Version 3.2 (updated June 2000, minor update March 2001, clarifications/addresses March 2002)

LONMARK® Application Layer Interoperability Guidelines

078-0120-01E

LONMARK Interoperability Association

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the LONMARK Interoperability Association.

Echelon, LON, Neuron, 3150, 3120, LonBuilder, LonTalk, LONWORKS, LONMARK, NodeBuilder, and LonManager are registered trademarks of Echelon Corporation. LonSupport is a trademark of Echelon Corporation. Other brand and product names are trademarks or registered trademarks of their respective holders.

Printed in the United States of America.
Copyright ©1992 - 2002 by the LONMARK Interoperability Association.

LONMARK Interoperability Association

550 Meridian Avenue

San Jose, CA 95126, USA

Contents

1

Introduction

LONWORKS® technology makes possible a new generation of smart control products which, working together, can form a control network. LONWORKS technology makes it possible to design enhanced control systems using products from multiple vendors. The LonTalk® protocol that is implemented in firmware on the Neuron® Chip provides the foundation for LONWORKS interoperability. The LONMARK interoperability guidelines provide detailed explanations and technical insight on how to design an interoperable LONWORKS technology-based product. All products that carry the LONMARK logo are certified to verify compliance with these guidelines.

The LONMARK interoperability guidelines are presented in separate volumes for layers 1-6 and layer 7 of the LonTalk Protocol. This document provides the Layer 7, or application layer design guidelines that address interoperability design issues for configuring, managing and communicating with products on a LONWORKS network. The term *node* is used to define a product with its own interface to the network. In addition, a series of LONMARK Functional Profiles--that define interoperable standards for application-specific functions--are published by the LONMARK Interoperability Association. Up-to-date information on LONMARK functional profiles is available on the LONMARK web site www.lonmark.org.

Computer programs, or instruction sets, that run on Neuron Chips are developed using the Neuron C programming language. However, the application program is not visible to other nodes on the network. The network interface to a node's application program is provided by shared data items called network variables. This interface provides the logical connection to other products on a LONWORKS control network. In multi-vendor networks the design of this interface is vital to providing interoperability and easy integration. Standardization of the logical connections between nodes is an important element of designing for interoperability.

The following chapters provide detailed information on how LONWORKS technology-based products should be designed so that the application-layer interface will support easy interoperation across a LONWORKS network. The basis for application-layer interoperability is the use of LONMARK Objects based on network variables. LONMARK objects maximize the number of products that will work together by providing agreed-upon definitions on collections of network variables that are used to implement standard functions. LONMARK objects defined for a specific application function are described by functional profiles.

LONMARK objects make use of a predefined set of Standard Network Variable Types (SNVTs). SNVTs further support interoperability since they allow network variable values to be exchanged between nodes in standard engineering and other predefined units. The definition of many SNVTs includes units, a range, an increment, and an identification (ID) number.

For a large number of products, the Neuron Chip executes the application program and implements the LonTalk protocol. In some products, the application program may execute on a non-Neuron Chip host such as a PC-compatible system, minicomputer, or another microprocessor. These products still contain a Neuron Chip; however, in these cases the Neuron Chip host is used solely as a LonTalk protocol processor. Any special considerations that should be observed by products that use a host processor are also outlined in the following chapters. However, it is very straightforward to design nodes to be interoperable, whether the application runs on the Neuron Chip or on an attached host.

LONMARK Program

A product that is certified as complying with these interoperability 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 systems integrators that a product can be easily linked with other products in a multi-vendor network. Contact the LONMARK Association Principal Engineer for more information on LONMARK certification.

Principal Engineer
LONMARK Interoperability Association
550 Meridian Avenue
San Jose, CA 95126
Tel: +1 408 938 5266 Fax: +1 408 790 3838
e-mail: cert@lonmark.org

Definition of Terms

Interoperability between products can be achieved at many different levels. This document uses the following terms to discuss interoperability:

Host	A device implementing layer 7 of the LonTalk protocol. A host may be based on a Neuron Chip, in which case it is called a *Neuron Chip-hosted node*. A host may be based on another processor, in which case it is called a *host-based node*. A host-based node uses a *LONWORKS Network Interface* to connect to a LONWORKS network.
Interoperability	A condition that ensures that multiple nodes (from the same or different manufacturers) can be integrated into a single network without requiring custom node or tool development.
Node	A device implementing layers 1 through 6 of the LonTalk protocol including a Neuron Chip, transceiver, memory, and support hardware.
Node Application Interface	The network visible interface to a node consisting of LONMARK objects, network variables, configuration properties, and node self-documentation.

Passive Tool

A tool that can be used on a LonMark device to assist in the successful commissioning of the device such that the passive tool has the following attributes and capabilities:

a) the passive tool consists of one or more means to monitor or alter configuration parameters or network variables solely for the purposes of replacing, commissioning, and /or installing the device, and

b) the passive tool may be used for device-specific configuration or monitoring, and

c) the passive tool shall not interfere with other tools or network management devices, and

d) the passive tool shall not make changes to any network configuration information (for example, address table entries) on any device both installed or not installed on the network, and

e) the passive tool shall leave a node in the same state as it found it, however, during its operation, it is free to modify the node's state and reset the node in the course of modifying the configuration properties, and

f) in recognition of the fact that a passive tool may take a node offline or reset a node, there can be system level disruptions while using a passive tool on a node without first coordinating the activity with the other nodes or systems or system operators that depend upon the normal operation of the node.

Proprietary Data

Data and message definitions in the node interface that are known only to the manufacturer and the manufacturer's agents. LONMARK certified nodes may contain proprietary data, however, there can be no requirement to access or modify the proprietary data in the course of successfully commissioning the node, and the lack of access to proprietary data shall not prevent the successful operation or use of the node's published, interoperable objects.

Sub-system

Two or more nodes working together to perform a function and bearing fixed, pre-defined relationships to one another. A sub-system may use one or more LonTalk protocol domains.

Successful Commissioning	The process of taking a node and integrating it into a LonWorks control 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 application interface and a choice of third party tools.
System	One or more independently managed sub-systems working together to perform a function. A system may use one or more LonTalk protocol domains.
User-Defined Data	The node application interface may be augmented by a device manufacturer with user-defined network variable types (UNVTs), user-defined objects and/or user-defined configuration property types (UCPTs). These user-defined data are data that have not been standardized by the Association. It is allowable to have the manipulation of user-defined data to be mandatory in order to be able to successfully commission a LONMARK certified device. The use of proprietary data cannot be required for the successful commissioning of a LONMARK certified device.

Audience

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

Contents

The *LONWORKS Application Layer Interoperability Guidelines* provide design guidelines that must be followed to qualify for LONMARK certification of products.

- *Chapter 1* introduces the scope of the application layer design guidelines.
- *Chapter 2* outlines the elements of an interoperable application layer interface.
- *Chapter 3* provides an introduction to LONMARK objects and describes their use.
- *Chapter 4* describes the handling of configuration information.
- *Chapter 5* describes product documentation.
- *Chapter 6* describes the requirements for host-based nodes.

- *Chapter 7* describes the design guidelines to facilitate network installation and maintenance of interoperable nodes.
- *Appendix A* describes the process for defining new SNVTs and SCPTs.
- *Appendix B* describes the process for defining new functional profiles.

Related Documentation

- ANSI/EIA 709.1-A specifies the services available at each of the seven layers of the LonTalk protocol embedded within every Neuron Chip.
- The LONMARK*Layers 1-6 Interoperability Guidelines* provide the interoperability design guidelines for layers 1-6 of the LonTalk Protocol.
- *The SNVT Master List* help file (SNVT.HLP and/or SNVT.PDF) documents the range, units, and resolution of all defined SNVTs.
- *The SCPT Master List* help file (SCPT.HLP and/or SCPT.PDF) documents the range, units, and resolution of all defined SCPTs.
- The LONMARK*Program Overview* describes the organizational structure and membership options of the LONMARKInteroperability Association, and rules for use of the LONMARKLogo.
- The LONMARK*Functional Profile*s provide detailed descriptions of all defined LONMARKobjects. Up-to-date documentation on all available functional profiles is available on the LONMARK web site www.lonmark.org.

2

The Application-Layer Interface

The LonTalk protocol employs a data-oriented application layer that supports a paradigm of communication of data, rather than one of commands between nodes on a LONWORKS network. In this approach, application data such as temperatures, pressures, states, and text strings can be sent to multiple nodes– each of which may have a different application for that datum. The data in the LonTalk protocol are called network variables and configuration properties. Standard Network Variable Types (or SNVTs) and Standard Configuration Parameter Types (SCPTs) provide a common framework for representing a wide range of data by specifying units, range, and resolution.

At the application layer, interoperability between LONWORKS technology-based products is facilitated through the use of application-specific LONMARK objects, generic LONMARK Objects, and SNVTs. LONMARK objects build upon network variables and provide a concise application-layer interface that incorporates semantic meaning for application-specific functions. LONMARK objects not only define which SNVTs and SCPTs to use to convey data but also provide semantic meaning about the information being communicated.

For communicating files of data between nodes, an interoperable file transfer mechanism is provided. This mechanism is managed via network variables, while the actual transfer of data is implemented with explicit messages, using a windowed protocol.

The Application-Layer Interface

The application-layer interface for an interoperable LONWORKS technology-based device comprises several elements, as shown in Figure 2.1. These elements fully describe the external interface of the interoperable node to the interoperable network. The interoperable interface is composed of several key elements:

- The node object (object 0), that provides for management of the individual objects within a node;
- Application-specific LONMARK objects;
- Generic LONMARK objects such as a sensor, actuator, or controller objects;
- Individual network variables;
- Configuration properties, and the interoperable file transfer mechanism as needed for the application program and for updating configuration parameter-based configuration data and product documentation.

The scope of the LONMARK Application Layer Guidelines is the node's application interface. The actual application software and hardware behind the interface is outside the scope of these guidelines. As such, the guidelines attempt 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 interfaces that describe the function of the device and how it exchanges information with other devices on the LonWorks network. The ability to choose among suppliers for system maintenance is realized by ensuring that LONMARK devices do not require any private information to be successfully commissioned.

LONMARK certified devices may contain proprietary data that are known only to the device manufacturer and the manufacturer's agents. These proprietary data are outside the scope of the LONMARK Application Layer Guidelines; however, the use of proprietary data cannot be required for the successful commissioning of the device.

The mandatory portion of the LONMARK application layer interface shall be composed of standard network variables, standard objects, and standard configuration properties, however, these standard interfaces may be augmented by a device manufacturer with user-defined network variables, user-defined objects and user-defined configuration properties. These user-defined data shall either have a public definition and associated documentation, or the data shall be accessible via the use of a passive tool. Thus, user-defined data are distinct from a manufacturer's proprietary data because user-defined data are intended to be manipulated by parties other than the manufacturer or the manufacturer's agents. It can be necessary to manipulate user-defined data to successfully commission a LONMARK certified device.

A brief description of each of the five components of the application interface is provided in the following sections– the detailed description is given in later chapters.

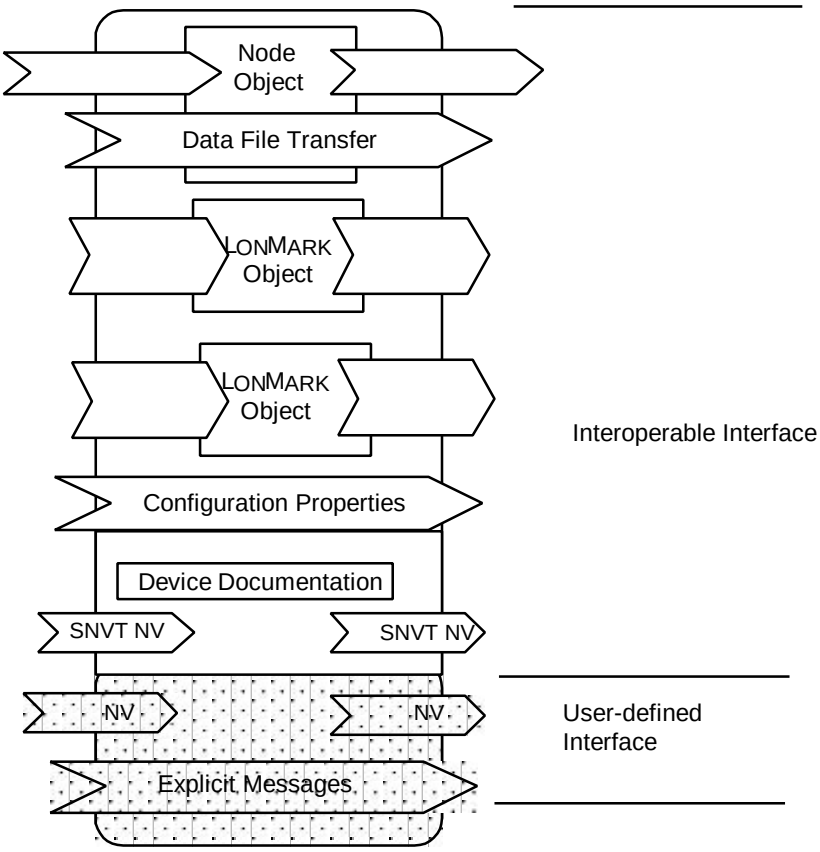

Figure 2.1 LONMARK Application-Layer Interface

Node Object

The node object provides the mechanism for requesting object modes and for reporting status of objects within a node. In addition, the node object includes network variables and configuration properties relating to the node as a whole, such as for network management support.

LONMARK Objects

LONMARK objects form the basis of interoperability at the application layer. The LONMARK objects describe standard formats for how information is input to and output from a node, and, how information is shared with other nodes on the network. LONMARK objects are defined as a set of one or more network variable inputs and/or outputs, implemented as SNVTs (with semantic definitions relating the behavior of the object to the network variable values) along with a set of configuration properties. By using turnaround network variables, a LONMARK object can send information to itself, however, the primary purpose of a LONMARK object is to send information to other nodes on the network. To provide for future expansion and to enable manufacturer differentiation, the LONMARK object definitions comprise both mandatory and optional network variables as well as a configuration section.

Included in this document is a set of generic LONMARK object types that can be used by developers as building blocks for the application-layer interface. The definitions are generic to apply to a broad range of applications, i.e., rather than exhaustively list the definitions for a temperature sensor, pressure sensor, flow sensor, etc., a more general "Sensor Object" is described. Details of the generic LONMARK object types are provided in Chapter 3.

In addition, more specialized object definitions are detailed in the LONMARK Functional Profile documents published and added to periodically by the LONMARK industry task groups. The functional profile definitions have been developed by the LONMARK Association members to specify LONMARK objects for application-specific functions. Up-to-date information on published LONMARK functional profiles is available on the LONMARK web site, which can be accessed at www.lonmark.org.

Standard Network Variable Types (SNVTs)

The use of SNVTs allows products made by different manufacturers to correctly interpret and exchange data by establishing standard representation formats for data. To use SNVTs, data within the device are converted to the particular datum definitions of the SNVTs after raw measurements have been appropriately linearized, calibrated, and filtered. This allows hardware characteristics to be hidden from the network. For example, a thermistor-based temperature sensor can be interchanged with a thermocouple-based sensor if each of them generates a calibrated temperature of type `SNVT_temp`.

The available SNVTs are listed in *The SNVT Master List* as SNVT.HLP (a Windows[i] Help file) or SNVT.PDF (an Acrobat[ii] document file). The SCPT Master list is SCPT.HLP or SCPT.PDF. These lists are augmented quarterly to accommodate new requirements for standard datum representations. There are two system-tested SNVT/SCPT releases per year, and two beta/prototype releases per year. All units are *Système Internationale* (SI) except as indicated. If a SNVT or a SCPT is not already available to satisfy your product requirements, refer to Appendix A for information on adding new SNVTs and/or SCPTs.

Configuration Properties

Application developers have a choice of how an application's parameters, called *configuration properties*, are set during the installation process. Configuration network variables can be used for relatively small amounts of configuration information. This method has the advantages of network variables, e.g. self-identification, self-documentation, external interface file support, and a simplified method for sending and receiving data.

For applications with large amounts of configuration information, a more compact mechanism for configuring a node is via the use of *configuration parameters* that are downloaded to a node using the LONMARK interoperable file transfer protocol or accessed by network management read/write memory commands. Use of configuration parameters can free-up network variables and on-chip EEPROM in Neuron Chip-hosted application programs.

The methods for downloading configuration parameters are described in detail in Chapter 4.

As with network variables, which can either be of user-defined type or of a standard network variable type (SNVT), configuration properties can be also either of standard configuration property types (SCPTs) or of user-defined configuration property types (UCPTs). SCPTs provide standard definitions for commonly used configuration properties including hysteresis thresholds, and message heartbeat rates.

New SCPTS can easily be defined using the procedures described in Appendix A. Existing object definitions can be revised by the LONMARK Interoperability Association to incorporate new SCPTS as additional options. However, a new object definition is required if a SCPT is to be specified as mandatory.

Data Transfer

Guideline 2-1: The LonMark interoperable file transfer protocol must be used for communicating files of data. Data must be sent in

[i] Windows is a registered trademark of Microsoft Corporation.
[ii] Acrobat is a registered trademark of Adobe Corporation

32-byte packets (not including protocol overhead) with a window size of 6 packets.

Data logging and monitoring applications can require sending files of data between nodes on a network. Additionally, host-based applications, as well as some Neuron Chip-hosted applications, receive their configuration information as a file. All of these files are transferred interoperably using LONMARK interoperable file transfer protocol described in the Echelon Corporation *File Transfer* engineering bulletin (005-0025-01). The file transfer protocol breaks up datum files into packets containing 32 bytes of datum, then transfers them sequentially. The file transfer protocol also supports random-access reads and writes of a variable number of bytes.

The setup of the file transfer is implemented with network variables and the actual data-transfer is implemented with explicit messages using a windowed protocol.

Guideline 2-2: Message Code 0x3E is reserved for the exclusive use of LonMark Interoperable file transfer protocol. The LonMark Interoperable file transfer protocol is used for the transfer of the Configuration Template file and the Configuration Parameter Value file as well as any user-defined files.

Guideline 2-3: Message Code 0x4E is reserved for the transfer of the ANSI/ASHRAE 135-1995 BACnet® NPDU frames[iii].

Guideline 2-4: Message Code 0x4F is reserved for future use.

Device Documentation

A hierarchical device documentation structure is supported by the LonTalk protocol. To support installation, each LONMARK node must contain self-documentation and self-identification information that identifies a number of key pieces of information. This information includes:

- The manufacturer of the device,
- The type of device,
- The ID of the Neuron Chip within the device,
- Any functional profiles supported by the node, any generic LONMARK objects used within the device,
- Their object type,

[iii] ASHRAE and BACnet are registered trademarks of the American Society of Heating, Refrigerating and Air-Conditioning Engineers, Inc.

- Type information for any additional network variables supported outside of the standard objects.

All of this data may be accessed over the network using standard network management tools. The documentation structures are defined to support language-independent documentation and are detailed in Chapter 5.

Functional Profiles

The initial set of LONMARK objects described in this document are generic and can be applied to a broad range of applications, such as temperature sensors, pressure sensors, motor actuators, damper actuators, power meters, circuit breakers, switches, and dimmers. The initial set of LONMARK object definitions leave the input and output datum types intentionally open to interpretation based on the particular application. In many cases it is desirable and beneficial to manufacturers and equipment specifiers to define the data types and associated configuration properties of new LONMARK objects that deliver a basic unit of functionality in a particular context. A control system specifier can use these new objects, described by functional profiles, as a shorthand for specifying the functionality required in a device. LONMARK task groups provide the forum for such definition work.

The intention of functional profiles is not to define (or standardize) devices, rather the purpose is to enable industry groups to define a shorthand way to describe common units of functional behavior. This will ultimately ease the specification process and enhance interoperability without compromising the ability of specifiers to call for unique capabilities, or the ability of manufacturers to differentiate their products. Devices can support one or more functional profiles.

The generic LONMARK objects: open and closed loop sensor and actuator objects, and the controller object, form templates for additional objects and functional profiles. In the case of basic sensor and actuator devices, the object definition will probably mirror the generic LONMARK sensor and actuator objects very closely– perhaps only adding-in the specific datum types for input or output. Since the LONMARK controller object is very loosely defined, functional profiles are required to provide definition for the network variables and configuration properties associated with a particular controller function.

Functional profiles are developed via LONMARK task groups. For current information on profiles specified or under development please refer to the LONMARK web site located at www.lonmark.org.

3

LONMARK Objects

This chapter provides an introduction to LONMARK objects: how they are graphically represented, how they are defined, and how they are used. Definitions for the basic set of generic LONMARK objects are provided. Specific LONMARK objects, described by functional profiles, are documented separately. Up-to-date information on published LONMARK functional profiles is available from the LONMARK web site www.lonmark.org.

Object Definitions

Graphical Representation

A standard graphical representation of LONMARK objects is shown in figure 3.1. Input network variables appear on the left of the diagram, output network variables on the right. Associated configuration properties are shown in the lower section of the object and are separated into mandatory and optional sections. Hardware input (if applicable) is shown at the bottom and hardware output (if applicable) at the top. Manufacturer-specific information also can be associated with an object in a manufacturer-defined section.

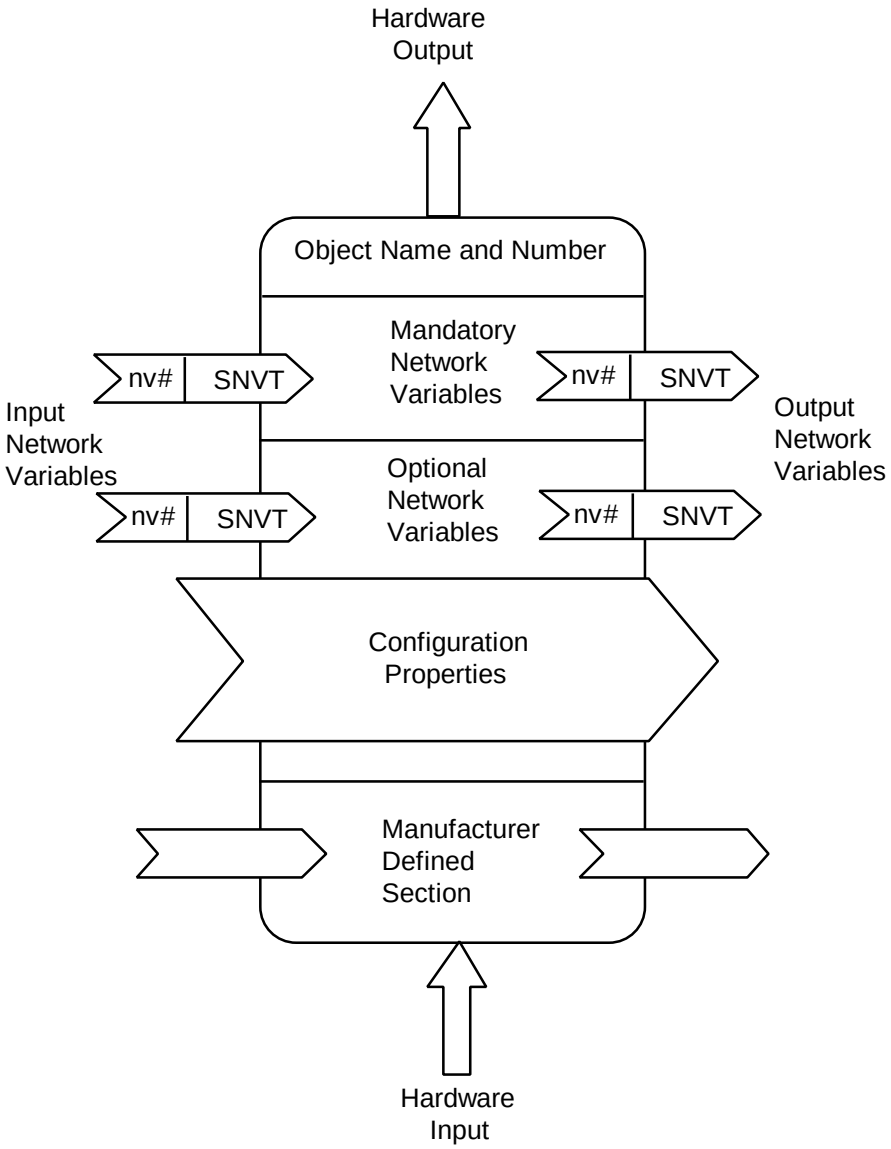

Figure 3.1 LONMARK Object Graphical Representation.

Network Variable Typing

For each LONMARK object defined in a node, the non-configuration network variables are numbered from 1 to *n* in the diagrams for use in self-documentation strings. This numbering provides a means to differentiate between two network variables of the same type within an object. The configuration network variables are numbered with the associated SCPT index, which provides a unique self-documentation index. See Chapter 5, *Documentation*, for information about how nodes map individual network variables to objects. The object definitions provided are generic in order to apply to a broad range of applications. For example, rather than exhaustively list the definitions for a temperature sensor, pressure sensor, flow sensor, etc. a more general "Sensor Object" is described.

