-modifiers*] [*connection-info*] *identifier* [ **[***array-bound***]** ] [= *initializer-list*] **;**

The syntax for declaring a configuration property to be implemented within a configuration file is shown below (see the *Neuron C Reference Guide* for a complete description of the syntax).

[**const**] *type* **cp_family** [*cp-modifiers*] *identifier* **[=** *initial-value*] **;**

**EXAMPLE**

```
SCPTgain cp_family cpGain = { 2, 3 };
```

The *cp-modifiers* specify optional configuration property flags described in Section 2.7.3.3 and range modifiers specified in Section 2.7.3.6. If included, the syntax for the configuration property flags is as follows:

**cp_info(***cp-option-list***)**

The *cp-option-list* is a comma-separated list of keywords for each specified configuration property flag as follows:

| | |
|---|---|
| *Constant* | **const** (this is not a cp-modifier, it is a class modifier) |
| *Device-Offline* | **offline** |
| *Device-Specific* | **device_specific** |
| *FB-Disabled* | **object_disabled** |
| *Manufacturing-Only* | **manufacturing_only** |
| *Reset-Required* | **reset_required** |

The **cp_family** declaration does not create an instance of the configuration property whereas the configuration network variable declaration does. Instances of the configuration property declared by the **cp_family** declaration are created when the CP family is associated with a network variable, functional block, or the device.

When declaring either a configuration network variable or a CP family, the type must be a configuration property type defined within a LONMARK resource file as described in Chapter 3, *Resource Files*.

To associate a configuration property with a network variable, functional block, or the device, include the configuration property name in a *property list* for the network variable, functional block, or device. A property list is a list of configuration properties associated with a network variable, functional block, or device. The syntax for a property list is one of the following:

**nv_properties** {*property-reference-list*}

**fb_properties** {*property-reference-list*}

**device_properties** {*property-reference-list*} **;**

To associate a configuration property with a network variable, the **nv_properties** clause must immediately follow the network variable declaration, preceding the closing semicolon. To associate a configuration property with a functional block, the **fb_properties** clause must immediately follow the **fblock** declaration, preceding the closing semicolon. The **device_properties** statement is a file-scope statement that may appear anywhere within a Neuron C Version 2 application. A Neuron C program may have multiple device-property lists. However, you cannot have more than one configuration property of any given type that applies to the device. Likewise, you cannot have more than one configuration property of any given type within the same **fb_properties** or **nv_properties** clause.

The *property-reference-list* contains a list of *property references*, separated by commas. Each property reference must be the name of a previously declared CP family or the name of a previously declared configuration network variable, using the name supplied as the *identifier* in the configuration property declaration. If the configuration network variable or CP family implements an array that is referenced in a **device_properties** clause or in an **nv_properties** clause that defines properties of a network variable that is not a member of a functional block, only a single array element may be chosen. An array index must be given as part of the property reference in that case. For configuration properties that implement arrays and that are referred to in an **fb_properties** clause, or that are referred to in an **nv_properties** clause that itself defines details of a network variable that implements a member of a functional block, CP sharing or CP dividing may be used as discussed earlier in this document. In this case, the CP array may also be referenced without the array index, allowing the entire CP array to apply to its application set as a whole.

Following the *property-identifier*, there may be an optional *initializer*, and an optional *range modifier*. These optional elements may occur in either order if both are given. These elements are described in the next section.

### 2.7.3.6. Configuration Property Initializers and Range Modifiers

If present, the property-list initializer for a CP-family member specifies the initial value for the CP-family instance created by the property list. The initializer overrides any initializer provided at the time of declaration of the CP family; thus, using this mechanism, some CP-family members can be initialized specially, with the remaining CP-family members having a more generic initial value.

If present, the range modifier allows you to specify a range-modification string that modifies the valid range for the configuration property defined by the resource file as described in Chapter 3. The range-modification string can only be used with fixed-point and floating-point types, and consists of a pair of either fixed-point or floating-point numbers delimited by a colon (“**:**”). The first number is the low limit while the second number is the high limit. If either the high limit or the low limit should be the maximum or minimum specified in the configuration property definition (from the functional profile) or specified in the configuration property type definition (from the resource file set outside of the functional profile), then the field is empty to specify this. In the case of a structure or an array, if one member of the structure or array has a range modification, then all members must have a range modification specified. In this case, each range modification pair is delimited by a vertical bar (“**|**”). To specify no range modification for a member of a structure (that is, revert to the default for that member), encode the field as a single terminating vertical bar with no other characters. Use the same encoding for structure members that cannot have their ranges modified due to their data type. Empty encoding is only allowed for members of structures. Whenever a member of a structure is not a fixed or floating-point number, its range may not be restricted. Instead, the default ranges must be used. In the case of an array, the specified range modifications apply to all elements of the array.

For example, to specify a range modification for a 3-member structure where the second member has the default ranges, and the third member only has an upper-limit modification, the range-modification string is encoded as: "**n:m||:m**". Positive values for range modifications and their exponents (if any) are implicit, while negative numbers and negative exponents must be explicitly designated as such with a preceding hyphen (“**–**”). Floating-point numbers use a period (“**.**”) for the decimal point. Fixed-point numbers must be expressed as signed 32-bit integers. Floating-point numbers must be within the range of an IEEE 754 32-bit floating-point number. To express an exponent, precede the exponent by an “**e”** or an “**E”** and then follow with an integer value. For example, to represent the lowest floating-point value possible, the encoding would be the following:

```
-3.40282E38
```

Both initial values and range modifiers may be used. Development and network tools must apply them in the following order:

1. Values provided with the CP reference within an **fb_properties**, **nv_properties**, or **device_properties** clause, if any.
2. Values provided with the declaration of the CP family, if any.
3. Values provided with the functional profile template definition in the resource files, if given.
4. Values provided with the configuration property or network variable type that is used with the functional profile, if given.

5 Initial values are considered to be zero unless defined elsewhere (in any of the places listed in this list above). Range modifiers default to the natural minimum and maximum value for the underlying base data type, unless defined in any of the items listed above.

### 2.7.3.7. Configuration Property Examples

Figure 9 demonstrates example configuration property declarations for configuration properties that apply to network variables, and that apply to entire functional blocks. The **nv_properties** and **fb_properties** Neuron C Version 2 programming keywords are used to make the associations, respectively. The solid arrows relate the functional profile network variable member definitions to the network variable references in the source code. The dashed arrows related the functional profile configuration property member definitions to the configuration property declarations in the source code. The nvoFixPtTemp and nciTmpOffset functional profile members are optional and are not implemented.

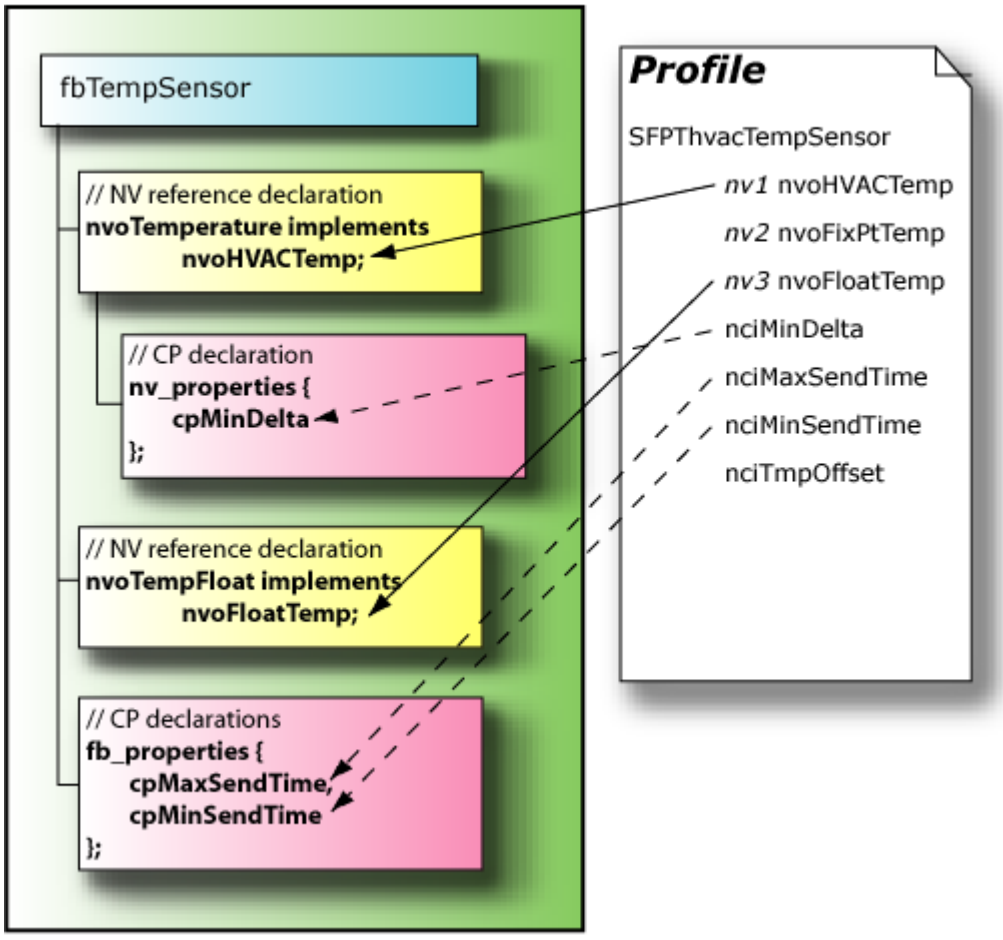


**Figure 9: fblock Configuration Property Member References**

The complete code for the declaration in Figure 9 is as follows:

```
SCPTmaxSendTime  cp_family cpMaxSendTime;
SCPTminDeltaTemp cp_family cpMinDelta;
SCPTminSendTime  cp_family cpMinSendTime;

network output SNVT_temp_p nvoTemperature
nv_properties {
      cpMinDelta
};

network output SNVT_temp_f nvoTempFloat;

fblock SFPThvacTempSensor {
      nvoTemperature implements nvoHVACTemp;
      nvoTempFloat implements nvoFloatTemp;
} fbTempSensor external_name("Temperature Sensor")
fb_properties {
      cpMaxSendTime,
      cpMinSendTime
};
```

## 2.8. Device and Functional Block Versioning

Versioning is an important part of upgrading and verifying systems. It can be useful information for a person tasked with such maintenance. Every certified device has a standard program ID (SPID). In many cases, when the device interface changes, the spid must be modified to indicate that the device is not the network-interface equivalent of other devices on the network. This convention performs basic device versioning. However, not all changes to a device interface require the SPID of a device to change. Additionally, it is possible to change the program within a device without changing its device interface, which also does not require the SPID to change. Details of allowable changes can be found in Appendix F.

A more flexible method of device versioning can be accomplished with the use of the SCPTdevMajVer and SCPTdevMinVer standard configuration property types. These two standard configuration property types are unsigned-short values with a range of 0–255 and a default value of 0. A configuration property based on SCPTdevMajVer must always specify the Constant flag, while a configuration property based on SCPTdevMinVer must always specify the Device-Specific flag. The Constant flag means that all devices with the same program ID will have the same value, while the Device-Specific flag means that devices with an identical program ID may have different values for this configuration property. See 2.7.3.3, *Configuration-File Access Methods*, for a complete list of flags and their values.

The presence of these configuration properties within a device defines the major version and minor version of the device. The major version number must be incremented when the network interface for the device changes, while the minor version number must be incremented when the network interface remains the same, but the device has a different behavior.

These device-versioning configuration properties are optional configuration properties of the Node Object functional profile. They should not be used outside of a Node Object functional block except for when the device does not implement a Node Object functional block.

For devices with multiple functional blocks declared within them, it is useful to know which functional blocks have changed. To support the versioning of individual functional blocks, the SCPTobjMajVer and SCPTobjMinVer configuration property types are defined. These two standard configuration property types are unsigned-short values with a range of 0–255 and a default value of 0. A configuration property based on SCPTobjMajVer must always specify the Constant flag, while a configuration property based on SCPTobjMinVer must always specify the Device-Specific flag. The Constant flag means that all devices with the same program ID will have the same value, while the Device-Specific flag attribute means that devices with an identical program ID may have different values for this configuration property. The presence of these configuration properties within a functional block defines the major version and minor version of the functional block. The major version number must be incremented when the network interface for the functional block changes, while the minor version number must be incremented when the network interface remains the same, but the functional block has a different behavior.

| Guideline 2.8: | If SCPTdevMajVer, SCPTdevMinVer, SCPTobjMajVer, or SCPTobjMinVer configuration properties are included on a certified device, they shall be implemented as described in 2.8, *Device and Functional Block Versioning*. |
|---|---|

## 2.9. Device Interface (XIF) File

The *device interface (XIF) file* is a standalone file that documents the device interface for a type of device. It also documents the default values for all the configuration properties on the device. The XIF file is an important component of a device's definition and must be submitted with the device's resource files when a device is submitted for certification.

The device interface file provides a summary and documentation of the device interface for a device with a specified SPID. The device interface file is created automatically by the LonBuilder® and NodeBuilder development tools, or it may be created manually based on the specifications in the *LONMARK Device Interface File Reference Guide*.

| | |
|---|---|
| Guideline 2.9A: | The device interface for a certified device shall be documented in a version 4.0 or newer device interface file as described in 2.9, *Device Interface (XIF) File*. The device interface file shall be submitted with the certification application. |

| | |
|---|---|
| Guideline 2.9B: | The device interface for a certified device shall include default values for all configuration properties as described in the *LONMARK Device Interface File Reference Guide*. |

# 3

# Resource Files

Resource files are files that define the functional profiles and types referenced by the device interface for one or more LONWORKS devices. These files allow network tools—such as installation tools and operator-interface applications—to interpret data produced by a device and to correctly format data sent to a device. They also help a system integrator or system operator to understand how to use a device and to control the functional blocks on a device. Resource files are available that define the standard components used in device interface definitions. Device manufacturers must create user resource files for any user-defined components used by their device interfaces.

# 3.1. Resource File Definitions

Resource files are used to publish definitions for both standard and manufacturer-defined resources. Standard resources include *standard functional profiles* (also called *LONMARK profiles*), *standard network variable types (SNVTs)*, *standard configuration property types (SCPTs)*, and *standard enumeration types*. Manufacturer-defined resources include *user functional profiles*, *user network variable types (UNVTs)*, *user configuration property types (UCPTs)*, and *user enumeration types*.

Resource files are grouped into *resource file sets*, where each set defines functional profiles, network variable types, configuration property types, enumeration types, strings, and formats for specified device types. The range of device types that a resource file set applies to is called the *scope* of the resource file set. For example, the scope may specify that the resource file set applies to an individual device type or to all device types. The available scopes are defined in 3.3, *Managing Resource Files*.

Each resource file set may contain definitions for the following resources:

| | |
|---|---|
| *Network Variable Types* | Type information for network variables. This information includes the size, units, scaling factors, and type category (float, integer, signed, *etc*) for each type. |
| *Configuration Property Types* | Type information for configuration properties. This information includes the size, units, scaling factors, and type category (float, integer, signed, *etc*) for each type. |
| *Functional Profiles* | Functional profiles define a template for functional blocks. Each functional profile is a collection of network variables and configuration properties designed to perform a single function on a device. |
| *Enumeration Types* | An enumeration type is a list of numerical values, each associated with a mnemonic name. |
| *Language Strings* | Language-specific strings that are referenced by type definitions, functional profiles, and self-documentation strings. |
| *Formats* | Formatting instructions for network variable and configuration property types. |

These resources are described in more detail in the following sections.

## 3.1.1. Type Definitions

A type definition may be a *network variable*, *configuration property*, or *enumeration type definition*. Network variable and configuration property types specify the data type, size, units, and scaling factors for the type. The data type may be a *base type*, an *enumeration type*, a *structure type*, a *union type*, or a *reference* to another network variable type.

### 3.1.1.1. Base Types

The following base types are available:

- Signed character (8 bits),
- Unsigned character (8 bits),
- Signed short (8 bits),
- Unsigned short (8 bits),
- Signed long (16 bits),
- Unsigned long (16 bits),
- Signed quad (32 bits),
- IEEE 754 single-precision floating point (32 bits).

The base types define size in bits, but—with the exception of floating point—do not define fractional values, nor do they define resolution and upper/lower limits. Limits are imposed by the range of values that can be represented using the number of bits of the type.

