ECHELON®

Neuron C Programmer's Guide

Describes how to write programs using the Neuron C programming language.

Welcome

This guide describes how to write programs using the Neuron® C Version 2.3 language. Neuron C is a programming language based on ANSI C that is designed for Neuron Chips and Smart Transceivers. It includes network communication, I/O, and event-handling extensions to ANSI C, which make it a powerful tool for the development of LONWORKS® applications. Key concepts in programming with Neuron C are explained through the use of specific code examples and diagrams. A general methodology for designing and implementing a LONWORKS application is also presented.

A subset of the Neuron C language is also used to describe the interoperable interface of host-based applications that are designed with the ShortStack® Developer's Kit, FTXL™ Developer's Kit, or the *i*.LON® SmartServer. This interoperable interface is contained within a file called a *model file*, which contains Neuron C declarations and definitions for the device interface. In addition to describing the Neuron C Version 2.3 language, this guide also provides a brief introduction to model file compilation, and points out syntactical differences between compilation for Neuron-hosted devices and compilation for host-based devices that use a model file.

The Neuron C Programmer's Guide:

- Outlines a recommended general approach to developing a LONWORKS application, and

- Explains key concepts of programming in Neuron C through the use of code fragments and examples.

Audience

The *Neuron C Programmer's Guide* is intended for application programmers who are developing LONWORKS applications. Readers of this guide are assumed to be familiar with the ANSI C programming language, and have some C programming experience.

For a complete description of ANSI C, consult the following references:

- —. 1989. *American National Standard for Information Systems Programming Language C*. Standard number X3.159-1989. New York, NY: American National Standards Institute.

- —. 2007. *International Standard ISO/IEC 9899:1999. Programming languages – C*. Geneva, Switzerland: International Organization for Standardization.

- Harbison, Samuel P. and Guy L. Steele, Jr. 2002. *C: A Reference Manual*, 5th edition. Upper Saddle River, NJ: Prentice Hall, Inc.

- Kernighan, Brian W. and Dennis M. Ritchie. 1988. *The C Programming Language*, 2nd edition. Upper Saddle River, NJ: Prentice Hall, Inc.

- Plauger, P.J. and Jim Brodie. 1989. *Standard C: Programmer's Quick Reference Series*. Buffalo, NY: Microsoft Press.

- Plauger, P.J. and Jim Brodie. 1992. *ANSI and ISO Standard C Programmer's Reference*. Buffalo, NY: Microsoft Press.

Related Documentation

The following manuals are available from the Echelon Web site (www.echelon.com) and provide additional information that can help you develop Neuron C applications for LONWORKS devices:

- *Introduction to the LONWORKS Platform* (078-0391-01A). This manual provides an introduction to the ISO/IEC 14908 (ANSI/CEA-709.1 and EN14908) Control Network Protocol, and provides a high-level introduction to LONWORKS networks and the tools and components that are used for developing, installing, operating, and maintaining them.

- *I/O Model Reference for Smart Transceivers and Neuron Chips* (078-0392-01C). This manual describes the I/O models that are available for Echelon's Smart Transceivers and Neuron Chips.

- *IzoT Commissioning Tool User's Guide* (**078-0514-01**). This manual describes how to use the IzoT Commissioning Tool to design, commission, monitor and control, maintain, and manage a LONWORKS network.

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

- *Neuron C Reference Guide* (078-0140-01G). This manual provides reference information for writing programs using the Neuron C Version 2.3 programming language.

- *Neuron Tools Errors Guide* (078-0402-01D). This manual documents and explains the various warning and error messages that can occur for the various Neuron C development tools.

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

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

Typographic Conventions for Syntax

Table 1 on page v lists the typographic conventions used in this manual for displaying Neuron C syntax.

<div align="center">**Table 1**. Typographic Conventions</div>

Typeface or Symbol	Used for	Example
boldface type	keywords literal characters	**network** **{**
italic type	abstract elements	*identifier*
[square brackets]	optional fields	[*bind-info*]
| vertical bar	a choice between two elements	**input | output**

Example: The syntax for declaring a network variable is:

network input | output [*netvar modifier*] [*class*] *type* [*bind-info*] *identifier*

- You type the keywords **network**, **input**, and **output** as shown

- You replace the abstract elements *netvar modifier*, *class*, *type*, *bind-info*, and *identifier* with the actual modifier, class, type, bind information, and identifier for the network variable

- The declaration must include either **input** or **output**, but not both

- The elements *netvar modifier*, *class*, and *bind-info* are all optional

When a particular element or expression includes punctuation, such as quotation marks, parentheses, and semicolons (but not including square brackets and vertical bars), you must type that punctuation as shown.

Code examples appear in the monospace Courier font:

```
#include <mem.h>

unsigned array1[40], array2[40];

// See if array1 matches array2
if (memcmp(array1, array2, 40) != 0) {
    // The contents of the two areas do not match
}
```

Table of Contents

1

Overview

This chapter introduces the Neuron C Version 2.3
programming language. It describes the basic aspects of the
language and provides an overview to using the LONWORKS
platform and the Neuron C programming language to
construct interoperable devices and systems. The chapter
also introduces key concepts of Neuron C such as event-
driven scheduling, network variables, configuration
properties, and functional blocks (which are
implementations of functional profiles).

A secondary purpose of this chapter is to introduce
fundamental material on Neuron C concerning Neuron C
types, storage classes, data objects, and how the Neuron C
language compares to the ANSI C language.

What Is Neuron C?

Neuron C Version 2 is a programming language based on ANSI C that is designed for Neuron Chips and Smart Transceivers. It includes network communication, I/O, and event-handling extensions to ANSI C, which make it a powerful tool for the development of LONWORKS applications. Following are a few of these new[1] features:

- A new network communication model, based on *functional blocks* and *network variables*, that simplifies and promotes data sharing between like or disparate devices.

- A new network configuration model, based on functional blocks and *configuration properties*, that facilitates interoperable network configuration tools.

- A new type model based on standard and user *resource files* that expands the market for interoperable devices by simplifying integration of devices from multiple manufacturers.

- An extensive built-in set of *I/O models* that support the powerful I/O capabilities of Neuron Chips and Smart Transceivers.

- Powerful *event-driven programming* extensions, based on new **when** statements, provide easy handling of network, I/O, and timer events.

- A high-level programming model that supports application-specific interrupt handlers and synchronization tools.