In the generic LONMARK object definitions included in this chapter, some SNVTs are shown as `SNVT_xxx` to indicate that the SNVT is determined by the specific application in which the object is used in. An example is temperature or pressure sensing. The `SNVT_xxx` declaration must be replaced by a specific SNVT declaration at compilation time. This is most easily accomplished using a `#define` statement to redefine the `SNVT_xxx` declaration as a SNVT declaration. In the Functional Profile definitions, exact specifications of which SNVTs to use are provided.

Naming Conventions

Network variables are referenced and used in two very different contexts by the programmer and the installation technician. The programmer often uses Neuron C naming conventions (programming reference), whereas the installation technician requires a legible name supplied in a native language (end-user reference).

Both the programming reference and the end-user reference may be determined when the application is written--provided that the final use for the node is already ascertained. However, when the node's use is not determined until installation time, as in the case of a generic analog sensor that uses a 4-20mA interface, only the programming reference can be determined when the application is created.

For all object definitions, network variables are referred to by their programming reference. See Chapter 5, *Documentation*, for information about how nodes may include documentation information using the minimum amount of node memory space.

Programming Reference

A convention is used for network variable programming references. The functional name of a network variable is prefixed with its storage class as defined below. Functional name lengths cannot exceed 11 characters. For compactness, underscores are not used and all characters are lowercase except the first character of a word. Note: This convention is for consistency only and is not required by the LONMARK application layer interoperability guidelines.

Network variable storage class prefixes

`nvi`	network variable input
`nvo`	network variable output
`nci`	config network variable input (EEPROM)
`nro`	network variable output (ROM)

Example:

```
nviValue        // a RAM-based network variable input
                // with a functional name of Value
```

Abbreviations Used

The following are the network variable programming reference functional name abbreviations used in the object definitions:

Actual	Act	Range	Rng
Clear	Clr	Rate	Rt
Delay	Dly	Table	Tbl
Feedback	Fb	Time	T
Hardware	Hw	Translation	Trans
Inhibit	Inh		
Input	In		
Maximum	Max		
Minimum	Min		
Object	Obj		
Output	Out		
Position	Posn		

Use of Object Definitions

LONMARK objects form the basis of interoperability at the application layer. The following sections provide the definitions for the generic set of LONMARK objects. The complete set of LONMARK object definitions is available on the LONMARK web site at www.lonmark.org. Developers can choose the objects that best fit the functions of the device being developed. *The primary function of the device must be handled via LONMARK objects (not manufacturer-defined objects).* New application-specific LONMARK objects are proposed and reviewed by the LONMARK association members on an ongoing basis. See Appendix B to propose a new LONMARK object definition.

Guideline 3-1: LONMARK objects must be used as the building blocks for designing the application-layer interface.

Each object definition includes a description, object diagram, details of the mandatory and optional network variables, and details of associated configuration properties. To implement an object within a node requires that the specified mandatory network variables must be implemented.

Guideline 3-2: If a LONMARK object is implemented then, as a minimum, the node must implement all the object's mandatory network variables.

Associated with each object definition is a set of configuration properties that are designated as either mandatory or optional. In a fully supported object, all the listed configuration properties are programmable either through the use of network variables or configuration parameters. In the following sections, the configuration property descriptions illustrate the use of configuration network variables. The use of configuration parameters is described in Chapter 4. If an optional configuration property is not implemented in an object then it is recommended that the node follow the specified default value, wherever possible, to ensure consistent behavior.

The Node Object is used for monitoring and reporting the status of objects within a node. In the case of a single-object node, other mechanisms may be available for control and monitoring of the object. In such a case, the Node Object may be omitted provided that the following conditions apply:

i) The application program does not need to continue operating when the object is disabled. In this case setting the device off-line will disable the object.

ii) The single-object node does not implement alarms, self-test, range checking, fault detection, file transfer, or other functions belonging to the node object.

Guideline 3-3: Any LONMARK Guidelines-compliant node that supports more than one LONMARK Object must include a Node Object to allow monitoring and control of the objects within the node. Single-object nodes also require a Node Object if the node implements alarms, self-test, range checking, fault detection, file transfer, or other functions belonging to the Node Object; or if the application program must continue to operate when the object is disabled.

Node and Object Versioning

Every LONMARK certified node has a standard program ID (see Chapter 5 of these guidelines). Whenever the external interface of the node changes, the standard program ID must be modified to indicate that the node is not the network interface equivalent of other nodes on the network. This convention performs the most basic function of node versioning.

For nodes with multiple objects defined within them, it is useful to know *which* objects have changed. Additionally, it is possible to change the program within a node without changing its external interface. This information can also be useful information for a person maintaining the system. To support the versioning of individual objects, two standard configuration property types are defined: `SCPTobjMajVer` and `SCPTobjMinVer`. These two standard configuration properties are of unsigned short size. Their range is 0-255 with a default value of 0. `SCPTobjMajVer` shall always use the `const_flg` attribute (see Chapter 4 of this document), while `SCPTobjMinVer` shall always use the `device_specific_flg` attribute. The `const_flg` attribute means that all nodes with the same program ID shall have the same value, while the `device_specific_flg` attribute means that nodes with an identical program ID may have different values for this configuration property. The presence of these configuration properties within an object defines the major version and minor version of the object. The major version number shall be incremented when the network interface for the object changes, while the minor version number shall be incremented when the network interface remains the same, but the object has a different behavior.

Standard configuration properties are also defined for the node. These are `SCPTdevMajVer` and `SCPTdevMinVer`. These configuration properties have the same semantics as `SCPTobjMajVer` and `SCPTobjMinVer` respectively. These node versioning configuration properties are optional configuration properties of the node object.

Node Object

Object Type 0: Node Object

The Node Object allows the function of objects within a node to be monitored. Upon receiving an update to the `nviRequest` network variable, the `nvoStatus` network variable is updated. If included in the node, the optional `nvoAlarm` network variable is updated also. In addition the `nvoAlarm` network variable will report alarm conditions as they occur. The definition of `SNVT_obj_request` includes an object ID field to allow the Node Object to report status and alarm conditions for all objects on a node.

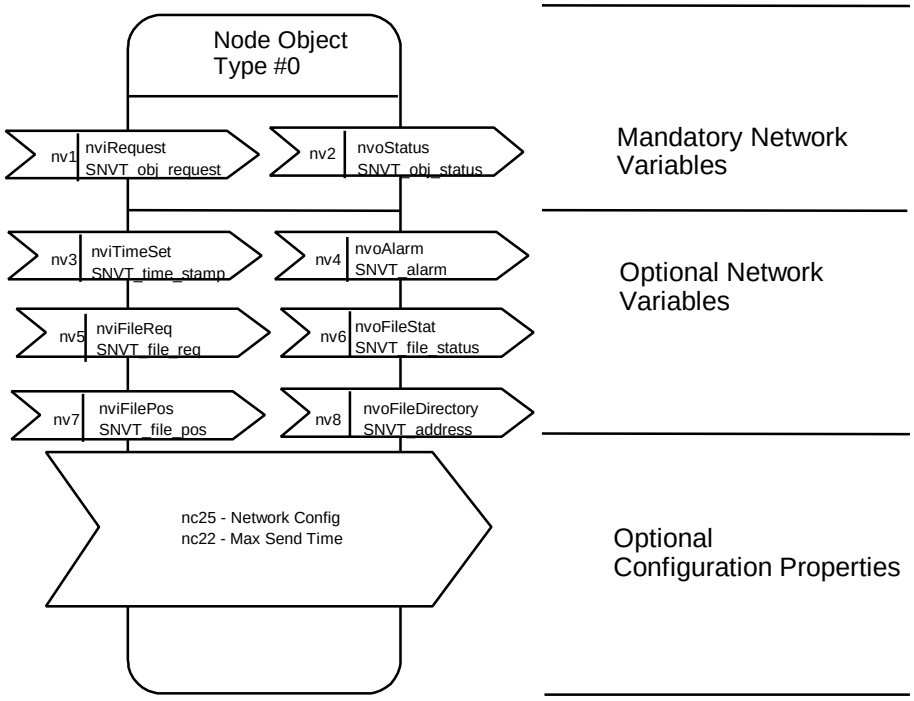

Figure 3.2 Node Object

Object Request

```
network input SNVT_obj_request nviRequest;
```

This input network variable provides the mechanism to request a particular mode for a particular object within a node. For a listing of all possible request codes and for the meaning of the function codes for SNVT_obj_request see the *SNVT Master List*.

Valid Range

The valid range is any value within the defined limits of SNVT_obj_request. The only mandatory request functions are RQ_NORMAL, RQ_UPDATE_STATUS and RQ_REPORT_MASK. Unsupported request functions must return invalid_request in the object status. It is not required to support object disable, self-test, override or alarm reporting.

The semantic definitions of the object request codes are as follows:

RQ_NORMAL - if the object was in the disabled or overridden state, this request cancels that state, and returns the object to normal operation. If the object was already in the normal state, a request to enter the normal state is not an error. After node reset, the state of objects on the node is application-specific.

RQ_UPDATE_STATUS - the status of the object is sent to the output network variable of type SNVT_obj_status. The state of the object is unchanged.

RQ_CLEAR_STATUS - Clears all status bits and reports that fact by outputting on SNVT_obj_sts. The state of the object is unchanged.

RQ_REPORT_MASK - the status bits that are supported by the object are sent to the output network variable of type SNVT_obj_status. If object disable is not supported, the disabled bit in the status mask must be zero for that object. See below for a description of the status mask. If self-test is not supported, the fail_self_test and self_test_in_progress bits in the status mask must be zero for that object. If alarm reporting is not supported, the in_alarm bit in the status mask must be zero for that object.

RQ_DISABLED - the object is placed in the disabled state. In the disabled state, output network variables belonging to the object are not propagated to the network. However, it must be possible to poll the output network variables of an object in this state. In the disabled state, the object must not respond to any updates received on its input network variables, but it must support reading and writing of any

configuration properties belonging to the object. If the object was already in the disabled state, a request to disable the object is not an error. An object may be in both the overridden and disabled states at the same time.

RQ_ENABLE - enables an object without modifying any overridden behavior. Used to enable an object without modifying the overridden state.

RQ_SELF_TEST - the object executes a self-test operation. If the self-test can be completed in the same critical section as the receipt of the request, the fail_self_test bit in the SNVT_obj_status output is set appropriately, and the self_test_in_progress bit remains zero. Otherwise, the self_test_in_progress bit must be set in the critical section in which the self-test request is received. When the test is complete, the fail_self_test bit is set appropriately, and the self_test_in_progress bit is cleared.

RQ_OVERRIDE - the object is placed in the overridden state. See the description of the Override Behavior and Override Value configuration properties in Chapter 3 for details. After node reset, the state of objects on the node is application-specific. An object may be in both the overridden and disabled states at the same time.

RQ_RMV_OVERRIDE - used to cancel an override without modifying the enabled state.

RQ_UPDATE_ALARM - the alarm status of the object are sent to the output network variable of type SNVT_alarm. The state of the object and of any alarms is unchanged.

RQ_CLEAR_ALARM - the alarm state of the object is cleared, SNVT alarm is updated to no alarms. State of the object is unchanged. Persistent alarms are reported as they are re-detected.

Node Object Behavior

Supporting requests to the node object (type 0) is optional. A request RQ_DISABLED to object type 0 means that all objects in the node should be disabled. If any of the objects in the node support the disable function, then those objects should be disabled, and invalid_request should not be reported. However, if none of the objects in the node supports the disable function, then invalid_request should be reported. Note that a request to disable the node object does not disable file transfer, or any other request to the node object. The SNVT_obj_status output and the SNVT_alarm output are not disabled when the node object is disabled.

A request RQ_NORMAL to the node object (type 0) specifies that all objects in the node leave the disabled and overridden states. If requests to the node object are supported, RQ_NORMAL is always a valid request to this object.

A request RQ_UPDATE_STATUS to the node object updates the status of the node object only. The status bits of the node object (with the exception of invalid_request and invalid_id) are defined to be the inclusive OR of the status bits of all the other objects in the node, with the possible addition of error and other conditions attributed to the node as a whole, rather than to any individual object. For example, if comm_failure is supported for the node object, then it should be set whenever any of the objects in the node reports communications failure, as well as when there is a communications failure at the node level.

A request RQ_UPDATE_ALARM to the node object may be an invalid request, if there are no node-level alarms in the node, but only object-level alarms. Node-level alarms may be total/service interval alarms, for example. See the *SNVT Master List and Programmer's Guide* (005-0027-01) for more details. The alarm status for object ID 0 does not reflect the status of any of the object alarms.

A request RQ_REPORT_MASK to the node object causes the object to respond with a mask of supported status bits in the SNVT_obj_status output network variable. A one bit in the mask means that the node may set the corresponding bit in the object status when the condition defined for that bit occurs. A zero means that the bit is never set by the node.

A request RQ_OVERRIDE to the node object causes all objects in the node that implement the overridden state to enter that state.

A request RQ_SELF_TEST to the node object initiates a node-level diagnostic self-test. It is not necessarily the same as self-testing each object in the node.

Object Status

```
network output SNVT_obj_status nvoStatus;
```

This output network variable reports the status for any object on a node.

Valid Range

The valid range is any value within the defined limits of SNVT_obj_status. The meaning of the fields of SNVT_obj_status are detailed in the *SNVT Master List and Programmer's Guide* (005-0027-01). The report_mask, invalid_id and invalid_request fields are the only mandatory bits in SNVT_obj_status, all other status bits are optional. Reporting a status of invalid_request is required if an unsupported request code is received on the object request input network variable. A status of invalid_id is reported whenever a request is received for an object ID that is not defined in the node. If the object ID is invalid and invalid_id is reported, then no further checking of the request code need be performed.

A self-documenting mask is used to describe the status bits that are supported beyond the two mandatory bits described above. The obj_request code RQ_REPORT_MASK causes the object to respond with a mask of supported status bits in the SNVT_obj_status output network variable. A one bit in the mask means that the object may set the corresponding bit in the object status when the condition defined for that bit occurs. A zero bit means that the bit may never be set by the object.

When setting the status in response to a RQ_REPORT_MASK, the report_mask bit must be set to distinguish this from other forms of status.

When Transmitted

Data are transmitted whenever a value is received on the Object Request input and when the Object Status Max Send Timer expires. When the Object Status Max Send

Timer expires, the status of each object (including the node object) is returned sequentially in round-robin fashion, one object status per expiration of the timer.

Update Rules

The application must update the status such that a poll of the status following the request is guaranteed to return a reasonable value. In a Neuron C program, this means setting the status in the same critical section as the request event is processed. Setting the status may mean setting the actual result or, for `RQ_SELF_TEST`, setting the `self_test_in_progress` indicator.

Update Rate

The default update rate for this network variable is 0 times per second.

Default Service Type

The default service type is acknowledged.

Alarm Output

```
network output sync SNVT_alarm nvoAlarm;
```

This output network variable transmits alarm data for each object on a node to a monitoring node. A message containing all the data relating to the alarm condition is sent whenever an alarm condition occurs, or is cleared, and upon the object receiving an RQ_UPDATE_ALARM request via the nviRequest network variable. The structure definition for SNVT_alarm is described in the *SNVT Master List and Programmer's Guide* (005-0027-01). The definition of SNVT_alarm includes an object ID field to allow the alarm object to report alarm conditions for multiple sensor and actuator objects within a node. Configuration properties for high and low limits, alarm set and clear times etc., are optionally set for each object.

When Transmitted

It is transmitted when an alarm condition occurs and also upon receiving an RQ_UPDATE_ALARM request via the nviRequest network variable.

Valid Range

The valid range for the value field is any value within the defined limits of the SNVT_alarm output.

Default Service Type

The default service type is acknowledged.

Time Stamp Input

```
network input SNVT_time_stamp nviTimeSet;
```

This input network variable is used to synchronize the node's internal real time clock with an external system clock, for the purpose of reporting alarms.

Valid Range

The valid range for all fields is any value within the defined limits of SNVT_time_stamp.

File Request

```
network input SNVT_file_req nviFileReq;
```

This input network variable contains an operation code and a file index. When an update is received on this network variable the node performs the indicated operation and returns the status of that operation in the network variable of type

SNVT_file_status. The request operation codes are defined in the Neuron C include file SNVT_FR.H and described in the Echelon *File Transfer* Engineering Bulletin (005-0025-01). Files are identified with a unique 16-bit number called the file index - up to 65,535 files can be identified on any node.

Valid Range

The valid range for the value field is any value within the defined limits of the SNVT_file_req.

Default Service Type

The default service type is acknowledged.

File Position

```
network input SNVT_file_pos nviFilePos;
```

Network variables of type SNVT_file_pos are used to control the position of the read/write pointer in a file used for random access, as well as to specify the length of the next file transfer. The rw_ptr field is a 32-bit value compatible with the Neuron C Extended Arithmetic type s32_type. For more details, see the Echelon *File Transfer* Engineering Bulletin (005-0025-01).

Valid Range

The valid range for the value field is any value within the defined limits of the SNVT_file_pos.

Default Service Type

The default service type is acknowledged.

File Status

```
network output SNVT_file_status nvoFileStat;
```

This output network variable transmits the status of the last file request to the node. The returned status codes are defined in the Neuron C include file SNVT_FS.H and described in the Echelon *File Transfer* Engineering Bulletin (005-0025-01).

When Transmitted

It is transmitted whenever a request is received on the File Request input network variable.

Valid Range

The valid range for the value field is any value within the defined limits of the type SNVT_file_status.

Default Service Type

The default service type is acknowledged.

NOTE: The node object implements the file request and file position network variables as inputs, and the file status network variable as an output. The node can therefore act as the Sender or the Receiver in a file transfer, but it cannot act as the Initiator of a file transfer using these network variables.

Configuration Parameter File Directory Address

```
const network output SNVT_address nvoFileDirectory;
```

A network variable of type `SNVT_address` is used to point to information in the address space of the Neuron Chip. This optional network variable in the node object points to a file directory containing descriptors for the files in the device. It is used when configuration properties are implemented using configuration parameter files accessed by network management read/write messages. For more details, see *Access to SCPTs via Network Management Read/Write Messages* in Chapter 4.

Valid Range

The valid range for the file directory address is any value within the user data memory space of the Neuron Chip

Default Service Type

The default service type is none. This variable may be polled by a configuration tool.

Optional Configuration Properties

Network Config - SCPT_nwrk_cnfg (25)

```
network input config SNVT_config_src nciNetConfig;
```

All nodes that support self-installation must provide a configuration property to allow a network management tool to also install the node.

Valid Range

When a node is self-installed this variable should be set to `CFG_LOCAL` at manufacture. A variable set to `CFG_EXTERNAL` signifies that a network manager will assign network addresses for the node.

Default Value

For a self-installed node the default value is `CFG_LOCAL`.

Max Send Time - SCPT_max_snd_t (22)

```
network input config SNVT_elapsed_tm nciMaxStsSendT;
```

This configuration property is used to control the maximum period of time that expires before the object automatically transmits the current value of the `nvoStatus` output network variable. This provides a heartbeat output that can be used by destination objects to ensure that the node is still healthy. The status of each object (including the node object) is returned sequentially in round-robin fashion, one object status per expiration of the timer.

Valid Range

	Heartbeat Active		Heartbeat Disabled
	Min	Max	
days	0	0	0
hours	0	17	0
minutes	0	59	0
seconds	0	59	0
milliseconds	0	999	0

Default Value

The default value is 0, heartbeat disabled.

Sensor Object - Open and Closed Loop

Introduction

The Sensor Object is a generic object that can be used with any form of sensor, such as an analog pressure, temperature, or humidity sensor, or even a digital switch. The Sensor Object can supply data directly to an Actuator Object or to a control loop located within a Controller Object. There are two versions of the sensor object, one without feedback named the "open loop sensor object" and one with feedback named the "closed loop sensor object." The particular version of the sensor object used will be determined by the application, as described in the following sections.

Object Type 1: Open Loop Sensor Object

The Open Loop Sensor Object is suitable for use with sensing devices that report absolute rather than relative values (such as potentiometers versus quadrature encoders) and for use with devices that do not require feedback information for correct operation.

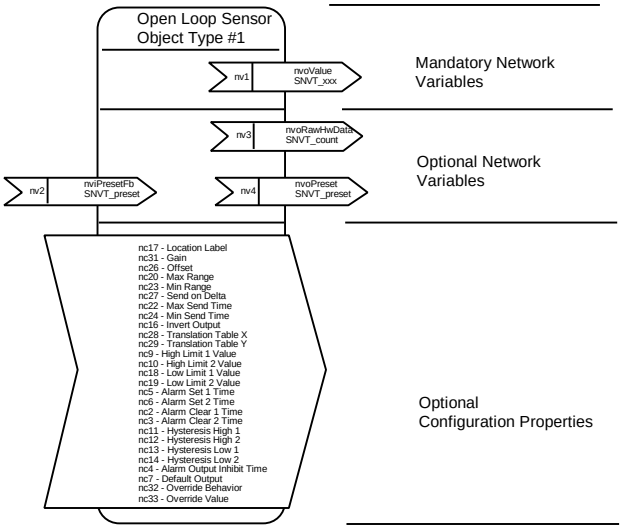

Figure 3.3 Sensor Object - Open Loop

Object Type 2: Closed Loop Sensor Object

The Closed Loop Sensor Object definition contains a feedback feature that makes it suitable for use in applications where multiple sensors can be combined in arbitrary combinations with multiple actuator devices. The purpose of the closed loop sensor object is to enable multiple sensors to control a common actuator, or a single sensor to control multiple actuators, while retaining synchronization between the actual and desired states of objects in both the sensors and actuators. Typical situations where these multiple relationships occur are when multiple remote set point controllers are positioned around a building and are all used to control the same parameter, or where multiple remote dimmer controls are used within a lighting system to control the same load.

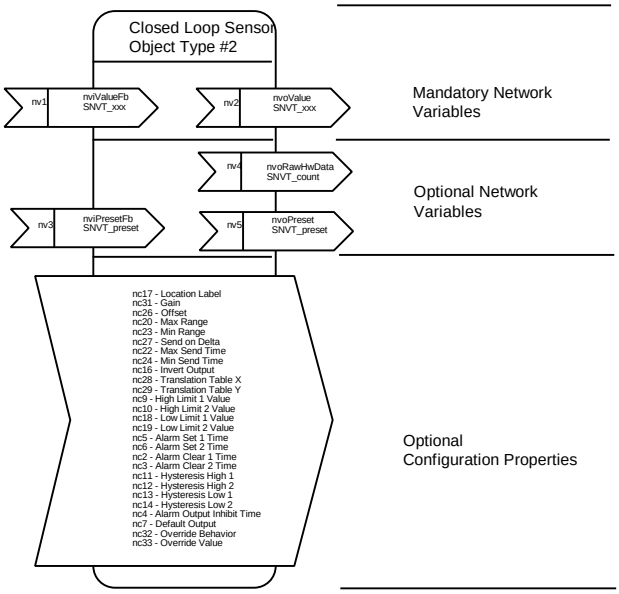

Figure 3.4 Sensor Object - Closed Loop

Figure 3.5 illustrates two possible connections between Closed Loop Sensor Objects and a Closed Loop Actuator Object. When connected in multiple relationships, there is sometimes a need to keep all sensor objects aware of any changes made to actuator objects by other sensor objects. This applies to multiple relationships between objects of all types, not specifically Sensors and Actuators. Synchronization is achieved by connecting the `nvoValueFb` output of destination objects to the `nviValueFb` input of source objects. This is method 1, where the `nvoValueFb` output always represents the requested value as received by the `nviValue` input and not the actual load value. Alternatively, synchronization may be achieved by connecting the `nvoValue` outputs of all sources to the `nviValueFb` inputs of all other sources controlling common actuators. This is method 2. Method 1 ensures that sensors are only synchronized based on updates from sensors which are actually received and processed by actuators; however, it does introduce additional traffic and delays. Method 2 generates the least network traffic and introduces the shortest delays between new sensor updates and synchronization of sensors. For

these reasons Method 2 may be more appropriate when connecting devices on low bandwidth channels. Note that, whichever method is used, only the connections are affected - the meaning and programming for feedback inputs and outputs remain unchanged.