### 3.1.1.2. Enumeration Types

An *enumeration type* is a list of numerical values, each associated with an enumerator name. If a network variable or configuration property type contains an enumeration, the definitions of the enumerated values are maintained separately as an enumeration type. Enumeration types are defined in a resource file with a .typ extension (along with network variable and configuration property types), and may also be defined in a separate C header file (.h extension). The C header file is typically required by development tools, but not by other network tools.

By convention, enumeration type names use all lower case, with each word in the name separated by an underscore, and ending with "_t" (for example: count_control_t). Enumeration type names are limited to 64 characters, including the "_t" suffix characters.

By convention, all the enumerator names within an enumeration type use a common, unique, prefix. The enumerator names use all upper case, with words separated by underscores (for example: DCM_SPEED_CONST and DCM_PRESS_CONST). Enumerator names are limited to 64 characters, including the unique prefix.

Each of the numerical values for an enumeration type is a signed 8-bit value. The range is –126 to 127. As convention, a value of –1 (0xFF) is reserved for an invalid or undefined value, and a value of 0 is typically used for the default value. The –1 (0xFF) value is always named <Prefix>_NUL.

The standard enumeration types are documented in the *LONMARK SNVT and SCPT Master List*.

### 3.1.1.3. Structure Types

A structure type defines an aggregate data type that consists of one or more fields. For example, the SNVT_date_cal network variable type contains 3 fields for the year, month, and day. Each field of a structure type may itself be a base type, a bitfield with a width of 1–8 bits, an enumeration type, another structure type, a union type, or a reference to another network variable type. There is no specified, programmatic naming convention for structures.

When used as the data type for network variables, all fields of a structure are updated when a network variable value update is sent on the network. There is no means to transmit an individual field of a network variable structure.

Structure type names are limited to 48 characters; but, if used as the data type of a network variable, they are limited to 16 characters. Field names are also limited to 48 characters.

### 3.1.1.4. Union Types

A union type defines an aggregate data type that consists of one or more fields. Unlike a structure type, the start of each of the fields of a union type overlaps. Like a structure type, each field may itself be a base type, a structure type, a union type, or a reference to another network variable type. There is no specified, programmatic naming convention for unions.

To allow for a network tool to determine the active portion of the union, a union is typically defined as a field within a structure, where the first field of the structure is a base-type value that is used as a selector for the union. This enables a format to be defined for the structure that uses one or more ternary operators to select the appropriate format based on the selector value.

**EXAMPLE**

The SNVT_file_status standard network variable type (SNVT) is defined as follows:

```
typedef struct  {
   file_status_t  status;
   unsigned long  number_of_files;
   unsigned long  selected_file;
   union adr  {
      struct  descriptor  {
         signed char  file_info[16];
         signed quad  size;
         unsigned long  type;
      };
      struct  address  {
         unsigned short  domain_id[6];
         unsigned short  domain_length;
         unsigned short  subnet;
         unsigned short  node;
      };
   };
}  SNVT_file_status;
```

The **status** field allows a network tool to determine the active portion of the **adr** union. The **status** field is defined as an enumeration with the following values:

```
typedef enum file_status_t  {
   /*  0 */   FS_XFER_OK,
   /*  1 */   FS_LOOKUP_OK,
   /*  2 */   FS_OPEN_FAIL,
   /*  3 */   FS_LOOKUP_ERR,
   /*  4 */   FS_XFER_UNDERWAY,
   /*  5 */   FS_IO_ERR,
   /*  6 */   FS_TIMEOUT_ERR,
   /*  7 */   FS_WINDOW_ERR,
   /*  8 */   FS_AUTH_ERR,
   /*  9 */   FS_ACCESS_UNAVAIL,
   /* 10 */   FS_SEEK_INVALID,
   /* 11 */   FS_SEEK_WAIT,
   /* -1 */   FS_NUL = -1
} file_status_t;
```

The format definition for the SNVT_file_status type uses a ternary operator to select one of two formats based on the value of the **status** field as follows:

```
SNVT_file_status:   text(("%m,%d %d ",
                    status,
                    number_of_files,
                    selected_file ),
                    ((status == 1) ? ("%d %d %s",
                    adr.descriptor.size,
                    adr.descriptor.type,
                    adr.descriptor.file_info) :
                    ("%d <%x %x %x %x %x %x> %d %d",
                    adr.address.domain_length,
                    adr.address.domain_id[0],
                    adr.address.domain_id[1],
                    adr.address.domain_id[2],
                    adr.address.domain_id[3],
                    adr.address.domain_id[4],
                    adr.address.domain_id[5],
                    adr.address.subnet,
                    adr.address.node)));
```

Union type names are limited to 48 characters; but if used as the data type of a network variable, then they are limited to 16 characters.

### 3.1.1.5. Network Variable Type Definitions

A *network variable type* definition specifies the data type, size, units, and scaling factors for a network variable type. Even though it is possible to declare a network variable directly using a base type, such a declaration cannot be used for certified devices—except in the case of a configuration network variable. Network variable types are defined in a resource file with a .typ extension.

Standard network variable type (SNVT) names always begin with a "SNVT_" prefix and are always lowercase (*e.g.*: SNVT_temp_p). By convention, user network variable type names begin with "UNVT_" and are lowercase (*e.g.*: UNVT_my_type).

Names of network variable types are limited to 16 characters, including the five prefix characters.

The type for a SNVT field may be based on a SNVT, but cannot be based on a user network variable type (UNVT). A UNVT field may be based on a UNVT or SNVT.

The standard network variable types are documented in the *LONMARK SNVT and SCPT Master List*.

### 3.1.1.6. Configuration Property Type Definitions

A *configuration property type* definition specifies the data type, size, units, and scaling factors for a configuration property type. Configuration property types are defined in a resource file with a .typ extension (this is the same file used for network variable types).

A network variable type may be used as the data type of a configuration property type, but this is not required. There is no requirement to create a new network

variable type for a new configuration property type. Standard configuration property types (SCPTs) and SCPT fields may be based on SNVTs, base types, or enumerations, but they cannot be based on UNVTs. User configuration property types (UCPTs) and UCPT fields may be based on UNVTs, SNVTs, base types, or enumerations.

Standard configuration property type names always begin with a "SCPT" prefix and are always a combination of uppercase and lowercase characters (*e.g.*: SCPTmaxPressureSetpoint). By convention, user configuration property-type names begin with a "UCPT" prefix and are a combination of uppercase and lowercase characters (*e.g.*: UCPTmyConfigurationType).

Names of configuration property types are limited to 63 characters, including the four prefix characters.

The standard configuration property types (SCPTs) are documented in the *LONMARK SNVT and SCPT Master List*.

## 3.1.2. Functional Profiles

A *functional profile* defines a template for a functional block. Each functional profile consists of a profile description and a specified set of network variables and configuration properties designed to perform a single function on a device. The network variables and configuration properties specified by the functional profile are called the *functional profile members*. A functional profile specifies whether or not each functional profile member is mandatory or optional. When a functional block implements a functional profile, it must implement all mandatory functional profile members defined by the functional profile, and it may implement some, all, or none of the optional functional profile members. A functional block may also add implementation-specific members that are not defined in the functional profile, but this is not recommended for certified devices. As described in 2.7.3.1, *Configuration Property Distribution Methods*, multiple configuration property members may share a single configuration property implementation on a device as long as they are the same type, are part of a compatible application set, and the sharing is not prohibited by the functional profile documentation.

Functional profiles are defined in a resource file with a .fpt extension. Functional profiles are also called *functional profile templates*.

In addition to the functional profile members, a functional profile also specifies the semantic meaning of the information being communicated. Thus, a functional profile provides additional information on usage, beyond the type information specified by a network variable or configuration property type.

Functional profiles that have been approved and published by the LONMARK Interoperability Association are called *standard functional profiles*. They are also called *LONMARK profiles*. The complete set of LONMARK profiles is available on the LONMARK Web site at www.lonmark.org. The primary function of a certified device

must be implemented using one or more LONMARK profiles. Developers can choose the profiles that best fit the functions of the device being developed, with the exception that a functional profile identified as obsolete on the LONMARK Web site may not be used for a new or recertified device, unless otherwise specified on the LONMARK Web site. If the appropriate LONMARK profiles are not available for a particular device, developers can work with a LONMARK task group to propose a new LONMARK profile as described in Appendix E. Information on how to propose a new LONMARK profile, as well as the templates to use, is provided in the profile.zip archive also available on the LONMARK Web site.

Functional profiles that have not been approved and published by the LonMark Interoperability Association are called *user functional profiles*. A user functional profile cannot be used to implement the primary function of a certified device.

> Guideline 3.1.2A: The primary function of a certified device shall be implemented with one or more functional blocks that conform to one or more LONMARK profiles as defined in 3.1.2, *Functional Profiles*.

> Guideline 3.1.2B: Each functional block shall, as a minimum, implement all the functional profile's mandatory network variables and configuration properties. Any optional members implemented by the functional block shall be implemented as specified in the profile; with the exception that the application set of an optional configuration property may be changed.

If an optional configuration property is not implemented in a functional block, then it is recommended that the device follow the specified default value, whenever possible, to ensure consistent behavior.

### 3.1.2.1. Functional Profile Names

Each functional profile must have a name that is unique within its resource file set. The name must start with "SFPT" for standard profiles and "UFPT" for user profiles. The name may not contain spaces. By convention, there is no underscore following the "SFPT" or "UFPT" prefix; the first letter after the prefix is lower case; and the name uses mixed case. For example, "SFPToccupancySensor" follows this convention. Functional profile names are limited to 64 characters, including the four-character prefix. A functional profile name may include underscores, but cannot use spaces or any other special characters.

### 3.1.2.2. Functional Profile Numbers

Each functional profile has a unique number, called the *functional profile number* or the *functional profile key*, which uniquely identifies the profile. The functional profile number need only be unique within the scope of the functional profile as described in 3.3, *Managing Resource Files*.

The LONMARK Association assigns profile numbers to standard functional profiles. For example, the SFPTswitch profile is profile number 3200. A standard profile number may be used as the device class for a device implementing the standard profile as the primary functional block for functional profile numbers between 100 and 19 999, inclusive. As described in the next section, inheriting profiles use the same functional profile number as the scope-0 profile from which they inherit their content. Manufacturers are free to assign any functional profile number to new non-inheriting profiles, as long as they are in the range of 20 000 to 25 000, and as long as the number is unique for the program ID template and scope of the resource file set containing the functional profile.

For example, a UFPTmyCreation user functional profile is defined as follows:

- Scope: 4
- Program ID template: 80:00:9F:20:00:00:00:00
- Functional profile number: 21 234

In this case, the manufacturer with MID 0:00:9F cannot define another functional profile with a profile number of 21 234 in any other resource file sets with a scope of 4 (see 3.3, *Managing Resource Files*, for a description of this value) and a device class of 20:00. The manufacturer can, however, create another functional profile with the same profile number in a resource file set at scope 3, 5, or 6. The manufacturer can also create another functional profile with the same number in a scope-4 resource file set as long as the device class is not 20:00. Other manufacturers can also create functional profiles that use profile number 21 234, subject to the same rules. If the same functional profile number is defined in multiple resource file sets, the definition in the resource file set with a matching program ID template and the numerically highest value applies. For example, if functional profile 21 234 is defined in both a scope-3 and scope-4 resource file set, the definition in the scope-4 resource file set applies if the program ID template matches.

### 3.1.2.3. Inheritance

If a device application implements a functional block based on a standard functional profile and adds additional members not defined in the standard profile, a user functional profile can be created that defines the additional members. To simplify development and maintenance of the user functional profile, the user functional profile may *inherit* the members defined in the standard functional profile and therefore only require the new members to be defined.

**EXAMPLE**

A developer can add a new network variable member to the Space Comfort Controller profile (SFPTspaceComfortController). The developer creates a new scope 3 – 6 functional profile named UFPTspaceComfortController with the same functional profile number as SFPTspaceComfortController, and specifies that the user functional profile inherits from the standard functional profile.

A functional profile that uses inheritance is called an *inheriting profile*. An inheriting profile includes a flag that specifies that it inherits members from a scope-0 profile. The inheriting profile must have the same functional profile number as the scope-0 profile. The inheriting profile may have a different name; however, a name similar to that of the scope-0 profile is recommended.

### 3.1.2.4. Member Names

Each network variable and configuration property member of a functional profile has a unique *member name* for that profile. Member names may be up to 64 characters. The member name may contain only letters, numerals, and the underscore character. A prefix is not required, but input network variable names may start with "nvi", output network variable names may start with "nvo", and configuration property names may start with "cp" (preferred) or "nci". By convention, the name is mixed case with no underscores, starting with a lower-case character if a standard prefix is used, and otherwise starting with an upper-case character. For example, "nviEnergyHoldOff" follows this convention. The abbreviations listed in 2.7.2.2, *Changeable-Type Network Variables*, should be used.

### 3.1.2.5. Member Numbers

Each network variable and configuration property member of a functional profile has a unique *member number* for that profile. This member number is used to associate a network variable or configuration property on a device with the corresponding network variable or configuration property member of the functional profile. Member numbers may be in the range of 1 to 4095, and need not be contiguous. Member numbers must be unique, with the exception that network variable and configuration property members may use the same number. There is a maximum of 255 mandatory members and 255 optional members of each type (scope 0 NV, inheriting NV, scope 0 CP, and inheriting CP).

Each member of an inheriting profile may be defined in one of two functional profiles: the inheriting profile itself and the inherited scope-0 profile with the same functional profile number. To correctly associate each network variable and configuration property on a device with either an inheriting profile or a scope-0 profile, the member number is prefixed by a *functional profile selector*. If the functional profile selector is an ASCII vertical bar ("|"), the member number identifies a member of a scope-0 profile. If the functional profile selector is an ASCII number sign ("#"), the member number identifies a member of the inheriting profile.

The number-sign functional profile selector is always used for members of user functional profiles, including profiles that do not use inheritance. The vertical-bar functional profile selector is always used for members of standard functional profiles. Two different functional profile members may have the same member number as long as they use different functional profile selectors. For example, the "**|1**" member of a functional profile is not the same as the "**#1**" member of the same profile. This prevents conflicts if new members are added to a standard functional profile that has already been used as the basis for inheriting profiles.

## 3.1.3. Language Strings

*Language strings* are text strings that are referenced by type definitions, functional profiles, and self-documentation strings. Language strings are stored in *language files*. There is one language file for every language supported by a resource file set. Language strings are referenced by index within the language file so that language-string references may be translated by looking up the reference in the appropriate language file. This index is called the *language-string index*.

Language strings may contain all printable ASCII characters except the tilde ("~"). C-like escape codes ("\n", "\x3D", *etc.*) are not supported.

A network tool may allow a user to specify a search order for language files, and can therefore control which set of strings are displayed, depending on the chosen and available language files. Each language file uses a unique file extension so that it can use the same base filename as the rest of the resource file set. The standard language file extensions are listed in Appendix B.

### 3.1.3.1. Self-documentation String Reference

The self-documentation text within a self-documentation string can reference a language string using the reserved 0x80 value (represented as an "`\x80`" ASCII string). Some network tools may not recognize these string references.

The syntax for a self-documentation string reference is as follows:

**\x80**[*scopeSpecifier***:**]*languageStringIndex*

The components of a self-documentation string reference are the following:

- A byte containing the value 0x80, represented by the "**\x80**" string.
- *scopeSpecifier* may be a "3", "4", "5", or "6" to specify a scope 3, 4, 5, or 6 resource. If not included, the scope is 0.
- A colon ("**:**") following the scope specifier. The colon is not included if the scope specifier is not included, otherwise it is mandatory.
- *languageStringIndex* is the index of the language string within the language file. This index ranges from 1 to 16 777 216.

**EXAMPLES**

The following string reference specifies language string index 522 within the standard resource file set. This string is the following in the standard.eng file: "Dictates the desired state of the actuator, determined by the specific application."

```
"@2|1;\x80522"
```

The following string reference specifies language string index 100 within a user resource file set at scope 3.

```
"@2#1;\x803:100"
```

### 3.1.4. Formats

*Formats* provide formatting instructions for network variable and configuration property types. Each network variable and configuration property type must have at least one format defined. This format describes how the value will be displayed to or entered by network integrators and network operators. It is possible to define multiple formats for a network variable type or configuration property type. Different formats can provide the information in a different order (if the value is a structure or union) or provide a different scaling factor (for example, the `SNVT_temp_f` network variable type has three formats: one for Fahrenheit, one for differential Fahrenheit, and one for Celsius). Formats are defined in format files with a .fmt extension. The *NodeBuilder Resource Editor User's Guide* provides instructions on how to define formats.