Neuron C provides a rich set of language extensions to ANSI C tailored to the unique requirements of distributed control applications. Experienced C programmers will find Neuron C a natural extension to the familiar ANSI C paradigm. Neuron C offers built-in type checking and allows the programmer to generate highly efficient code for distributed LONWORKS applications.

Neuron C omits ANSI C features that are not required by the standard for free-standing implementations. For example, certain standard C libraries are not part of Neuron C. Other differences between Neuron C and ANSI C are described in *Differences between Neuron C and ANSI C* on page 13.

Comparing Neuron C Version 2 to Version 1

Neuron C version 2 includes major enhancements to the language that greatly simplify the development of devices that use functional blocks and configuration properties. Prior to version 2, developers had to manually construct self-documentation strings for each device, network variable, and configuration property. This process could be both time-consuming and error-prone. With Neuron C version 2, the Neuron C compiler can automatically create and manage these strings.

If any of the following Neuron C keywords appear in an application, the Neuron C compiler automatically generates and manages the self-documentation strings:

- **fblock**

---

[1] "New" means relative to the ANSI Standard C language.

- **fb_properties**

- **nv_properties**

- **device_properties**

- **cp**

- **cp_family**

You can still manually create the self-documentation strings, if necessary, by avoiding the use of any of these keywords and by declaring the self-documentation strings using the Neuron C version 1 syntax. Using this syntax could be useful for migrating older applications (created with the NodeBuilder 1.5 or LonBuilder tools) to the IzoT NodeBuilder Software. Applications that do not use these keywords still get the benefit of access to resource definitions contained within resource files.

Unique Aspects of Neuron C

Neuron C implements all of the basic ANSI C types, and type conversions as necessary. In addition to the ANSI C data constructs, Neuron C provides some unique data elements. *Network variables* are fundamental to Neuron C and LONWORKS applications. Network variables are data constructs that have language and system firmware support to provide something that looks like a variable in a C program, but has additional properties for propagating across a LONWORKS network to or from one or more other devices on that network. The network variables make up part of the *device interface* for a LONWORKS device.

Configuration properties are Neuron C data constructs that are another part of the device interface. Configuration properties allow the device's behavior to be customized using a network tool such as the LonMaker tool or a customized plug-in created for the device.

Neuron C also provides a way to organize the network variables and configuration properties in the device into *functional blocks*, each of which provides a collection of network variables and configuration properties, that are used together to perform one task. These network variables and configuration properties are called the *functional block members*.

Each network variable, configuration property, and functional block is defined by a type definition contained in a *resource file*. Network variables and configuration properties are defined by *network variable types* (NVTs) and *configuration property types* (CPTs). Functional blocks are defined by *functional profiles* (which are also called *functional profile templates*).

Network variables, configuration properties, and functional blocks in Neuron C can use *standardized, interoperable types*. The use of standardized data types promotes the interconnection of disparate devices on a LONWORKS network. For configuration properties, the standard types are called standard configuration property types (SCPTs; pronounced *skip-its*). For network variables, the standard types are called standard network variable types (SNVTs; pronounced *snivets*). For functional blocks, the standard types are called standard functional profiles. If you cannot find standard types or profiles that meet your requirements, Neuron C also provides full support for user network variable types (UNVTs; pronounced *u-nivets*), user configuration property types (UCPTs; pronounced *u-keep-its*), and user functional profiles.

Neuron C is designed to execute in the environment provided by the Neuron system firmware. This firmware provides an *event-driven scheduling system* as part of the Neuron C language's run-time environment.

Neuron C also provides a lower-level *messaging service* integrated into the language in addition to the network variable model, but the network variable model has the advantage of being a standardized method of information interchange, whereas the messaging service is not standardized (with the exception of its usage by the LONWORKS file transfer protocol, LW-FTP). The use of network variables, both standard types and user types, promotes interoperability between multiple devices from multiple vendors. The lower-level messaging service allows for proprietary solutions in addition to the file transfer protocol.

Another Neuron C data object is the *timer*. Timers can be declared and manipulated like variables. When a timer expires, the system firmware automatically manages the timer events and notifies the program of those events.

Neuron C provides many built-in *I/O models*, which are instantiated as *I/O objects*. These I/O models are standardized I/O "device drivers" for the Neuron Chip or Smart Transceiver I/O hardware. Each I/O model fits into the event-driven programming model. A function-call interface is provided to interact with each I/O object.

Some I/O models, all I/O pins, and a dedicated, high-resolution system timer, can also be used to trigger asynchronous interrupts.

The rest of this chapter discusses these various aspects of Neuron C in more detail, and the remaining chapters cover these aspects in greater detail, accompanied by many examples.

Neuron C Integer Constants

Negative constants are treated as a unary minus operation on a positive constant, for example, -128 is a **signed long**, not a **signed short**. Likewise, -32768 is an **unsigned long**, not a **signed long.** To construct a **signed short** value of -128, you must use a cast:

> **((signed short)(-128))**

To construct a **signed long** value of –32768, you must also use a cast:

> **((signed long)(-32768))**

Decimal integer constants have the following default types:

```
0 .. 127                    signed short
128 .. 32767                signed long
32768 .. 65535             unsigned long
```

The default type can be modified with the **u**, **U**, **l**, and **L** suffixes. For example:

```
0L                          signed long
127U                        unsigned short
127UL                       unsigned long
256U                        unsigned long
```

Hexadecimal constants have the following default types, which can also be modified as described above with the **u**, **U**, **l**, and **L** suffixes:

```
0x0 .. 0x7F              signed short
0x80 .. 0xFF             unsigned short
0x100 .. 0x7FFF          signed long
0x8000 .. 0xFFFF         unsigned long
```

Octal constants have the following default types, which can also be modified as described above with the **u**, **U**, **l**, and **L** suffixes:

```
0 .. 0177                signed short
0200 .. 0377             unsigned short
0400 .. 077777           signed long
0100000 .. 0177777       unsigned long
```

Binary constants have the following default types, which can also be modified as described above with the **u**, **U**, **l**, and **L** suffixes:

```
0b0 .. 0b01111111                            signed short
0b10000000 .. 0b11111111                     unsigned short
0b0000000100000000 .. 0b0111111111111111     signed long
0b1000000000000000 .. 0b1111111111111111     unsigned long
```

Neuron C Variables

The following sections briefly discuss various aspects of variable declarations. Data types affect what sort of data the variable represents. Storage classes affect where the variable is stored, whether it can be modified (and if so, how often), and whether there are any device interface aspects to modifying the data.

Neuron C Variable Types

Neuron C supports the following C variable types. The keywords shown in square brackets are optional; if omitted, they are assumed by the Neuron C language, per the rules of the ANSI C standard.