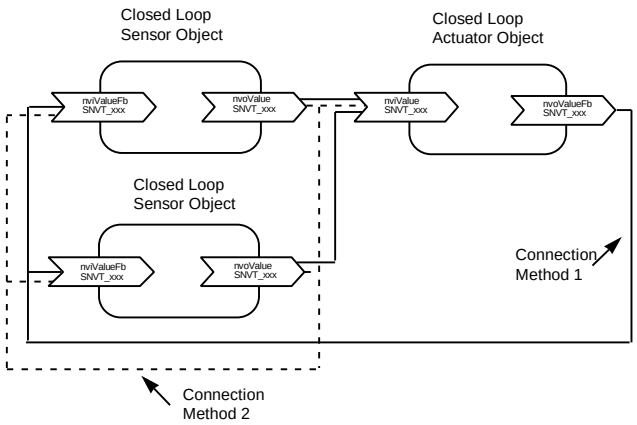

Figure 3.5 Typical Connections Between Closed Loop Sensor And Actuator Objects

Mandatory Network Variables - Open and Closed Loop Sensors

Value Output

```
network output SNVT_xxx nvoValue;
```

This output network variable transmits the value obtained by the data acquisition application after it has been converted to the correct engineering units. The data may have been scaled and/or linearized using a translation table prior to transmission.

Valid Range

The valid range is any value within the defined limits of the SNVT concerned, or it may be limited by the optional Min Range or Max Range configuration properties.

When Transmitted

Data is only transmitted when the input value has changed by a quantum which is either defined in the node's application program or configured by the optional Send on Delta configuration property. This network variable will also be transmitted as a heartbeat output as dictated by the optional Max Send Time configuration property.

Update Rate

The maximum update rate is configured using the optional Min Send Time configuration property. The default maximum update rate is a function of channel bandwidth as specified by the Min Send Time configuration property default values.

Default Service Type

The default service type is acknowledged.

Mandatory Network Variables - Closed Loop Sensor Only

Value Feedback Input

```
network input SNVT_xxx nviValueFb;
```

This network variable is used to synchronize Closed Loop Sensor Objects when they are connected in multiple relationships. It may be updated either by another source object, or by a destination object whenever the destination's `nviValue` input has received an update, although not necessarily immediately: the propagation of the feedback may be delayed relative to the rate of updating in order to avoid overloading the network with feedback values. See the Closed Loop Actuator Object definition for more details.

Valid Range

The valid range is any value within the defined limits of the SNVT concerned.

Optional Network Variables - Open and Closed Loop Sensors

Raw Hardware Data Output

```
network output SNVT_count nvoRawHwData;
```

This output network variable transmits the value obtained by the data acquisition application from a fixed point device such as an A/D converter, prior to any scaling or linearization which may be performed using a translation table.

Valid Range

The valid range is any 16 bit value.

When Transmitted

Data is transmitted when the output value has changed by a limit which is either defined in the node's application program or configured by the Send on Delta configuration property. This network variable will also be transmitted as a heartbeat output as dictated by the optional Max Send Time configuration property.

Update Rate

The update rate is configured using the optional Min Send Time configuration property. The default maximum update rate is a function of channel bandwidth as specified by the Min Send Time configuration property default values.

Default Service Type

The default service type is acknowledged.

Preset Output

```
network output SNVT_preset nvoPreset;
```

This network variable is used to program or control the preset function of a destination object. The preset function is an option that allows a destination object to be adjusted over time to various pre-set levels. The structure definition for `SNVT_preset` is detailed in *The SNVT Master List and Programmer's Guide* (005-0027-01).

Presets may be volatile or not. Non-volatile presets have the advantage of being retained across resets.

To program a preset, `nvoPreset` is transmitted with updated values for `SNVT_preset.value`, `SNVT_preset.selector`, and the time-related fields. In

addition SNVT_preset.learn is either set to LN_LEARN_VALUE or alternatively set to LN_LEARN_CURRENT which causes the actuator to learn whatever its current value is. A pre-programmed preset can be selected by transmitting nvoPreset with the relevant preset number set in SNVT_preset.selector and SNVT_preset.learn set to LN_RECALL.

The time-related fields specify the time period over which the actuator should progress from the current level to the newly selected preset level. A benefit of this mechanism is that any set of actuators that are preset with a common rate value for a particular preset number will all arrive at this new value at the same time regardless of the individual preset values to which they ramp.

Valid Range

The valid range for all fields and sub-fields is any value within the defined limits of the data types concerned. The exception is the range for the time-related fields, which has a maximum of 18 hours and a resolution of 1 second.

When Transmitted

This network variable is transmitted only when a preset is being recalled or programmed.

Update Rate

The default update rate for this network variable is 0 times per second.

Default Service Type

The default service type is acknowledged.

Preset Input Feedback

```
network input SNVT_preset nviPresetFb;
```

This network variable optionally receives preset function feedback information. It is used to synchronize source objects that are configured in multiple relationships. As discussed earlier, feedback information can come from Actuator Objects or from Sensor Objects.

Optional Configuration Properties - Open and Closed Loop Sensors

Transformations on Sensed Data

The sensor object includes various configuration properties that perform transformations on the sensed value. The behavior of the object is dependent on the order in which these transformations are applied. The following instructions for analog sensors and discrete sensors must be followed.

Analog Sensors

Raw hardware data comes directly from the physical sensor such as an A/D converter or shaft encoder. This data is not calibrated or linearized and probably is of type SNVT_count since hardware sensor devices are most likely to be fixed point devices. If a translation table is provided, the raw hardware value is converted to a floating point number and looked up in the translation table. The translation table can be replaced with a fixed point operation if there is no chance of overflow problems and the sensor is linear. After the translation table the value is converted to type SNVT_xxx. The calibration offset, if any, is applied. The value is pegged between the Max and Min Range. Send on Delta is applied to see if it should be sent. This means that a value that is continuously pegged at one limit will not be sent out (unless there is a Max Send Time).

Discrete Sensors

For purely discrete sensors none of the numerical transformations described above apply. Only the Location Label, Invert Output, Max Send Time, and Min Send Time configuration properties have significance for digital sensors.

Location Label - SCPT_location (17)

```
network input config SNVT_str_asc nciLocation;
```

This configuration property can optionally be used to provide more descriptive physical location information than can be provided by the Neuron Chip's 6 byte location string. The location relates to the object and not the node.

Valid Range

Any NUL terminated ASCII string less than 31 bytes total in length.

Default Value

The default value is an ASCII string containing all zeroes.

Maximum Send Time - SCPT_max_snd_t(22)

```
network input config SNVT_elapsed_t nciMaxSendT;
```

This configuration property is used to control the maximum period of time that expires before the object automatically transmits the current value of the nvoValue output network variable. This provides a heartbeat output that can be used by destination objects to ensure that the object is still healthy.

Valid Range

	Heartbeat Active		Heartbeat Disabled
	Min	Max	
days	0	0	0
hours	0	17	0
minutes	0	59	0
seconds	0	59	0
milliseconds	0	999	0

Default Value

The default value is 0, heartbeat disabled.

Minimum Send Time - SCPT_min_snd_t (24)

```
network input config SNVT_elapsed_tm nciMinSendT;
```

This configuration property is used to control the minimum period between output network variable transmissions (maximum transmission rate). It provides a way to tailor the output network variable transmission rate to available bandwidth.

Transmission rate limiting may be disabled by setting all fields to zero.

Valid Range

	Time Between Updates	
	Min	Max
days	0	0
hours	0	17
minutes	0	59
seconds	0	59
milliseconds	5	999

Default Value

If this configuration property is present the default value is set according to the bit rate of the transmission medium as follows:

Bit Rate	***Default Minimum Time Between Updates***
2 kbps	60 sec
4 kbps	60 sec
10 kbps	30 sec
39 kbps	15 sec
78 kbps	15 sec
1.25 Mbps	1 sec

Maximum Range - SCPT_max_rnge (20)

```
network input config SNVT_xxx nciMaxRng;
```

This configuration property is used to limit the maximum value of nvoValue. The data type is the same as nvoValue.

Valid Range

The valid range is any value within the defined limits of the SNVT concerned. The value must be greater than any specified Min Range configuration property.

Default Value

The default value is the maximum value of the SNVT concerned.

Minimum Range - SCPT_min_rnge (23)

```
network input config SNVT_xxx nciMinRng;
```

This configuration property is used to limit the minimum value of nvoValue. The data type is the same as the value field of nvoValue.

Valid Range

The valid range is any value within the defined limits of the SNVT concerned. The value must be less than any specified Max Range configuration property.

Default Value

The default value is the minimum value of the SNVT concerned.

Send on Delta - SCPT_snd_delta (27)

```
network input config SNVT_xxx nciMinDelta;
```

This configuration property is used to determine the amount by which the value obtained by the data acquisition application must change before nvoValue is transmitted. The data type is the same as nvoValue. For SNVT types that are not zero-based (for example SNVT_temp) this value is represented as a difference in two values of the SNVT concerned. For SNVT_switch the continuous (level) part follows this convention. Thus to represent a delta value of 0.5 deg C, the value of nciMinDelta should be set to 5. Some network monitoring and control tools may display this value as -273.5 deg C.

Valid Range

The valid range for this configuration property is any value within the defined limits of the data type in question.

Default Value

The default value is application specific.

Offset - SCPT_offset (26)

```
network input config SNVT_xxx nciOffset;
```

This configuration property is used to calibrate the external hardware by specifying the level that the nvoValue output should adopt based on the current data from the hardware. This offset applies after the use of any translation table or gain factor. The data type must be the same as that of nvoValue.

Valid Range

The valid range is any value within the defined limits of the SNVT concerned.

Default Value

The default value is application specific.

Invert Output - SCPT_invrt_out (16)

```
network input config SNVT_lev_disc nciInvert;
```

When a Sensor Object is used in conjunction with hardware that supplies only discrete information such as a switch, this parameter is used to invert the active polarity. This enables the use of either normally closed or normally open contacts. For SNVT_switch, the state part may use this option.

Valid Range

The valid range is ST_ON or ST_OFF. A value other than ST_OFF specifies that the output value should be inverted.

Default Value

The default value is no polarity inversion.

Default Output - SCPT_def_output (7)

```
network input config SNVT_xxx nciDefault;
```

This configuration property determines the position or level that the sensor should adopt, when no updates are received from the hardware within the maximum receive time, at power-on or reset, and when an override request is received for the object. The override behavior is defined by the configuration properties Override Behavior and Override Value.

Valid Range

The valid range of this configuration property is any value within the defined limits of the SNVT concerned.

Default Value

The default value is application specific.

Override Behavior - SCPT_ovr_behave (32)

```
network input config SNVT_override nciOverBehave;
```

This configuration property is used to define the behavior of a sensor when an override request is received for the object.

Valid Range

The valid range for this configuration property is any value within the defined limits of the `SNVT_override`. The sensor or actuator can retain its last setting, go to a specified value or go to the default output value.

Default Value

The default value is for the object to retain its last setting.

Override Value - SCPT_ovr_value (33)

```
network input config SNVT_xxx nciOverValue;
```

This configuration property is used to set the value a sensor should adopt when an object is overridden and the value of Override Behavior is `OV_SPECIFIED`.

Valid Range

The valid range for this configuration property is any value within the defined limits of the `SNVT_xxx` concerned.

Default Value

The default value is application specific.

Gain - SCPT_gain (31)

```
network input config SNVT_muldiv nciGain;
```

This configuration property is used to calibrate the external hardware by specifying a multiplication factor for the raw data. It applies when the sensor is linear and does not require a translation table. The gain is applied before any specified offset is applied. This SNVT specifies a 16-bit multiplier and a 16-bit divisor.

Default Value

The default value is application specific.

Translation Table X - SCPT_trns_tbl_x (28)

```
network input config SNVT_trans_table nciTransTblX;
```

This configuration property is used in conjunction with the Translation Table Y configuration property to create a translation table that dictates how to scale and linearize the raw input signal as received from the sensor.

Valid Range

The valid range for this configuration property is any value within the defined limits of `SNVT_trans_table`. A Translation Table Y value must be specified for each Translation Table X value.

Default Value

The default is application specific.

Translation Table Y - SCPT_trns_tbl_y (29)

```
network input config SNVT_trans_table nciTransTblY;
```

This configuration property is used in conjunction with the Translation Table X configuration property to create a translation table that dictates how to scale or linearize the raw input signal as received from the sensor.

Valid Range

The valid range for this configuration property is any value within the defined limits of `SNVT_trans_table`. A Translation Table X value must be specified for each Translation Table Y value.

Default Value

The default is application specific.

Alarm Configuration Parameter

The following configuration parameters relate to the optional alarm reporting provided by the Node Object using the network variable of type SNVT_alarm. Figure 3.6 illustrates the relationship of each of the alarm configuration parameters.

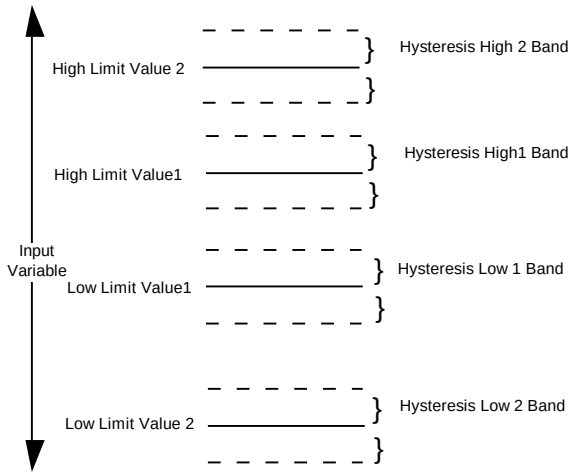

Figure 3.6 Alarm Configuration Parameters

High Limit Value 1 - SCPT_high_limit1 (9)

```
network config input SNVT_xxx nciHighLim1;
```

This configuration property is used to remotely set the alarm high limit 1 against which the value field of the output value is tested for alarm conditions. The data type is the same as the value field of the output value.

Valid Range

The valid range is any value within the defined limits of the SNVT concerned.

Default Value

The default is the maximum value of the SNVT concerned.

High Limit Value 2 - SCPT_high_limit2 (10)

```
network config input SNVT_xxx nciHighLim2;
```

This configuration property is used to remotely set the alarm high limit 2 against which the value field of the output value is tested for alarm conditions. The data type is the same as the value field of the output value.

Valid Range

The valid range is any value within the defined limits of the SNVT concerned.

Default Value

The default is the maximum value of the SNVT concerned.

Low Limit Value 1 - SCPT_low_limit1 (18)

```
config network input SNVT_xxx nciLowLim1;
```

This configuration property is used to remotely set the alarm low limit 1 against which the value field of the output variable is tested. The data type is the same as the value field of the output value.

Valid Range

The valid range is any value within the defined limits of the SNVT concerned.

Default Value

The default is the minimum value of the SNVT concerned.

Low Limit Value 2 - SCPT_low_limit2 (19)

```
config network input SNVT_xxx nciLowLim2;
```

This configuration property is used to remotely set the alarm low limit 2 against which the value field of the output variable is tested. The data type is the same as the value field of the output value.

Valid Range

The valid range is any value within the defined limits of the SNVT concerned.

Default Value

The default is the minimum value of the SNVT concerned.

Alarm Set Time 1 - SCPT_alrm_set_t1 (5)

```
network input config SNVT_elapsed_tm nciAlarmSetT1;
```

This configuration property is used to determine the time period that an alarm 1 condition must exist before it is regarded as a valid alarm.

Valid Range

	Min	Max
days	0	0
hours	0	0
minutes	0	0
seconds	0	59
milliseconds	0	999

Default Value

The default value is 0s.

Alarm Set Time 2 - SCPT_alrm_set_t2 (6)

```
network input config SNVT_elapsed_tm nciAlarmSetT2;
```

This configuration property is used to determine the time period that an alarm 2 condition must exist before it is regarded as a valid alarm.

Valid Range

	Min	Max
days	0	0
hours	0	0
minutes	0	0
seconds	0	59
milliseconds	0	999

Default Value

The default value is 0s.

Alarm Clear Time 1 - SCPT_alrm_clr_t1 (2)

```
network input config SNVT_elapsed_tm nciAlarmClearT1;
```

This configuration property is used to determine the time period that an alarm 1 condition must not exist before it is regarded as a valid cleared alarm.

Valid Range

	Min	Max
days	0	0
hours	0	0
minutes	0	0
seconds	0	59
milliseconds	0	999

Default Value

The default value is 0s.

Alarm Clear Time 2 - SCPT_alrm_clr_t2 (3)

```
network input config SNVT_elapsed_tm nciAlarmClearT2;
```

This configuration property is used to determine the time period that an alarm 2 condition must not exist before it is regarded as a valid cleared alarm.

Valid Range

	Min	Max
days	0	0
hours	0	0
minutes	0	0
seconds	0	59
milliseconds	0	999

Default Value

The default value is 0s.

Hysteresis High 1 - SCPT_hysthigh1 (11)

```
network input config SNVT_xxx nciLimHystHi1;
```

This configuration property determines the hysteresis level for the value field of the `nciHighLim1` comparison threshold. The data type must be the same as the value field of the output variable.

Valid Range

The valid range is any value within defined limits of the SNVT concerned.

Default Value

The default value is application specific.

Hysteresis High 2 - SCPT_hysthigh2 (12)

```
network input config SNVT_xxx nciLimHystHi2;
```

This configuration property determines the hysteresis level for the value fields of the `nciHighLim2` comparison threshold. The data type must be the same as the value field of the output variable.

Valid Range

The valid range is any value within defined limits of the SNVT concerned.

Default Value

The default value is application specific.

Hysteresis Low 1 - SCPT_hystlow1 (13)

```
network input config SNVT_xxx nciLimHystLow1;
```

This configuration property determines the hysteresis level for the value field of the `nciLowLim1` comparison threshold. The data type must be the same as the value field of the output variable.

Valid Range

The valid range is any value within defined limits of the SNVT concerned.

Default Value

The default value is application specific.

Hysteresis Low 2 - SCPT_hystlow2 (14)

```
network input config SNVT_xxx nciLimHystLow2;
```

This configuration property determines the hysteresis level for the value fields of the `nciLowLim2` comparison threshold. The data type must be the same as the value field of the output variable.

Valid Range

The valid range is any value within defined limits of the SNVT concerned.

Default Value

The default value is application specific.

Note: Hysteresis values for SNVTs that are not zero-based (for example SNVT_temp) are represented as a difference of two values of the SNVT concerned. Thus to represent a hysteresis value of 0.5 deg C, the configuration property should be set to 5. Some monitoring and control tools may display this value as -273.5 deg C.

Alarm Output Inhibit Time - SCPT_alrm_ihb_t (4)

```
network input config SNVT_elapsed_tm nciOutInhT;
```

This configuration property is used to determine the time period for which alarms are inhibited after an object is enabled or the node is reset or is put on-line.

Valid Range

	Min	Max
days	0	0
hours	0	17
minutes	0	59
seconds	0	59
milliseconds	0	999

Default Value

The default value is 0.

Actuator Object - Open & Closed Loop

Introduction

The Actuator Object is a generic object that can be used with any form of actuator, such as a motor or a valve. It may either be controlled by a remote control algorithm located within a Controller Object or directly by a Sensor Object. As with the sensor object there are two versions of the actuator object, one without feedback named the "open loop actuator" and one with feedback named the "closed loop actuator."

Object Type 3: Open Loop Actuator Object

The Open Loop Actuator Object is suitable for use in applications where the actuator provides no feedback information.

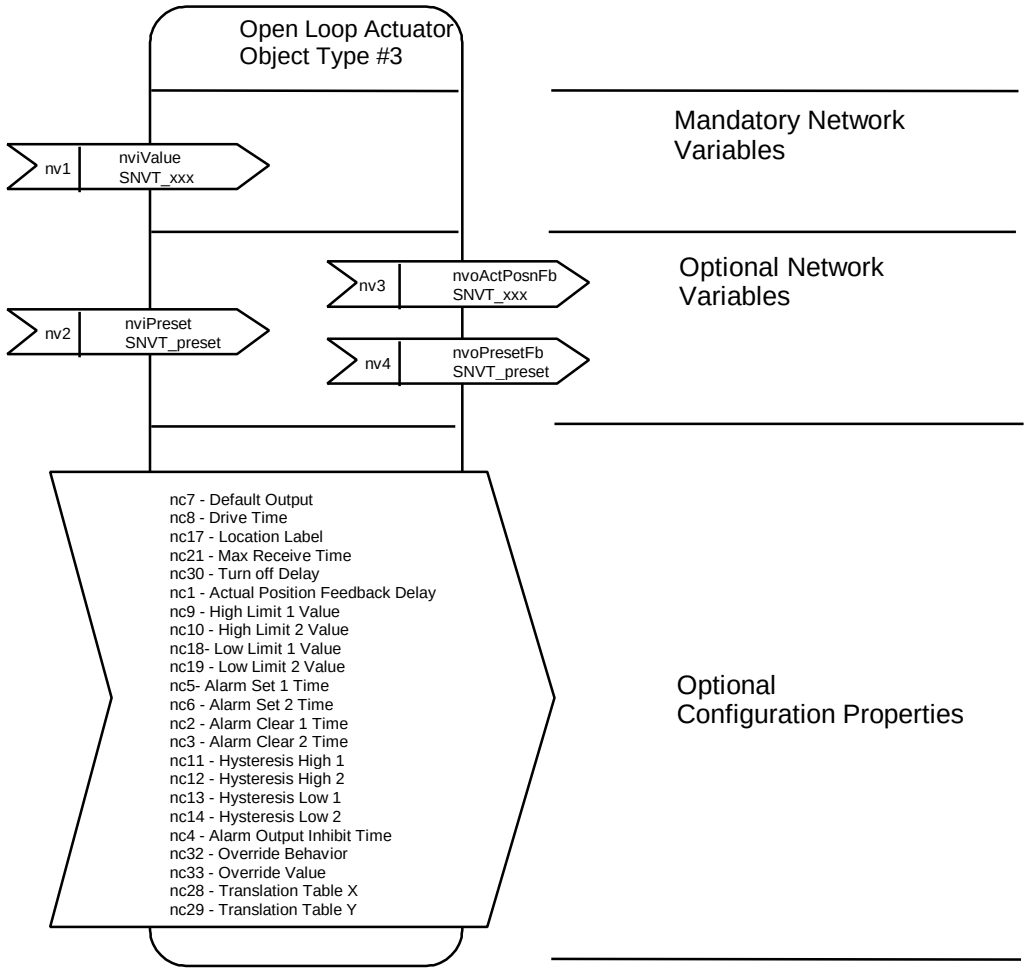

Figure 3.7 Open Loop Actuator Object

Object Type # 4 Closed Loop Actuator Object

The Closed Loop Actuator Object definition contains a feedback feature which makes it suitable for use in applications where multiple actuators can be combined in arbitrary combinations with multiple sensor devices. The feedback feature allows synchronization between the actual and desired states of objects in multiple sensors and actuators.

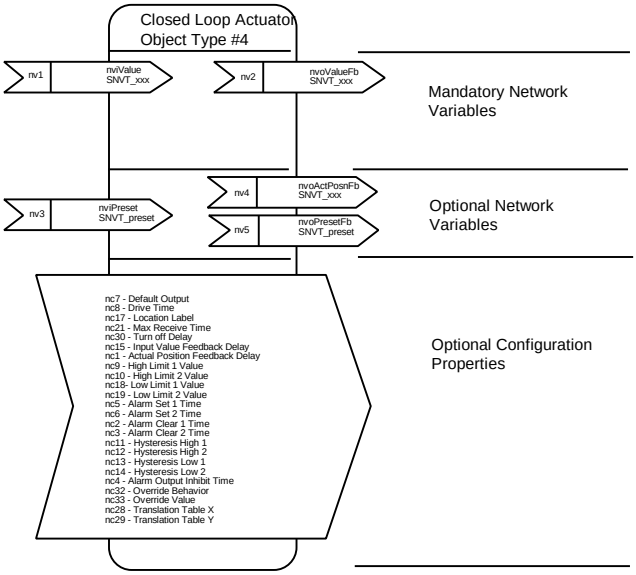

Figure 3.8 Closed Loop Actuator Object

Mandatory Network Variables - Open and Closed Loop Actuator

Value Input

```
network input SNVT_xxx nviValue;
```

This input network variable dictates the desired state of the actuator, such as speed, position, or state. The actual data type used is determined by the specific application.

Since the transmission rate from a Sensor Object may be restricted to conserve available bandwidth, application level smoothing is advised where necessary, as in the case of lighting control, to prevent the appearance of stepped transitions from level to level.

Valid Range

The valid range is any value within the defined limits of the SNVT concerned.

Mandatory Network Variables - Closed Loop Actuator Only

Value Feedback Output

```
network output SNVT_xxx nvoValueFb;
```

This output network variable transmits the current value of the `nviValue` input. It is used to synchronize source objects in multiple relationships.

When Transmitted

It is updated when the `nviValue` input is updated and under normal circumstances, only when the `nviValue` input is unlikely to be updated again immediately. The time period between the last change to `nviValue` and the output update may be configured using the optional Input Value Feedback Delay configuration property.

The feedback delay may be easily achieved using a non-repeating software timer that is retriggered on every update to the `nviValue` input, the expiration of the timer causing the `nvoValueFb` network variable to be transmitted.