## 3.2. Identifying Appropriate Resources

To promote interoperability between devices, Echelon and the LONMARK Interoperability Association have defined many standard resources. These standard resources specify standard functional profiles corresponding to specific functions that are common in specific controls industries such as temperature sensors and space comfort controllers, and also specify standard operational and configuration data types required by the controls industry.

Standard resources should be used in applications whenever possible. In some cases, a developer may find that there is a resource that they want to use that is not defined in the standard resource file set. In this case, the developer has two options—propose a new standard resource or develop a user resource.

If the required resource has general applicability within the developer's industry or across multiple industries, the developer should work with a LONMARK task group to propose a new standard resource. Section 3.2.2 identifies guidelines for new standard resources and Appendix E outlines the procedure for submitting new standard resource proposals.

If the required resource is specific to a particular implementation, installation, or company, the developer must create a *user resource file set* defining any required user functional profiles (UFPs or UFPTs), user network variable types (UNVTs), and user configuration property types (UCPTs) required by the interoperable interface of the device. Section 3.2.3 identifies guidelines for using user resources.

To facilitate reuse, a user functional profile should be defined as a general solution, rather than the specific one. Configuration properties should be used to configure a functional profile to meet specific requirements. This approach prepares for future reuse, and also prepares for proposing the user functional profile as a standard.

## 3.2.1. Using Standard Resources

The standard resources are defined in the *standard resource file set*. The standard resource file set includes definitions for standard functional profiles (SFPs or SFPTs), standard network variable types (SNVTs), standard configuration property types (SCPTs), standard enumeration types, standard language strings, and standard formats. The standard resource file set includes language-string definitions for American English and British English.

With the exception of the standard functional profiles, the standard resources are documented in the *LONMARK SNVT and SCPT Master List* included with the standard resource file set. The standard functional profiles are documented in individual functional profile documents available on the LONMARK Web site at www.lonmark.org.

The standard resource file set is also available on the LONMARK Web site at www.lonmark.org. It is updated periodically as Echelon and the LONMARK Interoperability Association define new standard profiles and types.

## 3.2.2. Proposing New Standard Resources

A device developer should propose a new standard resource whenever an appropriate standard resource is not available, but a resource with general applicability within the developer's industry or across multiple industries is required. The resource may be a functional profile, network variable type, configuration property type, enumeration type, format, or language string. Any member of the LONMARK Association may submit a proposal for a new or revised resource. This section outlines a few guidelines for new standard resources. Appendix E describes the procedure for submitting new standard resource proposals.

A new standard resource proposal must meet the following general guidelines:

1. The resource must not be specific to a particular manufacturer or product.
2. The resource must not duplicate an existing standard resource. An exception is a new standard resource that improves an existing standard resource. In this case,

the LONMARK Association should identify the existing standard resource as obsolete for new designs.

3 The resource should be based on ISO/IEC standard conventions, or existing conventions within the appropriate industry, if possible.

4 The resource name must be unique within the standard resource file set, and must follow similar naming conventions as existing resource names.

5 Echelon assigns functional profile numbers, type indices, and names. These may differ from the numbers, indices, and names assigned in the proposed resource file set. However, the proposed numbers, indices, and names must still follow the required guidelines.

#### 3.2.2.1. Guidelines for New Standard Functional Profiles

A new standard functional profile must meet the following guidelines:

1 The profile should represent an atomic function of a device; it should not be an aggregation of many different functions. Instead, the different functions should be broken down into separate profiles.

2 A profile must contain all the mandatory interfaces required to make it useful for a network integrator. It may contain additional, optional interfaces that make it more convenient to use. It must be possible to use the profile by using only the mandatory interfaces and none of the optional interfaces.

3 A profile must identify other profiles with which it will typically be used. The network variable types must be compatible between the profiles.

4 Changeable-type network variables, as described in 2.7.2.2, should be used for general-purpose profiles.

5 Floating-point types should be used for numeric inputs and outputs when a wide range of values with high resolution is required. This guideline does not apply if a specific numeric type has been standardized within the profile's industry.

6 The SNVT_switch type should be used for Boolean (one-of-two levels) inputs and outputs as well as any discrete data types (one-of-n levels) requiring up to 201 discrete levels. The SNVT_lev_percent type should be used for discrete data types requiring 202 to 32 766 discrete levels.

7 Enumeration types should be used for state inputs and outputs.

8 A profile should apply to multiple industries, if possible.

9 A profile should be defined as a general solution rather than a specific one. Configuration properties should be used to configure a functional profile to meet specific requirements.

10 Profiles must not embed documentation within the text of the profile or other documents that is better expressed as part of a standard resource. For example, a

profile that requires different state inputs cannot use a SNVT_count, SNVT_state, or SNVT_str_asc input and then provide text documentation that identifies an interpretation for each of the SNVT_count values, SNVT_state bits, or SNVT_str_asc strings. Instead, a new enumeration type must be defined with enumeration values for each of the states, or a structure type with a bitfield must be created to represent each of coexisting states.

**11** A new SNVT should not be proposed solely to support the creation of a new SCPT. There is no requirement to base a SCPT on a SNVT. A new SNVT may be proposed for a new SCPT if the type required for a new SCPT would be suitable for non-configuration network variable inputs and outputs.

**12** Network variable members to be shared by multiple functional blocks on a device should be proposed as new optional members for the Node Object functional block.

**13** Functional profile names should follow the naming guidelines in 3.1.2.1 (*e.g.*: SFPToccupancySensor).

**14** Functional profile names must be no more than 64 characters, including the four prefix characters.

**15** Profile member names should follow the naming guidelines in 3.1.2.4 (*e.g.*: nviEnergyHoldOff).

**16** Profile member names must be no more than 16 characters, including the prefix characters.

**17** Profile member numbers must be unique, be between 1 and 4095, and use the appropriate functional profile selector.

#### 3.2.2.2. Guidelines for New SNVTs and SCPTs

A new SNVT or SCPT must meet the following guidelines:

**1** A new fixed-point numeric SNVT should not be proposed if an existing floating-point SNVT exists for the same measurement type, unless the floating-point SNVT will not meet the target-application performance requirements.

**2** Numeric values must be represented as Système Internationale (SI) units if an appropriate SI unit is available, except when the generally accepted industry convention worldwide is not in SI units. A new numeric resource in non-SI units will not be approved when an existing numeric resource in SI units already exists.

**3** A numeric value might be represented as SI units using the standard multipliers listed in Table 2. The standard multiplier identifier or its full name should be included in the type name.

**Table 2. Standard Multipliers**

| *Multiplier Name* | *Value* | *Identifier* |
|---|---|---|
| Pico | $10^{-12}$ | p |
| Nano | $10^{-9}$ | n |
| Micro | $10^{-6}$ | u |
| Milli | $10^{-3}$ | m |
| Kilo | $10^{+3}$ | k |
| Mega | $10^{+6}$ | M |

For example, a type that is used to describe the current output wattage of a power station could describe the value in Megawatts. A possible type name would be SCPTmegaWatt.

**4** A new aggregate (structure or union) SNVT or SCPT that aggregates existing SNVTs or SCPTs should only be proposed if multiple quantities must be communicated simultaneously in a single update (due to time-stamping needs or something similar), and several similar products are expected to operate in the same way. A new aggregate SNVT must not be proposed solely to gather information into a single variable for the purpose of reducing the number of network variables required on a device.

**5** A new SNVT that is based on a new enumeration type may be proposed when there is an industry-accepted set of modes, states, functions, or other mutually exclusive conditions that need to be communicated between products of different manufacturers.

**6** A new aggregate SNVT or SCPT may include bitfields to hold enumerated or numerical value to reduce the total size of the SNVT or SCPT.

**7** A new SNVT, SCPT, or aggregate field representing a Boolean flag should be based on the boolean_t enumeration type.

**8** A new SNVT that is not used exclusively for monitoring and control applications cannot contain embedded type information, unless that type information is static. For example, the SNVT_reg_val type contains an embedded **unit** field that specifies the type of the **raw** field. Because of this, the reg_val_unit_t enumeration type used by the **unit** field is defined with a static definition.

**9** SNVT and SCPT names should follow the naming guidelines in 3.1.1.5 (*e.g.*: SNVT_temp_p), and 3.1.1.6 (*e.g.*: SCPTmaxPressureSetpoint).

**10** SNVT names must be no more than 16 characters, including the five prefix characters.

**11** SCPT names must be no more than 63 characters, including the five prefix characters.

**12** Every SNVT and SCPT must have at least one format defined. For types that don't lend themselves to formatting for textual display, this format may be defined so that no data is shown. In such a case, the format must be defined so that the integrator can see some suitable replacement text. For example, the following format displays a static text string:

```
UNVT_my_type: text("<value not shown (binary data)>");
```

### 3.2.2.3. Guidelines for New Standard Enumeration Types

A new standard enumeration type must meet the following guidelines:

**1** A value of –1 (0xFF) must be used for an invalid or undefined value.

**2** A value of 0 must be used for the default value if a default value exists.

**3** Enumeration type names should follow the naming guidelines in 3.1.1.2 (for example: DCM_SPEED_CONST and DCM_PRESS_CONST).

**4** Enumerator values can be added to an existing enumeration type following the same approval procedure for new enumeration types, with the exception that new enumerator values cannot be added to an enumeration type if the documentation for the enumeration type states that the set of enumerator values is fixed and may not be extended. Except for these fixed enumeration types, applications must be able to handle unexpected enumerator values.

**5** All enumerator names within an enumeration type must have a common, unique, prefix.

**6** Enumeration type names must be no more than 16 characters including the "_t" suffix.

**7** Enumerator names must be no more than 64 characters including the unique prefix.

### 3.2.2.4. Guidelines for New Standard Formats

A new standard format must meet the following guidelines:

**1** Numerical formats that have different US and SI representations must have US and SI formats defined with appropriate scaling as required.

**2** Formats requiring list separators should use the localized list separator and specify the localized modifier ("#LO").

**3** Formats for date and time values should use the date and time localization functions and specify the localized modifier ("#LO").

**4** Formats for unions should use the ternary operator to select alternate formats as required.

#### 3.2.2.5. Guidelines for New Standard Language Strings

A new standard language string must meet the following guidelines:

1. US language strings must be provided for all standard resources.
2. A language string may not contain a tilde ("~").
3. Do not put periods at the ends of strings unless they contain two or more sentences.
4. Use title case for type and profile names (all major words—nouns, verbs, and adverbs—capitalized; minor words—articles, prepositions, and coordinating conjunctions—are not capitalized unless they are the first or last word); use lower case for unit names—unless the name is a proper name in which case it should have an initial capital; use sentence case (initial capital only) for all other strings.
5. Avoid useless phrases at the beginning of strings. For example, use "Outdoor temperature reading" instead of "This is used to provide an outdoor temperature reading," or "Used to provide an outdoor temperature reading," or "Provides an outdoor temperature reading."

## 3.2.3. Using User Resources

User resources are distinct from a manufacturer's proprietary data because user data are intended to be manipulated by parties other than the manufacturer or the manufacturer's agents. It is allowed that it may be necessary to manipulate user data to successfully commission a certified device, but manipulation of manufacturer data cannot be a requirement to successfully commission a certified device. Manufacturer data may be used for calibration, diagnostic, test, and repair interfaces used solely for manufacturing or field troubleshooting operations that are not required for normal commissioning and operation of the device.

| | |
|---|---|
| Guideline 3.2.3A: | All user functional profiles, user network variable types, user configuration property types, and user enumeration types required for the interoperable interface of a certified device shall be documented within a LONMARK resource file set as described in Chapter 3, *Resource Files*. A minimum of one language file shall be included defining any required language strings. A format file shall be included defining a minimum of one format for each user network variable type and user configuration property type. This guideline does not apply to any resources required exclusively for configuration if a passive configuration tool is supplied. The resource file set, if any, shall be submitted with the certification application; and the passive configuration tool, if any, shall be made available at the time of application submittal. |

| | |
|---|---|
| Guideline 3.2.3B: | There shall be no requirement to access or modify proprietary data in the course of successfully commissioning a certified device. The lack of access to proprietary data shall not prevent the successful operation or use of the device's published, interoperable functional blocks. |

When creating a user functional profile, manufacturers can either inherit from an existing standard functional profile or define a new user functional profile. A user functional profile that does not inherit from a standard functional profile, as defined in 3.1.2.3, *Inheritance*, cannot form the primary function or basis of a certified device, but can be supplemental and complementary to the standard functional profiles on the device.

Developers who choose to add manufacturer-specific network variable and configuration property members, as a part of their interoperable interface, to the functional blocks on their devices must provide resource files that contain user functional profiles defining the manufacturer-specific members. If the user network variables or configuration properties are added to a LONMARK profile, then the LONMARK profile must be inheriting or redefined in the user resource files. Inheritance should be used for all new profiles. Additionally, if the network variable or configuration property type is not a standard network variable or configuration property type (SNVT or SCPT), then the user network variable or configuration property type (UNVT or UCPT) must be defined within the resource file set.

> Guideline 3.2.3C: User network variables or configuration properties that are intended to be a part of a functional block's interoperable interface shall be documented in a user functional profile within the device's resource file set. The functional profile shall define the network variable and configuration property members.

Network variables or configuration properties that are not associated with a particular functional block, but pertain to the device as a whole, can be assigned to the Node Object functional block as manufacturer-defined network variables or configuration properties.

## 3.3. Managing Resource Files

There may be multiple resource file sets on a computer. In addition to the standard resource file set, there may be one or more user resource file sets from one or more manufacturers. Each resource file set must be contained in a single folder, but there may be multiple resource file sets in a folder. For example, a network may contain devices from several different manufacturers, and each manufacturer may supply their own resource file set with type, functional profile, format, and language strings specific to their devices.

Each resource file set may be kept in a separate folder. These folders are typically installed in the LONWORKS Types\User folder (this is c:\LonWorks\Types\User by default), each identified by the manufacturer name. Large manufacturers may use additional subdirectories to organize their resource files. For example: "c:\LonWorks\Types\User\Manufacturer A\Division B."

To enable network tools to find resource files, a *resource catalog* is maintained that contains a list of resource folders. The resource catalog is contained within a *resource-catalog file*, which is a file with a .cat extension. By default, the resource catalog file is contained in the LONWORKS Types folder and is named ldrf.cat (the full path is C:\LonWorks\Types\ldrf.cat by default), but both the folder and filename may be changed. There can only be a single resource catalog per computer, and all applications on the computer must use the same resource catalog.

To be able to associate a resource file with a network variable, configuration property, or functional block on a device, each resource file set must be associated with a particular standard program ID (SPID), a range of SPIDs, or with all SPIDs. Each resource file set includes a *program ID template* that is compared to the SPID of a device when searching for resources for that device. The type of association is called the *scope* of the resource file. The scope for a resource file specifies what part or parts of a device's SPID should be used when selecting the resource file set. The scope is an integer value between 0 and 6 as defined in the following table:

| Scope | Scope Definition |
|---|---|
| 0 | Used for the standard resource file set. The standard resource file set contains standard definitions for all devices from any manufacturer. This scope value can only be used for the standard resource file set published by Echelon and distributed by the LONMARK Interoperability Association. |
| 1 | Reserved for future use. |
| 2 | Reserved for future use. |
| 3 | Used for a user resource file set containing user resources for all devices with a specified manufacturer ID (MID). This scope value can be used by a manufacturer for a resource file set that applies to all of the manufacturer's devices. |
| 4 | Used for a user resource file set containing user resources for all devices with a specified MID and device class. This scope value can be used by a manufacturer for a resource file set that applies to all of the manufacturer's devices of a specific device class. |
| 5 | Used for a user resource file set containing user resources for all devices with a specified MID and device class, usage, and channel. This scope value can be used by a manufacturer for a resource file set that applies to all of the manufacturer's devices of a specific device class, usage, and channel type. |
| 6 | Used for a user resource file set containing user resources for all devices with a specified SPID. This scope value can be used by a manufacturer for a resource file set that applies to a single device type. |

**EXAMPLE**

A manufacturer produces a scope-3 resource file set with all type, format, and language information for all its devices. The resource file set has a program ID template of 80:00:9F:00:00:00:00:00. All applications with 0:00:9F (the LONMARK Technical Staff MID) as the MID portion of their SPID would use the types in this file set.