[**signed**] **long** [**int**]	16-bit quantity
unsigned long [**int**]	16-bit quantity
signed char	8-bit quantity
[**unsigned**] **char**	8-bit quantity
[**signed**] [**short**] [**int**]	8-bit quantity
unsigned [**short**] [**int**]	8-bit quantity
enum	8-bit quantity (**int** type)

Neuron C provides some predefined enum types. One example is shown below:

> **typedef enum {FALSE, TRUE} boolean;**

Neuron C also provides predefined objects that, in many ways, provide the look and feel of an ANSI C language variable. These objects include Neuron C timer and I/O objects. See the *I/O Model Reference* for more details on I/O objects, and see the *Timers* chapter in the *Neuron C Reference Guide* for more details on timer objects.

The extended arithmetic library also defines **float_type** and **s32_type** for IEEE 754 and signed 32-bit integer data respectively. These types are discussed in great detail in the *Functions* chapter of the *Neuron C Reference Guide*.

Neuron C Storage Classes

If no class is specified and the declaration is at file scope, the data or function is global. *File scope* is that part of a Neuron C program that is not contained within a function or a task. Global data (including all data declared with the **static** keyword) is present throughout the entire execution of the program, starting from the point where the symbol was declared. Declarations using **extern** references can be used to provide forward references to variables, and function prototypes must be declared to provide forward references to functions.

Upon power-up or reset of a Neuron Chip or Smart Transceiver, the global data in RAM is initialized to its initial-value expression, if present, otherwise to zero (variables declared with the **eeprom** or **config** class, as well as configuration properties declared with the **config_prop** or **cp_family** keywords, are only initialized when the application image is first loaded).

Neuron C supports the following ANSI C storage classes and type qualifiers:

auto Declares a variable of local scope. Typically, this would be within a function body. This is the default storage class within a local scope and the keyword is normally not specified. Variables of auto scope that are not also static are not initialized upon entry to the local scope. The value of the variable is not preserved once program execution leaves the scope.

const Declares a value that cannot be modified by the application program. Affects self-documentation (SD) data generated by the Neuron C compiler when used in conjunction with the declaration of CP families or configuration network variables.

extern Declares a data item or function that is defined in another module, in a library, or in the system image.

static Declares a data item or function which is *not* to be made available to other modules at link time. Furthermore, if the data item is local to a function or to a **when** task, the data value is to be preserved between invocations, and is not made available to other functions at compile time.

In addition to the ANSI C storage classes, Neuron C provides the following classes and class modifiers:

config Can be combined only with an input network variable declaration. A **config** network variable is used for application configuration. It is equivalent to **const eeprom**. Such a network variable is initialized only when the application image is first loaded. The **config** class is obsolete and is included only for legacy applications. The Neuron C compiler does not generate self-documentation data for **config**-class network variables. New applications should use the configuration network variable syntax described Chapter 4, *Using Configuration Properties to Configure Device Behavior*, on page 89.

network Begins a network variable declaration. See Chapter 3, *How Devices Communicate Using Network Variables*, on page 45, for more details.

system Used in Neuron C solely to access the Neuron firmware function library. Do not use this keyword for data or function declarations.

uninit   When combined with the **eeprom** keyword (see below), specifies that the EEPROM variable is not initialized or altered on program load or reload over the network.

The following Neuron C keywords allow you to direct portions of application code and data to specific memory sections:

- **eeprom**

- **far**

- **offchip**   (only for Neuron Chips and Smart Transceivers with external memory)

- **onchip**

- **ram**   (only for Neuron Chips and Smart Transceivers with external memory)

These keywords are particularly useful on the Neuron 3150 Chip and 3150 Smart Transceivers or members of the Series 5000 or 6000 chip families, because a majority of the address space for these parts is mapped off chip. See *Using Neuron Chip Memory* on page 188 for a more detailed description of memory usage and the use of these keywords.

Variable Initialization

Initialization of variables occurs at different times for different classes. The **const** variables, except for network variables, *must* be initialized. Initialization of **const** variables occurs when the application image is first loaded into the Neuron Chip or Smart Transceiver, or when execution enters the local scope in which a constant *auto* variable is declared. The **const ram** variables are placed in off-chip RAM that must be non-volatile. Therefore, the **eeprom** and **config** variables are also initialized at load time, except when the **uninit** class modifier is included in these variable definitions.

Global RAM variables are initialized at reset (that is, when the device is reset or powered up). By default, all global RAM variables (including **static** variables) are initialized to zero at this time. Initialization to zero costs no extra code space, as it is a firmware feature.

Initialization of I/O objects, input network variables (except for **eeprom**, **config**, **config_prop**, or **const** network variables), and timers also occurs at reset. Zero is the default initial value for network variables and timers.

Local variables (except **static** ones) are not automatically initialized, nor are their values preserved when the program execution leaves the local scope.

Neuron C Declarations

Both ANSI C and Neuron C support the declarations listed in **Table 2**.

Table 2. ANSI C and Neuron C Declarations

Declaration	Example
Simple data items	**int a, b, c;**
Data types	**typedef unsigned long ULONG;**
Enumerations	**enum hue {RED, GREEN, BLUE};**
Pointers	**char *p;**
Functions	**int f(int a, int b);**
Arrays	**int a[4];**

Structures and unions	**struct s {**
	  int field1;
	  unsigned field2 : 3;
	  unsigned field3 : 4;
	};

In addition, Neuron C Version 2 supports the declarations listed in **Table 3**.

Table 3. Neuron C Version 2 Declarations

Declaration	Example
I/O Objects	**IO_0 output oneshot relay_trigger;** (See Chapter 2)
Timers	**mtimer led_on_timer;** (See Chapter 2)
Network Variables	**network input SNVT_temp temperature;** (See Chapter 3)
Configuration Properties	**cp_family SCPTdefOutput defaultOut;** (See Chapter 4)
Functional Blocks	**fblock SFPTnodeObject { ... } myNode;** (See Chapter 5)
Message Tags	**msg_tag command;** (See Chapter 6)

Network Variables, SNVTs, and UNVTs

A *network variable* is an object on one device that can be *connected* to network variables on one or more additional devices. A device's network variables define

its inputs and outputs from a network point of view and allow the sharing of data in a distributed application. Whenever a program writes into one of its *output* network variables (with the exception of output network variables that are declared with the **polled** modifier), the new value of the network variable is *propagated* across the network to all devices with *input* network variables connected to that output network variable. If the output network variable is not currently a member of any network variable connection, no transaction and no error occurs.