Valid Range

The valid range is any value within the defined limits of the SNVT concerned.

Default Service Type

The default service type is unacknowledged.

Preset Input

```
network input SNVT_preset nviPreset;
```

This network variable is used to program or control the preset function. See the Sensor Object description and *The SNVT Master List and Programmer's Guide* (005-0027-01) for more details.

Valid Range

The valid range for all fields and sub-fields is any value within the defined limits of `SNVT_preset`.

Preset Feedback Output

```
network output SNVT_preset nvoPresetFb;
```

This network variable transmits the setting associated with the current recalled or programmed preset. It is used to synchronize source objects in multiple relationships and to extract the contents of the node's internal preset tables.

When Transmitted

It is transmitted when a change is made to the `nviPreset` input.

Update Rate

There is no maximum update rate.

Valid Range

The valid range for all fields and sub-fields is any value within the defined limits of `SNVT_preset`.

Default Service Type

The default service type is unacknowledged.

Actual Position Feedback Output

```
network output SNVT_xxx nvoActPosnFb;
```

This output network variable reflects the current position of the actuator and can be used as part of a control loop and for monitoring purposes.

When Transmitted

The variable is transmitted on a regular basis when the actual position of the actuator does not match the requested position as dictated by `nviValue`.

Update Rate

The update rate is configured using the optional Actual Position Feedback Delay configuration property. The default maximum update rate is a function of channel bandwidth as specified by the Actual Position Feedback Delay configuration property default values.

Valid Range

The valid range for the value field is any value within the defined limits of the SNVT concerned.

Default Service Type

The default service type is unacknowledged.

Optional Configuration Properties - Open and Closed Loop Actuators

Location Label - SCPT_location (17)

```
network input config SNVT_str_asc nciLocation;
```

This configuration property can optionally be used to provide more descriptive physical location information than can be provided by the Neuron Chip's 6 byte location string. The location relates to the object and not the node.

Valid Range

Any NUL terminated ASCII string of 31 bytes total length.

Default Value

The default value is an ASCII string containing all zeroes.

Translation Table X - SCPT_trns_tble (28)

```
network input config SNVT_trans_table nciTransTblX;
```

This configuration property is used in conjunction with the Translation Table Y configuration property to create a translation table that dictates how to scale or linearize actuator movement with respect to `nviValue`.

Valid Range

The valid range for this configuration property is any value within the defined limits of the SNVTs used for the values in `SNVT_trans_table`. A Translation Table Y value must be specified for each Translation Table X value.

Default Value

The default is application specific.

Translation Table Y - SCPT_trns_tble (29)

```
network input config SNVT_trans_table nciTransTblY;
```

This configuration property is used in conjunction with the Translation Table X configuration property to create a translation table that dictates how to scale and linearize actuator movement with respect to `nviValue`.

Valid Range

The valid range for this configuration property is any value within the defined limits of the `SNVT_trans_table`. A Translation Table X value must be specified for each Translation Table Y value.

Default Value

The default is application specific.

Input Value Feedback Delay - SCPT_in_fb_dly (15) Closed Loop Only

```
network input config SNVT_elapsed_tm nciInFbDly;
```

This configuration property is used to set the time period between the last update in a succession of changes to the `nviValue` input and the `nvoValueFb` output being updated.

Valid Range

	Min	Max
days	0	0
hours	0	0
minutes	0	0
seconds	0	59
milliseconds	0	999

Default Value

The default value is 0ms.

Actual Position Feedback Delay - SCPT_act_fb_dly (1)

```
network input config SNVT_elapsed_tm nciActFbDly;
```

This configuration property sets the period for updating the `nvoActPosnFb` output when the actuator position does not match the requested position as specified by `nviValue`.

Valid Range

	Min	Max
days	0	0
hours	0	0
minutes	0	0
seconds	0	59
milliseconds	0	999

Default Value

The default value is 0s.

Drive Time - SCPT_drive_t (8)

```
network input config SNVT_elapsed_tm nciDriveT;
```

This configuration property is used to inform the application program of the time to be taken by the actuator to move from one extreme to the other.

Valid Range

	Min	Max
days	0	0
hours	0	17
minutes	0	59
seconds	0	59
milliseconds	0	999

Default Value

The default value is application specific.

Turn Off Delay - SCPT_off_dely (30)

```
network input config SNVT_elapsed_tm nciOffDly;
```

This configuration property is only used when the data type of the nviValue input is SNVT_switch or SNVT_lev_disc. It is used to determine the length of time that the load remains energized after a change from ON to OFF has been received by the nviValue input. If this parameter is set to a non-zero value, the load will be kept on for the specified time after the nviValue is set to OFF. The turn off delay feature may be disabled by setting all fields to zero.

Valid Range

	Turn Off Dly Active		Turn Off Dly Inactive
	Min	Max	
days	0	0	0
hours	0	17	0
minutes	0	59	0
seconds	0	59	0
milliseconds	0	999	0

Default Value

The default value is zero.

Default Output - SCPT_def_output (7)

```
network input config SNVT_xxx nciDefault;
```

This configuration property determines the position or level that the actuator should adopt, when no updates are received by nviValue within the maximum receive time, at power-on or reset, and when an override request is received for the object. The override behavior is defined by the configuration properties Override Behavior and Override Value.

Valid Range

The valid range of this input parameter is any value within the defined limits of the SNVT concerned.

Default Value

The default value is application specific.

Override Behavior - SCPT_ovr_behave (32)

```
network input config SNVT_override nciOverBehave;
```

This configuration property is used to define the behavior of a sensor or actuator when an override request is received for the object.

Valid Range

The valid range for this configuration property is any value within the defined limits of the SNVT_override. The sensor or actuator can retain its last setting, go to a specified value, or go to the default output value.

Default Value

The default value is for the object to retain its last setting.

Override Value - SCPT_ovr_value (33)

```
network input config SNVT_xxx nciOverValue;
```

This configuration property is used to set the value that a sensor or actuator should adopt when an object is overridden and the value of Override Behavior is OV_SPECIFIED.

Valid Range

The valid range for this configuration property is any value within the defined limits of the SNVT_xxx.

Default Value

The default value is application specific.

Maximum Receive Time - SCPT_max_rcv_t (21)

```
network input config SNVT_elapsed_tm nciMaxReceiveT;
```

This configuration property is used to control the maximum time that elapses after the last update to nviValue before the actuator adopts the Default Output.

Valid Range

	Min	Max
days	0	0
hours	0	17
minutes	0	59
seconds	0	59
milliseconds	0	999

Default Value

The default value is 0s, meaning that the default output is never adopted.

Alarm Configuration Parameter

The following configuration parameters relate to the optional alarm reporting provided by the Node Object using the network variable of type `SNVT_alarm`. Figure 3.9 illustrates the relationship of each of the alarm configuration parameters.

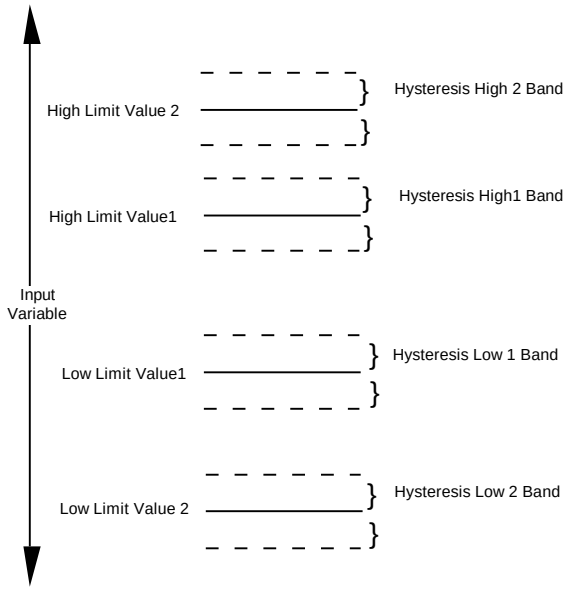

Figure 3.9 Alarm Configuration Parameters

High Limit Value 1 - SCPT_high_limit1 (9)

```
network config input SNVT_xxx nciHighLim1;
```

This configuration property is used to remotely set the alarm high limit 1 against which the value field of the input variable is tested for alarm conditions. The data type is the same as the value field of the input variable.

Valid Range

The valid range is any value within the defined limits of the SNVT concerned.

Default Value

The default is the maximum value of the SNVT concerned.

High Limit Value 2 - SCPT_high_limit2 (10)

```
network config input SNVT_xxx nciHighLim2;
```

This configuration property is used to remotely set the alarm high limit 2 against which the value field of the input variable is tested for alarm conditions. The data type is the same as the value field of the input variable.

Valid Range

The valid range is any value within the defined limits of the SNVT concerned.

Default Value

The default is the maximum value of the SNVT concerned.

Low Limit Value 1 - SCPT_low_limit1 (18)

```
config network input SNVT_xxx nciLowLim1;
```

This configuration property is used to remotely set the alarm low limit 1 against which the value field of the input variable is tested. The data type is the same as the value field of the input variable.

Valid Range

The valid range is any value within the defined limits of the SNVT concerned.

Default Value

The default is the minimum value of the SNVT concerned.

Low Limit Value 2 - SCPT_low_limit2 (19)

```
config network input SNVT_xxx nciLowLim2;
```

This configuration property is used to remotely set the alarm low limit 2 against which the value field of the input variable is tested. The data type is the same as the value field of the input variable.

Valid Range

The valid range is any value within the defined limits of the SNVT concerned.

Default Value

The default is the minimum value of the SNVT concerned.

Alarm Set Time 1 - SCPT_alrm_set_t1 (5)

```
network input config SNVT_elapsed_tm nciAlarmSetT1;
```

This configuration property is used to determine the time period that an alarm 1 condition must exist before it is regarded as a valid alarm.

Valid Range

	Min	Max
days	0	0
hours	0	0
minutes	0	0
seconds	0	59
milliseconds	0	999

Default Value

The default value is 0s.

Alarm Set Time 2 - SCPT_alrm_set_t2 (6)

```
network input config SNVT_elapsed_tm nciAlarmSetT2;
```

This configuration property is used to determine the time period that an alarm 2 condition must exist before it is regarded as a valid alarm.

Valid Range

	Min	Max
days	0	0
hours	0	0
minutes	0	0
seconds	0	59
milliseconds	0	999

Default Value

The default value is 0s.

Alarm Clear Time 1 - SCPT_alrm_clr_t1 (2)

```
network input config SNVT_elapsed_tm nciAlarmClearT1;
```

This configuration property is used to determine the time period that an alarm 1 condition must not exist before it is regarded as a valid cleared alarm.

Valid Range

	Min	Max
days	0	0
hours	0	0
minutes	0	0
seconds	0	59
milliseconds	0	999

Default Value

The default value is 0s.

Alarm Clear Time 2 - SCPT_alrm_clr_t2 (3)

```
network input config SNVT_elapsed_tm nciAlarmClearT2;
```

This configuration property is used to determine the time period that an alarm 2 condition must not exist before it is regarded as a valid cleared alarm.

Valid Range

	Min	Max
days	0	0
hours	0	0
minutes	0	0
seconds	0	59
milliseconds	0	999

Default Value

The default value is 0s.

Hysteresis High 1 - SCPT_hysthigh1 (11)

```
network input config SNVT_xxx nciLimHystHi1;
```

This configuration property determines the hysteresis level for the value field of the `nciHighLim1` comparison threshold. The data type must be the same as the value field of the input variable.

Valid Range

The valid range is any value within defined limits of the SNVT concerned.

Default Value

The default value is application specific.

Hysteresis High 2 - SCPT_hysthigh2 (12)

```
network input config SNVT_xxx nciLimHystHi2;
```

This configuration property determines the hysteresis level for the value fields of the `nciHighLim2` comparison threshold. The data type must be the same as the value field of the input variable.

Valid Range

The valid range is any value within defined limits of the SNVT concerned.

Default Value

The default value is application specific.

Hysteresis Low 1 - SCPT_hystlow1 (13)

```
network input config SNVT_xxx nciLimHystLow1;
```

This configuration property determines the hysteresis level for the value field of the `nciLowLim1` comparison threshold. The data type must be the same as the value field of the input variable.

Valid Range

The valid range is any value within defined limits of the SNVT concerned.

Default Value

The default value is application specific.

Hysteresis Low 2 - SCPT_hystlow2 (14)

```
network input config SNVT_xxx nciLimHystLow2;
```

This configuration property determines the hysteresis level for the value fields of the `nciLowLim2` comparison threshold. The data type must be the same as the value field of the input variable.

Valid Range

The valid range is any value within defined limits of the SNVT concerned.

Default Value

The default value is application specific.

Controller Object

Object Type 5: Controller Object

Controller Objects allow control algorithms to be introduced between data producing objects such as the Sensor Object and data consuming objects such as the Actuator Object. Figure 3.10 illustrates an example of several sensing devices providing data to a controller object which in turn is supplying data to several actuator devices. In the controller object definition there is no limit on the number of sensor or actuator objects that can be interfaced to a controller object. The designer can support as few or as many interfaces to additional sensor and actuator devices as desired. To aid in the description of a controller object the network variables are grouped into receiver and sender sections. A receiver group of network variables interfaces to a sensor and a sender group interfaces to an actuator.

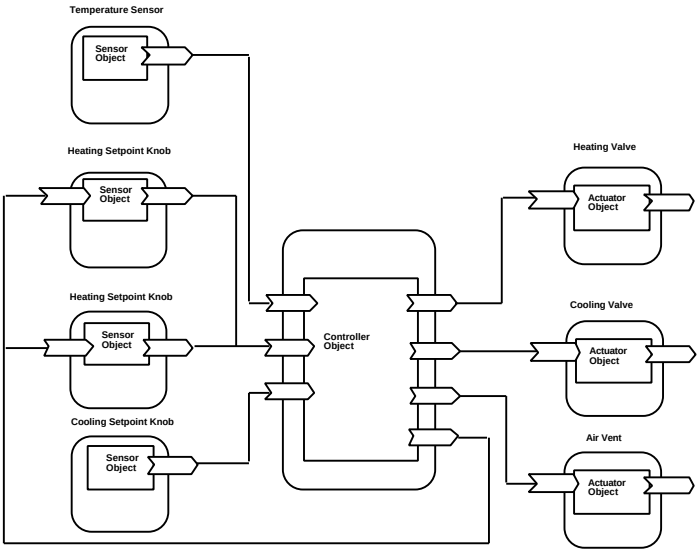

Figure 3.10 Interface Between Controller Object and Sensor and Actuator Objects

Description

The Sender section of the Controller Object comprises instances of the two network variables, nvoValue and nviValueFb which connect with the respective nviValue and nviValueFb variables of an actuator object. The nviValueFb network variable is used with closed loop actuator objects to ensure synchronization in multiple relationships. The data type used in both nvoValue and nviValueFb must match those used in the actuator object. The data types of the different sender objects do not need to match each other.

The Receiver section of the Controller Object comprises instances of the network variables nviValue and nvoValueFb which connect to the nvoValue and nviValueFb network variables of a sensor object. The data type used in nviValue and nvoValueFb must be the same as that used in the sensor object. The variable nvoValueFb is only used with closed loop sensor objects and is updated immediately when a new value of nviValue is received. The data types of the different receiver objects do not need to match each other.

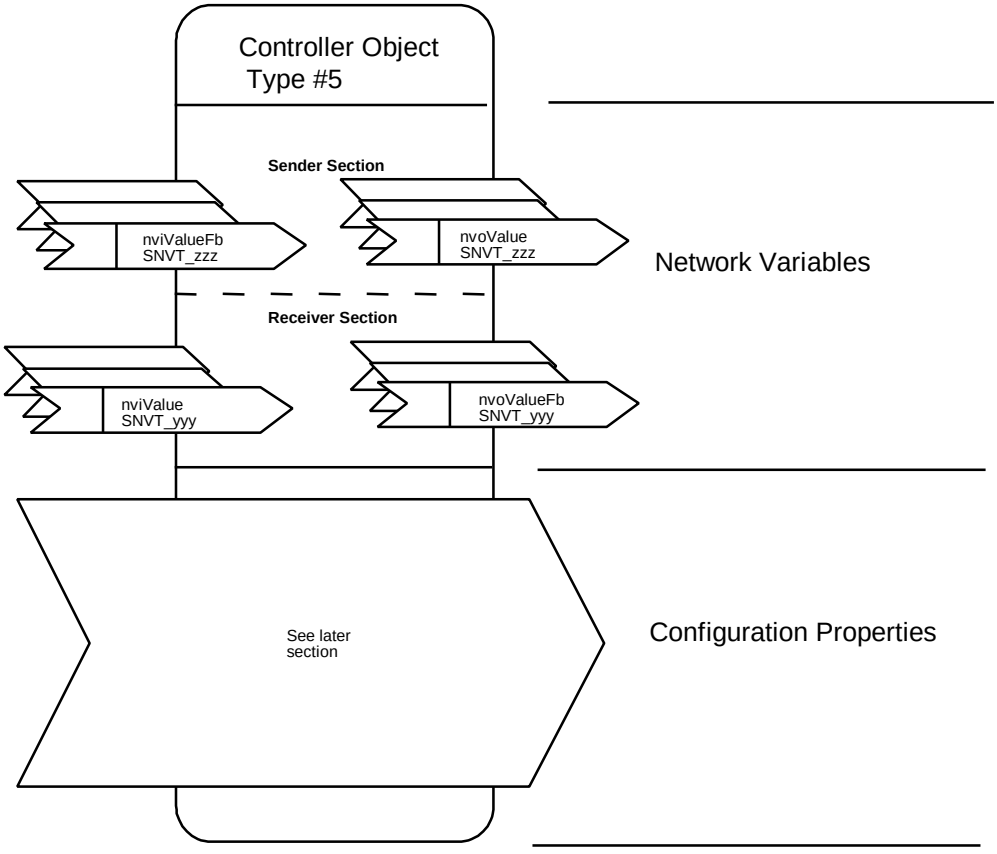

Figure 3.11 Controller Object

The Controller Object is a generic object that is designed to interface to multiple sensor and actuator objects. Functional profiles are used to describe application-specific controller objects.

Sender Section

Value Output

```
network output SNVT_xxx nvoValue;
```

These output network variables transmit data to data consuming objects.

Valid Range

The valid range is any value within the defined limits of the SNVT concerned, or it may be limited by one of the optional Min Range or Max Range configuration properties.

When Transmitted

Data is only transmitted when the input value has changed by a limit which is either defined in the node's application program or configured by the optional Send on Delta configuration property. This network variable will also be transmitted as a heartbeat output as dictated by the optional Max Send Time configuration property.

Update Rate

The maximum update rate is configured using the optional Min Send Time configuration property. The default maximum update rate is a function of channel bandwidth as specified by the Min Send Time configuration property default values.

Default Service Type

The default service type is acknowledged.

Value Feedback Input

```
network input SNVT_xxx nviValueFb;
```

These input network variables are used as a feedback return whenever a destination object's `nviValue` input has received an update. The `nviValueFb`*`nn`* network variable is used with closed loop data consuming objects to ensure synchronization in multiple relationships. The data type used in both `nvoValue`*`nn`* and `nviValueFb` must match those used in the corresponding data consuming object.

Valid Range

The valid range is any value within the defined limits of the SNVT concerned.

Receiver Section

Value Input

```
network input SNVT_yyy nviValue;
```

This network variable receives data from a data source object. Any subsequent operations on the data are manufacturer specific. The behavior will typically be determined by configuration properties in the controller object.

Default Service Type

The default service type is acknowledged.

Value Feedback Output

```
network output SNVT_yyy nvoValueFb;
```

This output network variable is used as a feedback return for `nviValue` that it uses for synchronization in multiple relationships.

When Transmitted

Immediately after an update to `nviValue` has been received, this network variable re-transmits the new value of `nviValue` as `nvoValueFb`.

Default Service Type

The default service type is acknowledged.

Configuration Properties

Depending on the network variables implemented within the Controller Object, the configuration properties will include some combination of sensor and actuator configuration properties as earlier defined. Functional profiles provide the mechanism to describe configuration properties relating specific sensor input signals to specific actuator output signals and to parameterize the specific controller algorithm used. For example, a functional profile for a PID controller would describe configuration properties for the proportional, integral, and differential gain, plus the sampling rate.

4

Configuration Properties

Even the most basic sensor node can require considerable amounts of configuration data to customize and optimize its performance for a particular application. This configuration data is referred to as Configuration Properties (CPs) throughout the LONMARK standard. Configuration properties characterize the behavior of a device in the system. Network installation tools realize this attribute and provide database storage 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.

There are three interoperable ways to implement configuration properties:

- With configuration network variables,
- With direct memory read/write standard and user-defined configuration parameter types (SCPTs and UCPTs), and
- With SCPTs and UCPTs using the LONMARK interoperable file transfer protocol.

When using user-defined configuration properties, supporting information must be provided so that the node can be integrated into a system. This information consists of the definition of the data type in equivalent format as the standard types. Documentation standards for all data types (user-defined and standard) are discussed in Chapter 5 of this document.

The first approach, to use configuration network variables to configure nodes, has been described in the object definitions in Chapter 3. Node documentation standards for this approach are discussed in Chapter 5 of this document. Using configuration network variables to implement node configuration properties has all the advantages of network variables, e.g. self-identification, self-documentation, external interface file support, and a simplified method for sending and receiving data.

For applications with large amounts of configuration data, the use of configuration parameters is a more compact mechanism for configuring a node. An additional benefit of configuration parameters is that they do not consume network variable resources.

Configuration parameters are either downloaded to a node via the LONMARK interoperable file transfer protocol or accessed via network management read and write messages.

The use of configuration parameters can free up network variable resources for use with application programs. This chapter describes how you can make use of configuration parameters in the design of interoperable LONWORKS nodes.

As with network variables that can either be user-defined or of a standard network variable type (SNVT), configuration parameters can be either of a user-defined type or of one of the standard configuration parameter types (SCPTs). SCPTs provide standard type definitions for commonly used configuration parameters such as dead-bands, hysteresis thresholds, and message heartbeat rates.

Declaration of Configuration Properties

Application developers have a choice of how configuration properties are implemented. Programmable support can be provided either via configuration network variables or via configuration parameters.

Configuration network variables can be used for relatively small amounts of configuration information. However, when the amount of configuration information is large, or if the requirements of the application demand that most of the available network variables be used for control, then the application developer can use configuration parameters instead.

Node developers are free to declare configuration properties as configuration network variables and as configuration parameters on the same device, however a given configuration property can not be implemented as both a configuration class network variable AND as a configuration property on the same device.

Configuration properties can be declared to belong to the entire node, to an object(s) or to a network variable(s). This declaration defines the *scope* of the configuration property. Configuration properties that belong to an object shall be declared as a part of the object and not declared as a part of the node. Similarly a configuration property that is associated with a specific network variable(s) shall be declared as belonging to the network variable(s) and not the object or the node.

Each configuration property within its specified scope must have a unique configuration property type. For example, if an object has three output network variables that each require an independent max send time configuration property, then the maximum send-time properties must be declared to belong to the network variables within the object. Declaring them to belong to the object would be ambiguous because it would be impossible to tell which maximum send-time value controlled which network variable in the object.

User-defined configuration properties are permitted within the Interoperability Guidelines because the configuration data for a given LONMARK object is often implementation specific. Modification of this user-defined configuration data may be necessary for the successful commissioning of the device. Any user-defined configuration properties necessary to successfully commission the device must either be defined with documentation and machine-readable resource files or must be modifiable with a passive tool. Utilities for creating and reading resource files and associated documentation about resource files can be obtained from the LONMARK site (www.lonmark.org).

Guideline 4-1: A given configuration property can be implemen-ted as either a config class network variable or as a configuration parameter.

Guideline 4-2: User-defined configuration properties, which must be modified for successful commissioning of the device, must either be defined with resource files and user documentation or a passive tool must be provided. The documentation and resource files, if any, must be supplied at the time the node is certified. The passive tool, if any, must be available at the time the node is certified.

Guideline 4-3: If a passive tool is required for successful commissioning of a device, the tool shall conform to the definition of a passive tool in Chapter 1 of the document, and shall be available to anyone owning the device on equivalent business terms, and such availability shall be demonstrably free of any discriminatory terms and conditions.

Guideline 4-4: If no passive tool is required for the successful commissioning of a device, any and all configuration information required to be modified to successfully commission the device shall be

implemented and exposed via LONMARK con-figuration properties and user documentation.

Config Class Network Variables

Configuration properties can be declared as network variables using the *config* storage class keyword. The programming reference for the network variable name should follow the programming reference conventions.