By using scope, resource files are treated as a hierarchy of type definitions, with scope 0 at the top. Resource files may refer to other resource files above them in the scope hierarchy. For example, a file with a scope of 5 could contain references to scope 4, 3, and 0 resource files, each with program ID templates that match the relevant parts of the scope-5 program ID template.

## 3.4. Implementing Resource Files

Developers can use the NodeBuilder Resource Editor to view standard and user resource file sets, and to create and maintain user resource file sets. The NodeBuilder Resource Editor is available as a free download from the LONMARK Web site to current LONMARK members, and is also included with Echelon's NodeBuilder Development Tool.

The LONMARK Web site also includes an open application programming interface (API) for accessing these files. This API is called the *LONMARK resource file API*. It is available in two versions: a Windows dynamic link library version, including an optional COM interface, that can be used to read and write resource file sets, and a source-code version that can be ported to any processor and used to read resource file sets.

A *LONMARK resource file set* is a resource file set that is compatible with the LONMARK resource file API, and that conforms to the resource file guidelines outlined in this chapter.

Each file in a resource file set has a *data version number* and a *format version number*, with the exception of format files that do not include embedded version numbers. The data version number identifies the version of the data contained within the file. The format version number identifies the file format that the file conforms to.

Table 3 lists the format version numbers as of the publication date of these guidelines. The table also lists the version number of the resource file API required to access each format version.

Two conversion utilities are available for converting between different format versions. The LONMARK Resource File Conversion Utility converts format version-3 format files to format version 2. The NodeBuilder Resource File Converter converts all other resource-file types and supports conversion between any of the format versions supported by the resource file API. Both utilities are available from the LONMARK Web site at www.lonmark.org.

**Table 3. Resource File Format Version Numbers**

| *File Type* | *Format Version* | *Format Changes* | *Minimum Required Resource File API* |
|---|---|---|---|
| ***Functional Profile*** | 1 | Initial release. | 1.0 |
| | 2 | Added support for larger profiles and for marking profiles as obsolete. | 2.0 |
| | 3 | Added support inheriting profiles and for non-contiguous member numbers. | 2.1 |
| | 4 | Added support for CP arrays and for deleting profiles. | 2.2 |
| ***Format File*** | 1 | Initial release. | N/A |
| | 2 | Added support for scale factors. | N/A |
| | 3 | Added support for language localization. | N/A |
| ***Language File*** | 1 | Initial release. | 1.0 |
| | 2 | Added support for larger language files. | 2.0 |
| | 3 | Added support for deleting language strings. | 2.2 |
| ***Type File*** | 1 | Initial release. Included NVTs only. | N/A |
| | 2 | Added CPTs and enumeration types. | 1.0 |
| | 3 | Added support for invalid values and for marking types as obsolete. | 2.1 |
| | 4 | Added support for CP arrays and for deleting types. | 2.2 |

# 4

# Network Installation

The physical attachment of devices to a communications channel, such as a twisted-pair wire or a power-line circuit, is not enough to commission a control network. The physical attachment only provides a path for the device to send and receive messages. The device also needs information on the system to which it belongs and the other devices with which it should share data. Specifying and loading this additional information is a necessary step for installing a device into a control network. Address assignment, binding, and configuration are the network-management tasks associated with managing this information.

The device design plays an essential role in how it will be installed into an interoperable control network. Regardless of whether a single device or a subsystem consisting of a collection of devices needs to be installed into an interoperable network, a network tool must be able to manage the logical connections between devices. Bringing devices and systems on-line, making connections, polling, and querying devices, are all services that a network tool may perform and to which a device or subsystem must be able to respond.

This chapter outlines the design guidelines that must be followed so that devices can be installed into an interoperable network. It is also important that the network tools used to install the interoperable network support installation of devices that follow these guidelines.

# 4.1. Network Addressing

Devices use their network addresses to send messages and to determine if messages are destined for them. A device's network address consists of the following three components defined by the ANSI/EIA/CEA 709.1 protocol:

- The *domain* to which it belongs.
- The *subnet* to which it belongs within the domain.
- The *node ID* within the subnet.

Using the ANSI/EIA 709.1-A Control Network Protocol, a device can be a member of up to two domains. Under ANSI/EIA/CEA 709.1-B, a device can be a member of up to 65 535 domains. A key function of a network tool is to ensure that in any domain, no two devices are assigned the same subnet and node ID.

A device can also be addressed by using *group* addresses, assigned during the binding process. A single message can be addressed to all members of a group. Under ANSI/EIA 709.1-A, a device can be a member of up to 15 different groups. Under ANSI/EIA/CEA 709.1-B, that limit is raised to 65 535 groups.

The binding process also allocates network variable *selectors*. Network variable selectors are 14-bit numbers used to identify network variables. All network variables in a connection must have the same network variable selector value. In addition, the assigned network variable selector must allow each device to uniquely associate an incoming network variable update with one of its network variables. As with network address assignment, a network tool is responsible for allocating group addresses, tracking group membership, assigning network variable selectors, and reassigning network variable selectors as needed to produce the desired logical connectivity. That is, for each network variable, the network tool must ensure that messages are only sent to, received by, and processed by the desired set of devices.

Network addresses may be defined in a number of ways, including the following:

- Programmed into the device when it is manufactured. This is typically used for closed, self-contained systems.
- Self-installed by each device during field installation. This is typically used for small, closed systems.
- Assigned by a network tool during field installation. This is used for most systems.

Each of these methods represents a trade-off in terms of ease of initial installation, flexibility, and cost of tools. The ANSI/EIA/CEA 709.1 protocol and these guidelines have been designed to make all of these installation scenarios compatible. Systems installed with one of the simple scenarios can migrate at a later date to a more sophisticated network-management scenario, without having to change device application code or hardware.

To correctly allocate network resources, such as device addresses and network variable selectors, a network tool must have the freedom to reassign resources as needed. To support interoperable systems, installation dependencies must not be built into the devices. A device's functional behavior must be independent of its address or the details of its connections to other devices. All messages should be sent using either *implicit addressing*, with addresses assigned by a network tool, or using *explicit addressing* where the explicit addresses are determined at installation time using configuration properties.

| Guideline 4.1: | A certified device application shall not be dependent upon its network configuration. |
|---|---|

## 4.1.1. Address-Table Entries

Each distinct implicit destination address for an outgoing network variable update, poll, or application message requires an address-table entry. In addition, each group to which the device belongs requires an address-table entry. The maximum number of address-table entries under ANSI/EIA 709.1-A is 15, and each requires five bytes of on-chip EEPROM. For devices that support ANSI/EIA/CEA 709.1-B, the limitation of 15 address-table entries has been raised to 65 535.

The number of address-table entries directly affects the ease of installation of the device, since network variable and message-tag binders may fail if there are an insufficient number of these entries on the device. However, in a memory-limited application, such as those implemented on a Neuron 3120 Chip, or similar, there is a tradeoff between application functionality and these table entries. Wherever possible, at least 15 address-table entries should be supported to avoid binder failure.

| Guideline 4.1.1: | Wherever possible, a certified device should have sufficient address-table entries to support every bindable network variable and message tag. This will often not be possible for ANSI/EIA 709.1-A devices. As a minimum, all certified devices shall support a number of address-table entries equal to the number of non-configuration network variables plus the number of bindable message tags, or the protocol limit (15 for ANSI/EIA 709.1-A or 65 535 for ANSI/EIA/CEA 709.1-B), whichever is fewer. |
|---|---|

## 4.1.2. Network Variable Aliases

Network variable aliases are another tool for the device developer to conserve address-table entries, and also to prevent limitations due to network variable connection constraints. For example, network variable aliases are required on a device when a single network variable output on the device must be connected to two or more network variable inputs on another device. For example, a single switch connected to a device containing four actuators—where the single switch must simultaneously control all four of the actuator inputs.

Network variable aliases can also conserve address-table and group-address entries on monitoring devices. For example, when an output network variable on a device is connected to one or more other network inputs on another device—and that same output variable needs to be bound to the monitoring device—there are two alternatives:

- Have the monitoring device be a member of the group in the original connection.
- Allocate an alias to the output network variable, and send the alias as a unicast update to the monitoring device (unicast addressing does not consume address-table entries on the receiver device).

Aliases are often required when installing devices in open networks. The number of aliases to implement on a specific device depends upon the application, the available device resources, and the network topology of the network where it is installed. For these reasons, the guideline regarding network variable aliases only requires that device developers provide a reasonable number by using their application knowledge and understanding, and taking into account the devices' available memory resources. In lieu of more specific guidance by the device's application, the following formula may be used as a rule of thumb for a minimum value:

```
NumAliases = (NVcount==0) ? 0 : min(62, 10+(NVcount/3));
```

| Guideline 4.1.2: | A certified device shall support a reasonable number of network variable aliases to avoid binding errors due to network variable connection constraints. |
|---|---|

## 4.1.3. Domain-Table Entries

With regard to the number of domain-table entries, it is often useful to have a device be a member of the zero-length domain so that it may be queried without knowing its Neuron ID. This is useful when the network database is lost and must be recovered from the network itself. While the Neuron ID may be acquired by activating the device's service pin, and the domain table read with a second command using the Neuron ID, the service pin may not be easily accessible on

devices in some applications. For example, the device may be on a roof or behind a wall. If it is inconvenient, or not practical, to activate the service pin on a device which has only a single domain-table entry, and that device's configured domain is unknown, then the device cannot be recovered. In these cases, the Query ID network-management message must be used to get the Neuron ID. While the service-pin message is always sent as a domain-wide broadcast on the zero-length domain, the Query ID network-management message is domain specific. Thus, a network tool must know one of the domains of the device to use the Query ID network-management message, or it must already know the Neuron ID. Since the zero-length domain is not typically used for normal system operation, the need for the second domain entry arises from the need for devices to be members of their own system domain and the zero-length domain so that the Query ID network-management message may be used on a known domain to assist in database recovery. Once the system domain is known, all devices that are members of that domain may be recovered.

> Guideline 4.1.3: A certified device shall support at least two domains.

## 4.1.4. Self-Installed Devices

A *self-installed device* updates its own network-addressing information based on local inputs—with no interaction with other devices on the network during the installation process. In a typical self-installed system, the only information set at installation time is a domain number and group number. The rest of the installation information—including the majority of the binding information—is set at the time of manufacture. The user interface at each device is usually very simple; for example, push-button switches, DIP switches, rotary switches, or a backplane slot ID.

Self-installed devices can communicate across an interoperable network in one of the following two ways:

- Using a subsystem gateway (described later).
- After re-installation onto the network using a network tool.

Each self-installed device must contain a Node Object functional block with a SNVT_config_src configuration network variable as specified in the Node Object functional profile. When the device is manufactured, the value of this variable should be set to CFG_LOCAL to support self-installation. When the device is installed into a network using a network tool, the network tool changes the value to CFG_EXTERNAL to indicate to the device application that the network tool has taken over management of the device. The device application must not set its own network addresses when the SNVT_config_src input is set to CFG_EXTERNAL.

| Guideline 4.1.4: | A certified self-installed device shall implement a Node Object functional block with a SNVT_config_src input as described in 4.1.4, *Self-Installed Devices*. It must be possible to configure the device with a network tool, and have that device's address be set to any legal ANSI/EIA/CEA 709.1 Control Network Protocol address. |
|---|---|

### 4.1.5. Field-Installed Devices

Field-installed devices are installed using a network tool. The tool is typically one of the following two types:

- The tool is invisible to the user and is embedded in the network. It performs installation and maintenance behind the scenes. This is known as *automatic installation*. To the end user, the network appears to install itself. In reality, the tool is analyzing the network contents, and is automating installation based on a set of rules.
- The user interacts with the network tool to configure the network. In this case, the tool might be embedded in the network—for example, integrated into a monitoring and control station—or it might be a portable tool that is attached to the network only during installation and maintenance.

## 4.2. Passive Configuration Tools

A *passive configuration tool* is a network tool that can be used on a device to assist in the successful commissioning of the device without disrupting the operation of other network tools. It may be a plug-in, standalone software, hardware attachment, or other tool. A passive configuration tool has the following attributes and capabilities:

- It provides one or more means to monitor or alter configuration properties or network variables solely for the purposes of replacing, commissioning, or installing the device.
- It may be used for device-specific configuration or monitoring.
- It does not interfere with other tools or network management devices.
- It does not make changes to any network-configuration information (for example, address-table entries) on any device both installed and not installed on the network.
- It leaves a device in the same state as it found it; however, during its operation, it is free to modify the device's state and reset the device in the course of modifying the configuration properties.
- In recognition of the fact that a passive configuration tool may take a device offline or reset a device, there can be system-level disruptions while using a

passive configuration tool on a device without first coordinating the activity with the other devices, systems, or system operators that depend upon the normal operation of the device.

- It is available to anyone owning the device on equivalent business terms, and such availability must be demonstrably free of any discriminatory terms and conditions.

| Guideline 4.2: | If a passive configuration tool is required for successful commissioning of a certified device, the tool shall conform to the definition of a passive configuration tool in 4.2, *Passive Configuration Tools*. |
|---|---|

## 4.3. Service Pin

The *service pin* is a physical or logical button on a device that causes the device to broadcast its Neuron ID and program ID. It is used during installation to uniquely identify a device and its application to a network tool. The network tool then uses Neuron ID addressing to assign a network address as described in 4.1, *Network Addressing*.

The method used to activate the service pin varies from application to application. Examples of mechanical methods include activating via an accessible push-button switch, or a magnetic-reed switch located within an enclosure. A service-pin message can also be sent under software control. For example, the device can send the message when the device is powered up or when a predefined series of I/O events occur. Sending the service-pin message exactly at power-up is not recommended because it will cause a spike in network traffic when power is restored after a power failure.

Even if a service pin will not be used as the default identification method for installing the device, some method for activating the service-pin input must be accessible to a maintenance technician. The service pin is a simple way to ensure that an installer can always identify, and thereby establish communication with, a given device. If necessary, the service pin can be located inside the device such that it is accessible only to service personnel. However, the activation of the physical or logical service pin at an asynchronous and arbitrary moment must not cause adverse device or network function. For example, activation of the service pin will not cause physical or logical reset of the device, nor will it cause extraneous network traffic.

| | |
|---|---|
| Guideline 4.3: | A certified device shall provide internal or external access to its service pin. The device shall respond with a service-pin message as defined in the ANSI/EIA/CEA 709.1 protocol when the service pin is activated. For example, the service pin may be activated when a service button is momentarily depressed. |

## 4.4. Gateways to Command-Based Systems

In a command-based system composed of multiple devices, commands are sent between the devices to initiate system actions. This implies that the devices sending and receiving the commands agree on the command semantics and actions. Building a gateway to such a system and simply propagating the command structure across the gateway would not allow the command-based system to interoperate with a LONMARK system because the LONMARK devices were not programmed to use these commands. In fact, to get interactions between the devices on both sides of the gateway, the LONMARK devices would have had to be designed to send and receive the other system's commands. Since LONMARK devices communicate via functional blocks, this method of gateway construction severely limits interoperability.

A better method for constructing a gateway to a command-based system is to think of the entire command-based system as a single LONMARK device with a set of standard functional blocks that accomplish the interoperable functions of the command-based system. Once this abstraction of the command-based system is defined, it then becomes the interface between the gateway device and the LONMARK devices. Within the gateway, translations between the commands and the LONMARK functional blocks are accomplished by the gateway software. In this way, knowledge of the command set is confined to the gateway and the command-based system. Any LONMARK device with functional blocks defined that are compatible with those defined on the gateway can interact with the command-based system without the foresight of the device developer.