Although the propagation of network variables occurs through LONWORKS messages, these messages are sent implicitly. The application program does not require any explicit instructions for sending, receiving, managing, retrying, authenticating, or acknowledging network variable updates. A Neuron C application provides the most recent value by writing into an output network variable, and it obtains the most recent data from the network by reading an input network variable.

Example:

```
network input SNVT_temp nviTemperature;
network output SNVT_temp nvoTemperature;

void f(void)
{
    nvoTemperature = 2 * nviTemperature;
}
```

Network variables greatly simplify the process of developing and installing distributed systems because devices can be defined individually, then connected and reconnected easily into many new LONWORKS applications. Network variables are discussed in detail in Chapter 3, *How Devices Communicate Using Network Variables*, on page 45, and also in the *Neuron C Reference Guide*.

Network variables promote interoperability between devices by providing a well-defined interface that devices use to communicate. Interoperability simplifies installation of devices into different types of networks by keeping the network configuration independent of the device's application. A device can be installed in a network and logically connected to other devices in the network as long as the data types (for example, **SNVT_switch** or **SNVT_temp_p**) match. To further promote interoperability, the LONWORKS platform provides standard functional profiles that define standard functional interfaces for devices, and standard network variable types (SNVTs) that define standard data encoding, scaling, and units, such as degrees C, volts, or meters. There are standard functional profiles for a variety of functions and industries. There are SNVT definitions for essentially every physical quantity, and other more abstract definitions tailored for certain industries and common applications.

You can also create your own user functional profiles and user network variable types (UNVTs). You can define resource files for your custom types and profiles to enable your devices to be used with devices from other manufacturers. The IzoT Resource Editor included with the NodeBuilder tool provides a simple interface for viewing existing resources and defining your own resources.

Configuration Properties

A configuration property is a data item that, like a network variable, is part of the device interface for a device. A configuration property can be modified by a network tool. Configuration properties facilitate interoperable installation and configuration tools by providing a standardized network interface for device configuration data. Like network variables, configuration properties also provide a well-defined interface. Each configuration property type is defined in a resource file that specifies the data encoding, scaling, units, default value, invalid value, range, and behavior for configuration properties based on the type. A rich variety of standard configuration property types (SCPTs) are available. You can also create your own user configuration property types (UCPTs) that are defined in resource files that you create with the IzoT Resource Editor.

Functional Blocks and Functional Profiles

The *device interface* for a LONWORKS device consists of its functional blocks, network variables, and configuration properties. A *functional block* is a collection of network variables and configuration properties that are used together to perform one task. These network variables and configuration properties are called the *functional block members*.

Functional blocks are defined by *functional profiles*. A functional profile is used to describe common units of functional behavior. Each functional profile defines mandatory and optional network variables and mandatory and optional configuration properties. Each functional block implements an instance of a functional profile. A functional block must implement all of the mandatory network variables and configuration properties that are defined by the functional profile, and can also implement any of the optional network variables and configuration properties that are defined by the functional profile. In addition, a functional block can implement network variables and configuration properties that are not defined by the functional profile – these are called *implementation-specific* network variables and configuration properties.

Functional profiles are defined in *resource files*. You can use standard functional profiles or you can define your own functional profiles in your own resource files by using the IzoT Resource Editor. A functional profile defined in a resource file is also called a *functional profile template* (FPT).

LONMARK International provides a procedure for developers to certify devices. LONMARK interoperable devices conform to all LonTalk® protocol layer 1 – 6 requirements as specified by the *LonMark Layer 1 – 6 Interoperability Guidelines*, and conform to all aspects of application design, as described in the *LonMark Application Layer Interoperability Guidelines*.

Contact LONMARK International at www.lonmark.org for more details about becoming a member and certifying your devices.

You can automatically embed data within your device that identifies its device interface to network tools that are used to install the device. This data is called *self-identification* (SI) data and *self-documentation* (SD) data. The Neuron C compiler generates this data based on the functional blocks, network variables, and configuration properties that you declare, as well as the resource files that you provide. You can add your own documentation to the SD data to further document your device and its interface.

You can include network variable names in the SD data using the **#pragma enable_sd_nv_names** directive. You can also include a rate estimate in tenths-of-messages/second and a maximum rate estimate in tenths-of-messages/second in the SD data for each network variable. The rate estimate and maximum rate estimate values are provided through the **bind_info** feature. (See the discussion of this feature in Chapter 3, *How Devices Communicate Using Network Variables*, on page 45, and also in the *Neuron C Reference Guide*.)

An application image for a device created by the Neuron C compiler contains SD information unless the **#pragma disable_snvt_si** directive is used. (See the *Compiler Directives* chapter of the *Neuron C Reference Guide* for more information.)

Data-Driven Compared with Command-Driven Protocols

Network variables are used to communicate data and state information between devices. This data-driven model provides a different communication model than in command-based systems. In command-based messaging systems, designers are faced with having a large number of commands, specific to each application, that must be managed, updated, and maintained. Each device has to have knowledge of every command. This leads to ever-growing command tables and application code.

With network variables, the command or action portion of a message is not in the message. Instead, with network variables, this information is in the application program, and each application program only needs have the knowledge required to perform its function. A network integrator can add new types of devices at any time, and connect them to existing devices in the network to perform new applications not envisioned by the original designers of the devices.

Event-Driven Scheduling or Polled Scheduling

Although the Neuron C language is principally designed to make event-driven scheduling natural and easy, Neuron C also allows you to construct polled applications that implement a centralized control application. Chapter 3, *How Devices Communicate Using Network Variables*, on page 45, provides further information on polling.

Low-Level Messaging

In addition to the functional block and network variable communication model, Neuron C also supports application messages. You can use application messages – in place of or in conjunction with the network variables approach – to implement proprietary interfaces to your devices. They are also used for the LONWORKS file transfer protocol. Application messages are described in Chapter 6, *How Devices Communicate Using Application Messages*, on page 123.

I/O Devices

A Neuron Chip or Smart Transceiver can be connected to one or more physical I/O devices. Examples of simple I/O devices include temperature and position

sensors, valves, switches, and LED displays. Neuron Chips and Smart Transceivers can also be connected to other microprocessors. The Neuron firmware implements numerous *I/O models* that manage the interface to these devices for a Neuron C application. I/O models are discussed in detail in Chapter 2, *Focusing on a Single Device*, on page 17, and in the *I/O Model Reference*.