For example, the following statement declares network variable configuration data for the Max Send Time configuration property.

```
network input config SNVT_elapsed_tm nciMaxSendT;
```

Associated with each configuration network variable declaration is a mandatory self-documentation string (not shown in the above example) that describes which object or object member is associated with that configuration network variable, the configuration property index, whether the configuration property is standard or user-defined, etc. These documentation standards are described in *Documentation of Configuration Network Variables* in Chapter 5 of this document. The self-documentation strings for configuration network variables resemble the configuration parameter template file records defined in this chapter. In this way, the capabilities allowed by configuration parameters are also provided for configuration network variables when used as configuration properties.

Configuration Parameters & CPTs

Configuration parameters can either be user-defined configuration parameter types (UCPT), or a standard configuration parameter type (SCPT). To facilitate interoperability a number of SCPTs have been defined for commonly used configuration properties such as those detailed in the LONMARK Object definitions. Since nodes can require considerable amounts of configuration information, the list of SCPTs is expected to grow considerably over time.

A current list of SCPT indexes is provided in the *SCPT Master List* help file or PDF file, downloadable from the LONMARKwebsite, www.lonmark.org. Additionally, the association provides the standard resource files containing the definitions of all standard network variable types, configuration property types, and enum values.

The SCPT or UCPT index along with the standard resource file or a user resource file, allows an installation tool to correctly and consistently interpret the data type and semantic meaning for a particular configuration parameter. Note that the limits imposed on data types by network variable constraints (e.g. no more than 62 network variables total on a Neuron Chip hosted application) do not apply to configuration parameters. The limit to the size of configuration property space is only limited by the 32 bit file size of the file transfer protocol. The limit on the size (in bytes) of a SCPT or UCPT is 65,534 bytes.

Configuration parameters either apply to the node as a whole, a specified LONMARK object, a collection of LONMARK objects, a specified network variable, or collection of network variables within the node.

Configuration Parameters vs. Network Variables

Configuration parameters and network variables are different in several key aspects. In particular:

- Configuration parameters may be of any length, including arrays of any size, unlike network variables that are of fixed size and limited to 31 bytes in length. Note that this means that CPTs can be used to represent more types of data than network variables; while data represented by a config class network variable can always be represented by a CPT, the converse is not true. Thus there may not be (and in many cases will not be) a corresponding SNVT for every SCPT.
- Use of configuration parameters does not reduce the number of network variables that can be declared on a node.

Configuration Parameter Files

When using configuration parameters, at least two files are embedded within the node. The first file is a template file that contains information about how to interpret the second file. The second file contains the actual values of the configuration parameters. An optional third file contains constant configuration parameters. Constant configuration parameters may be put into this third file or may be interspersed in the second file at the implementer's discretion. In the case of both Neuron and host-based nodes, these files may be stored in the node's memory, in an external serial memory, or in the case of a host-based node, in a real file system as well. These files are accessed with either the LONMARK interoperable file transfer protocol or with direct memory access via network management messages. To use the direct memory access method to read and write these files, the files must be accessible with the network management read/write memory messages. If the files reside anywhere else, e.g. on an external serial EEPROM, in the host's memory, or in a file system, then the LONMARK interoperable file transfer mechanism must be used to read and write the files.

Guideline 4-5: Host-based nodes that support configuration parameters shall use the Configuration Parameter Template file and the Configuration Parameter Value file. Other applications may also optionally use this method, or may use a derivative form of this method called direct memory read/write configuration parameters as long as the configuration parameters may be read and written with the standard read/write memory messages in the LonTalk protocol.

Configuration Parameter Template File

File Type 2

The first file defined on a node (file index 0) must be the Configuration Parameter Template File (file type 2). This file is an ASCII text file which contains a sequence of configuration records which map configuration parameters to either a node, LONMARK object, network variable or a collection of LONMARK objects or network variables. File type 2 supersedes file type 0 (described in the LONMARK *Application Layer Interoperability Guidelines Version 2.0)* as the Configuration Parameter Template File. File type 0 is obsolete and must no longer be used for any purpose. Use of SCPT config indexes which are defined after the January 1995 version of the SCPT Master List in file type 0 may not be supported by the installation tool.

Guideline 4-6: File type 0 must not be used.

Guideline 4-7: File type 2 must be used for the SCPT file template.

The format of the File Type 2 record is described below.

Configuration Record

This file has the following records:

The first record is:

A major version number followed by an ASCII period followed by a minor version number, followed by a semicolon for a record terminator. The major and minor version numbers shall each be of the range from 0 to 255 inclusive, and are implemented as unsigned 8 bit integers. The version numbers are used to describe the format of the data to follow, and are used by the installation tool as a means to interpret the subsequent records. The current major and minor version numbers are specified for each record by these guidelines. Node developers may not use these version numbers for the versioning of their own iterations of these files. The current version of this file is version number 1.1, which includes the new CP array features described below. The record should appear as follows:

```
1.1;
```

The minor version number will increment with any backward compatible changes to this file format. An example of a backward compatible change would be the addition of information at the end of the record. An example of a non-backward compatible change would be a change to the meaning of one of the fields already defined.

Subsequent records are:

```
<hdr>,<select>,<flg>,<config_index>,<length>,<dimension>,<range_modification>;
```

Each field in the record is delimited by the ASCII comma character. The record is terminated by an ASCII semicolon character. Two consecutive commas indicate that the field is null, except when a semicolon is encountered. In this case, all remaining

fields are considered null. Only the <select> field, the <dimension> field , and the <range_modification> field may be null. If the <range_modification> field is null, the record is terminated with a single semicolon after the <dimension> field. If the <range_modification> field and <dimension> fields are null, the record is terminated with a single semicolon after the <length> field.

The first field is an enumeration called <hdr> which is used to specify whether the configuration record applies to the entire node (indicated by a zero (0) in the <hdr> field), to objects (indicated by a one (1) in the field), or to network variables (indicated by a two (2) in the <hdr> field).

The second field, a variable length field called <select>, is a sequence of zero to n bytes. The <select> field is used to specify whether the record applies to a single object or network variable within the node, or to a group of objects or network variables defined within the node. If the <hdr> field specified that the record applied to the whole node then this field is a null field. To encode a null field in the middle of a record, delimit the null field with the required comma. Thus a null field looks like two consecutive commas-- ,,.

Network variables (or network variables within a LONMARK object) are identified by network variable index (corresponding to the order of declaration in the application program. LONMARK objects are identified by object index. These indices are used in the <select> field as follows

- To encode the record to apply to a single network variable or to a single network variable embedded in a LONMARK object, encode the <select> string with the network variable index in ASCII
- To encode the record to apply to a single object, encode the <select> string with the object index in ASCII.
- To specify a range (of network variables or objects) where the entire CP (array or scalar) is shared among the range, specify the first index and the last index within the range. Separate the two indices by a hyphen (-). For example, to have a record apply to network variable indexes 4 through 12, encode the <select> string with the following ASCII characters: 4-12.
- To specify a collection of either network variables or objects with indices 2, 4, and 12 where the entire CP (array or scalar) is shared among the range, encode the <select> string with the object indices separated by a single period (.), i.e., 2.4.12.
- To specify a range (of network variables or objects) where the individual CP array elements are distributed sequentially among the members in the range, specify the first index and the last index within the range. Separate the two indices by a tilde (~). For example, to have a record containing an array of CPs have each element of the array

apply sequentially to network variable indexes 4 through 12, encode the <select> string with the following ASCII characters: 4~12.

- To specify a collection of either network variables or objects with indices 2, 4, and 12 where the individual CP array elements are distributed sequentially among the members in the range, encode the <select> string with the object indices separated by a forward slash (/), i.e., 2/4/12.

When multiple network variables or multiple objects are specified with this field using either the hyphen (-) or period (.) delimiters it is intended that a *single CPT* will apply to *all* the specified network variables or objects. In this way, the storage of a single CPT and any associated code and timers to use with it can be shared among multiple network variables or objects. This can result in substantial memory and code space savings.

When multiple network variables or multiple objects are specified with this field using either the tilde (~), or forward slash (/) delimiters it is intended that an array of CPs will have a single element of the CP array applied to each object/network variable in sequence. 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. This encoding cannot be used unless the CP array has exactly the same number of elements as the number of network variables or objects specified in the range or list of network variables or objects.

`<flg>` is a two digit hexadecimal number encoded in ASCII. This field is encoded as a sequence of two ASCII digits. The first digit selects between a standard configuration parameter type and a user-defined configuration parameter type while the second digit selects the options for modification of the parameter. If the first digit is encoded as an ASCII '0', the `flg` designates a standard configuration parameter type. Instructions for encoding this first digit to designate a user-defined configuration parameter type and a user-defined resource file are provided later in this proposal with the discussion of UCPTs and management of UCPT resource files.

The second digit of the `<flg>` field encodes attributes for the configuration parameter that instruct the installation tool as to:

- Whether the configuration parameter is constant or can be modified by an installation tool,
- Whether the object needs to be disabled prior to modification,
- Whether the node must be reset for the change to take effect, etc.

This second digit is encoded as an ASCII character representing two hexadecimal digits. The rationale for this is that some of the conditions are not mutually exclusive. For example, the changing of a configuration parameter could both require that the object be disabled and *also that* a reset MUST occur for the change to take effect. For this reason, 7 bit masks are defined which can be bitwise ORed together to produce the instructions for the installation tool. Once any ORing has been done, the result is an ASCII escape character in the range of `0x00 - 0x7F`. To avoid problems with non-printable characters, and with the possible encoding of a

null within a string, 0x80 is then ORed into the second character of <flg>. This results in the high order bit always being set. The means for entering non-printable characters into an ASCII file is dependent upon the tool used to create the file, however, most text editors have a means to enter non-printable characters. In any case, the flags below must be only one byte of the ASCII template file. One of the flags is reserved for future use. The remaining flags control when a configuration parameter may be modified. The definition of the flags after the most significant bit is set is:

```
0xC0 reserved            /* reserved for possible future use  */
0xA4 device_specific_flg /* Constant, but varies by device    */
0x90 mfg_flg             /* modified only during manufacture  */
0x88 reset_flg           /* reset after modifying             */
0x84 const_flg           /* 0=>Read/Write, 1 => constant      */
0x82 offline_flg         /* 1=>modify only when offline       */
0x81 obj_disabl_flg      /* 1=>modify when object is disabled */
0x80 no_restrictions     /* modify anytime                    */
```

The device_specific_flg designates a configuration parameter that will always be read from the device instead of relying upon the value in the node interface file or a value stored in the network management database. Network management tools must never change this parameter except as a side effect of a new program download. A flag value with 0xA0 set but without the 0x04 ORed with it is reserved for future use.

The const_flg designates a configuration parameter that is never changed by a network management tool. However, network management tools may write such configuration parameters when residing in value file index 1 as long as the value is not changed. Configuration parameters with the const_flg but without the device_specific_flg can be assumed to have the same value on all nodes using the same program ID.

The mfg_flg is used to store factory settings which need to be read/write during manufacture, but are not normally (or ever) modified in the field. In this way a standard installation tool may be used during manufacture to calibrate a node, while a field installation tool would observe the flag in the field and prevent or require a password to modify the value.

The offline_flg is only applicable to direct memory read/write configuration parameters or network variable configuration properties. The offline_flg is ignored for file transfer configuration parameters. This is because file transfer can not function while an application is in the offline state. In fact, an offline application must be placed into the online state for the duration of any file transfer configuration parameter operations.

It is recommended that for direct memory read/write configuration parameters that the obj_disabl_flg or offline_flg be used to prevent problems arising from configuration parameters being modified via the network and being read by the application concurrently. For file transfer configuration parameters, it is recommended that the obj_disabl_flg be used or that the application provide some other means for safeguarding configuration parameter access while they are being modified.

A network management tool may elect not to disable a LONMARK object prior to modifying a configuration parameter with the `obj_disabl_flg` restriction if that node is currently offline (and can be updated while offline). This is allowed because an offline node has all its objects implicitly disabled, and because an object may not be disabled when the node is offline.

`<config_index>` contains either the SCPT index or the UCPT index within the file specified in the `<flg>` field above. Note that selecting a SCPT in the `<flg>` field above causes the SCPT type file on an installation tool to be referenced to locate the SCPT specified in the `<config_index>` field. SCPT indexes 250 through 255 are special indices that are never used in this ASCII version of the file and are reserved in the definition for future expansion.

`<length>` is a field specifying the length of the SCPT/UCPT in bytes. If an array is specified in the `<dimension>` field, the length refers to only one element of the array.

`<dimension>` contains the dimensions of the SCPT. The primary use for the `<dimension>` field is for translation tables used for the linearization of sensor data. To make a linearization table with 10 X-Y points, a node developer supplies a dimension to the SCPT or UCPT in this field with the integer number of entries, e.g. 10.

`<range_modification>` is a pair of either fixed point or floating point numbers delimited by a colon, ':'. Other data types, e.g. enum, char, boolean, etc. cannot have their ranges restricted. The first number is the lower 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 parameter definition, then the field is empty to specify this. In the case of a structure or an array, if one member has a range modification, then all members must have a range modification specified. In this case, each range modification pair is delimited by the ASCII '|'. To specify no range modification for a member of a structure (that is, revert to the default for that member), encode the field as '|'. The same encoding is to be used for structure members which cannot have their ranges modified due to their data type. Note that the encoding '|' is only allowed for members of structures. Whenever a member of a structure is not a fixed or floating point number, its range 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>` field 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 '-' character. Floating point numbers use a '.' character for the decimal point. Fixed point numbers must be expressed as a signed 32 bit integer. Floating point numbers must be within the range of an IEEE 32 bit floating point number. To express an exponent, precede the exponent by an 'e' or an 'E' and then follow with a integer value.

Specifying UCPTs in the <flg> Field

When a UCPT is specified by <flg> there must be a pointer to a resource file which contains the UCPT definition. This file is analogous to the STANDARD.TYP and SNVT.TYP file, but with some additional complexity. The complexity arises from the fact that there is one and only one STANDARD.TYP file, but there may be many files which contain type definitions for user-defined configuration properties. Thus the installation tool must be able to pick the correct UCPT definition file and use the UCPT definition file to find the UCPT and its accompanying type definition as specified by the UCPT index in the node configuration parameter template file. Note that UCPT indexes, unlike SCPT indexes are not unique. Every node developer is free to make up their own UCPT indexes and they must only be unique within their own files. A similar problem exists for text based resource file information which may be pointed to in the various node self documentation strings. This section-specifies how resource files containing UCPT definitions are specified by the node developer, and also how the correct files are found and accessed by the installation tool.

To point to a resource file containing UCPT definitions, the node developer needs to reference the correct file among some number of files that a company may be maintaining. Files are referenced by information within the standard program ID field. Thus, knowing the node's standard program ID one can reference a particular record within the file. The possible references can be thought of as combinations of the meanings of the fields within the standard program ID. These fields are Manufacturer Number (mfg), Device Class (dc), Device SubClass (dsc), and Model Number (mod). A given manufacturer may provide a UCPT definition file that contains the UCPT type definitions for all his nodes of a specific Device Class. Alternatively, a manufacturer may provide several files arranged hierarchically. One file would correspond to all the products. Another family of files provided would correspond to that manufacturer's products of each device class, while a third family of files that are even more specific would go down to the device sub class. Finally, a fourth family of files are specific to a single model within a mfg, dc, dsc designation.

The scope of the file containing the UCPT type definition information must be specified in the first record of the file. Along with the scope, a simple version number with the same rules for incrementing as file type 2 shall be in the first record. After that, a file containing the UCPT indices and type definitions, ranges, etc. follows.

The node developer's requirement is that the method of referencing the file must be compact in terms of the storage required. If we look at the combinations of mfg, dc, dsc, mod we get six combinations which make sense. These are enumerated as:

1. All nodes of a specified device class from any manufacturer. This scope designation is reserved for the exclusive use of the LONMARK association.
2. All nodes of a specified device class and device subclass from any manufacturer. This scope designation is reserved for the exclusive use of the LONMARK association.
3. All nodes of a specified manufacturer.
4. All nodes of a specified manufacturer and device class.
5. All nodes of a specified manufacturer, device class and device subclass.

6. All nodes of a specified manufacturer, device class, device subclass and model -- a single node type from a single manufacturer.

Taking this information together, a node developer specifies a UCPT resource file by specifying an ASCII encoded number from 1 to 6 in the file type 2 data as the first digit of the <flg> field. This number is called the selector (and in some documents it is also called the mode). (Recall that the first digit encoded as the ASCII character '0' specifies that a SCPT is being used.) By using the character encoding of '1' to '6' along with the standard program ID of the node a UCPT type file is specified. The role of the installation tool is to install the UCPT type files so that it can determine if it has a type file corresponding to the encoded character plus standard program ID.

Such a UCPT type file is a binary file. Its first record is encoded with a version number and following that, the various fields in the standard program ID. Fields that match any pattern are to be given an otherwise impossible value. For this reason, the 8-byte standard program ID will be encoded in this record since all values are possible legal values. Subsequent records are also in binary and contain the UCPT typing information.

With the information in the Parameter Template File as described above it is possible to parse the Parameter Value File to elaborate its self-documentation using supplied resource files and to access any desired configuration parameter value.

The LONMARK Association provides a set of resource file APIs for node developers to generate resource files. These APIs are provided in binary format for Windows based computers, and the subset necessary to read resource files Is provided as a set of C source files for porting to other platforms.

Special Considerations - File Type 2

In cases where an object has multiple input and output network variables some of the configuration parameters may apply exclusively to a network variable member of the object rather than to the object as a whole. For example, a single object with multiple outputs may require different heartbeats, send on delta values, etc. depending on the individual output. In cases like these, do not use the object specifier in the <hdr> field. Instead, specify that the configuration parameter applies to a network variable and encode the network variable index in the <select> field. In this way, the ambiguity of which configuration parameter applies to which variable within the object can be handled while retaining the information that the configuration parameter is a member of the object. The network variable index maps the configuration parameter to a specific network variable while the object documentation maps specific network variables to objects.

A simple test can be defined for whether a configuration parameter applies to an object or to a member within the object. The test is, if one were to add another output to the object, would the configuration parameter need to be duplicated. An example where the answer is likely no is location labels (an entire object has a single location). Examples where the answer is likely 'yes' are those having mostly to do with the control of the propagation of a value like heartbeat rates, guard band parameters, etc.

Some Examples of File Type 2 Records

Suppose object index 2 in a node is a closed loop sensor object (type 4) with the mandatory network variables implemented along with the optional nvoActPosnFB also implemented. The network variables implemented are nviValue (nv index 6), nvoValueFB (nv index7), and nvoActPosnFB (nv index 8).

The configuration parameters implemented are:

1. Maximum receive time, SCPT index 21, (with a range modification and instructions to take the node offline and then reset after completing the modification),
2. Feedback delay, SCPT index 15, for the mandatory input, and
3. A single Turn off delay value, SCPT index 30, for both outputs.

An example of each of these three records is provided here.

```
1,2,0\x8A,21,7,1,:0|:0||||; /* select object, range restr, offline & reset   */
2,6,0\x8A,15,7;             /* select nv, offline & reset, no range restr    */
2,7.8,0\x8A,30;             /* select 2 nvs, offline & reset, no range restr */
```

Configuration Parameter Value File

File Type 1

The second file (file index 1) defined on the node is the Configuration Parameter Value file (file type 1). Optionally, there may be a third file (file index 2) defined on the node that is a constant configuration parameter value file. This third file is of type 1. Both files are binary files containing the actual values of the configuration parameters. The Configuration Parameter Template file described above can be thought of as a structure template for these files. The data values in the Configuration Parameter Value file must appear in exactly the same order as the configuration records in the Configuration Parameter Template file. If no constant value file is present, the value file contains all values, constant and modifiable. Otherwise each of the two value files contains the values of appropriate type in exactly the order they appear in the template file. The Configuration Parameter Value and Constant Value files maintain the bit and byte ordering of the Neuron Chip irrespective of the host processor.

Example

This example is an open-loop 16-bit analog sensor object with configurable minimum and maximum send timers, low and high alarm limits with hysteresis, and alarm set and clear timers. Following are the declarations of the configuration parameter template file SCPT_File2, the configuration parameter value file SCPT_File1, the file directory table FileDirectory, and the output network variable in the node object nvoFileDirectory.

```
const char SCPT_File2[] = // Configuration Parameter Template
                          // File, Type 2
"1.1;"              // current version number of file type 2
"1,1,0\x83,22,7;"   // SCPT_max_snd_t
"1,1,0\x83,24,7;"   // SCPT_min_snd_t
"1,1,0\x83,5,7;"    // SCPT_alrm_set_t1
"1,1,0\x83,2,7;"    // SCPT_alrm_clr_t1
"1,1,0\x83,9,2;"    // SCPT_high_limit1
"1,1,0\x83,18,2;"   // SCPT_low_limit1
"1,1,0\x83,11,2;"   // SCPT_hyst_high1
"1,1,0\x83,13,2;"   // SCPT_hyst_low1
;
far eeprom struct { // Configuration Parameter Value File
             SNVT_elapsed_tm     MaxSendT;
             SNVT_elapsed_tm     MinSendT;
             SNVT_elapsed_tm     AlarmSetT1;
             SNVT_elapsed_tm     AlarmClrT1;
             SNVT_xxx            HighLim1;
             SNVT_xxx            LowLim1;
             SNVT_xxx            LimHystHi1;
             SNVT_xxx            LimHystLow1;
} SCPT_File1 = {
             ZERO_ELAPSED_TM,           // MaxSendT
             DFLT_MIN_SEND_TM,          // MinSendT
             ZERO_ELAPSED_TM,           // AlarmSetT1
             ZERO_ELAPSED_TM,           // AlarmClrT1
             MAX_xxx,                   // HighLim1
             MIN_xxx,                   // LowLim1
             ZERO_xxx,                  // LimHystHi1
             ZERO_xxx };                // LimHystLow1

#define NUM_FILES 2              // directory size
typedef struct {
      unsigned long   Size;     // file size
      unsigned long   Type;     // file type
      void *          pData;    // pointer to data
} FileDescriptor;                       // directory entry

const struct {   // File Directory Table
      int       Version;
      int       NumFiles;
      FileDescriptor Files[NUM_FILES];
} FileDirectory = {
      0x20, // Major & Minor version number in a single byte
      NUM_FILES, {
      { sizeof(SCPT_File2), 2, &SCPT_File2 }, // type 2,template file
      { sizeof(SCPT_File1), 1, &SCPT_File1 } // type 1,value f ile
      }
};
```

Access to CPTs via the LONMARK Interoperable File Transfer Protocol

Each file defined on a node has both an index and a file type. Files are indexed from 0 to 65,535 on the node. File indices identify an instance of a file type on a node. Thus there may be multiple files of the same type defined on a single node, but with different file indices. When a node implements the LONMARK interoperable file transfer protocol, a network variable of type `SNVT_file_status` is used in conjunction with the `SNVT_file_req` variable by the installation tool to find out which files are present on that node. These file transfer network variables are declared as a part of the node object. Part of the file status structure is the type of the file. The Configuration Parameter Template file is of file type 2 and the Configuration Parameter Value file is of file type 1. So that directory search time is limited, the Configuration Parameter Template and Configuration Parameter Value files must be defined as file indexes 0 and 1. When using the file transfer method of accessing the configuration parameter template file, node developers have a choice to either mix constant and read/write parameters into a single file or alternatively declare a second file of type 1, index 2 on the node, that holds the constant parameters.

Guideline 4-8: The first file index defined on the node must be the Configuration Parameter Template File (file type 2) and the second file index defined on the node must be the Configuration Parameter Value File (file type 1).

Guideline 4-9: If the optional constant Configuration Parameter Value file is defined on a node, it must be the third file defined (file index 2), and it must be of file type 1.

Configuration parameters may be downloaded to a node in a file using the LONMARK interoperable file transfer protocol. A tool that sets the node's configuration parameters would typically first sequentially upload a file from the node containing the template of configuration parameters defined on the node. Secondly, it would upload the current values of the configuration parameters. Finally, the installer would change some or all of the configuration parameters and the tool would download the new values to the node.

To support the definition of a system prior to physically installing the system, the configuration parameter template and value files may be stored in a external interface file (.XIF and .XFB extensions), so that a network management tool need not upload the template file or value file(s). Both the template file and the values of the constant configuration parameters, other than device specific configuration parameters, must be the same on all devices with the same program ID. Read/Write values present in an external interface file indicate default values.

Guideline 4-10: The transfer of the Configuration Parameter Template file and the Configuration Parameter Value file must be accomplished using message code 0x3E.

Guideline 4-11: File type 0 through file type 127 and file type 256 through file type 32,767 are reserved for LONMARK usage. File types 128 through file type 255 and file type 32,768 through 65,535 are available for user-defined file types.

The LONMARK Interoperable File Transfer Protocol uses explicit messaging as described in the LonWorks Engineering Bulletin entitled *File Transfer* (005-0025-01). As with all explicit messages, a message code must be provided by the application. For LONMARK certified nodes sending file transfer messages the message code must be set to 0x3e.