This same technique may be used to create gateways to proprietary LONWORKS devices that do not meet the requirements of these guidelines. It can also be used to create gateways between network subsystems that are installed using a network tool, and those that are self-installed. This enables a proprietary device or a self-installed subsystem to be viewed as an interoperable subsystem—the proprietary or self-installed network is independently managed and it interfaces to other devices and subsystems through one or more gateway devices.

| Guideline 4.4: | Under no conditions shall a certified LONMARK gateway pass commands from a command-based system directly into a LONWORKS network. Instead, these commands shall be mapped to LONMARK standard and user functional blocks. |
|---|---|

## 4.5. Shared-Media Considerations

A power-line channel and a radio-frequency channel that contain devices within communicable range of several network tools are two examples of *shared-media* channels. When two or more network tools share such a medium, messages can leak between one tool and devices belonging to another tool. If the tools and installers do not directly coordinate their activities, the tools and devices must follow conventions to avoid conflicting network changes or installing the wrong devices. The following guidelines apply to devices on a shared medium. The term *shared media* refers not only to communications-medium sharing but also uncoordinated network-management activities as described above. It also refers to open, shared media, like a power-line channel or RF channel; and closed, shared media, like a twisted-pair channel.

If a foreign network tool inadvertently acquires a device and installs it with network-management authentication, the device's owner is unable to reclaim the device over the network. To prevent this, devices intended for installation on shared media must provide some means for locally causing the device to go unconfigured. An unconfigured device does not have a network address. For example, invoking the Neuron C **go_unconfigured()** function unconfigures a device and resets its authentication key, thereby allowing the device's owner to reclaim the device by reinstalling it. A typical implementation requires a pushbutton, often the service-pin button, to be pressed and held for 15 seconds, to cause the device to unconfigure itself.

| Guideline 4.5A: | A certified device that is intended for installation on shared media must provide some means for locally causing the device to go unconfigured. |
|---|---|

Since the service-pin message can be received by foreign network tools, a means is required for a network integrator to determine if the correct device was installed upon installing a new device. This can be provided by a *wink function* as defined in the ANSI/EIA/CEA 709.1 protocol. The wink function allows a network integrator to physically confirm that an intended device has been installed. Device winking, whether due to the installation protocol itself, or post-installation testing, may cause activity in an unintended device if an incorrect device was installed on a foreign network sharing the shared-media channel. Since the existence of the local network may not even be known to people working on the foreign network, the effects of

winking must be benign. For example, an LED may flash on a device, but a motor should not be powered on.

A device that responds to a wink command must automatically stop winking after a maximum of 30 seconds. A device must not require special means, like the receipt of a second wink command, to leave the wink state.

| | |
|---|---|
| Guideline 4.5B: | A certified device that is intended for installation on shared media must support the wink function as described in 4.5 and must provide a wink that does not create a potentially dangerous or costly situation if invoked at any arbitrary time in the operational life of the device. |

# Appendix A

## Glossary

# A.1. Definition of Terms

**ANSI/EIA 709.1-A** — A control-network protocol encompassing all seven layers of the ISO OSI protocol model. It is implemented in Neuron® Chip microprocessors and Echelon Smart Transceivers as Echelon's LonTalk Protocol. It can be ported to different processors.

**ANSI/EIA/CEA 709.1-B** — An enhanced version of the **ANSI/EIA 709.1** control-networking protocol that allows for greater than 15 address-table entries per device.

**Application set** — The object or objects to which a configuration property applies. The application set may be a network variable, a series or compilation of network variables, a functional block, a series or compilation of functional blocks, or the entire device.

**Base type** — A fundamental type that may be used as the basis of a network variable type or configuration property type. The available base types are defined in 3.1.1, *Base Types*.

**Certified device** — A device that has been certified by the LONMARK Interoperability Association to comply with the *LONMARK Application-Layer Interoperability Guidelines* and the *LONMARK Layer 1-6 Interoperability Guidelines*.

**Changeable-type network variable** — A network variable whose type can be changed during installation. See *Changeable-Type Network Variables* for details.

**Configuration property (CP)** — A data value used to configure the application program in a device. Configuration properties are used to set parameters such as maximum, minimum, default, and override values. They can be implemented using configuration network variables or as data items within configuration files. Configuration property data is kept in a device's non-volatile memory.

Configuration property interfaces are indicated with arrows with magenta shading in color versions of this document. Configuration property implementations are indicated with magenta rectangles with shadows.

**Configuration property member** — See *functional profile member*.

| | |
|---|---|
| **Configuration property member number** | See *functional profile member number*. |
| **Configuration property type index** | A 16-bit number that uniquely identifies a configuration property type within the scope defined by the scope number and program ID template of the resource file that contains the configuration property type definition. For example, the configuration property type index for the SCPTmaxSendTime type is 49 within the scope-0 standard resource file set. |
| **Device** | See *LONWORKS device*. |
| **Device channel ID** | A number that optionally specifies the channel to which a device is attached. |
| **Device class** | The Device Class field is a two-byte value identifying the primary function of a device. It is part of the SPID of the device. The value is drawn from a registry of pre-defined Device Class definitions that is maintained and published by the LONMARK Interoperability Association. |
| **Device interface** | The network-visible interface to a device consisting of the Neuron ID, program ID, channel ID, location field, device self-documentation string, device configuration properties, and functional blocks. |
| **Device-location field** | A string or number that optionally specifies the location of a device. |
| **Device self-documentation string** | A string that specifies the structure of the contents of the self-documentation strings, the functional blocks, and optionally describes the function of a device. |
| **Device subclass** | A two-byte value specifying the usage in the first byte and the channel type in the second byte. It is part of the SPID of a device. See the *usage* and *channel type* definitions. |
| **Dynamic network variable** | A network variable that is added to a device by a network tool after the device is installed. |
| **Format** | *In the context of the program ID*: A four-bit value defining the structure of the program ID and device self-documentation strings in the device. It is part of the SPID of a device. The format must be 8 or 9, where |

format 8 is reserved for devices that have completed certification by the LONMARK Interoperability Association, and format 9 is used for all other devices. Format 9 must be used for devices that will not be certified, for devices that will be certified but are still in development, and for devices that have not yet completed the certification process. Device formats 0 – 2, and 10 – 15 (0xA – 0xF) are reserved by Echelon for future use. Device formats 3 – 7 are used by network interfaces and legacy non-interoperable devices and must not be used for other interoperable devices.

*In the context of a resource file:* A string that provides formatting instructions for a network variable or configuration property type. Each network variable and configuration property type must have at least one format defined. This format describes how the value will be displayed to or entered by network integrators and network operators. It is possible to define multiple formats for a network variable type or configuration property type. Different formats can provide the information in a different order (if the value is a structure or union) or provide a different scaling factor (for example, the SNVT_temp_f network variable type has three formats, one for Fahrenheit, one for differential Fahrenheit, and one for Celsius).

**Functional block**

A functional block is a portion of a device's application that performs a task by receiving configuration and operational data inputs, processing the data, and sending operational data outputs. A functional block may receive inputs from the network, from hardware attached to the device, or from other functional blocks on a device. A functional block may send outputs to the network, to hardware attached to the device, or to other functional blocks on the device.

A functional block is an implementation of a functional profile. A standard functional block is also known as a *LONMARK object*. A standard or user functional block is also known as an *object*.

Functional block interfaces are indicated with rounded rectangles with light-blue shading in color versions of this document. Functional block implementations are

indicated with light-blue rectangles with shadows.

**Functional block index** A sequentially assigned number identifying a functional block implementation on a device. For Neuron C applications, the functional block index is assigned by the Neuron C compiler in the order of declaration. The first functional block on a device has index 0, the second index 1, *etc.* Also called the *global index*.

**Functional profile (FP)** A template that describes common units of functional behavior. Functional profiles are also known as *profiles*, *functional profile templates*, *FPs*, and *FPTs*. Standard functional profiles are also known as *LONMARK profiles*. Each functional profile consists of a profile description and a specified set of network variables and configuration properties designed to perform a single function on a device. The network variables and configuration properties specified by the functional profile are called the *functional profile members*. A functional profile specifies whether or not each functional profile member is mandatory or optional. A profile is uniquely identified by a program ID template, scope, and functional profile number.

**Functional profile key** See *functional profile number*.

**Functional profile member** A network variable or configuration property member of a functional profile. Each functional profile member is identified as mandatory or optional by the functional profile. Each member also includes a text description of the member for the functional profile. For example, the nviRequest member of the SFPTnodeObject functional profile defines it as being a SNVT_obj_request type and having to support RQ_NORMAL, RQ_UPDATE_STATUS, and RQ_REPORT_MASK inputs.

**Functional profile member number** A two-byte number that uniquely identifies a network variable or configuration property member of a functional profile. This member number is used to associate a network variable or configuration property on a device with the corresponding network variable or configuration property member of the functional profile. Member numbers must be in the range of 1 to 4095, and need not be contiguous. Member numbers must be

unique, with the exception that network variable and configuration property members may use the same number. There may be a maximum of 255 mandatory members and 255 optional members of each type (scope 0 NV, inheriting NV, scope 0 CP, and inheriting CP). A member number may be preceded by a *functional profile selector*. For example, the nviRequest member of the SFPTnodeObject functional profile has a network variable member number of 1.

**Functional profile number**

A two-byte number that uniquely identifies a functional profile within the scope defined by the scope number and program ID template of the resource file that contains the functional profile definition. For example, the functional profile number for the SFPTswitch profile is 3200. The functional profile number of the primary functional profile on a device can be used as the device class for the device. Functional profile numbers 0 through 99 (inclusive) cannot be used as device classes for a SPID. Functional profile number 5 is obsolete and cannot be used for any new devices certified under these guidelines. Functional profile number 5 cannot be used to implement the primary functional block on an existing device recertified to these guidelines. Also called the *functional profile key*, or the *FPT key*.

**Functional profile selector**

Associates a functional profile member with either the functional profile itself or a scope-0 profile. The functional profile selector may be an ASCII vertical bar ("|") or an ASCII number sign ("#"). If the functional profile selector is a vertical bar, the member number identifies a member of a scope-0 profile. If the functional profile selector is a number sign, the member number identifies a member of the inheriting profile. The number-sign functional profile selector is always used for members of user functional profiles, including profiles that do not use inheritance. The vertical-bar functional profile selector is always used for members of standard functional profiles. Two different functional profile members may have the same member number as long as they use different functional profile selectors. For example, the "|**1**" member of a functional profile is not the same as the "**#1**" member of the same profile. This prevents conflicts if new members are added to a

| | |
|---|---|
| | standard functional profile that has already been used as the basis for inheriting profiles. |
| **Functional profile template** | See *functional profile*. |
| **Global index** | See *functional block index*. |
| **Host** | A device implementing layer 7 of the ANSI/EIA/CEA 709.1 protocol. A host may be based on a Neuron Chip or Smart Transceiver, in which case it is called a *Neuron Chip-hosted device*. A host may be based on another processor, in which case it is called a *host-based device*. A host-based device uses a LONWORKS network interface to connect to a LONWORKS network. |
| **Host-based device** | A device that executes its application program on a processor that is not a Neuron Chip or a Smart Transceiver. |
| **Host selection** | A form of network variable selection that occurs on the host processor. When using host selection, the host processor must manage the network variable configuration table. |
| **Inheriting Profile** | A functional profile that inherits members from a scope-0 profile. |
| **Interoperability** | A condition that ensures that multiple devices – from the same or different manufacturers – can be integrated into a single network without requiring custom device or tool development. |
| **LONMARK Association** | See *LONMARK Interoperability Association*. |
| **LONMARK brand** | A branding program for devices that have been tested and certified by the LONMARK Association for compliance to the LONMARK guidelines. |
| **LONMARK Interoperability Association** | The LONMARK Interoperability Association's mission is to enable the easy integration of multi-vendor systems based on LONWORKS networks. The Association provides an open forum for member companies to work together on marketing and technical programs to promote the availability of open, interoperable control devices. |

| | |
|---|---|
| **LONMARK object** | See *functional block*. |
| **LONWORKS device** | Hardware and software that runs an application and communicates with other devices using the ANSI/EIA/CEA 709.1 protocol. It may optionally interface with input/output hardware. A LONWORKS device includes at least one processor and a LONWORKS transceiver. A LONWORKS device typically includes a Neuron Chip or Echelon Smart Transceiver. Also called a *LONWORKS node*, or simply a *node*. Devices and device interfaces are indicated with green shading in color versions of this document. |
| **LONWORKS network** | A collection of intelligent devices that communicate with each other using the ANSI/EIA/CEA 709.1 protocol over one or more communications channels. |
| **Manufacturer ID (MID)** | A 20-bit number that uniquely identifies the device manufacturer of a device. It is part of the device's SPID. Manufacturer IDs are assigned by the LONMARK Interoperability Association. Permanent manufacturer IDs are assigned upon request from a LONMARK Partner or Sponsor Member, and are unique to that particular manufacturer. Temporary manufacturer IDs are assigned by request of any developer by submitting the form at www.lonmark.org/mid. Temporary MIDs are not guaranteed to be unique. |
| **Network-interface selection** | A form of network variable selection that occurs on the network interface. When using network-interface selection, the Neuron Chip or Smart Transceiver in the network interface manages the network variable configuration table. |
| **Network variable (NV)** | A data item that a particular device application program expects to get from other devices on a network (an *input network variable*) or expects to make available to other devices on a network (an *output network variable*). Examples are a temperature, switch value, and actuator position setting. Network variable data is typically stored in a device's volatile memory.<br><br>Network variable interfaces are indicated with arrows with yellow shading in color versions of this document. Network variable implementations are indicated with |

yellow rectangles with shadows.

**Network variable declaration** — The establishment of an instance of a network variable type within the code of an application. For example, "`network input SNVT_obj_request nviRequest;`" is a network variable declaration.

**Network variable index** — A sequentially assigned number identifying a network variable implementation on a device. For Neuron C applications, the index is assigned by the Neuron C compiler in the order of declaration. The first network variable on a device has index 0, the second index 1, *etc.*

**Network variable member** — See *functional profile member*.

**Network variable member number** — See *functional profile member number*.

**Network variable programmatic name** — The name assigned to a network variable implementation by the device application developer. The programmatic name is limited to 16 characters, including any optional prefixes. The programmatic name is not significant for interoperability, but conventions are suggested in 2.7.2.3, *Network Variable Naming Conventions*, to make programmatic names easier to use for integrators.

**Network variable selection** — The process of associating a network variable selector with a network variable on a device.

**Network variable type** — A specification of the length, units, valid range, and resolution of the data contained within a network variable. A network variable type may be a simple, one, two, or four-byte scalar type; or a more complex structure or union of up to 31 bytes.