Neuron-Hosted and Host-Based Compilation

Compilation for Neuron-hosted devices, that is, devices based on a Neuron Chip or Smart Transceiver as the main processor, use Neuron C to define all aspects of the device's application:

- The application's interoperable interface, including its set of network variables, configuration properties, and functional blocks, and the self-identification and self-documentation data

- Device configuration information, such as buffer configuration

- Application code including, when tasks, interrupt tasks, I/O device declarations, and functions

Compilation for host-based devices, that is, devices that implement the application on a different processor (the *host processor*), often use a subset of the Neuron C language to model the device's interoperable interface. For these devices, the Neuron C source code acts as a model for the finished application, and the primary source file is called the *model file*. Compilation for host-based devices uses Neuron C to define only the following aspect of the device's application:

- The application's interoperable interface, including its set of network variables, configuration properties, and functional blocks, and the self-identification and self-documentation data

The ShortStack Developer's Kit, FTXL Developer's Kit, and the *i*.LON SmartServer all use model files. See the related product documentation for information about compiling a model file, and about implementing the device's application (including the device configuration information and application code).

Syntactically, a Neuron C application and a model file differ in the following ways:

- Declaration of the interoperable interface is largely identical, however, a few exceptions exist which are identified throughout this book, as necessary.

- Device configuration is typically adjusted using a product-specific tool, such as the LonTalk Interface Developer utility, and is not specified in a model file. When encountered within a model file, device configuration constructs can trigger a compiler error or warning.

- Application code, including I/O device declarations, application timer declarations and any form of executable code, is ignored in a model file. Such code results in a warning for model file compilation, and the code has no effect.

You can use conditional compilation to prepare the same source code for use as Neuron C application and a model file. The **_MODEL_FILE** preprocessor symbol is automatically predefined for model file compilation, and is not

predefined for application compilation. See the *Neuron C Reference Guide* for more information about predefined preprocessor symbols.

Differences between Neuron C and ANSI C

Neuron C adheres closely to the ANSI C language standard; however, Neuron C is not a "conforming implementation" of Standard C, as defined by the American National Standards Institute committee X3-J11.

The following list outlines the major differences between Neuron C and ANSI C:

- Neuron C does not support floating-point computation with C syntax or operators. However, a floating-point library is provided to allow use of floating-point data that conforms to the IEEE 754 standard.

- ANSI C defines a **short int** as 16 bits or more and a **long int** as 32 bits or more. Neuron C defines a **short int** as 8 bits and a **long int** as 16 bits. In Neuron C, **int** defaults to a **short int**. A 32-bit signed integer library is available to allow use of 32-bit quantities.

- Neuron C does not support the **register** or **volatile** classes. These storage classes can be specified but are ignored.

- Neuron C does not implement initializers in declarations of automatic variables which implement aggregate types such as structures, unions or arrays. However, Neuron C 2.3 supports explicit initialization of those automatic variables which implement scalar types.

  Example:

  ```
  void example(void) {
      int a = -18;        // expl. init. for scalar auto OK
      struct { int a, b; } x = { -5, +9 };     // ERROR!
      …
  }
  ```

- Neuron C does not support structures or unions as procedure parameters or as function return values.

- Neuron C does not support declaration of bitfields as members of unions. However, an equivalent declaration can be accomplished by defining a structure as a member of the union, where the structure contains the bitfields.

- Network variable structures cannot contain pointers. Configuration property structures also cannot contain pointers.

- Pointers to timers, to message tags, or to I/O objects are not supported.

- Pointers to network variables, configuration properties, and EEPROM variables are treated as pointers to constants (that is, the contents of the variable referenced by the pointer can be read, but not modified). Under special circumstances, and with certain restrictions, the pointers can be used to modify the memory. See the discussion of the **eeprom_memcpy( )** function in Chapter 8, *Memory Management*, on page 185, and also in the *Functions* chapter of the *Neuron C Reference Guide*. Also refer to the discussion of the **#pragma relaxed_casting_on**

compiler directive in the *Compiler Directives* chapter of the *Neuron C Reference Guide*.

- Macro arguments are not rescanned until after the macro is expanded, thus the macro operators **#** and **##** might not yield results as defined in the ANSI C standard when they occur in nested macro expansions.

- Names of network variables and message tags are limited to 16 characters. Names of functional blocks are limited to 16 characters unless they are declared using the **external_name** feature, in which case the external name is limited to 16 characters, and the internal name of the functional block is limited to 64 characters.

- A few ANSI C library functions are included in Neuron C, such as **memcpy( )** and **memset( )**. A string and byte operation library is provided to allow use of a subset of the ANSI C functions defined in the **<string.h>** include file. Other ANSI C library functions, such as file I/O and storage allocation functions, are not included in Neuron C. Consult the *Neuron C Reference Guide* for a complete and detailed list.

- The Neuron C implementation includes three ANSI include files: **<stddef.h>**, **<stdlib.h>**, and **<limits.h>**.

- Neuron C requires use of the function prototype feature whenever a call to the function precedes the function definition (see Chapter 2, *Focusing on a Single Device*, on page 17).

- Neuron C does not support the use of the ellipsis (. . .) in function prototypes or definitions.

- Neuron C contains additional reserved words and syntax not found in ANSI C. See the *Neuron C Reference Guide* for the syntax summary and the list of reserved words.

- Neuron C supports binary constants in addition to octal and hexadecimal. Binary constants are specified as 0b<*binary_number*>. For example, **0b1101** equals decimal 13.

- Neuron C supports the **//** comment style from C++, in addition to the traditional **/* */** C coment style. In the **//** style, two slashes (**//**) begin a comment. The comment is terminated by the end of the line, without further punctuation.

```
C code      /* An ANSI C and NEURON C comment */
C code      // A line-style NEURON C comment
```

- The **main( )** construct is not used. Instead, a Neuron C program's executable objects consist of **when** statements in addition to functions. A thread of execution always begins with a **when** statement, as described in Chapter 2, *Focusing on a Single Device*, on page 17.

- Neuron C does not support multiple source files in separate compilation units (however, the **#include** directive is supported).

-

- The Neuron C implementation of the **#error** directive supports both the standard-compliant unquoted form and the backwards compatible quoted form. The quoted form is recommended for compliance with Neuron

Compiler version 5, otherwise, use the unquoted form.   This change
supports both the quoted and unquoted form of a warning directive.

See Appendix D, *Neuron C Language Implementation Characteristics*, on page
253 for a description of how the Neuron C language conforms to "implementation-
specific" aspects of the ISO and ANSI C language definitions.