Example

The following is an example of an installation scenario where the installation tool knows nothing about the device from any previous installations.

1) The installation tool first uses the `SNVT_file_status` and `SNVT_file_req` network variables to determine which files are on the node.

2) If a file of index 0 exists on the node and is of type 2 then the installation tool selects the file of type 2 on the node and reads it.

3) Then the installation tool selects file index 1 and reads the file of type 1. If file index 2 exists, and is also of type 1, then this file is a constant file and is read at this time as well.

4) The information contained in file type 2 is then translated into the native language of the installer via a resource file and the current values for each configuration parameter are displayed next to the configuration parameter description.

5) In the case of non-standard configuration parameters, the installation tool refers to a manufacturer provided resource file which describes the proprietary configuration parameters to the installer.

6) The installer then modifies some values and the installation tool converts them into the binary format of the Configuration Parameter Value file and transfers the file into the node.

7) If random access of the file is supported by the node, the installation tool may calculate the byte offset to the modified portion of the file and transfer only the records that were modified by the installer. This method is described in detail in the LonWorks *File Transfer* engineering bulletin (005-0025-01).

Access to SCPTs via Network Management Read/Write Messages

Configuration parameters can also be accessed on a node via network management read and write memory messages. The file directory network variable of type `SNVT_address` is included in the node object. The value of this network variable is

the address of a file descriptor table that points to memory areas that contain the files on this node. This mechanism is usable only if the configuration parameters are accessible by the standard read/write memory network management commands. Other nodes must support either file transfer or configuration network variables.

Example

Following are the declarations of the file directory table `FileDirectory`, and the output network variable in the node object `nvoFileDirectory`. In this example the value file is in a single, contiguous read/write memory area. For this example, the version number is encoded to 2.0 to indicate the most recent version of the directory structure that allows for a 16-bit file type. This change makes the direct memory read/write files capable of handling the same range of file types as the files implemented with the LONMARK interoperable file transfer protocol. Version 2.0 supports up to 256 distinct files on the node. The file type has been expanded to 16 bits so user-defined file types may be defined and use the direct memory read/write access method. The first two files defined must be the configuration parameter template and configuration parameter value file, respectively. If a third file is of type 1, it is the constant configuration parameter value file. Additional, application specific, files may then be defined. Note that version 2.0 is incompatible with previous versions and may not be supported by older installation tools. For this reason, node developers are free to continue to use version 1.1 of this data structure. An example declaration of the two structures is shown below.

If the configuration values were composed of parameters that were of class read/write as well as constant, the file directory structure would contain a second pointer to a file of type 1. This second file of type 1 holds the constant configuration parameters. Note that the template file will point to which memory area to access, and that the constant parameters do not have to be contiguous in the template file.

```
#define NUM_FILES 2             // directory size
typedef struct {
      unsigned long       Size;  // file size
      unsigned long       Type;  // file type
      void *              pData; // pointer to data
} FileDescriptor;                // directory entry

const struct {        // File Directory Table
      int                 Version;
      int                 NumFiles;
      FileDescriptor      Files[NUM_FILES];
} FileDirectory = {
      0x20,  //major/minor version number in 1 byte
             // major version number incremented due to
             // file type being expanded to 16 bits
      NUM_FILES, {
      { sizeof(SCPT_File2), 2, &SCPT_File2 },
      { sizeof(SCPT_File1), 1, &SCPT_File1 }
      }
};
```

```
const network output sd_string("&0|8;File Directory")
bind_info(ackd)SNVT_address
                nvoFileDirectory = (SNVT_address)&FileDirectory;
```

The self-documentation string in the declaration above, "0|8;", indicates that the network output variable belongs to the node object and the variable is member 8 of that object.

In the next example, the declarations necessary to provide a mixed configuration parameter value area containing both constant and read/write configuration parameters is provided. Note that there are two files of type 1 defined and that the version number is 2.0.

```
#define NUM_FILES 3             // directory size
typedef struct {
unsigned long       Size;       // file size
unsigned long       Type;       // file type
void *  pData;                  // pointer to data
} FileDescriptor;               // directory entry


const struct {// File Directory Table
int                 Version;
int                 NumFiles;
FileDescriptor      Files[NUM_FILES];
} FileDirectory = {
       0x20,    // major/minor version number in 1 byte
       NUM_FILES, {
       { sizeof(SCPT_File0), 2, &SCPT_File2 },
       { sizeof(SCPT_File1RW), 1, &SCPT_File1RW }, // read-write
       { sizeof(SCPT_File1RO), 1, &SCPT_File1RO }  // Constant
       }
};

const network output sd_info("0|8;") SNVT_address
               nvoFileDirectory = (SNVT_address)&FileDirectory;
```

The directory structure for version 1.1 supports only 256 files on the node. The Neuron C definition of this structure is:

```
typedef struct {
                    unsigned long fileSize;
                    unsigned int  fileType;
                    const  void *  fileData;
} FileDescriptor;
```

```
typedef struct {
    int             version;
    int             numFiles;
    FileDescriptor  files[NUM_FILES];
} FileDirectory = {
0x11,    // major/minor version number in 1 byte
       NUM_FILES, {
       { sizeof(SCPT_File0), 2, &SCPT_File2 },
       { sizeof(SCPT_File1RW), 1, &SCPT_File1RW }, // read-write
       { sizeof(SCPT_File1RO), 1, &SCPT_File1RO }  // Constant
};
```

File Considerations

A node may use direct memory read/write access for configuration parameters and also for application specific files defined on the node for other purposes. A node may support both direct memory read/write access and file transfer only if all files defined on the node are accessible using both methods. A node may also exclusively use file transfer for configuration parameters and also for application specific files defined on the node for other purposes.

Assuming the delay in file access is largely attributable to networking delays, the direct memory read/write access method is substantially slower for large files since every packet is acknowledged across the network.

5

Documentation

The documentation of a product's network interface and functionality is a critical element in allowing the product to be integrated and operated effectively. The LONMARK guidelines support a hierarchical product documentation structure that provides a very efficient mechanism for products to be self-documenting. Although specific documentation details are covered throughout the Interoperability Guidelines, this chapter provides a summary of all documentation services and interoperability requirements.

LONMARK documentation supports a series of documentation structures that allow an installer to find out basic information regarding a product either over the network from the node itself or through the node external interface (*.XIF) file. These documentation structures are filled with information at compilation time and, for changeable types and location strings, at installation time.

At the application layer the documentation structures identify which functional profiles are supported, which additional LONMARK objects are being used, what separate network variables are supported, what configuration parameters are supported and unique product information. The benefit of providing self-identification and self-documentation information is that all of the information needed to integrate and manage the product can be read by the network installation tool over the network, and no accompanying manufacturer documentation is needed. Special considerations for multiple language support are detailed in the following sections and summarized in the section entitled Multiple Language Support.

Node Self-documentation

Self-documentation String

At the highest level of the node documentation is the node self-documentation string. This string is up to 1024 bytes in length and is specified by the following Neuron C compiler directive.

```
#pragma set_node_sd_string "sss"
```

For LONMARK products, this string is used to map object indices defined on the node to their LONMARK object types.

The LONMARK self-documentation string begins with a header to identify it as such. The header is the ASCII character '&' followed by two ASCII numbers that represent the major and minor version number of the interoperability guidelines that the node implements. These two numbers are separated by a period. This substring is followed by the ASCII character, @, that is followed by ASCII encoded numbers of the standard object types defined on the nodes. These numbers are delimited by the ASCII comma and terminated by the ASCII semicolon if additional documentation follows, otherwise the string is terminated by ending the string with the ASCII double quote character ". After the semicolon, the developer is free to include any additional textual information as needed. In cases where the text must be language independent, a second header is reserved ("\x80" the ASCII string for the Hex character 80) to point an installation tool to a resource file (described later in this chapter). A third header is reserved to allow future extensions to the documentation of LONMARK Objects on a node. This third header "\x81" is the ASCII string for the Hex character 81.

For example, suppose a node had the following five LONMARK objects defined within it: one Node Object (type 0), and four Closed Loop Sensor Objects (type 2). Suppose that the developer planned to include additional installation information about the node. To set the node self-documentation string in this example node, the compiler directive would be the following:

```
#pragma set_node_sd_string "&3.2@0,2,2,2,2;Installation Text"
```

The object types 0 through 19999 are all considered standard object types (standard Functional Profile Template Ids). Object types 20000 through 25000 are set aside to be used as manufacturer specific IDs. Each manufacturer can use any or all numbers from the manufacturer specific ID range without concern about choosing unique IDs across other manufacturers.

In the example node self-documentation string above, the objects had no names associated with them. In some cases, the node developer may wish to name the objects to help the installer decide how the node should be installed. For example, naming each of multiple sensor objects that report the same type of data may aid the installer in picking the correct object

for the application. Naming of an object is accomplished by inserting either text or a resource file pointer between the object type specifier and the delimiting comma or semicolon. If the name is text, it must consist of printable characters, be of no more than 16 characters in length, and not contain any square brackets (left or right), commas, or periods, and it must not begin with any number. An example of a node self-documentation string that contains a node object and two, named open loop sensor objects (type 1) where the names are encoded as ASCII strings is shown below. In this example there was no other installation text so the terminating semicolon was omitted. Note that the text names immediately follow the object type designators.

```
#pragma set_node_sd_string “&3.2@0,1Indoor Temp,1Outdoor Temp”
```

To encode the same example, but with pointers to language independent resource files the resource file character followed by the resource index would be provided in place of the text strings. A resource index is a decimal number expressed in ASCII that identifies a string resource in a resource file. In the syntax examples below 'rinx' stands for an index into a resource file.

```
#pragma set_node_sd_string “&3.2@0,1\x80<scope>:<rinx>,1\x80<rinx>”
```

The syntax `\x80<rinx>` specifies a index into the standard language resource files: `STANDARD.<Language>`, where `<language>` is any of the supported extensions which designate the language within the resource file, for example, `STANDARD.ENU` is for U.S. English. The syntax `\x80<scope>:<rinx>` specifies a file scope level, 1 through 6, for a user file scope (see chapter 4 of this document for the file scope definitions). As with the previous syntax, `\x800:<rinx>` specifies an index into the standard language resource files.

These syntax examples can be further extended to communicate an array of objects to the installer. Suppose that the node contained an array of 16 digital inputs (declared as open loop sensor objects) and 16 digital outputs (declared as open loop actuator objects). If the installation tool knows that a set of objects is an array, it can apply the name supplied plus an index to the installer to designate the object. The node self-documentation string for this type of node is:

```
#pragma set_node_sd_string “&3.2@0,3[16DO,1[16DI”
```

In this example the arrays of objects were dimensioned just like in ANSI C with the exception that the closing brackets are optional and can be omitted to save space in the node. Arrays of objects, just like arrays of network variables, have an index in the node for each element of the array.

Guideline 5-1: The node documentation string format specified in Chapter 5 of the LONMARK Interoperability Guidelines must be used to encode instances of objects to their object types.

Guideline 5-2: The node documentation string must be stored in the node's memory.

Standard Program ID

This ID is used by a network management tool to identify the type of device. In order to complete the LONMARK conformance review a node must be assigned a Standard Program ID. Instructions for selecting a Standard Program ID are provided on the LONMARK site http://www.lonmark.org. This ID is specified by the Neuron C compiler directive:

```
#pragma set_std_prog_id fm:mm:mm:cc:cc:ss:ss:nn
```

Guideline 5-3: Standard Program IDs must be used.

The program ID is an 8 byte value formatted as shown in figure 5.1.

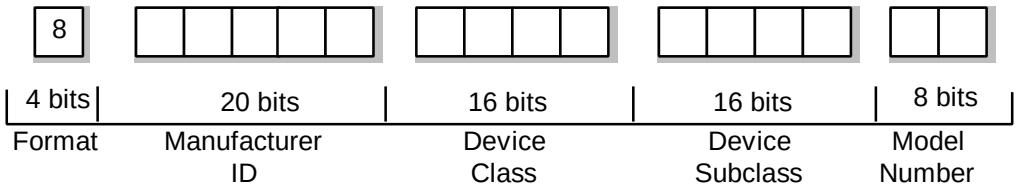

Figure 5.1 Diagram of Standard Program Identification Fields

The fields within the Standard Program ID are the following:

- ***Format***. A 4-bit value defining the structure of the program ID. Program ID formats 8 and 10 - 15 are reserved for interoperable LONMARK nodes. ID format 8 is used for Standard Program IDs. Format 9 can be used during development to test decoding of standard IDs by a network management tool. For the format shown above, format number 8 or 9 should be used.
- ***Manufacturer ID***. A 20-bit unique ID identifying each manufacturer of interoperable LONMARK nodes. This ID is assigned to a manufacturer when it becomes a member of the LONMARK Interoperability Association.
- ***Device Class***. A 16-bit ID identifying the device class. This ID is drawn from a registry of pre-defined class definitions. If an appropriate class designation is not available, one will be assigned, upon request, by the LONMARK Association Secretary.

- ***Device Subclass***. A 16-bit ID identifying a subclass within the device class. This ID is drawn from a registry of pre-defined subclass definitions. If an appropriate subclass designation is not available one will be assigned upon request.
- ***Model Number***. An 8-bit ID identifying the specific product model. Model numbers are assigned by the product manufacturer and must be unique within the device class and subclass for the manufacturer. 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 program ID does not have to conform to the manufacturer's model number.

For nodes with modifiable network variable data types, the most significant bit of the sixth byte in the standard program ID must be set to 1. The Device Sub-Class must be in the range of 80-BF (hex) for nodes with modifiable interfaces and in the range of 00-3F (hex) for nodes with non-modifiable interfaces.

Once a Standard Program ID is selected it can be installed into the LONMARK product and is available for use by any network management tool.

Node Location String

This 6-byte field is part of the configuration structure located in EEPROM. The node location string describes the physical location of the node to a network management tool. This string may be read and written over the network using the Read Memory and Write Memory network management messages. Some devices can determine their physical location by reading external physical inputs such as DIP switches, keyed connectors, card-cage slot numbers and the like. Such devices use the location field to communicate their physical location information to a network management tool that can use this information to identify the physical assignment of the device.

Guideline 5-4: A node's application program that wishes to communicate its physical location or ID assignment to a network management tool may write this information into the location ID field of its configuration structure when the node is reset. This information must be either a 15-bit unsigned integer, or a 0-6 character ASCII string. If the physical ID is a simple integer in the range 0 .. 32,767, the first byte (location[0]) contains the upper byte of this integer in binary, with the most significant bit set to one. The second byte (location[1]) contains the lower byte of the physical ID. If the most significant bit of the first byte is zero, the location field will be interpreted as a string of ASCII characters of from 0 to 6 bytes in length. If the string is shorter than 6 characters, it must be null-terminated.

Network Variable Self-documentation

As described above, the node self-documentation string is used to map objects declared on the node to their standard types. In a similar fashion, the Network Variable self-documentation string is used to define membership of a network variable to an object on the node. For example, suppose the second object declared in a node was a closed loop actuator object. This object has a single 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 node corresponds to the first output versus the second output in the closed loop actuator object.

This mapping of network variables to objects and to specific network variables within the object is done with a self-documentation string modifier. Different self-documentation string formats are used for regular network variables, configuration network variables, and manufacturer defined network variables as described below.

In the case of the closed loop actuator object, the `nviValue` is the first network variable within the object (network variables are numbered in the object definition diagrams), the `nvoValueFb` is the second network variable within the object, while the `nvoActPosnFb` is the fourth network variable within the object. The Neuron C declarations to map each of these network variables to the second object declared on the node are as follows:

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

Note that the names of the network variables used in the Neuron C program are not significant for interoperability. The notation used in this document and in the reference implementations is for convenience only.

The standard self-documentation string for network variables belonging to LONMARK objects always begins with the ASCII "@" character. Following this there are two fields with the first field indicating which object the network variable belongs to on the node. The second field is the network variable member number within the object that the declaration refers to. The two fields are delimited by the ASCII vertical bar "|" and this mapping string is delimited with a semicolon. After the semicolon, the user is free to optionally include additional documentation information. For example:

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

The ASCII string "\x80", for the Hex character 80, is reserved as a header to point the installation tool to an external resource file exactly as for the node self-documentation. Network variables that are not associated with a particular object but pertain to the node as a whole can be assigned to the Node Object as manufacturer defined network variables.

Documentation of arrays of network variables mapped onto objects is accomplished using the following format for the LONMARK portion of the self-documentation string: *"*@mm-nn|vv[ss]*"* where mm is the index of the first object, nn is the optional index of the last object in the object array, vv is the variable index and ss

is the member array size. If the last object index is omitted (nn), then the array is assumed to be a member array within the single object. The array size is only needed to convey multiple objects each containing member arrays. In this case, the NV array is mapped as if it were a two dimensional array as follows: nvs[object count][member array size].

The following example self-documentation strings illustrate how the above can be used. The first example shows an array of network variables mapped onto an array of objects. The second example shows a single object with multiple network variables as object members that are implemented as a network variable array. The third example shows an array of objects where each object has multiple network variables as object members that are implemented as a single network variable array.

```
network input sd_string("@3-6|5") nvs[4]; // Each of objects 3-6
                                          // has a member number
                                          // 5. Member 5 of ob-
                                          // ject 3 is nvs[0].
                                          // Member 5 of object
                                          // 4 is nvs[1], etc.

network input sd_string("@3|5[4]") nvs[4]; // Object 3 has a
                                          // member array of
                                          // length 4 that oc-
                                          // cupies member num-
                                          // bers 5 through 8

network input sd_string("@3-6|5[4]") nvs[16]; // Each of objects
                                          // 3-6 has a member ar-
                                          // ray of length 4 that
                                          // occupies member num-
                                          // bers 5-8. nvs[0..3]
                                          // is object 3's member
                                          // array, nvs[4..7] is
                                          // object 4's member
                                          // array, etc.
```

Manufacturer defined network variables associated with an object must use a modified self-documentation string. For example:

```
network input sd_string ("@2#3;mfr info") SNVT_xxx nviMfrin;
network output sd_string("@2#4;mfr info") unsigned long nvoMfrout;
```

The number before the # character is the object index. The number after the # character is manufacturer-assigned, and must be unique within each object. Developers who choose to add manufacturer specific network variables as a part of their interoperable interface to the objects on their nodes must provide a user defined function profile template resource file that defines member numbers for the user defined network variables. In addition, these node developers must provide a resource file that contains the typedef for the user defined network variable. These two resource files allow user defined network variable interfaces to be added to objects.

Guideline 5-5: User defined network variable interfaces within objects that are intended to be a part of the object's interoperable interface shall have a user functional profile template resource file created for the device.

Note that the names of the network variables used in the Neuron C code are not significant for interoperability. The notation used in this document and in the reference implementations is for convenience only.

Manufacturer defined network variables must have accompanying documentation and resource files so that they may be correctly displayed by a human-machine interface. This documentation shall be in the form of resource files.

Network variables may be of changeable type. Such network variables are used when the node developer cannot know the correct type of the variable in advance, as is the case with a generic sensor node which can attach to any standard sensor and report any sensed value. The self-documentation of a non-configuration network variable that may have its type modified is indicated by a ?'character after the object NV index and before the semicolon (if any). For example, the mandatory output network variable of an open loop sensor object may be declared as

```
network output sd_string("@1|1?") SNVT_xxx   nvo01Value;
```

Dynamic data typing may also be applied to manufacturer-defined network variables. In the example above, SNVT_xxx should be replaced with the default type for the network variable.

Configuration parameters associated with network variables of changeable type inherit the type of the network variable. Configuration properties that are implemented as configuration network variables must use the ?'character in their declaration.

Network variables that may change type must have their network variable self-documentation information stored in read/write memory so that an installation tool may modify the SNVT index stored in the node when the type is changed. Nodes that support one or more network variables with changeable type must reflect this in the Device Sub-Class of its Program ID, as described above.

Documentation of Configuration Network Variables

The syntax for the documentation of configuration network variables as configuration properties is similar to the method described in Chapter 4 for SCPT/UCPT usage. The self-documentation string for configuration network variables begins with the ASCII "&" character. Following this, there are the same fields as in the records (after the first record) of file type 2. The mandatory bytes are for the header field <hdr>, <select>, <flg>, <config_index>. Following this there are two optional fields for <range_modification> and <chg_type>. Note that the <length> and <dimension> fields are omitted since this information is available by virtue of the configuration network variable declaration. The dimension of a configuration network variable is its array bound.

A configuration network variable may be specified as belonging to a contiguous range of objects by specifying the first and last object index separated by an ASCII '-'character in the select field of the documentation. For example, a Maximum Send Time configuration parameter (SCPT index 22) that applies to objects 0 through 3 may be specified as shown below. In this example the <flg> field is encoded to indicate that the objects must be disabled prior to the modification of this configuration network variable.

```
config network input sd_string ("&1,0-3,0\x81,22") SNVT_elapsed_tm nciMaxSendTime;
```

Just as with SCPTs and UCPTs, one can have a single configuration network variable apply to a single network variable input or output, a list of network variables, a range of network variables, a single object, a list of objects, or a range of objects. This is accomplished by selecting the object, network variable and its range or list if appropriate in the select field. See chapter 4 for additional examples.

If a separate maximum send time configuration parameter is needed the encoding described in chapter 4 can be applied to the same example:

```
config network input sd_string ("&1,0~3,0\x81,22") SNVT_elapsed_tm nciMaxSendTime[4];
```

The self-documentation of configuration network variables that may have their type changed must include the optional <chg_type> field denoted by a ? character after the <range_modification> field and separated from it by a comma. If the <range_modification> field is null, this must be indicated by two successive commas. For example, the send-on-delta configuration network variable may be declared as:

```
config network input sd_string("&1,1,0\x80,27,,?")
SNVT_xxx nci01MinDelta = DELTA_xxx;
```

Guideline 5-6: Network variable self-documentation strings as specified in this chapter of the LONMARK Interoperability Guidelines must be used to map network variables to the objects declared on the node.

Guideline 5-7: The network variable documentation string must be stored in the node's memory.

Guideline 5-8: If the application is hosted on a processor other than the Neuron Chip then the network variable documentation strings must be provided as a response to the Query SNVT command issued by a network management tool.

Guideline 5-9: Either a passive tool must be available at certification time or all user defined network variable types that are part of the application's interoperable interface must have definitions contained in a resource file that is supplied along with the other

materials required for node certification. These files shall define the user types and enum values (.TYP), formatting information for rendering the user types (*.FMT), and string resource files for at least one language (*.<language extension>).*

External Interface File

A node's external interface file (.XIF and .XFB extensions) provides a summary of the node's interface and documentation and is automatically created by the LonBuilder and NodeBuilder tools if the node is created using Neuron C. For host-based nodes instructions for creating an external interface file are described in the *LONWORKS Host Application Programmer's Guide* (078-0016-01). The external interface file documents the node's network variables and message tags and hardware-related parameters, including the Neuron Chip type, the clock rate, the bit rate, and the medium type. The file format consists of a number of records, one record per network variable or message tag, preceded by some header information and some global values for use by the network management tool. The global values include the following information:

- The node's Standard Program ID
- The maximum number of domains to which the node can belong
- The number of address table slots
- Whether the node handles incoming explicit messages
- The number of network variables defined
- The number of message tags defined
- The node self-documentation string
- The self-documentation strings for each network variable declared on the node.

The external interface file can be used by a network management tool to allow configuration of the node. A manufacturer can deliver this file with the product or provide it to the network management tool manufacturer. The configuration property template and default value files for either direct memory read/write access or file transfer can be included in the node interface (*.XIF) file. When this information is included, the version number of the *.XIF should be version 4.0 or higher. The LONMARK Association maintains a database of the external interface files for all LONMARK-certified nodes on the LONMARK website at www.lonmark.org.

Multiple Language Support

LONWORKS nodes are installed and maintained by people with different native languages. To allow a LONWORKS node to represent information in a native language requires a language-independent way to represent strings. The need for multiple language support has implications for LONWORKS nodes and for the tools that install and maintain networks. To avoid burdening every node with extra memory requirements, the bulk of the responsibility for supporting multiple languages falls on the installation and maintenance tools.

Resource Files

To provide language independence, nodes must use numeric indices rather than explicit text strings to identify each piece of named data. The installation and maintenance tool will map indices into appropriate strings (words, phrases, or sentences) via the use of resource files. Due to the difficulties of natural language translation, conversion is done using one-to-one string substitution. Within an application program, strings are referenced using an index into an installation tool resource file.

For example, Echelon currently provides the resource files STANDARD.TYP, STANDARD.ENU, and STANDARD.ENG. The STANDARD.TYP file is language-independent, because it does not have any language-specific strings contained within it. Instead, when the STANDARD.TYP file's contents need to refer to descriptive text strings, strings for unit names, etc., it uses indices.