**Network variable type index** — A 16-bit number that uniquely identifies a network variable type within the scope defined by the scope number and program ID template of the resource file that contains the network variable type definition. For example, the network variable type index for the SNVT_switch type is 95 within the scope-0 standard resource file set.

| | |
|---|---|
| **Neuron Chip-hosted device** | See *host*. |
| **Neuron ID** | A unique 48-bit identifier within the read-only data structure of a device as defined by the ANSI/EIA/CEA 709.1 protocol. It is also called the *unique node ID*. |
| **Node** | In common usage, a *node* is the same as a *device*. A more precise definition is that a *node* is a physical and logical presence on a LONWORKS network with a unique Neuron ID and network address. The Neuron ID relates to the identification of a single instance of an implemented ANSI/EIA/CEA 709.1 protocol stack. A *device* is also a network presence with an application processor and one or more nodes. A device with multiple Neuron IDs would consist of multiple nodes. Some infrastructure devices, such as routers, also consist of more than one Neuron ID and thus consist of multiple nodes. |
| **Object** | See *functional block*. |
| **Passive configuration tool (PCT)** | A network tool that can be used on a device to assist in the successful commissioning of the device without disrupting the operation of other network tools. It may be a plug-in, standalone software, hardware attachment, or other tool. A passive configuration tool has attributes and capabilities as defined in 4.2, *Passive Configuration Tools*. |
| **Primary functional block** | The functional block on a device that implements the most important function for the device. The primary functional block for a certified device must implement a standard functional profile. |
| **Primary functional profile** | The standard functional profile that defines the primary functional block on a device. |
| **Proprietary data** | Data and message definitions in the device interface that are known only to the manufacturer and the manufacturer's agents. Certified devices can contain proprietary data, however, there can be no requirement to access or modify the proprietary data in the course of successfully commissioning the device; and the lack of access to proprietary data must not prevent the |

| | |
|---|---|
| | successful operation or use of the device's published, interoperable functional blocks. |
| **Self-documentation string** | A text string associated with a device, network variable, or configuration property that is stored within a device and within the device interface (XIF) file for a device. Network tools can read the self-documentation strings from the device itself or from the device interface file. |
| **Self-documentation text** | Optional text within a device, network variable, or configuration property self-documentation string that provides documentation of the intended use of the device, network variable, or configuration property respectively for use by integrators. |
| **Shared media channel** | A communications channel where messages can leak between tools and devices belonging to different systems. |
| **Standard configuration property type (SCPT)** | A **configuration property** type that has been standardized by the LONMARK Interoperability Association. A SCPT is a standardized definition of the units, scaling, encoding, valid range, and meaning of the contents of configuration properties. |
| **Standard network variable type (SNVT)** | A **network variable** type that has been standardized by the LONMARK Interoperability Association. |
| **Standard program ID (SPID)** | An 8-byte number that uniquely identifies the device interface for a device, encoded according to rules specified in 2.3, *Standard Program ID*. |
| **Static network variable** | A network variable that is statically defined for a device; that is, a network variable that is not a dynamic network variable. |
| **Subsystem** | Two or more devices working together to perform a function and bearing fixed, pre-defined relationships to one another. A subsystem may use one or more ANSI/EIA/CEA 709.1 domains. |

| | |
|---|---|
| **Successful commissioning** | The process of taking a device and integrating it into a LONWORKS network. Successful commissioning means that the device can be physically installed in a network and made to perform its application function with the exclusive use of its device interface and a choice of third-party tools. |
| **System** | One or more independently managed subsystems working together to perform a function. A system may use one or more ANSI/EIA/CEA 709.1 domains. |
| **Unconfigured device** | A device without a valid network configuration. |
| **Usage** | A one-byte value describing the intended usage of the device. It is part of the SPID of a device. The Usage field consists of a one-bit Changeable-Interface flag, a one-bit Functional Profile-Specific flag, and a 6-bit usage ID. |
| **Usage ID** | A 6-bit value in the least-significant portion of the Usage field that identifies the primary intended usage of a device. |
| **User data** | User functional blocks, user network variables, and user configuration properties used by a device manufacturer to augment the device interface. These user data are data that have not been standardized by the LONMARK Association. It is allowable to have the manipulation of user data to be mandatory in order to be able to successfully commission a certified device. |
| **Wink function** | A function provided by a device that allows a network integrator to physically identify the device. For example, a wink function may blink an LED on the device. |

# Appendix B

## Language File Extensions

This appendix lists the file extensions used for language files as described in 3.1.3.

# B.1. Language File Extensions

Network variable types, configuration property types, functional profiles, enumeration types, and self-documentation strings can all reference text information used to describe their name, units, and function. This text information is contained in separate *language files*. There is one language file for every language supported by a resource file set. When a language file is translated, the language string references still point to the appropriate strings. The file extension of each language file depends on the language, and is one of the following:

**Table 4. Language File Extensions**

| *Language* | *Extension* |
|---|---|
| Czech | csy |
| Danish | dan |
| Dutch (Belgian) | nlb |
| Dutch (default) | nld |
| English (UK) | eng |
| English (US) | enu |
| Finnish | fin |
| French (Belgian) | frb |
| French (Canadian) | frc |
| French (default) | fra |
| French (Swiss) | frs |
| German (Austrian) | dea |
| German (default) | deu |
| German (Swiss) | des |
| Greek | ell |
| Hungarian | hun |
| Icelandic | isl |
| Italian (default) | ita |
| Italian (Swiss) | its |
| Norwegian (Bokmål) | nor |
| Polish | plk |
| Portuguese (Brazilian) | ptb |
| Portuguese (default) | ptg |
| Russian | rus |
| Slovak | sky |
| Spanish (default) | esp |
| Spanish (Mexican) | esm |
| Swedish | sve |
| Turkish | trk |

# Appendix C

## Self-Documentation Syntax and Data Representations

This section details the self-documentation string syntax required for Neuron C Version 1 and some host-based devices to declare functional blocks, network variables, and configuration properties.

# C.1. Device and Functional Blocks

The device self-documentation string specifies the self-documentation string structure, the functional blocks, and optionally describes the function of a device. Network access to the device self-documentation string is defined by the ANSI/EIA/CEA 709.1 protocol. This string is up to 1024 bytes in length and is created automatically by the Neuron C Version 2 (or newer) compiler. The device self-documentation string can be declared using Neuron C Version 1 syntax as follows:

**#pragma set_node_sd_string "***sdString***"**

The syntax for *sdString* is as follows:

**&3.3@***fblockList*[**;**[*selfDocText*]]

The components of the documentation string are the following:

- An ampersand ("**&**") prefix.
- A "**3.3**" substring identifying the major and minor version number of the guidelines implemented by the device.
- An at sign ("**@**") separator.
- *fblockList* is a list of functional profile numbers with optional array indices and names for each of the functional blocks implemented on the device. These numbers and names are delimited by a comma ("**,**"). The functional profile numbers must be listed in order of the functional block indices, with the first functional profile number corresponding to functional block index 0, the second to functional block index 1, *etc.* The Node Object functional block (SFPTnodeObject, or any UFPT inheriting from SFPTnodeObject) if implemented, must be first with an index of 0.

  An array of functional blocks can be specified by appending an opening left-square bracket ("**[**") and array dimension to the functional profile number. These may be followed by a closing right-square bracket ("**]**"), but the closing bracket may be omitted to save a byte of non-volatile memory.

  A device-specific name can be provided for a functional block by appending a string or string reference (as defined in 3.1.3.1, *Self-documentation String Reference*) immediately following the functional profile number and array dimension (if any). If the name is text, it must consist of printable characters, be of no more than 16 characters in length, and must not contain any left or right square brackets ("[" or "]"), commas, or periods, and it must not begin with any number. Functional block names can improve device usability, especially when there are multiple functional blocks of the same type. For example, naming each of multiple sensor functional blocks that report the same type of data may aid the installer in picking the correct functional block on a device.

- An optional semicolon (“**;**”) terminator. The terminator is required if any *selfDocText* self-documentation text is included.
- *selfDocText* is optional self-documentation text. A description of the intended device usage for network integrators. The self-documentation text may include references to language strings as described in 3.1.3.1, *Self-documentation String Reference*. A 0x80 value (represented as a “`\x80`” ASCII string) is reserved for these references. A 0x81 value (represented as a “`\x81`” ASCII string) is reserved for future expansion.

  **EXAMPLE**

  The following Neuron C Version 1 directive defines a device self-documentation string for a device with the following five functional blocks: one Node Object (profile 0), two Closed-Loop Sensors (profile 2), one Switch (profile 3200), and one manufacturer-specific functional block (profile 20001). The directive also specifies “XYZ Company Installation Text” as the self-documentation text for the device.

  ```
  #pragma set_node_sd_string "&3.3@0,2,2,3200,20001;
  XYZ Company Installation Text"
  ```

  The following Neuron C Version 1 directive adds the following functional block names to the previous example: “Node object,” “Widget Opener,” “Gadget Mover,” “Single-Pole X5,” and “XYZ Config”:

  ```
  #pragma set_node_sd_string "&3.3@0Node object,
  2Widget Opener,2Gadget Mover,
  3200Single-Pole X5,20001XYZ Config; XYZ Company
  Installation Text"
  ```

  The following Neuron C Version 1 directive changes the functional block names and self-documentation text in the previous example to string references in the scope-0 standard resource file set and a scope-3 user resource file set.

  ```
  #pragma set_node_sd_string
  "&3.3@0\x80468,2\x803:11,2\x803:12,3200\x803:13,
  20001\x803:14;\x803:10"
  ```

  The first string reference (“\x80468”) references the following string in the standard resource file set:

  ```
  …
  string 468: “Node object”
  …
  ```

  The remaining string references reference the following American English strings in a scope-3 user resource file set:

```
…
string 10: “XYZ Company Installation Text”
string 11: “Widget Opener”
string 12: “Gadget Mover”
string 13: “Single-Pole X5”
string 14: “XYZ Config”
…
```

The advantage of using resource strings is twofold. First, the string text can be removed from the device’s memory, thus saving memory space. Second, the strings from resource files can be translated to other languages by providing additional language files. For example, a French language file can be provided that translates each of the above strings to French as follows, with no changes necessary to the application program in the device:

```
…
string 10: “Texte D'Installation de XYZ Company”
string 11: “Ouvreur De Widget”
string 12: “Moteur De Gadget”
string 13: “Poteau Simple X5”
string 14: “Config de XYZ”
…
```

The following Neuron C Version 1 directive creates an array of 2 Closed-Loop Actuator functional blocks named “Widget Opener.”

```
#pragma set_node_sd_string "&3.3@0Node object,
2[2Widget Opener,3200Single-Pole X5,20001XYZ Config;
XYZ Company Installation Text"
```

## C.2. Network Variables

Documentation of network variables is accomplished through the use of network variable self-documentation (NVSD) strings. A network variable self-documentation string is used to define membership of a network variable to a functional block. Network access to the network variable self-documentation strings is defined by the ANSI/EIA/CEA 709.1 protocol.

**EXAMPLE**

The third functional block declared on a device (functional block index 2) is based on functional profile number 4, the Closed-Loop Actuator profile. This functional block has one mandatory input network variable, and two output network variables of the same type—one of which is mandatory and one of which is optional. The two output variables are of the same type, so it is important to know which network variable within the device corresponds to the first output versus the second output in the Closed-Loop Actuator functional profile.

This mapping of network variables to functional blocks, and to specific network variables within the functional block, is done within the self-documentation string.

Different self-documentation string formats are used for regular network variables, configuration network variables, and manufacturer-defined network variables as described below.

The syntax for a self-documentation string for a network variable or network variable array belonging to one or more functional blocks is as follows:

*@fbIndex*[–*endFbIndex*]||*#memberNumber*[**[***mArraySize***]**][**?**][**;**[ *selfDocText*]]

The components of the self-documentation string are the following:

- An ASCII at-symbol ("@") prefix.
- *fbIndex* is the functional block index of the functional block that contains the network variable or network variable array, or the index of the first functional block in a functional block array that contains the first network variable in a network variable array, if *endFbIndex* is specified. The first functional block on a device is index 0.
- *endFbIndex* is the functional block index of the last functional block in a functional block array that contains the last network variable in a network variable array. If a network variable array is specified and the *endFbIndex* value is omitted, the array is assumed to be a member array within the single functional block specified by *fbIndex*. The array size is required to convey multiple functional blocks each containing member arrays. In this case, the network variable array is mapped as if it were a two-dimensional array as follows:

  *nvName***[***fblockCount***][***memberArraySize***]**
- An ASCII vertical bar ("|") or ASCII number sign ("#") *functional profile selector*. If the functional profile selector is a vertical bar, the member number identifies a member of a standard profile (scope 0). If the functional profile selector is a number sign, the member number identifies a member of a user profile (scope 3 to 6).
- *memberNumber* is the network variable member number within the functional profile, or the index of the first member number in a member array if *mArraySize* is specified.
- *mArraySize* specifies the array size for a member array.
- An optional ASCII question mark ("**?**") *changeable-type specifier*. The changeable-type specifier must be included if the type of the network variable may change after installation.
- An optional semicolon ("**;**") terminator. The terminator is required if any *selfDocText* self-documentation text is included.
- *selfDocText* is optional self-documentation text. A description of the intended network variable usage for network integrators. The self-documentation text may include references to language strings as described in 3.1.3.1, *Self-*

*documentation String Reference*. A 0x80 value (represented as a "`\x80`" ASCII string) is reserved for these references. A 0x81 value (represented as a "`\x81`" ASCII string) is reserved for future expansion.

**EXAMPLES**

The following Neuron C Version 1 code maps network variables to network variable members of the third functional block declared on the device (functional block index 2). This functional block is based on the Closed-Loop Actuator profile. The following code maps the nviValue network variable to network variable member number 1 within the functional profile, the nvoValueFb network variable to network variable member number 2 within the profile, and the nvoActPosnFb network variable to network variable member number 4 within the profile.

```
network input sd_string("@2|1") SNVT_lev_cont nviValue;
network output sd_string("@2|2") SNVT_lev_cont nvoValueFb;
network output sd_string("@2|4") SNVT_ lev_cont nvoActPosnFb;
```

The following Neuron C Version 1 example maps an nviValue network variable to network variable member number 1 of the Closed-Loop Actuator profile. This example includes the optional documentation for the network variable.

```
network input sd_string("@2|1;boiler pressure")
SNVT_press_p nviValue;
```

The following Neuron C Version 1 example declares the mandatory output network variable of an Open Loop Sensor profile with an initial SNVT_temp_p type and specifies that the type may change to a different type at installation time.

```
network output sd_string("@1|1?") SNVT_temp_p nvoValue;
```

The following Neuron C Version 1 example shows a network variable array mapped onto a functional block array. Each of functional blocks 3 through 6 has a member number of 5. Member 5 of functional block 3 is nviExample[0]. Member 5 of functional block 4 is nviExample[1], *etc.*

```
network input sd_string("@3-6|5") SNVT_press_p nviExample[4];
```

# C.3. Configuration Properties

Documentation of configuration properties implemented as configuration network variables is accomplished through the use of network variable self-documentation (NVSD) strings, but using a different syntax as detailed in this section. Documentation of configuration properties implemented within configuration files is accomplished through the use of declaration strings within the configuration file. In either case, the documentation defines whether the configuration property applies to the entire device, one or more functional blocks, or one or more network variables. Configuration properties must be documented if they are to be part of the interoperable interface of a certified device.

The syntax for a documentation string for a configuration property (which may be a configuration property array) implemented as a configuration network variable is as follows:

**&**_header_**,**[_select_]**,**_flag_**,**_index_**,**[_dim_]**,**[_rangeMod_]**,**[**?**][**;**[_selfDocText_]]

The syntax for a documentation string for a configuration property (which may be a configuration property array) implemented within a configuration file is as follows (the only differences are removal of the ampersand prefix and addition of the *length* value):

_header_**,**[_select_]**,**_flag_**,**_index_**,**_length_**,**[_dim_]**,**[_rangeMod_]**,**[**?**][**;**[_selfDocText_]]

The components of the configuration property documentation string are the following:

- An ampersand ("**&**") prefix. It is only included for a configuration network variable.
- *header* specifies whether the configuration property applies to the entire device ("**0**"), a functional block or functional blocks ("**1**"), or a network variable or network variables ("**2**") on the device.
- *select* optionally specifies to which functional blocks or network variables the configuration property applies. This field is not specified if the configuration property applies to the entire device. The field values are listed in Table 5. A single configuration property may apply to multiple network variables or functional blocks. Therefore, unlike a network variable, a single configuration property may correspond to multiple members of multiple functional profiles. The association with the member or members in the functional profile or profiles is made by matching the type of a configuration property and the application set objects to which it applies. This means that multiple functional profile configuration property members may be implemented by a single configuration property.

**Table 5. Select-Field Values**

| ***CP Applies To*** | ***Select-Field Value*** |
|---|---|
| Entire device | Null |
| One functional block | Functional block index. If the CP is an array, all elements of the array apply to the functional block. |

| *CP Applies To* | *Select-Field Value* |
|---|---|
| Contiguous series of functional blocks, with the CP shared by all functional blocks | First functional block index and last functional block index separated by a hyphen ("–"): *firstFbIndex–lastFbIndex*. If the CP is an array, all elements of the array are shared by all functional blocks. |
| Contiguous series of functional blocks, with the CP divided among all functional blocks | First functional block index and last functional block index separated by a tilde ("~"): *firstFbIndex~lastFbIndex*. This option may only be used for CP arrays. |
| Non-contiguous compilation of functional blocks, with the CP shared by all functional blocks | Functional block indices separated by periods ("."). If the CP is an array, all elements of the array are shared by all functional blocks. |
| Non-contiguous compilation of functional blocks, with the CP divided among all functional blocks | Functional block indices separated by slashes ("/"). This option may only be used for CP arrays. |
| One network variable | Network variable index. If the CP is an array, all elements of the array apply to the network variable. |
| Contiguous series of network variables, with the NV shared by all functional blocks | First network variable index and last network variable index separated by a hyphen ("–"): *firstNvIndex–lastNvIndex*. If the CP is an array, all elements of the array are shared by all network variables. |
| Contiguous series of network variables, with the NV divided among all functional blocks | First network variable index and last network variable index separated by a tilde ("~"): *firstNvIndex~lastNvIndex*. This option may only be used for CP arrays. |
| Non-contiguous compilation of network variables, with the NV shared by all functional blocks | Network variable indices separated by periods ("."). If the CP is an array, all elements of the array are shared by all network variables. |

| CP Applies To | Select-Field Value |
|---|---|
| Non-contiguous compilation of network variables, with the NV divided among all functional blocks | Network variable indices separated by slashes ("/"). This option may only be used for CP arrays. |

- *flag* is a two-digit hexadecimal number encoded as a sequence of two ASCII digits. The first digit specifies the scope of the resource file set that defines the configuration property type. The value may be a "0", "3", "4", "5", or "6" digit as defined in 3.3, *Managing Resource Files*. The second digit encodes the flags described in 2.7.3.4, *Configuration Property Flags*. The values for each flag are listed in Table 6. These values may be or'd together, with the exception of the flags identified as exclusive. For example, both the Device-offline and the FB-disabled flag may be specified by or'ing 0x82 with 0x81, yielding a value for the second byte of 0x83. At least one of the values in Table 6 must be specified. When specified in a C or Neuron C application, the value must be encoded as "**x***hexDigits**"**, where *hexDigits* are the two hex-encoded values for the second byte.