2

Focusing on a Single Device

This chapter describes the Neuron C event scheduler and I/O objects. The concepts of *predefined events* and *user-defined events* are introduced. Code examples in this chapter illustrate the use of events, I/O and timer objects, and I/O functions.

Objects that can be defined for each Neuron C application include *timers* and *input/output (I/O) objects*, described here; *network variables*, described in Chapter 3; *configuration properties*, described in Chapter 4; *functional blocks*, described in Chapter 5; and *application messages*, described in Chapter 6.

What Happens on a Single Device?

In this chapter, you begin to learn about programming a Neuron Chip or Smart Transceiver by focusing first on a single device. Each Neuron Chip and each Smart Transceiver has standard firmware, called the *Neuron firmware*, and hardware support that implement a scheduler, timers, and I/O device drivers and interfaces. Series 5000 and 6000 chips also provide hardware support for interrupts; see *Interrupts* on page 165 for more information.

The Neuron C language includes predefined objects that provide access to these firmware features. These objects are described briefly here, and in more detail later in this chapter:

- The Neuron firmware's *event scheduler* handles task scheduling for the application program. This chapter explains how to use the Neuron C language to define events and tasks, how the scheduler evaluates nonpriority events, and how you can define priority events.

- The Neuron C language offers two types of *timer* objects: millisecond and second timers. These timers can be used to affect the scheduling of tasks, as described in *Timers* on page 27.

- A number of *I/O objects* can be declared using Neuron C extensions to ANSI C. These I/O objects, as well as related I/O functions and events, are described in *Input/Output* on page 29.
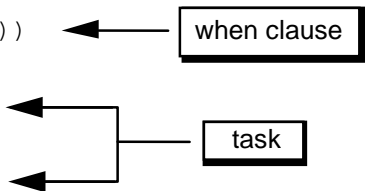
The Scheduler

The scheduling of application program tasks is event driven: when a given condition becomes TRUE, a body of code (called a *task*) associated with that condition is executed. The scheduler allows you to define tasks that run as the result of certain events, such as a change in the state of an input pin, receiving a new value for a network variable, or the expiration of a timer. You can also specify certain tasks as **priority** tasks, so that they receive preferential service (see *Priority When Clauses* on page 25). Series 5000 and 6000 chips also allow you to specify interrupt tasks that are serviced independently of the scheduler; see *Interrupts* on page 165 for more information.

When Clauses

Events are defined through **when** clauses. A **when** clause contains an expression that, if evaluated as TRUE, causes the body of code (the *task*) following the expression to be executed to completion. Multiple **when** clauses can be associated with a single task. A simple **when** clause and its associated task are shown below. The **when** clause or clauses and the associated task are frequently referred to as one entity known as a *when task* or a *when statement*.

```
when (timer_expires(led timer))        ◄──  when clause

{
        // Turn off the LED
        io_out(io_led, OFF);                        task
}
```

In this example above, when the **led_timer** application timer (definition not shown in this example) expires, the body of code (the task) that follows the **when** clause is executed to turn off the specified I/O object, **io_led** (also defined elsewhere in the program). After this task has been executed, the **timer_expires** event is automatically cleared. Its task is then ignored until the LED timer expires again and the **when** clause again evaluates to TRUE.

The following examples demonstrate various ways of using tasks and events. More information about tasks and events can be found in Chapter 7, *Additional Features*, on page 157, and **Figure 14** on page 159.

```
when (reset)
when (io_changes(io_switch))
when (!timer_expires)
when (flush_completes && (y == 5))
when (x == 3)
{
    // Turn on the LED and start the timer
    . . .
}
```

The **when** clauses cannot be nested. For example, the following nested **when** clause is not valid:

```
when (io_changes(io_switch))
{
    when (x == 3) {        // Can't nest!
    ...
    }
}
```

An equivalent result may be achieved by testing the event with an **if** statement:

```
when (io_changes(io_switch))
{
    if (x == 3) {
    ...
    }
}
```

When Statement

The syntax for a **when** statement (the **when** clause or clauses plus the associated task) is:

> *when-clause*
> *[when-clause ...]*
> *task*

The syntax for *when-clause* is:

[priority] [preempt_safe] when (*event***)**

priority   Forces evaluation of the following **when** clause each time the scheduler runs. See *Priority When Clauses* on page 25.

preempt_safe   Allows the scheduler to execute the associated **when** task even if the application is in preemption mode. See the discussions on preemption mode in Chapter 6, *How Devices Communicate Using Application Messages*, on page 123.

event   This expression is either a predefined event (see the following section) or any valid Neuron C expression (which can contain a predefined event). Predefined events as well as expressions are enclosed in parentheses. One or more **when** clauses can be associated with the same task.

task   A Neuron C compound statement, consisting of a series of Neuron C declarations and statements, enclosed in braces, which are identical to those found in a Neuron C function definition. The task is identical to the body of a **void** function (that is, it cannot return a value). A **return** statement can be used to terminate execution of the task but is not required.

When-tasks always compile into resident code, but code within a *when*-task can call resident or transient utility functions alike.

Types of Events Used in When Clauses

The events defined in a **when** clause fall into two general categories: predefined events and user-defined events. *Predefined events* use keywords built into the compiler. Examples of predefined events include input pin state changes, network variable changes, timer expiration, and message reception. *User-defined events* can be any valid Neuron C expression that evaluates or converts to a Boolean value.

The distinction between user-defined events and predefined events is not critical. Use predefined events whenever possible, because they require less code space.

There is one exception to the statement that a **when** clause can be any valid C expression. The **offline**, **online**, and **wink** predefined events must appear by themselves if used. All other predefined events may be combined into any arbitrary expressions. This restriction only applies to **when** clauses.

Examples:

```
when (msg_arrives)                // O.K.
when (msg_arrives && flag == TRUE)  // O.K.
when (online)                     // O.K.
when (online && flag == TRUE)       // Not permitted.
```

Predefined Events

The **timer_expires** event shown earlier is one type of predefined event. **Table 4** lists other predefined events that are represented by unique keywords.

Table 4. Predefined Events

Predefined Event	Where Described in This Manual
flush_completes	Chapter 7
io_changes	this chapter
io_in_ready	this chapter
io_out_ready	this chapter

Predefined Event	Where Described in This Manual
io_update_occurs	this chapter
msg_arrives	Chapter 6
msg_completes	Chapter 6
msg_fails	Chapter 6
msg_succeeds	Chapter 6
nv_update_occurs	Chapter 3
nv_update_completes	Chapter 3
nv_update_fails	Chapter 3
nv_update_succeeds	Chapter 3
offline	Chapter 7
online	Chapter 7
reset	this chapter
resp_arrives	Chapter 6
timer_expires	this chapter
wink	Chapter 7

A modifier that narrows the scope of the event may follow some predefined events, such as the I/O events and network variable events. If the modifier is optional and not supplied, any event of that type qualifies.