These indices refer to language-dependent strings within the STANDARD.ENU (U.S. English) and the STANDARD.ENG (British English) files, and could in the future refer to strings in similar files for Spanish, French, German, Italian, Chinese, Arabic, or whatever languages may be supported in the future. The index references the string with identical meaning in all variations of the language-dependent string resource file. In other words, suppose the index 14 referenced a string in the U.S. English file that was 'color,' then the index 14 would reference a string in the British English file that was 'colour'.

In this way the SNVT names do not have to be stored in the node; only the SNVT IDs are stored in the node's network variable self-identification information, and the SNVT ID can be used as an index into a file which can be changed depending on the native language of the installer.

Table 5.1 summarizes the method for indexing each type of data stored in a node for interpretation and native language translation by an installation tool using a resource file. Table 5.1 also identifies the source for default naming information. Default naming information is used when the installation tool has no other string information available to it.

Table 5.1 Multiple Language Support

Item	***Identifying Index***	***Source for Default Naming Information***
Node device type	Standard Program ID	Language-dependent device description information can be looked up in resource files based on the standard program ID.
Node location string	Location string	Language dependent location information can be looked up in resource files based on indexes stored in the location string.

Object Documentation	Node self-documentation string	Language-dependent self-documentation can be looked up in resource files based on the standard program ID and indices stored in the self-documentation string. Default self-documentation is extracted from an external interface file or the self-documentation string.

Table 5.1 Multiple Language Support (continued)

Network variable names	Network variable index/program ID	Language-dependent names can either be looked up in resource files based on indices stored in the network variable self-documentation string or can be looked up in resource files based on the program ID and network variable index. The default network variable name is extracted from an external interface file or the self-documentation string.
Network variable types	SNVT or user type index (augmented with program ID if needed)	Language-dependent names for standard types are available via the SNVT type file. For non-standard types, language-independent strings can be looked up in resource files based on the user type.

6

Host-based Nodes

A special class of nodes called host-based nodes use the Neuron Chip as a communications processor and run the application on a non-Neuron host. This chapter covers additional guidelines for host-based nodes that communicate to a LONWORKS network via a LONWORKS network interface. Nodes that use a non-LONWORKS network interface e.g. a serial or parallel interface to a host processor are already covered by the guidelines that govern Neuron-Chip-hosted nodes.

Host-based nodes must meet the guidelines described in this chapter to ensure their compatibility with Neuron Chip-hosted nodes

Network Variable Selection

Host-based nodes may be built in one of two ways, depending on where network variable selection occurs. If network variables selection occurs on the host, then the host must manage the network configuration table.

Guideline 6-1: If network variable selection is performed on the host, then the host application must manage a network variable configuration table, and respond correctly to the Update and Query NV Config messages using the data structure defined in the Neuron Chip Data Book.

Guideline 6-2: Host-based nodes must preserve network variable configuration table contents across hardware resets and power cycles.

Guideline 6-3: The host must determine whether authentication is required for a network variable, 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 must reject an unauthenticated update or poll to that network variable.

The Neuron Chip network processor informs the host processor whether or not an incoming message is authenticated. It is the responsibility of the host to determine whether authentication is required for a given network variable by checking the corresponding network variable configuration table entry.

Guideline 6-4: The host application must respond to Network Variable Fetch and Network Variable Poll request messages with the appropriate data from the requested network variable. If the application is in the Off-line mode when a Network Variable Poll is received, the response must contain no data. Otherwise, the response must contain the correct number of bytes of data as specified by the network variable type.

Guideline 6-5: The host application must deliver the niONLINE and niOFFLINE commands to the network interface when the Mode On-line and Mode Off-line messages are received.

Documentation

Guideline 6-6: The host application must ensure that the Standard Program ID is present in the network interface.

LONMARK certification is granted to the combination of the host application and the network interface. If either of these components is absent, the node should not appear to be certified. If the network interface may be detached from the host (for example, if it is a Serial LonTalk Adapter), then the program ID must be loaded into the network interface each time the host application is launched. Ideally, the program ID should be set to all zeroes when the host application terminates.

Guideline 6-7: The host application must implement the LonMark self-documentation requirements. It must respond to the Query SNVT request message with the same data that a Neuron Chip-based node would respond with. The appropriate data structures are defined in the Neuron Chip Data Book.

Dynamic Network Variables

Host nodes may have a maximum of 4096 network variables and 8192 network variable aliases in their interfaces. Because these limits are much higher than the Neuron Chip limits, host nodes are extensively used for monitoring and control applications. In the case where a bound connection from the monitoring and control node is needed, it is required that the monitoring and control node present the correct network variable interface (that is, a variable of the same type as the one being monitored). In the general case, this interface is impossible to know in advance of the installation. To solve this problem, nodes may use dynamic network variables. These network variables may be created and deleted at will, rather than being statically declared. The only static declaration required is the number of dynamic network variables and aliases supported on the node. This information appears in the .XIF file for the node and can be queried from the node using the commands described below.

Support of Dynamic Network Variables is optional, however, if a node can dynamically create and delete network variables after being installed in a network, then the method described in this section shall be used.

Guideline 6-8: If a developer chooses to incorporate the optional feature of dynamic Network Variables, the implementation shall conform to the three requirements listed immediately below.

1. The node must have a Program ID set according to the directions in chapter 5 of this document to specify its interface is changeable.
2. The node must have an external interface file of version 4.1 or later that specifies the fixed and changeable portions of the interface.
3. The node must support and respond to the extended network management messages to add and delete network variables and aliases, to query their attributes, bind them, etc. These extended messages are based upon the Install command in the LonTalk protocol. These commands are documented below.

These commands, which are an extension of the Install command (message code 0x70), provide methods to query Self Identification/Self Documentation(SI/SD) data, update SI/SD data, inform the node of a new Network Variable addition, or the removal of an existing Network Variable. The general format of the install commands for all three purposes is:

```
<Application Command> <Application Specific Data Fields>
```

The first byte of the response should either be 0x30 to indicate command successful or 0x10 to indicate command failed. Additional bytes of the response (if any) depend on the application command. The values of the application commands are:

APP_WINK (0)	Execute the application's wink clause.
APP_INSTALL (1)	Find all the network interfaces for a host.
APP_NV_DEFINE (2)	Create a new dynamic network variable definition.

APP_NV_REMOVE (3)	Remove an existing dynamic network variable definition.
APP_QUERY_NV_INFO (4)	Query SI/SD data for an network variable.
APP_QUERY_NODE_INFO (5)	Query SI/SD data for the node.
APP_UPDATE_NV_INFO (6)	Update SI/SD data for an network variable.

Following are the definitions of the application specific data fields by command type.

APP_NV_DEFINE

This command adds a new or modifies an existing network variable definition. If an attempt is made to add or modify the definition of a network variable that is part of the node's fixed network variable interface, or the index exceeds the maximum supported by the node, the node will report command failed.

After a successful define, the host is responsible for setting the network variable configuration to default values (unbound selector, correct direction, etc) such that a subsequent network variable config query command shows the correct default attributes for the newly defined network variable.

The message data structure contains the following fields:

APP_NV_DEFINE (2)		As defined above.
Unsigned	nvIndexHi	The high order byte of the network variable index.
Unsigned	nvIndexLo	The low order byte of the network variable index.
unsigned	arrayLenHi	The high order byte of the number of elements in the network variable array. The value is 0 if the network variable is not an array.
unsigned	arrayLenLo	The low order byte of the number of elements in the network variable array. The value is 0 if the network variable is not an array.
unsigned	nvLen	The number of bytes in the network variable value (1-31).
unsigned	nv_dflts	Default configurable attributes of the network variable. Encoded as a single byte. Includes direction, priority, authentication, and service type.
unsigned	nv_attr	Attributes of the network variable. Encoded as a single byte. (See definition below.)

APP_NV_REMOVE

This command removes one or more existing network variable definitions. The network variable index must be within the area of the interface that is specified to be dynamic. If the network variable index specified is within the node's fixed interface, the command will fail. The command will succeed even if the specified set of network variables is not currently defined or exceeds the maximum number of

network variables supported by the node. The command contains the following fields:

APP_NV_REMOVE(3) As defined above.

unsigned	nvIndexHi	The high order byte of the network variable index of the first network variable definition to be removed.
unsigned	nvIndexLo	The low order byte of the network variable index of the first network variable definition to be removed
unsigned	nvCountHi	The high order byte of the number of network variables to remove.
unsigned	nvCountLo	The low order byte of the number of network variables to remove.

APP_QUERY_NV_INFO

This command queries self-documentation data about a specific network variable. It contains the following fields:

APP_QUERY_NV_INFO (4)

unsigned	nv_info	The self-documentation data being requested. Current values are:
	NV_INFO_DESC (0)	Basic attributes of the network variable, array information, how the network variable was defined, and indication of available additional self-documentation.
	NV_INFO_RATE_EST (1)	Average and maximum rate estimates defined for the network variable.
	NV_INFO_NAME (2)	The name of the network variable.
	NV_INFO_SD_TEXT (3)	The self-documentation text string associated with the network variable.
	NV_INFO_SNVT_INDEX (4)	The index of the SNVT.
unsigned	nvIndexHi	The high order byte of the network variable index.
unsigned	nvIndexLo	The low order byte of the network variable index.
unsigned	offset_hi	If nv_info is NV_INFO_SD_TEXT, the high order byte of the offset of the requested data.
unsigned	offset_lo	If nv_info is NV_INFO_SD_TEXT, the low order byte of the offset of the requested data.
unsigned	length	If nv_info is NV_INFO_SD_TEXT, the number of bytes requested.

The response data structure depends upon the data requested.

If the data requested is NV_INFO_DESC, the following response is sent:

unsigned length :5	The length of the network variable value, in bytes. Encoded as 0 if the network variable is currently undefined.
unsigned origin :3	How the network variable was created. One of the following values:
NV_UNDEFINED (0)	The network variable is not currently defined.
NV_STATIC (1)	The network variable was statically defined (cannot be removed).
NV_DYNAMIC (2)	The network variable was dynamically defined (can be removed).
unsigned nv_dflts	Default configurable attributes of the network variable. Encoded as a single byte. Includes direction, priority, authentication, and service type.
unsigned nv_attr	Basic attributes of the network variable.
unsigned nv_exten	Extension bits. Indicators of additional SI/SD data that is available.
unsigned nv_array	Network variable array attributes.
unsigned nv_name[16]	Optional field. If included, the nm_supplied bit must also be set in the ext field. The name of the network variable without array subscripts.

If the requested data is NV_INFO_RATE_EST, the following response is sent:

unsigned nv_rate_est	The encoded average rate estimate for the network variable.
unsigned nv_rate_est	The encoded maximum rate estimate for the network variable.

If the requested data is NV_INFO_NAME, the following response is sent:

unsigned nv_name[16]	The name of the network variable. If network variable is part of an array, the name does not include the array subscript. Zero-terminated if length is less than 16 characters.

If the requested data is NV_INFO_SD_TEXT, the following response is sent:

unsigned length	The number of bytes of SD Text data returned.
unsigned text[*]	The SD text data segment. The actual text array size is dependent upon the preceding length value.

If the requested data is NV_INFO_SNVT_INDEX, the following response is sent:

unsigned snvt_type_index	The index of the Standard Network Variable Type associated with the network variable. Encoded as zero if the network variable is not a SNVT.

APP_QUERY_NODE_INFO

This command queries self-documentation data about the node. It contains the following fields:

APP_QUERY_NODE_INFO (5)

unsigned	node_info	The self-documentation data being requested. Values are:
NODE_INFO_SD_TEXT (3)		The self-documentation text string associated with the node.
unsigned	offset_hi	If node_info is NODE_INFO_SD_TEXT, the high order byte of the offset of the requested data.
unsigned	offset_lo	If node_info is NODE_INFO_SD_TEXT, the low order byte of the offset of the requested data.
unsigned	length	If node_info is NODE_INFO_SD_TEXT, the number of bytes requested.

The response data structure depends upon the data requested. If the requested data is NODE_INFO_SD_TEXT, the following response is sent:

unsigned	length	The number of bytes of SD Text data returned.
unsigned	text[*]	The SD text data segment. The actual text array size is dependent upon the preceding length value.

APP_UPDATE_NV_INFO

This command updates self-documentation data about a specific network variable. In general, update messages cannot be relied upon to be validated by the target application. All validation should be performed by the initiator. This message contains the following fields:

APP_UPDATE_NV_INFO (6)

unsigned	nv_info	The self-documentation data being updated. Current values are:
NV_INFO_RATE_EST (1)		Average and maximum rate estimates defined for the network variable.
NV_INFO_NAME (2)		The name of the network variable.
NV_INFO_SD_TEXT (3)		The self-documentation text string associated with the network variable.
NV_INFO_SNVT_INDEX (4)		The index of the SNVT.
unsigned	nvIndexHi	The high order byte of the network variable index.
unsigned	nvIndexLo	The low order byte of the network variable index.

The remaining data fields are dependent upon the nv_info value specified. If the specified nv_info is NV_INFO_RATE_EST, the following data is provided:

unsigned clear_mre : 1		If set, clear the network variable's maximum rate estimate value, and indicate that this data is not available.
unsigned clear_re : 1		If set, clear the network variable's rate estimate value, and indicate that this data is not available.
unsigned update_mre : 1		If set, update the network variable's maximum rate estimate value. Also, indicate that the maximum rate estimate data is available.
unsigned	update_re : 1	If set, update the network variable's rate estimate value. Also, indicate that the rate estimate data is available.
unsigned	nv_rate_est	The encoded average rate estimate for the network variable.
unsigned	nv_rate_est	The encoded maximum rate estimate for the network variable.

If the specified nv_info is NV_INFO_NAME, the following data is provided:

unsigned	nv_name[16]	The null (zero) terminated name of the network variable. The name does not include the array subscripts.

If the specified nv_info is NV_INFO_SD_TEXT, the following data is provided:

unsigned	length	The number of bytes of SD Text data being updated.
unsigned	offset_hi	The high order byte of the offset of the data to be updated.
unsigned	offset_lo	The low order byte of the offset of the data to be updated.
unsigned	text[*]	The SD text data segment. The actual text array size is dependent upon the preceding length value.

If the specified nv_info is NV_INFO_SNVT_INDEX, the following data is provided:

unsigned snvt_type_index	The index of the Standard Network Variable Type associated with the network variable. Encoded as zero if the network variable is not a SNVT.

The following data structures are referenced in the preceding message definitions:

nv_attr — This data structure describes the basic attributes of a specific network variable. It contains the following fields:

unsigned nv_sync		: 1 If set, all values assigned to the network variable are propagated, in their original order. Mutually exclusive with nv_polled.
unsigned nv_polled	: 1	If set, the output network variable's value is sent only in response to a poll. Updates are not generated when the application updates the network variable value.
unsigned nv_offline	: 1	If set, the node should be taken offline before updating the value of the network variable.
unsigned nv_service_type_config	: 1	If set, the network variable's service type attribute is configurable.
unsigned nv_priority_config	: 1	If set, the network variable's priority attribute is configurable.
unsigned nv_auth_config	: 1	If set, the network variable's authentication attribute is configurable.
unsigned nv_config_class	: 1	If set, the network variable is a configuration network variable.
unsigned snvt_type_index		The index of the Standard Network Variable Type associated with this network variable. Zero if the network variable is not a SNVT.

nv_exten — This data structure indicates what additional self-documenting data is available for a specific network variable. It contains the following fields:

unsigned	mre	: 1	If set, the network variable's maximum rate estimate is available.
unsigned	re	: 1	If set, the network variable's average rate estimate is available.
unsigned	nm	: 1	If set, the network variable's name is available.

unsigned	sd	: 1	If set, the network variable has a self-documenting text string.
unsigned	nm_supplied	: 1	Applies only to APP_QUERY_NV_INFO response for then the nv_name is provided in the response, following the existing data fields.

nv_array — This data structure provides the array attributes for a specific network variable. It contains the following fields:

unsigned	count_hi	Total number of network variables in the array. An unsigned 16 bit number
unsigned	count_lo	
unsigned	element_hi	The index of the network variable within the array. Encoded as a zero
unsigned	element_lo	if the current network variable is not a member of an array.

nv_dflts — This data structure provides the default values for the configurable attributes for a specific network variable. It contains the following fields:

unsigned	nv_direction	: 1	The default direction of the network variable.
unsigned	nv_auth	: 1	The default authentication setting of the network variable.
unsigned	nv_priority	: 1	The default priority setting of the network variable.
unsigned	nv_service	: 2	The default service type of the network variable.

The data structures corresponding to the formats of these commands are given below. These structures are in big-endian order with an int defined as a 16 bit quantity.

```
typedef struct {
   unsigned    reserved            :3;     /* Must be set to 0 */
   unsigned    nv_direction        :1;
   unsigned    nv_auth             :1;
   unsigned    nv_priority         :1;
   unsigned    nv_service          :2;
} nv_dflts;

typedef struct {
   unsigned    reserved                :1;       /* Must be set to 0 */
   unsigned    nv_sync                 :1;
   unsigned    nv_polled               :1;
   unsigned    nv_offline              :1;
   unsigned    nv_service_type_config  :1;
   unsigned    nv_priority_config      :1;
   unsigned    nv_auth_config          :1;
   unsigned    nv_config_class         :1;
   byte       snvt_type_index;                 /* use enum SNVT_t */
} nv_attr;

typedef struct {
   unsigned    mre         :1;     /* Max Rate Est data is available */
   unsigned    re          :1;     /* Rate Estimate data is available */
   unsigned    nm          :1;     /* NV Name is available */
   unsigned    sd          :1;     /* NV SD text is available */
   unsigned    nm_supplied :1;     /* For QUERY_NV_INFO/NV_INFO_DESC,
                                      name is appended to message */
   unsigned   reserved1    :3;     /* Must be set to 0 */
} nv_exten;

typedef struct {
   unsigned       length   :5;   /* # of bytes in NV value */
   unsigned       origin   :3;   /* How NV was defined */
   nv_dflts       dflts;         /* Default NV Configuration settings */
   nv_attr        attr;          /* Basic NV attributes */
   nv_exten       ext;           /* Available NV Extension attributes */
   nv_array       array;         /* NV array attributes */
} nv_desc;

typedef enum{
   APP_WINK            = 0,      /* basic command        */
   APP_INSTALL         = 1,      /* get information from multiple stacks     */
   APP_NV_DEFINE       = 2,      /* define a new network variable on a node  */
   APP_NV_REMOVE       = 3,      /* remove a network variable from the node  */
   APP_QUERY_NV_INFO   = 4,      /* get network variable information         */
   APP_QUERY_NODE_INFO = 5,      /* get node self documentation information  */
   APP_UPDATE_NV_INFO  = 6       /* modify nv info. Change its type, etc.    */
} AppCommand;
```

```
struct install_msg {
   byte command;           /* value = 16 */
   byte subcommand;        /* one of AppCommand above */
   byte subdata[*];        /* length dependent upon the command/subcommand pair   */
};

struct install_response {
   byte command;     /* value = 16 */
   union {
      struct service_pin_message;   /* if subcommand == 0 */
   } data;
};

/* Request to define new NV(s) on the node. The app should respond successfully even
   if the NV is already defined.
*/
typedef struct
{
   byte        app_command;    /* APP_NV_DEFINE (AppCommand enum) */
   int         nv_index;       /* New NV index */
   int         array_len;      /* # of elements in NV array,
                                   0 if new NV is not an array */
   byte        nv_length;      /* # of bytes in NV value (1-31) */
   nv_dflts    dflts;          /* Default NV Configuration settings */
   nv_attr     attr;           /* NV self-documentation attributes */
} NM_app_define_nv;

typedef struct
{
    byte                code;    /* NM_app_cmd */
    NM_app_define_nv    data;
} CMD_app_define_nv;

typedef struct
{
    byte    code;             /* NM_app_cmd_fail/NM_app_cmd_succ */
} CMD_app_response;

/* Request to remove a set of NVs. The host should respond successfully
   even if 1 or more of the specified NVs does not exist.
*/
typedef struct
{
   byte              app_command;        /* APP_NV_REMOVE (AppCommand enum) */
   unsigned long     nv_index;           /* Index of first NV to remove*/
   unsigned long     nv_count;           /* # of NVs to remove */
} NM_app_remove_nv;

typedef struct
{
    byte                code;               /* NM_app_cmd */
    NM_app_remove_nv    data;
} CMD_app_remove_nv;
```

```
/*************************************************************************
    Query Network Variable Self-documentation data
 *************************************************************************/

#define NV_INFO_DESC        0
#define NV_INFO_RATE_EST    1
#define NV_INFO_NAME        2
#define NV_INFO_SD_TEXT     3
#define NV_INFO_SNVT_INDEX  4

typedef struct
{
    byte             app_command;        /* APP_QUERY_NV_INFO (AppCommand enum) */
    byte             nv_info;            /* Requested NV Info (NV_INFO...) */
    unsigned long    nv_index;
    union   {
        struct {
               unsigned long  offset;
               byte           length;
        } sd;                   /* For requesting NV_INFO_SD_TEXT */
    } addl;
} NM_app_query_nv_info;

typedef struct
{
    byte                    code;               /* NM_app_cmd */
    NM_app_query_nv_info    data;
} CMD_app_query_nv_info;

typedef union
{
    /* Response for requested info NV_INFO_DESC */
    struct {
         nv_desc     desc;
         byte        nv_name[NV_NAME_LEN]; /* Optional field  - Included only if
                                              desc.ext.nm_supplied is set */
    } desc;

    /* Response for requested info NV_INFO_RATE_EST */
    struct {
        byte    nv_rate_est;        /* Encoded rate estimate. Only valid if 're'
                                       Is set in NV desc */
        byte    nv_max_rate_est;    /* Encoded max rate estimate. Only valid if
                                       'mre' is set in NV desc */
    } rate;                         /* Response for requested info NV_INFO_NAME  */
    byte    nv_name[NV_NAME_LEN];   /* NV name. Only valid if 'nm' set in NV desc */

    /* Response for requested info NV_INFO_SD_TEXT */
    struct {
        byte    length;
        byte    text[*];            /* depends upon length field      */
    } sd;

    /* Response for requested info NV_INFO_SNVT_INDEX */
    byte    snvt_type_index;
} NM_app_query_nv_info_response;
```

```
typedef struct
{
    byte                              code;  /* NM_app_cmd_fail/NM_app_cmd_succ */
    NM_app_query_nv_info_response     data;
} CMD_app_query_nv_info_response;

/*****************************************************************
    Update Network Variable Self-documentation data
 *****************************************************************/

typedef struct
{
    byte          app_command;        /* APP_UPDATE_NV_INFO (AppCommand enum) */
    byte          nv_info;            /* NV Info to be updated (NV_INFO...)   */
    unsigned long nv_index;           /* NV index - If the target NV represents an
                                          array, then the update applies to all
                                          elements of the array.             */
    union {
        /* Data for updated info NV_INFO_NAME */
            nv_name[NV_NAME_LEN];   /* NV name (ascii) - 0 terminated
                                       if < 16 characters */
        /* Data for updated info NV_INFO_SD_TEXT */
        struct {
            byte              length;
            unsigned long     offset;
            byte              text[*];            /* May not be 0 terminated */
} sd;

        /* Data for updated info NV_INFO_RATE_EST */
     struct {
      unsigned    reserved    : 4;    /* Must be set to 0 */
      unsigned    clear_mre   : 1;    /* Clear 'ext' rec 'mre' field */
      unsigned    clear_re    : 1;    /* Clear 'ext' rec 're' field */
      unsigned    update_mre  : 1;    /* Update max rate est with nv_max_rate_est */
      unsigned    update_re   : 1;    /* Update rate est with nv_rate_est */
      byte        nv_rate_est;        /* Encoded rate estimate.  */
      byte        nv_max_rate_est;    /* Encoded max rate estimate.  */
     } rate;

        /* Data for updated info NV_INFO_SNVT_INDEX */
        byte    snvt_type_index;
    } info;
} NM_app_update_nv_info;

typedef struct
{
    byte                    code;       /* NM_app_cmd */
    NM_app_update_nv_info   data;
} CMD_app_update_nv_info;
```

```
/*************************************************************************
    Query Node Self-documentation data
 *************************************************************************/

#define NODE_INFO_SD_TEXT     3

typedef struct
{
    byte    app_command;        /* APP_QUERY_NODE_INFO (AppCommand enum) */
    byte    node_info;          /* Requested Node Info (NODE_INFO...) */
    union   {
        struct {
            unsigned long  offset;
            byte           length;
        } sd;                   /* For requesting NODE_INFO_SD_TEXT */
    } addl;
} NM_app_query_node_info;

typedef struct
{
    byte                        code;               /* NM_app_cmd */
    NM_app_query_node_info    data;
} CMD_app_query_node_info;

typedef union
{
    /* Response for requested info NODE_INFO_SD_TEXT */
    struct {
        byte    length;
        byte    text[*];        /* May not be 0 terminated */
    } sd;
} NM_app_query_node_info_response;

typedef struct
{
    byte                              code;  /* NM_app_cmd_fail/NM_app_cmd_succ */
    NM_app_query_node_info_response data;
} CMD_app_query_node_info_response;
```

7

Network Installation

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

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

This chapter outlines the design guidelines that must be followed so that nodes can be installed into an interoperable network. It is also important that the network tools used to install the interoperable network support installation of nodes that follow these guidelines. Refer to the *LonWorks Installation Overview* engineering bulletin (005-0006-02) for more detail about installation tools and options. Specific interoperability design guidelines for network management tools are in review.