**Table 6. Flag Values**

| Flag | Value | Exclusive |
|---|---|---|
| Constant | 0x84 | Yes |
| Device-offline | 0x82 | No |
| Device-specific | 0xA4 | Yes |
| FB-disabled | 0x81 | No |
| Manufacturing-only | 0x90 | No |
| No restrictions | 0x80 | Yes |
| Reset-required | 0x88 | No |

- *index* specifies the configuration property index within the specified resource file set. For example, the configuration property index for the SCPTmaxSendTime type in the standard resource file set is specified as "49".
- *length* specifies the configuration property size in bytes. If a CP array is specified using the *dim* field, the length refers to only one element of the array. The *length*

field is only specified for a configuration property implemented within a configuration file.

- *dim* optionally specifies the dimension of a configuration property array. If not specified, the configuration property is not an array. If specified, the dimension must be at least two.
- *rangeMod* optionally narrows the range specified by the configuration property definition in the functional profile, or specified by the configuration property type. The rangeMod value is a string as described in 2.7.3.6, *Configuration Property Initializers and Range Modifiers*.
- An optional question mark ("**?**") *changeable-type specifier*. The changeable-type specifier must be included if the type of the configuration property may change after installation. The changeable-type specifier is required for configuration properties with inheritable types (like SCPTlowLimit1) if the configuration property is implemented as a configuration network variable. If a configuration property is implemented within a configuration file, the question mark is not needed since the base type of the configuration property is not specified in the configuration file.
- A semicolon ("**;**") terminator.
- *selfDocText* is optional self-documentation text. A description of the intended configuration property usage for network integrators. The self-documentation text may include references to language strings as described in 3.1.3.1, *Self-documentation String Reference*. A 0x80 value (represented as a "`\x80`" ASCII string) is reserved for these references. A 0x81 value (represented as a "`\x81`" ASCII string) is reserved for future expansion.

Each field in the documentation string is delimited by a comma ("**,**"). Two consecutive commas ("**,,**") indicate that the field is null (empty or unspecified), except when a semicolon is encountered. In the event that a semicolon is encountered, all remaining fields of the string are considered null.

#### FIXED-TYPE EXAMPLES

The following configuration network variable self-documentation string declares a 31-byte SCPTlocation configuration property (CPT index 17 within the scope-0 standard resource file set) that applies to functional block index 0. No CP flags or range modifications are specified.

```
"&1,0,0\x80,17;"
```

The following configuration-file documentation string declares a 2-byte user configuration property (CPT index 1 within a scope-3 resource file set) that applies to the entire device. No CP flags or range modifications are specified.

```
"0,,3\x80,1,2;"
```

The following configuration-file documentation string declares a 10-element CP array, with each element requiring 2-bytes (CPT index 1 within a scope-3

resource file set) that applies to the entire device. No CP flags or range modifications are specified.

```
"0,,3\x80,1,2,10;"
```

The following configuration-file documentation string declares a 12-byte SCPTsetPnts configuration property (CPT index 60 within the scope-0 standard resource file set) that applies to functional block index 1. No CP flags are specified, but range restrictions are provided.

```
"1,1,0\x80,60,12,,|-2500:3000|-2750:3200|||:2000"
```

The SCPTsetPnts type is defined in the standard resource file set as follows:

```
typedef struct {
  signed long    occupied_cool;
  signed long    standby_cool;
  signed long    unoccupied_cool;
  signed long    occupied_heat;
  signed long    standby_heat;
  signed long    unoccupied_heat;
} SNVT_temp_setpt;
```

The range modification uses the default ranges for each member of the structure except for the standby_cool and unoccupied_cool members, and the high value of the unoccupied_heat member. The range modification can be interpreted as follows:

```
default:default|
standby_cool low:standby_cool high|
unoccupied_cool low:unoccupied_cool high|
default:default|
default:default|
default:unoccupied_heat high
```

#### CHANGEABLE-TYPE EXAMPLES

The following configuration-file documentation string declares a 12-byte SCPTmaxRnge configuration property (CPT index 60 within the scope-0 standard resource file set) that applies to functional block index 1. No CP flags are specified. The type is changeable and inherited from a changeable-type network variable with an initial type of SNVT_temp_setpt. A range modification is specified that uses the default low and high values, except for standby_cool and unoccupied_cool values, and the high value for unoccupied_heat.

```
"1,1,0\x80,20,12,,|-2500:3000|-2750:3200|||:2000,?"
```

The following configuration-file documentation string is the same as the previous example, deleting the range modifications and adding an array specification with a dimension of 2:

```
"1,1,0\x80,20,12,2,,?"
```

The following configuration-file documentation string is the same as the previous example, deleting the array dimension.

```
"1,1,0\x80,20,12,,?"
```

# Appendix D

## Host-Based Devices

This appendix describes additional guidelines for host-based devices. A *host-based device* is a device that executes its application program on a processor that is not a Neuron Chip or a Smart Transceiver. A host-based device may use a Neuron Chip or Smart Transceiver as a communications processor.

# D.1. Network Variable Selection

Host-based devices must be able to receive network variable updates from other devices over the network and must be able to send network variable updates to other devices over the network. To do this, host-based devices must correctly manage *network variable selectors* as defined by the ANSI/EIA/CEA 709.1 protocol. The process of associating a network variable selector with a network variable is called *network variable selection*. Network variable selection is performed by looking up a network variable selector in a *network variable configuration table*. A host-based device may implement network variable selection in one of two ways, depending on where network variable selection occurs: *host selection* or *network-interface selection*. When using host selection, network variable selection occurs on the host processor and the host processor must manage the network variable and alias configuration tables and preserve their contents across hardware resets and power cycles. When using network-interface selection, network variable selection occurs on the Neuron Chip or Smart Transceiver within the network interface, and the Neuron firmware manages the network variable and alias configuration tables.

| | |
|---|---|
| Guideline D.1A: | If a host-based certified device implements host selection, the host application shall manage the network variable and alias configuration tables and shall respond correctly to the Update and Query NV Config messages as specified by the ANSI/EIA/CEA 709.1 protocol. |

| | |
|---|---|
| Guideline D.1B: | If a host-based certified device implements host selection, the host application shall preserve network variable configuration table, alias configuration table, and configuration file contents across hardware resets and power cycles. |

A host-based device may support the following maximum numbers of network variables and alias table entries:

**Table 7. Maximum Number of Static NVs and Aliases**

| | Network-Interface Selection | Host Selection | |
|---|---|---|---|
| | | ANSI/EIA 709.1-A | ANSI/EIA/CEA 709.1-B |
| ***Maximum total NV count (static plus dynamic)*** | 62 (only static NVs possible) | 4 096 | 65 535 |
| ***Maximum number of aliases*** | 62 | 8 192 | 65 535 |

When using a network interface, the network processor within the network interface informs the host processor whether or not an incoming message is authenticated. It is the responsibility of the host application to determine whether authentication is required for a given network variable by checking the corresponding network variable configuration-table entry.

| | |
|---|---|
| Guideline D.1C: | When a host-based certified device receives a network variable update or poll request, the host application shall determine whether authentication is required for the update or poll request based on the authentication bit in the corresponding network variable configuration-table entry on the host. If a network variable is configured to be authenticated, then the host application shall reject an unauthenticated update or poll to that network variable. |

| | |
|---|---|
| Guideline D.1D: | When a host-based certified device receives a network variable fetch and network variable poll request as specified by the ANSI/EIA/CEA 709.1 protocol, the host application shall respond with the appropriate data from the requested network variable. If the application is in the Offline mode when a network variable poll request is received, the response shall contain no data. Otherwise, the response shall contain the correct number of bytes of data as specified by the network variable type. |

| | |
|---|---|
| Guideline D.1E: | The host application for a host-based certified device shall deliver the niONLINE and niOFFLINE commands to the network interface when the Online and Offline Device Mode messages are received. |

## D.2. Device Interface

A host-based certified device must meet all the device interface requirements in Chapter 2, *Device Interfaces*. The section lists additional device interface requirements for host-based devices.

Certification for a host-based device is granted to the combination of the host application and the network interface. If either of these components is absent, the device should not appear to be certified. If the network interface can be detached from the host (*e.g.*, if it is a Serial LonTalk Adapter), then the SPID must be loaded into the network interface each time the host application is launched.

| | |
|---|---|
| Guideline D.2A: | The host application for a host-based certified device shall ensure that a certified standard program ID (SPID) is present in the network interface during normal operation of the device. |

| | |
|---|---|
| Guideline D.2B: | The host application for a host-based certified device shall implement the self-documentation requirements specified in Appendix C. It shall respond to the Query-SNVT request message with the correct data as specified by the ANSI/EIA/CEA 709.1 protocol. |

## D.3. Dynamic Network Variables

A host-based device that implements host selection may implement dynamic network variables. A *dynamic network variable* is a network variable that is added to a device by a network tool after the device is installed. These network variables may be created and deleted at will, rather than being statically declared. A network variable that is not dynamic is called a *static network variable*. The only static declaration required for a host-based device is the maximum number of dynamic network variables and aliases supported on the device. This information appears in the device interface (XIF) file for the device, and it can be queried from the device using the commands described in this section.

Support of dynamic network variables is optional; however, if a device can dynamically create and delete network variables after being installed in a network, then the method described in this section must be used. A device that does not support dynamic network variables may ignore the commands described in this section.

| Guideline D.3: | If a host-based certified device implements dynamic network variables, the implementation shall conform to requirements listed in D.3, *Dynamic Network Variables*. |
|---|---|

A host-based device that supports dynamic network variables must implement the following:

**1** The Changeable-Interface flag must be set in the program ID as described in 2.3.4.1, *Changeable-Interface Flag*.

**2** The device must have a version 4.0 or later device interface file as specified in the *LONMARK Device Interface File Reference Guide*. The device interface file must specify the static and maximum dynamic portions of the interface.

**3** The device must support and respond to the extended network-management commands to manage dynamic network variables including the commands to add and delete network variables and aliases, to query their attributes, and to bind them. These extended commands are based upon the Install command defined by the ANSI/EIA/CEA 709.1 protocol, and are listed in the next section.

## D.4. Extended Network Management Commands

The *extended network management command*s are an extension of the ANSI/EIA/CEA 709.1 protocol Install command (message code 0x70). They provide methods to query self-identification/self-documentation (SI/SD) data, update SI/SD data, inform the device of a new network variable addition, and remove an existing network variable. Optional methods are also provided to increase the capacity of the domain and address tables, as well as other features. The syntax and usage of the commands are described in the *Install* and *Install Command Data Structures* sections of the ANSI/EIA/CEA 709.1-B protocol specification.

Host-based devices must implement support for a Wink request, which is the APP_WINK (0) application command within the Install command. Host-based devices that support dynamic network variables must also support the following additional application commands within the Install command:

| | |
|---|---|
| APP_NV_DEFINE (2) | Create a new dynamic network variable declaration. |

| | |
|---|---|
| APP_NV_REMOVE (3) | Remove an existing dynamic network variable declaration. |
| APP_QUERY_NV_INFO (4) | Query SI/SD data for a network variable. |
| APP_QUERY_DEVICE_INFO (5) | Query SI/SD data for the device. |
| APP_UPDATE_NV_INFO (6) | Update SI/SD data for a network variable. |

The command formats, data values, and additional commands to support expanded capacities and other extended features are described in the ANSI/EIA/CEA 709.1-B protocol specification.

Except for the Wink command, the extended commands require the device to implement a version-2, or newer, SI data structure as described in the next section.

## D.5. Version 2 SI-Data Structure

The following SI data structure diagram in Figure 10 is a representation of the version 0 and 1 data structures outlined in ANSI/EIA 709.1-A protocol.

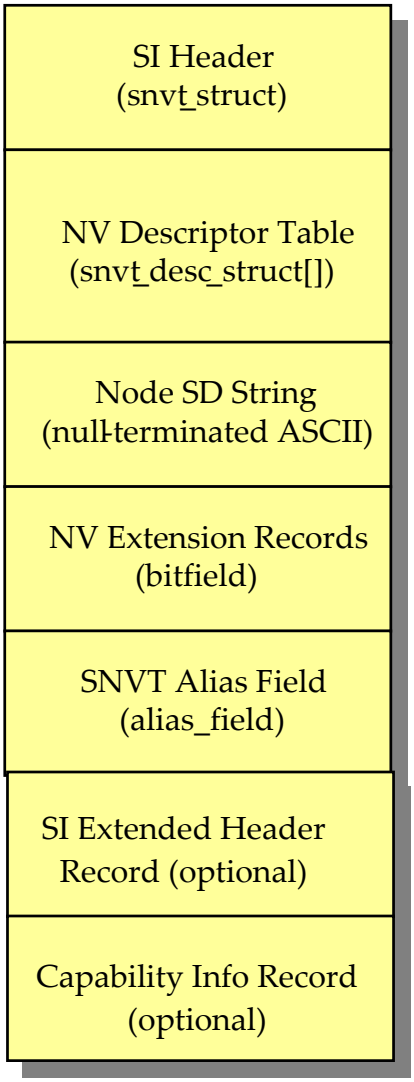


**Figure 10: Version-0 and –1 SI-Data Structures**

The optional SI Extended Header Record enhances parsing of SI data. The optional Capability Info Record indicates the expanded domain and address table capacities. These records are optional; if they are not present, then the SI Header's length field should not include the size of the records. The records are described in the *SI Extended Header Record* and *Capability Info Record (snvt_capability_info)* sections of the ANSI/EIA/CEA 709.1-B protocol specification.

The version 2 SI-data structure is shown in Figure 11.

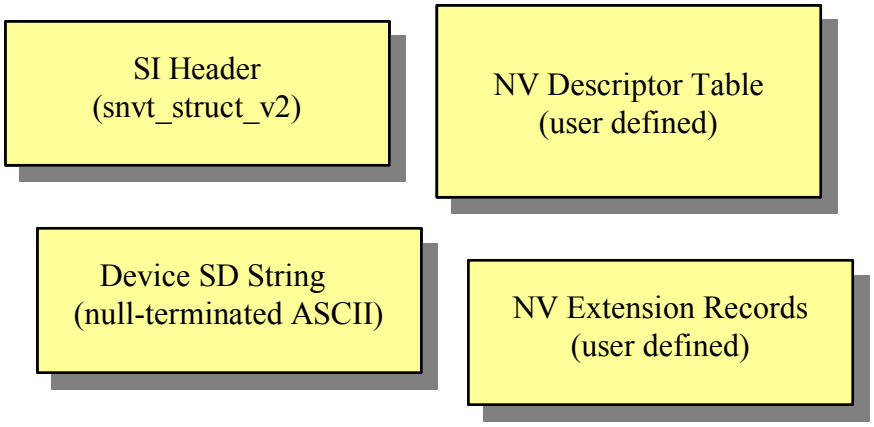


**Figure 11: Version-2 SI-Data Structure**

Each of the sections is separate from the others. The organization and implementation of each is left to the device developer. This is because the Query SNVT command only reads the SI Header.