Predefined events can also be used as any sub-expression, including within the control expression of **if**, **while**, and **for** statements. This method is termed *direct event processing*. An example of direct event processing is:

```
mtimer t;
when (event)
{
    . . .
    if (timer_expires(t)) {
        io_out(io_led, OFF);
    }
    . . .
}
```

Any built-in event keyword or keyword expression (such as **timer_expires(t)**) is treated the same as any other sub-expression and any combination allowed by standard C expression syntax is allowed when programming in Neuron C.

The special case of the **io_changes** event expression *must be treated carefully*. The **to** and **by** qualifier keywords are treated as general expression operators for purposes of precedence (although they are only permitted in combination with **io_changes**). These operators are of equal precedence with each other, but they are mutually exclusive. They are of higher precedence than relational operators (that is, comparisons), but lower in precedence than shift and arithmetic operators.

Following are examples of how the **io_changes** event expression is parsed:

> **io_changes (***device***) by a + b**
> <u>as</u>:
> **io_changes (***device***) by (a + b)**

> <u>and</u>

> **io_changes (***device***) by a < b**
> <u>as</u>:
> **(io_changes (***device***) by a) < b**

As with any other C operators, the implied precedence can be explicitly changed by parenthesization. Parentheses should *always* be used to improve clarity of the code if there is any doubt. Use of extra parentheses has no negative effect upon the compilation or the code generated.

The Neuron C compiler detects the use of predefined event keywords in **when** clauses and treats them specially for code optimization purposes. However, when event keywords are used as sub-expressions within **when** clauses, event table optimizations cannot be used. In the examples below, the first case uses the event table optimization, the second and third do not:

> **when (timer_expires) { }**

> **when (! timer_expires) { }**

> **if (timer_expires)**

Although the **io_changes** expression (**by** and **to** varieties) does not *require* a constant value, only *constant-valued* **io_changes** expressions are optimized into the **when** clause event table.

Event Processing

Events related to network activity are processed using two separate queues. One queue serves the following events related to incoming network messages:

> **nv_update_occurs**

> **msg_arrives**

> **online**

> **offline**

> **wink**

The other queue serves the remaining network events pertaining to completion events and responses:

> **nv_update_completes**

> **nv_update_succeeds**

nv_update_fails

msg_completes

msg_succeeds

msg_fails

resp_arrives

Most network events, except **resp_arrives**, are enqueued only if the Neuron C compiler has determined that the application checks for the event. The **online**, **offline**, and **wink** events are always enqueued but are discarded by the scheduler if no corresponding **when** clause is found.

When it reaches the head of the queue, an event remains there until processed by the application. Therefore, any network event that is checked for by an application must be checked for frequently, or the event remains at the head of the queue, effectively blocking that queue. A blocked queue prevents the application from continuing normal processing of events and can cause the device to fail to respond to any subsequent application or network management messages.

This is a particularly critical consideration for **nv_update_occurs** and **msg_arrives** events, which can arrive unsolicited at any time; in comparison, completion events and responses arrive only as the result of application-initiated outgoing network activity. The Neuron C compiler determines that an event is handled by the application by virtue of its presence in the program, even if it is never checked for in a **when** clause, or is only checked for in special circumstances.

Reset Event

The **reset** event is TRUE the first time this event is evaluated after the Neuron Chip or Smart Transceiver is reset for any reason. Note that I/O object and global variable initializations are performed before processing any events. The reset event task is the first task to be executed after reset of the Neuron Chip or Smart Transceiver.

The reset event task executes only if the device is in the configured state (that is, if the device is not applicationless, hard-offline, or unconfigured). Also, the reset event task runs when the device is unconfigured if the directive **#pragma run_unconfigured** is specified in the application program. The task runs regardless of whether the device is soft-offline or not. The soft-offline state is not reset-retained, so the only case where this is meaningful is when the device transitions from unconfigured or hard-offline to configured state after a reset, as would typically happen during initial commissioning. In this case, the device executes the reset task followed by the offline task.

A reset occurs as a natural part of the process of commissioning a LONWORKS device, and the reset process includes the execution of the reset event task. The device undergoes a state transition to complete the commissioning process, and that state transition can only be completed after the reset event task has been executed. Consequently, you should keep the reset event task short so that the device can be commissioned at maximum speed. You must keep the total reset event task processing time under 18 seconds to prevent commissioning failures. Reset event task processing time includes Neuron firmware initialization time as described in the Smart Transceivers databooks.

User-Defined Events

A user-defined event can contain assignments and function calls. Such events using complex expressions can affect the response time for all events within a program, so you must minimize calls to complex functions within user-defined events. Assignments within user-defined events can only be done to global variables.

Furthermore, the evaluation of an event keyword or an event expression, such as **timer_expires(t)**, clears any pending event, regardless of whether the entire expression is TRUE or FALSE, as below:

> **when ((timer_expires(t)) && (flag = = TRUE))**

As with ANSI C compilers, the Neuron C compiler evaluates logical expressions only as much as needed. For example, in an **if (a && b)** expression, the **b** term is only evaluated if **a** is TRUE, and in an **if (a | | b)** expression, the **b** term is only executed if **a** is FALSE. This is called *short-circuit evaluation*, and is specified by the ANSI C language definition.

When combining user-defined expressions with a predefined event using the logical operators discussed in the paragraph above, you must make sure that this does not prevent the predefined events from being evaluated as needed, in order to avoid blockage of the event queue as discussed earlier in this chapter.

For example, the following user-defined event expression is okay:

> **when ((timer_expires(t)) && (flag = = TRUE))**

But, if the expression above is reversed, as shown below, it is likely to cause a blockage of the event queue if the **flag** variable is true for any significant time, because the short-circuit nature of the logical-and operator can prevent the timer expiration event from being checked at all. Thus, the reversed expression shown below must be avoided:

> **when ((flag = = TRUE) && (timer_expires(t)))**

Scheduling of When Clauses

The scheduler evaluates **when** clauses in round-robin fashion: Each **when** clause is evaluated by the scheduler and, if TRUE, the task associated with it is executed. If the **when** clause is FALSE, the scheduler moves on to examine the following **when** clause. After the last **when** clause, the scheduler returns to the top and moves through the group of **when** clauses again.