Network Addressing

Overview

Nodes use their network addresses to send messages and to determine if messages are destined for them. A node's network address consists of three components – the domain to which it belongs, the subnet to which it belongs within the domain, and its node number within the subnet. A node can be a member of up to two domains. A key function of a network tool is to ensure that in any domain no two nodes are assigned the same subnet and node number.

A node can also be addressed by using group addresses, assigned during the binding process. A node can be a member of up to 15 different groups. The binding process also allocates network variable selectors. Network variable selectors are 14-bit numbers used to identify connected network variables. All network variables in a connection must have the same network variable selector value. Also, the assigned network variable selector must allow each node to uniquely identify each of its connected 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 nodes.

Network addresses may be defined in a number of ways, for example:

- At the time of node manufacture.
- By each node during field installation.
- By a network tool during field installation.
- By a combination of the methods above.

Each of these methods represents a trade-off in terms of ease of initial installation, flexibility, and cost of tools. The LonTalk protocol and the interoperability 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 node application code or hardware.

Address Table Entries

Each distinct destination address for an outgoing network variable update, poll or explicit message requires an address table entry. In addition, each group to which the node belongs requires an address table entry. The maximum number of address table entries is 15, and each requires 5 bytes of on-chip EEPROM. The number of address table entries directly affects the ease of installation of the node, since network variable and message tag binders may fail if there are an insufficient number of these entries on the node. However in a memory-limited application, such as those implemented on a Neuron 3120 Chip, there is a tradeoff between

application functionality and these table entries. Wherever possible the maximum number of 15 address table entries should be supported to avoid binder failure.

Network variable aliases are another tool for the node developer to use to conserve address table entries and also to prevent network variable connection constraints. Network variable aliases are required on a node when a single network variable output on the node must be connected to two or more network variable inputs on another node. For example a single switch connected to a node containing 4 actuators where all four of the actuator inputs must be controlled by the single switch.

Network variable aliases can also conserve address table and group address entries on monitoring nodes. For example, when an output network variable on a node is connected to one or more other network inputs on another node, and that same output variable needs to be bound to the monitoring node, there are two alternatives: either have the monitoring node be a member of the group in the original connection, or allocate an alias to the output network variable and send the alias as a unicast update to the monitoring node (unicast addressing does not consume address table entries on the receiver node).

From this discussion, one can imagine that some number of network variable aliases should be available on all LONMARK certified nodes. It is also clear that the number of aliases to implement on a specific node depends upon the application and the available node resources. For these reasons, the guideline regarding network variable aliases only requires that the node developer provide a reasonable number by using their application knowledge and understanding, and taking into account the node's available memory resources.

Guideline 7-1: Wherever possible the maximum number of 15 address table entries should be supported on a node to avoid binder failure. As a minimum nodes must support a number of address table entries equal to the number of non-configuration network variables plus the number of bindable message tags, or 15, whichever is less.

Guideline 7-2: A node must support a reasonable number of network variable aliases to avoid network variable connection constraints.

Domain Table Entries

With regard to the number of domain table entries, it is often useful to have a node be a member of the zero length domain so that it may be queried without knowing its 48-bit ID. This is useful when the network data base is lost and must be recovered from the network itself. While the 48-bit ID may be acquired by activating the service pin, and the domain table read with a second command using the 48-bit ID, the service pin may not be easily accessible on nodes in some applications. For example, the node may be on a roof or behind a wall. If it is

inconvenient or not practical to activate the service pin on a node which has only a single domain table entry and that node's configured domain is unknown, then the node cannot be recovered. In these cases, the query ID network management message must be used to get the 48-bit 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 node must know the domain(s) of the other node to use the query ID network management message or it must already know the 48 bit 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 nodes 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 nodes which are members of that domain may be recovered.

Guideline 7-3: All LONMARK compliant nodes must support 2 domains to facilitate integration into subsystems, and subsystems into systems.

Self-Installed Nodes

A self-installing node updates its own network addressing information based on local inputs with no interaction with other nodes 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 node is usually very simple, for example push-buttons, DIP switches, rotary switches, or a backplane slot ID.

Self-installed nodes can communicate across an interoperable network in one of two ways; either via a subsystem gateway (described later) or by being re-installed onto the network using a network tool.

Guideline 7-4: It must be possible to configure a self-installed node with a network management tool and have that node's address be set to any legal LonTalk protocol address.

Each self installing node must contain a configuration network variable of type SNVT_`config_src` contained in the Node Object, Object 0. When the node is manufactured, the value of this variable should be set to `CFG_LOCAL`. When the value of the variable is set to `CFG_EXTERNAL`, it means that a network tool has taken over management of the node, and the node must not set its own network addresses.

Guideline 7-5: Self-installed nodes shall not modify their channel ID field.

The channel ID field is reserved for network management tools to track the physical location of a node. A value of zero indicates that the node's channel ID is unassigned.

Field Installed Nodes

Field installed nodes are installed using a network management tool. The tool can take on two basic forms. In many cases, the tool is invisible to the user. That is, the network management tool is embedded in the network and performs installation and maintenance "behind the scenes". This is known as automatic installation. To the end user, the network appears to install itself, while in reality the tool is analyzing the network contents and automating installation based on a set of rules.

In other cases, the user interacts with the 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.

Sub-Systems and Nodes

The difference between a control system built up from individual nodes, and one built-up from sub-systems is important. An example helps to illustrate the point. Consider a building control application consisting of lighting, heating, and access control functions. The "building control system" could be built out of individual nodes, or it could be a collection of three sub-systems. While in both cases the function of the building system is the same – to provide integrated environment and security control – how it is constructed determines the system's capabilities and how the operator interacts with it.

An example of a building control system implemented as a collection of sub-systems is shown in figure 7.1.

The first advantage of this type of control system is that each sub-system can choose its own installation and maintenance scenario. For example, the access control system might be self-installed while the lighting control system uses a field installation tool. The other advantage is that access to the nodes within the sub-system is controlled. Each sub-system uses one or more subsystem gateway nodes (see the guidelines for more details) to connect to the integrated building system. The only access to the sub-system is through the gateway nodes which "police" the requests of the other systems to protect the integrity and privacy of the nodes within the sub-system.

There are also disadvantages to implementing the building as a collection of sub-systems. For example:

- non-uniformity of user interface for different building functions
- reduced ability to share common resources such as cabling etc.
- need for more system management tools
- more complex network administration - a functional change that affects all three subsystems requires interaction with each subsystem's management tool

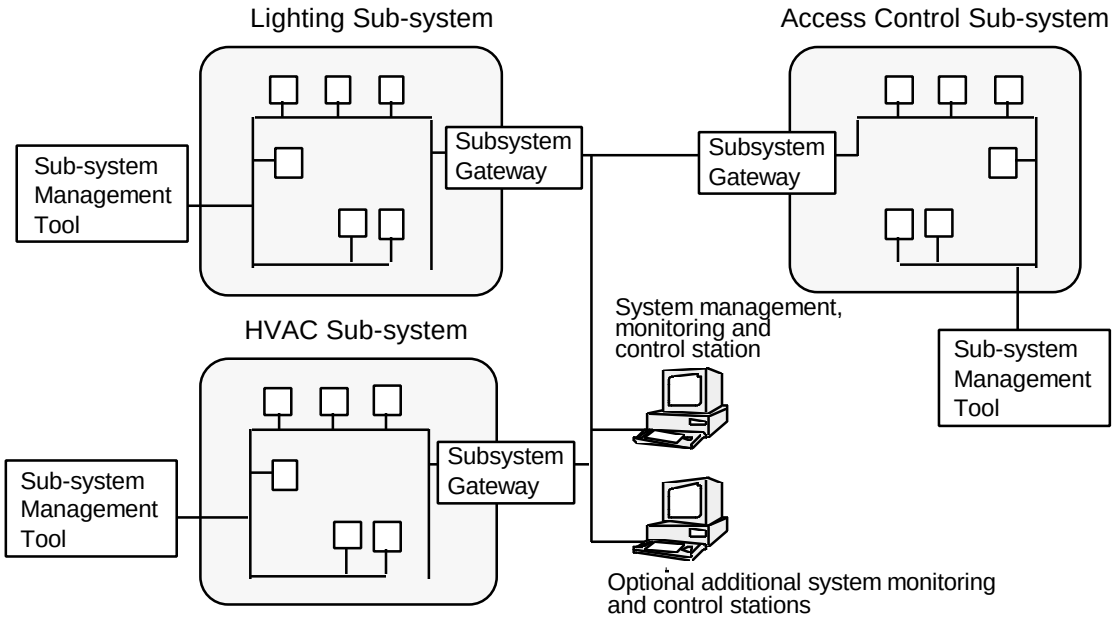

Figure 7.1 An Example Building Control System with Sub-systems

An example of the building implemented from individual nodes is shown in figure 7.2. The lighting, HVAC, and access control components may still have been manufactured and sold by three separate vendors, but in this installation the system can be managed as a single unit from a central point. This design overcomes the disadvantages of the system consisting of multiple sub-systems.

First, since access to the nodes within a functional group is no longer limited by the interface nodes, the building system integrator is free to make any connections– opening the possibility of sharing functions across systems and eliminating duplicate components. For example, the occupancy detectors supplied by the lighting system could also be used by the HVAC system. Likewise, the system monitoring and control stations can display any data from anywhere on the network.

The disadvantage of this type of design is the same as its strength– the sub-systems are no longer autonomous. A system-wide view of network design and management is required to ensure the integrity of all attached subsystems.

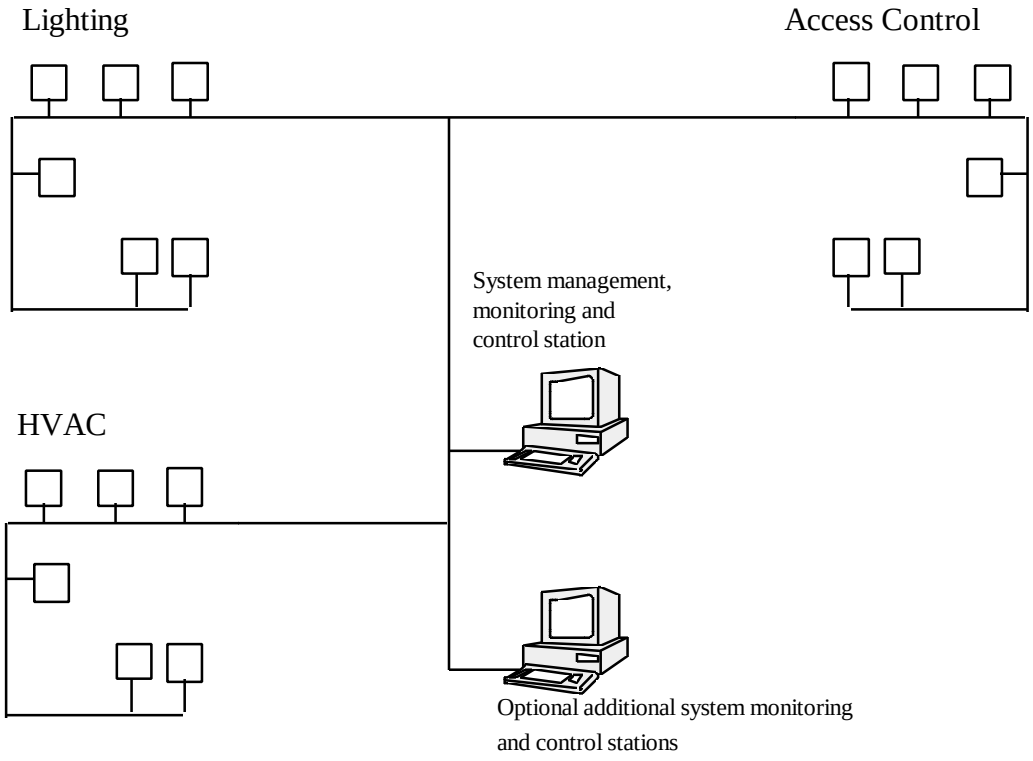

Figure 7.2 An Example Building Control System Comprised of Individual Nodes

The following sections outline the design guidelines for individual nodes and sub-systems.

Nodes

The following sections apply to implementing interoperable nodes. These rules also apply to the gateway nodes of interoperable products which contain multiple nodes, one or more of which do not meet these guidelines. In the second case, the manufacturer's product should be viewed as an interoperable sub-system– the product network is independently managed and it interfaces to other nodes and sub-systems through one or more gateway nodes.

Guideline 7-6: Nodes must be manufactured with a zero channel ID.

The channel ID field is reserved for network tools to track the physical location of a node. A value of zero indicates that the node's channel ID is unassigned. The node's

application should not require specific values in this field since the network tool may change the value as needed.

Guideline 7-7: A node's application program must not be dependent upon its network configuration.

To correctly allocate network resources such as node 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 nodes. A node's functional behavior should be independent of its address or the details of its bindings to other nodes. All messages should be sent using either implicit addressing or using explicit addressing where the explicit addresses are determined at installation time.

Guideline 7-8: All nodes must provide internal or external access to the service pin.

The process of installing nodes requires that a physical node be identified with a logical address. This process is done using the node's Neuron ID, a unique 48-bit number written to the Neuron Chip during manufacture. The most direct way to identify a node is by using its service pin. When a Neuron Chip's service pin is activated, the node sends a message containing its Neuron ID and program ID.

The method used to activate the service pin varies from application to application. Examples of mechanical methods include activating via an accessible push button or a magnetic reed switch located within an enclosure. A service pin message can also be sent under software control. For example, the node can send the message when the node is powered up or when a predefined series of I/O events occur.

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– it is a simple way to ensure that an installer can always identify and thereby establish communication with a given node. If necessary, the service pin can be located inside the device such that it is accessible to service personnel only.

Sub-Systems

Guideline 7-9: A sub-system must include one or more sub-system gateways to allow connection to an interoperable system.

A sub-system is an independently managed collection of nodes. To allow sub-systems to be connected into an interoperable system, the sub-system must provide one or more subsystem gateways. The sub-system gateway acts as an agent for the sub-system and provides the interoperable interface between the interoperable system and the sub-system. Any functions that are provided between the sub-system and the interoperable system must be provided through the sub-system gateway.

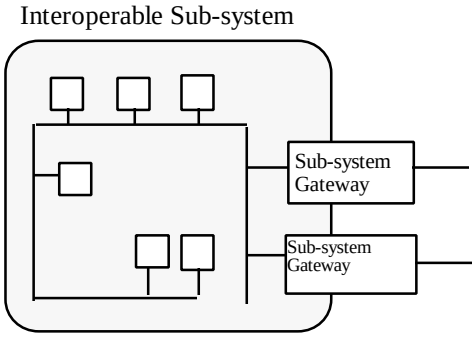

Figure 7.3 Interoperable Sub-system

The sub-system gateway is an application-level gateway containing two nodes: an internal interface node and an external interface node. All communication between the two nodes in the gateway is done at the application layer, using any desired method. See figure 7.4. This allows the subsystem gateway to "police" which information is accessible to the outside and allow or deny access to the nodes within the sub-system to protect the integrity of the sub-system.

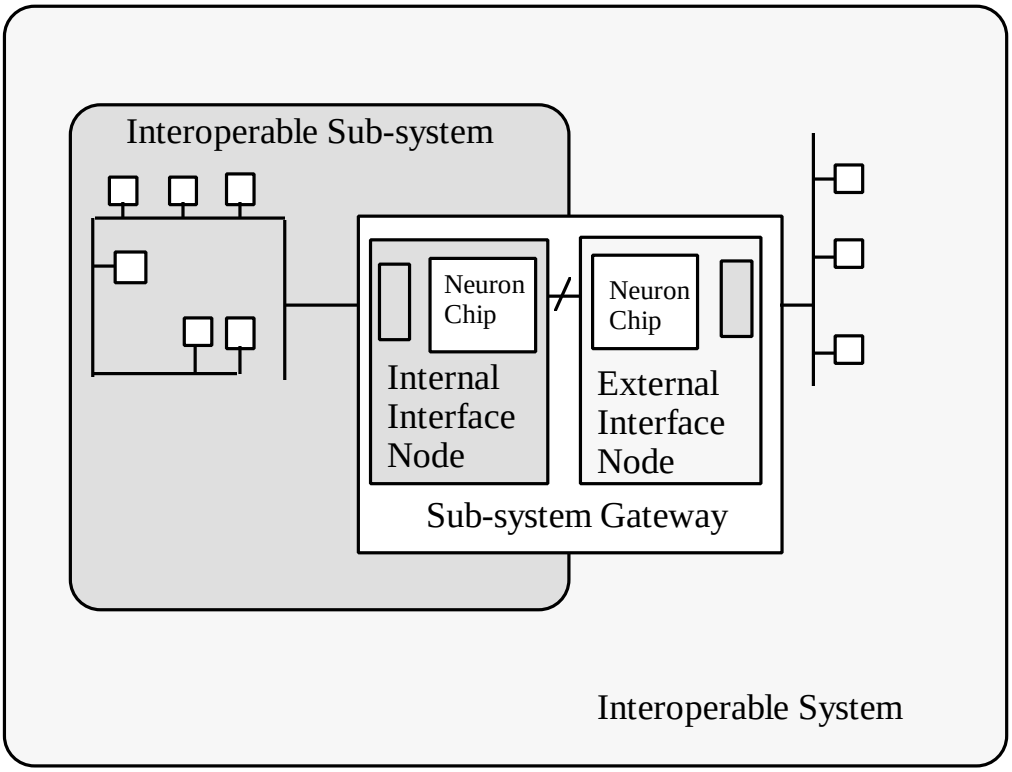

Figure 7.4 Close-up of a Sub-system Gateway

The internal interface node "belongs" to the sub-system and is managed by the sub-system network manager. The external interface node "belongs" to the interoperable system and is managed by the interoperable system network management tool. Note: Either the internal or external interface nodes may optionally contain non-Neuron-based host processors.

Guideline 7-10: A sub-system gateway external interface node must meet all the interoperability guidelines.

The external interface node must comply with all the interoperability guidelines. The internal interface node and other nodes within the sub-system do not have to comply with any interoperability guidelines.

Gateways to Command Based Systems

Guideline 7-11: Under no conditions shall a LONMARK gateway pass commands from a command based system directly into a LONWORKS network. Instead, these commands shall be mapped to LONMARK objects.

In a command based system composed of multiple nodes, commands are sent between the nodes to initiate system actions. This implies that the nodes 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 the LONMARK system because the LONMARK nodes were not programmed to use these commands. In fact, to get interactions between the nodes on both sides of the gateway, the LONMARK nodes would have had to be designed to send and receive the other system's commands. Since LONMARK nodes communicate via LONMARK objects, this method of gateway construction severely limits interoperability.

A better method for constructing a gateway to a command based system would be to think of the entire command based system as a single LONMARK node with a set of standard objects which 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 node and the LONMARK nodes. Within the gateway, translations between the commands and the LONMARK objects 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 node with objects defined which are compatible with those defined on the gateway can interact with the command based system without the foresight of the node developer.

Shared Media Considerations

A power line channel, that contains nodes within earshot of several network tools, is an example of a shared medium. When two or more network tools share such a medium, messages can leak between one tool and nodes belonging to another tool. If the tools and installers do not directly coordinate their activities, the tools and nodes must follow conventions to avoid conflicting network changes or installing the wrong nodes. The following guidelines apply to nodes on shared media. The term *shared media* refers not only to media sharing but also uncoordinated network management

activities as described above. It also refers to open shared media like powerline and closed shared media like twisted pair.

Reclaiming Nodes

If a foreign network tool inadvertently acquires a node and installs it with network management authentication, the node's owner is unable to reclaim the node over the network. Invoking the `go_unconfigured()` function resets the node's authentication key allowing the node's owner to use the service pin to reclaim the node.

Guideline 7-12: Nodes intended for installation on shared media must provide some means for locally causing the node to go unconfigured, for example by invoking the Neuron C `go_unconfigured()` function.

Since the service pin message can be received by foreign network management tools the wink function must be supported on the node to confirm the intended installer. Node winking, whether due to the installation protocol itself or post installation testing, may cause activity in a foreign node. 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, e.g. an LED is flashed locally on the node.

Guideline 7-13: Nodes intended for installation on shared media must support the wink function 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 node.

Appendix A

New SNVT and SCPT Definitions

If a SNVT or SCPT is not available to satisfy your product requirements, this appendix outlines the procedure for adopting new SNVTs and SCPTs.

Creating New SNVTs

In general, the following considerations will be made when reviewing the adoption of new SNVTs and SCPTs:

- Is this request applicable to more than one manufacturer's product?
- Are there any existing SNVTs or SCPTs that can be used to fulfill this request?
- Is this a recognized standard data convention?

Systeme Internationale (SI) units will be used, except when the generally accepted industry convention worldwide is not in SI units.

Submission of SNVTs and SCPTs for Review

All SNVT and SCPT requests must be sent to SNVTrequest@lonmark.org for review. There is a template document available from the Tech Resources section of the Members Area at www.lonmark.org that should be used for documenting proposed SNVT and SCPTs. LMSNVT.ZIP is the name of the template document archive containing LMSNVT.DOC.

SNVTs or SCPTs required as part of a Functional Profile proposal should be submitted to SNVTrequest@lonmark.org and also included in the Profile review to ensure that there are no unexpected delays or corrections. Please do not wait until a Profile has been reviewed before sending in a SNVT or SCPT request since this will delay final approval of the Profile.

Numeric-type SNVTs and SCPTs

Additional numeric-type SNVTs and SCPTs will be considered when:

- An existing SNVT or SCPT does not meet requirements in terms of resolution or maximum range.
- Use of floating point SNVTs or SCPTs would not meet performance requirements.
- A physical quantity not already represented is needed to interoperate with other products.

All new numeric-type SNVTs and SCPTs are realized in SI units. A new SNVT or SCPT in non-SI units will not be approved when an existing SNVT or SCPT in SI units already covers this same variable.

Enumeration-type SNVTs and SCPTs

Additional enumeration-type SNVTs and SCPTs will be added 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. Enumeration values can be added to an existing SNVT or SCPT following the same approval procedure for new SNVTs and SCPTs, with the following exception:

Enumerations will not be added to a SNVT or SCPT when the SNVT or SCPT is self-defining based on the values in the enumeration. In this latter case adding enumeration values would change the definition of the SNVT or SCPT, which would render the variable non-interoperable with its previous definition.

Structured SNVTs and SCPTs

Additional structured SNVTs and SCPTs, comprising `struct` and/or `union` combinations, will be added when multiple quantities are to be communicated simultaneously in a single update (due to timestamping needs or something similar), and several similar products are expected to operate in the same way. This should not be used to gather information into a single variable for the purpose of reducing the number of network variables required on a node.

If you have any questions about this process please contact:

LONMARK Principal Engineer

e-mail address - tech@lonmark.org

Phone +1 408 938 5266

Fax +1 408 790 3838

Appendix B

Functional Profiles

Functional profiles describe application-specific LONMARK objects. The procedures used by the LONMARK Interoperability Association for developing and adopting new LONMARK objects are described in this Appendix.

LONMARK Functional Profile Review & Approval Process

LONMARK functional profiles can be drafted by any LONMARK member and submitted for member review, comment, and subsequent approval by the LONMARK Board of Directors.

Proposals should be sent via email to the appropriate LONMARK task group leader or to the LONMARK Principal Engineer if there is doubt on which group or forum is appropriate.

There is a functional profile template document available for download from the Tech Resources section of the Members Area at www.lonmark.org that should be used for documenting proposed LONMARK object definitions. HNDBK_20.ZIP is the name of the Association Handbook archive containing the functional profile template.

Once a working draft document is posted to the Member Forum on www.lonmark.org, it is available for comment. The review period for draft profiles is thirty (30) days. All members can comment on proposed drafts.

Based on the scope of the comments received, a revised proposal will either be submitted for another review cycle, or may be voted upon by the task group at a scheduled task group meeting or by email ballot. This determination is made jointly by the document author and the task group leader. Once approved by the task group, the task group leader will request to the Principal Engineer that the proposal be passed directly to the Board of Directors for a two-week review and final approval.

If you have any questions about this process please contact:

LONMARK Principal Engineer

email address - tech@lonmark.org

Phone +1 408 938 5266

Fax +1 408 790 3838