The NV descriptor table, device SD string, and extension records are accessed via dynamic network variable protocol commands that the application must process. The device's responses to these commands are sent back to the network, and they must follow the dynamic network variable protocol. Thus, the developer is free to choose the most appropriate method of internal representation. However, the header of the SI data, containing the version of the SI data, must be formatted as shown in the code example below.

This structure is requested multiple bytes at a time by the Query SNVT command, and is then parsed by a network tool. The data types given are Neuron C types. These must be translated as appropriate for other platforms when responding to the Query SNVT command, taking into account big-endian vs. little-endian storage.

The capability info record (**cap** field) indicates the expanded address table and domain table capacities as well as additional features of the device. This record is optional; if it does not exist, then the SI Header's length should not include the size of the record. The record is described in the *Capability Info Record (snvt_capability_info)* section of the ANSI/EIA/CEA 709.1-B protocol specification.

```
typedef struct {
    unsigned length_hi;
           /* length of header only. Others are read via the WINK's subcommands */
    unsigned length_lo; /* APP_QUERY_NV_INFO and APP_QUERY_NODE_INFO */
    unsigned num_netvars_lo;
           /* Max # of NVs which can be defined (static + dynamic) */
    unsigned version; /* version 2 format */
    unsigned num_netvars_hi;
           /* Max # of NVs which can be defined (static + dynamic) */
    unsigned mtag_count;
    unsigned long static_nv_count;
    unsigned long current_nv_count; /* Number of currently defined NVs */
    unsigned long max_nv_in_use; /* Maximum NV index. 0xffff indicates none defined */
    unsigned long alias_count;
    unsigned long node_sd_text_length;
#ifdef LITTLE_ENDIAN
    unsigned unused :6;
    unsigned query_stats :1;
    unsigned binding_II :1;
#else
    unsigned binding_II :1;
    unsigned query_stats :1;
    unsigned unused :6;
#endif
    snvt_capability_info cap;     /* capability info record (optional) */
} snvt_struct_v2;
```

# Appendix E

## New Standard Profile and Type Proposal Procedure

If a standard functional profile, SNVT, or SCPT is not available to satisfy your product requirements, this appendix outlines the procedure for creating, proposing, and adopting new LONMARK profiles, SNVTs, and SCPTs.

# E.1. Submitting a New Proposal

Any member of the LONMARK Association may submit a proposal for a new or revised LONMARK profile, SNVT, or SCPT. To create a proposal, the proposal author must create a Zip archive containing proposed documentation and a proposed resource file set. The proposal author does not have the means to create a proposed scope-0 resource file set; the proposal may therefore be submitted with any suitable program ID template and a scope value of 3, 4, 5, or 6. The documentation must document the new profiles and types using the documentation template available from the Tech Resources section of the Members Area at www.lonmark.org/members/techinfo.cfm. Proposal.zip is the name of the template document archive containing the proposal templates. The resource file set must include any new profiles and types, including any required enumeration types and formats. The resource file set must meet the guidelines described in this appendix and in Chapter 3, *Resource Files*.

To submit the proposal, the author must post the proposal for 30-day member review and comment on the appropriate area of the LONMARK Member Forum. The appropriate area is typically a task-group conference area within the Member Forum. Proposals can also be sent to the Association Principal Engineer (tech@lonmark.org) if there is any doubt on which area of the Member Forum is appropriate. All SNVT and SCPT proposals must also be sent to SNVTrequest@lonmark.org for review. All members can comment on any proposals in the Member Forum. All comments should be posted as a thread to the initial posting in the Member Forum. Commenting is a conditional requirement to vote in the approval process.

Comments should be responded to as they are posted. Based upon the scope of the comments received, the author may prepare a revised proposal addressing the comments received, including a summary of the comments received and the resolution of each, and submit it for another review cycle. Alternatively, the proposal may be voted upon within the task group area of the Member Forum. The document author and the task group leader will jointly determine when a revised proposal is required and will determine when a proposal is ready for a task group vote.

When a task group vote is called for, a request for a vote will be posted within the thread of the original posting on the Member Forum. The call for a vote must include an end date for the voting period. The only votes required are from companies that had posted one or more comments (even if such comment was simply an approval of the original proposal). The voting period can end early if all required votes are received.

Once the Member Forum and SNVTrequest review is complete, the Principal Engineer submits the proposal for a two-week review by the Technical Advisory Committee. The Technical Advisory Committee consists of the Principal Engineer and five members appointed by the Board of Directors. Each year, up to three seats are made available to allow rotation of committee participants. The role of the Technical Advisory Committee is to advise the board on particular focus topics as requested by the Executive Director or Board of Directors, such as reviewing and approving profiles.

Based on the scope of comments received from the Technical Advisory Committee, the Principal Engineer may either request a revised proposal from the task group leader or author, or forward the final proposal along with recommendations to the Executive Director for forwarding to the LONMARK Board of Directors. Upon approval by the Board of Directors, the final proposal is then submitted to Echelon for incorporation into the standard resource file set.

## E.2. Contact

If you have any questions about this process or need assistance please contact the LONMARK technical staff using one of the following:

LONMARK Technical Services
E-mail: tech@lonmark.org
Tel: +1-408-938-5266
Fax: +1-408-790-3838

# Appendix F

## Requirements for Retesting, Upgrading, and Recertifying Devices

This appendix describes the situations where a device must change its standard program ID (SPID), undergo recertification, be retested by the LONMARK technical staff, and be upgraded from earlier versions of the guidelines.

# F.1. Certified Device and Resource File Changes

With the exceptions described in this appendix, any change to the device interface of a device or its device interface or resource files as defined in Chapter 2, *Device Interfaces*, and Chapter 3, *Resource Files*, requires a new standard program ID, recertification, and resubmittal to the LONMARK Association. A device interface change includes, but is not limited to, any addition, deletion, or re-ordering of network variables and configuration properties. A resubmittal consists of giving any modified files from the original certification to the LONMARK Association if a change has been made.

The following table lists exceptions to the above guideline. A check mark (☑) in a Required column means that the particular change requires a SPID change, recertification, or resubmittal as indicated by the check marks. Numbers refer to footnotes with detailed requirements.

The XIF Ref column identifies the corresponding field in the text-format XIF file. The following format is used for references to the global section of the XIF file: "*line*.*field*". For example, "6.3" represents global line 6 field 3. The following format is used for references to named sections of the XIF file: "*section*.*line*.*field*" where *section* is "**NV**" for the network variable section, "**MT**" for the message tag section, "**FILE**" for the file definition section, and "**NVVAL**" for the NV values section. For example, "NV.1.2" represents line 1 field 2 in the network variable section.

**Table 8. SPID, Recertification, and Resubmittal Requirements**

| *XIF Ref* | *Change* | *SPID Change Required* | *Recertification Required* | *Resubmittal Required* |
|---|---|---|---|---|
| 1 to 3 | XIF file header information changes as follows: changes to version numbers and other information in XIF file lines one through three, inclusive (including: XIF creation date, APC revision, and XIF version) | ☐ | ☐ | ☐ |
| 6.3 | Line 6, field 3 of the XIF file (whether a device can receive incoming application messages) | ☐ | ☐ | ☐ |
| 6.6 | Changes to the number of network input buffers | ☐ | ☐ | ☐ |
| 6.7 | Changes to the number of network output buffers | ☐ | ☐ | ☐ |
| 6.8 | Changes to the number of network priority output buffers | ☐ | ☐ | ☐ |

| *XIF Ref* | *Change* | *SPID Change Required* | *Recertification Required* | *Resubmittal Required* |
|---|---|---|---|---|
| 6.11 | Changes to the number of application input buffers | ☐ | ☐ | ☐ |
| 6.10 | Changes to the number of application output buffers | ☐ | ☐ | ☐ |
| 6.9 | Changes to the number of application priority output buffers | ☐ | ☐ | ☐ |
| 6.12 | Changes to a network input buffer size | ☐ | ☐ | ☐ |
| 6.13 | Changes to a network output buffer size | ☐ | ☐ | ☐ |
| 6.15 | Changes to an application input buffer size | ☐ | ☐ | ☐ |
| 6.14 | Changes to an application output buffer size | ☐ | ☐ | ☐ |
| 6.18 | Changes to the receive transaction buffer count | ☐ | ☐ | ☐ |
| 6.21 | Changes to statistics-relative address references | ☐ | ☐ | ☐ |
| 6.22 | Changes to maximum write-memory block size | ☐ | ☐ | ☐ |
| 7.1 | Changes from one Neuron Chip model to another: *e.g.*, 3150 to 3120E2 | **1** | **1** | **1** |
| 7.2 | Changes to the input clock speed: *e.g.*, 5MHz to 10MHz | ☐ | ☐ | ☐ |
| 7.3 | Changes to the firmware version: *e.g.*, 7 to 12 | **1** | **1** | **1** |
| 7.4 | Receive transaction block size | ☐ | ☐ | ☐ |
| 7.5 | Transaction control block size | ☐ | ☐ | ☐ |
| 7.6 | On-Chip RAM | ☐ | ☐ | ☐ |
| 7.7 | Off-chip RAM | ☐ | ☐ | ☐ |
| 7.8 | Domain-table entry size | ☐ | ☐ | ☐ |
| 7.9 | Address-table entry size | ☐ | ☐ | ☐ |

| *XIF Ref* | *Change* | *SPID Change Required* | *Recertification Required* | *Resubmittal Required* |
|---|---|---|---|---|
| 7.10 | Network variable configuration table size | ☐ | ☐ | ☐ |
| 7.11 | EEPROM size | ☐ | ☐ | ☐ |
| 7.12 | Alias table entry size | ☐ | ☐ | ☐ |
| 8.1 & 8.2 | Changes to the transceiver type with the same channel type: *e.g.*, FTT-10A to LPT-10 | ☐ | ☐ | ☑ |
| 8.4 to 11 | Changes to the channel type: *e.g.*, TP/FT-10 to PL-20N | ☑ | **2** | ☑ |
| 12 | Changes to the functional block names within the device self-documentation string (*e.g.*, FCUnit to FanCoilUnit) | ☐ | ☐ | ☐ |
| NV.1.1 | Changes to an NV programmatic name | ☐ | ☐ | ☑ |
| N/A | Adding or deleting a changeable-type flag to or from an NV | ☑ | **2** | ☑ |
| NV.1.3 & 1.4 | Changes to rate estimates for NVs | ☐ | ☐ | ☐ |
| NV 2.1 | Changes to offline update | ☐ | ☐ | ☐ |
| NV.2.5 | Changes to default NV service type (*e.g.*, acknowledged to unacknowledged) | ☐ | ☐ | ☐ |
| NV.2.7 | Changes to default authentication attribute of an NV | ☐ | ☐ | ☐ |
| NV.2.9 | Changes to default priority of an NV | ☐ | ☐ | ☐ |
| NV.2.6, 2.8, 2.10 | Changes to the configurable/non-configurable modifier for service type, authentication, or priority | ☑ | **2** | ☑ |
| NV.2.12 | Changes to whether an NV is synchronized | ☐ | ☐ | ☑ |
| MT.1.3 & 1.4 | Changes to rate estimates for message tags | ☐ | ☐ | ☐ |

| *XIF Ref* | *Change* | *SPID Change Required* | *Recertification Required* | *Resubmittal Required* |
|---|---|---|---|---|
| 12 | Change the guideline version number in the device self-documentation string: *e.g.*, "&3.2@0NodeObject,8020FanCoilUnit; Device SD Text" to "&3.3@0NodeObject,8020FanCoilUnit; Device SD Text" | **3** | **2, 3** | **3** |
| 12 | Changes to the self-documentation text in the device self-documentation string (text after the semicolon): *e.g.*, "&3.3@0NodeObject,8020FanCoilUnit; Node SD Text" to "&3.3@0NodeObject,8020FanCoilUnit; Device SD Text" | ☐ | ☐ | ☑ |
| NV.3-N | Changes to the self-documentation text in an NV self-documentation string (text after the semicolon): *e.g.*, "@0\|1;ObjRequest" to "@0\|1;Request" | ☐ | ☐ | ☑ |
| NV.3-N and FILE.2-N | Changes to the self-documentation text in a CP documentation string (text after the semicolon): *e.g.*, "&1,0,0\x80,49;NodeSendTime" to "&1,0,0\x80,49;-Send Time" | ☐ | ☐ | ☑ |
| N/A | Bug fixes that do not modify the device interface or resource files including the XIF file | **4** | ☐ | ☐ |
| N/A | Bug fixes that do not modify the device interface or resource files including the XIF file with the exception of changes to the model number field in the SPID. | ☑ | ☐ | ☑ |
| FILE.1 | Addition of CP template or values FILE tables to the XIF file | ☐ | ☑ | ☑ |
| NVVAL 1 | Addition of an NVVAL section to the XIF file | ☐ | **2** | ☑ |
| FILE.2 | Changes to existing CP value FILE entries in a XIF file | ☐ | ☐ | ☑ |
| NVVAL 2 | Changes to existing NVVAL entries in a XIF file | ☐ | ☐ | ☑ |
| N/A | Changes and fixes to existing format files | ☐ | ☐ | ☑ |

| XIF Ref | Change | SPID Change Required | Recertification Required | Resubmittal Required |
|---|---|---|---|---|
| N/A | Language-file strings added to an existing language file | ☐ | **2** | ☑ |
| N/A | Creation of an alternate language file (other language) | ☐ | ☐ | ☑ |
| N/A | Fixes (spelling, clarity, *etc.*) to existing language strings | ☐ | ☐ | ☑ |
| N/A | Changes to any portion of the resource file set not listed above | ☐ | **2** | ☑ |
| N/A | Changes to any part of the device interface not listed above | ☑ | ☑ | ☑ |

**Notes:**

**1** In general, a change in the Neuron firmware version or Neuron model number requires a new SPID, recertification, and resubmittal. This is because new firmware versions or Neuron models tend to have different capabilities and limitations of which network tools must be aware, and the SPID is the best way to communicate this information. An exception is an upgrade from firmware version 7 to version 12 or 13. A second exception is a change from a Neuron 3120 Chip to a compatible FT 3120 Smart Transceiver, or vice versa; or a change from a Neuron 3150 Chip to a compatible FT 3150 Smart Transceiver, or vice versa. These particular upgrades do not require a new SPID, recertification, or resubmittal. Echelon may identify additional firmware upgrade exceptions in the future. Contact the LONMARK Principal Engineer at tech@lonmark.org for a list of the current firmware upgrade exceptions.

**2** Full recertification is not required, but resubmittal is required and an administrative fee will be charged. The LONMARK technical staff will examine the resubmitted files for errors.

**3** In general, a change in the guideline version number requires a new SPID, recertification, and resubmittal. Exceptions are identified in F.2, *Upgrading to the Version 3.3 Guidelines*.

**4** A new SPID is not required, but SCPTminObjVer/ SCPTmajObjVer or SCPTminDevVer/SCPTmajDevVer configuration properties should be used for tracking revisions to functional blocks. See 2.8, *Device and Functional Block Versioning*, for guidelines on versioning.

# F.2. Upgrading to the Version 3.3 Guidelines

According to the LONMARK Logo License agreement, all devices certified to earlier versions of these guidelines must be recertified within 18 months of their initial certification, or six (6) months after the release of new guidelines—whichever is later. This section lists exceptions to this requirement. The following cases are exempted from the recertification requirement until the next release of the guidelines:

- All previously-certified devices that were certified to version 3.0 of the *LONMARK Application Layer Guidelines*.
- All devices certified to the version 3.1 or 3.2 *LONMARK Application Layer Guidelines* that meet all requirements of the version 3.3 guidelines with the exception of the guideline version number and the following guidelines: 2.5B, 2.7.2.2, 2.8, 2.9A, and 2.9B. This exemption does not apply to devices that use functional profile number 5, the Controller profile, to implement their primary function. All devices with Controller functional blocks implementing their primary function must be updated to comply with the version 3.3 guidelines with a new SPID, and must be resubmitted for certification. Devices that do not use the Controller functional profile to implement their primary function, and that satisfy this exemption, may be identified as complying with the version 3.3 guidelines, and the guideline version number may, at the manufacturer's discretion, be updated in the device self-documentation string and device interface (XIF) file.

# Index

## D



## E

## F



## G



## H



## I

## L



## M



## N

## O



## P



## Q



## R

## S



## T



## U

## V



## W



## X



## Z