Example: A group of **when** clauses might look like the following:

```
when (nv_update_occurs) // Event A
  //  {task to execute}

when (nv_update_fails)  // Event B
  //  {task to execute}

when (io_changes)       // Event C
  //  {task to execute}

when (timer_expires)    // Event D
  //  {task to execute}
```

Letter names shown above are used for the clauses in **Figure 1** and the following narration of events. This shows how the order of execution of tasks differs from the order the **when** clauses appear in a program.

At the start of this example, no event has occurred, thus no when clause event expression is TRUE.

1   The scheduler begins with A. Since A is FALSE, its task is not executed.

2   Event C occurs and the expression C becomes TRUE.

3   The scheduler moves to B. Since B is FALSE, its task is not executed.

4   The scheduler moves to C. Since C is TRUE (item 2, above), its task is executed.

5   A becomes TRUE.

6   The scheduler moves to D. Since D is FALSE, its task is ignored.

7   The scheduler moves back to A. Since A is TRUE (item 5, above), its task is executed.
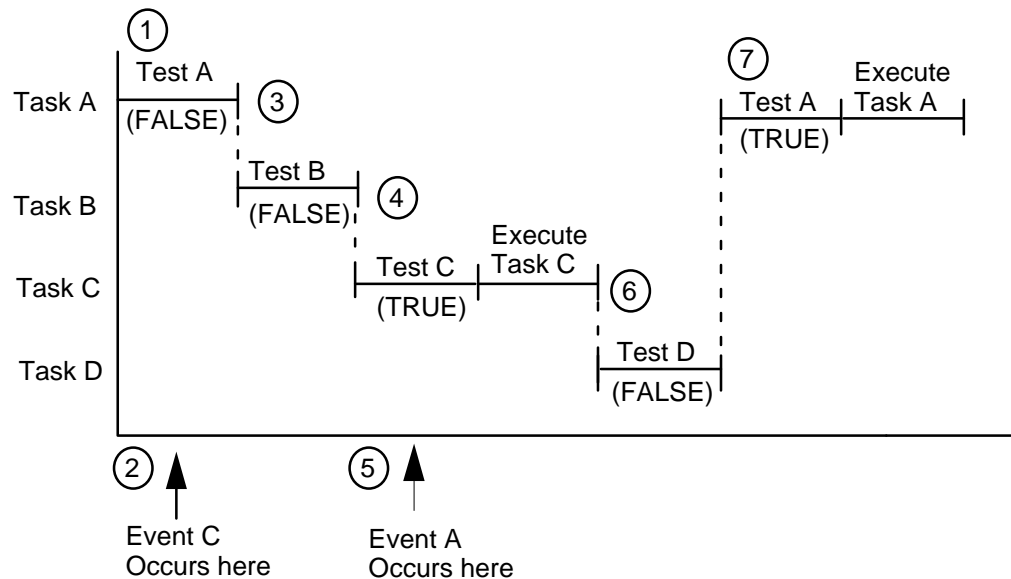
Figure 1. Example Scheduler Timeline

Priority When Clauses

The **priority** keyword can be used to designate **when** clauses that should be evaluated more often than nonpriority **when** clauses. Priority **when** clauses are evaluated in the order specified *every* time the scheduler runs. If any priority **when** clause evaluates to TRUE, the corresponding task is executed and the scheduler starts over at the top of the priority **when** clauses.

If none of the priority **when** clauses evaluates to TRUE, then a nonpriority **when** clause is evaluated, selected in the round-robin fashion described earlier. If the selected nonpriority **when** clause evaluates to TRUE, its task is executed. The scheduler then resumes with the first priority **when** clause. If the nonpriority **when** clause selected evaluates to FALSE, its task is ignored and the

scheduler resumes with the first priority **when** clause. See **Figure 14** on page 159.

The scheduling algorithm described above can be modified through use of the **scheduler_reset** pragma, discussed in Chapter 7, *Additional Features*, on page 157.

Important: Excessive use of priority **when** clauses might starve execution of nonpriority **when** clauses. If a priority **when** clause is true the majority of the time, it monopolizes processor time. Priority **when** clauses should be designed to be true only rarely, or the remaining tasks must be tolerant of not being executed frequently and responsively.

Interrupts

Neuron C supports application-specific asynchronous interrupts from various interrupt sources, and provides a semaphore for synchronization in a multiprocessing environment.

Each **interrupt** statement consists of an **interrupt** clause, followed by an interrupt task. Unlike **when** statements, which can include more than one **when** clause, **interrupt** statements support only a single **interrupt** clause.

The **interrupt** clause defines the interrupt source and the conditions that trigger the interrupt. The interrupt task contains code that runs as a result of the interrupt.

Example:

```
interrupt(IO_3, clockedge(-)) {
   ...
}
```

Interrupt clauses support interrupt requests from signal conditions on I/O pins, from timer or counter I/O objects, or from the high-resolution system timer.

A simple **__lock{ }** construct implements synchronization through a hardware semaphore.

See *Interrupts* on page 165 for more information.

Function Prototypes

Neuron C requires the use of function prototypes if a function is to be called before it is defined. Examples of valid prototypes include the following:

```
void f(void);
int g(int a, int b);
```

The following are not considered prototypes because they do not have argument lists. They are merely forward declarations:

```
void f();
g();  // defaults to 'int' return value
```

If you define a function before you call it, Neuron C automatically creates an internal prototype for you. Only one prototype is created for a given function. The following examples are technically not prototypes, but Neuron C creates function prototypes for them:

```
void f()
{ /* body */ }

g (a,b)
int a;
int b;
{ /* body */ }
```

Although Neuron C can create prototypes, it does *not* employ the ANSI C
Miranda prototype rule. According to the Miranda prototype rule, if a function
call does not already have a prototype, a prototype is automatically created for it.
In Neuron C, a function prototype is automatically created only when the
function is defined.

Timers

Two types of software timer objects are available to a Neuron C application:
millisecond timers and second timers. The millisecond timers provide a timer
duration of 1 to 64,000 milliseconds (or .001 to 64 seconds). The second timers
provide a timer duration of 1 to 65,535 seconds. For more accurate timing of
durations of 64 seconds or less, use the millisecond timer. These timers are
separate from the two hardware timer/counters in the Neuron core (see also
Input Clock Frequency and Timer Accuracy on page 38).

For Series 5000 and 6000 chips, a high-resolution hardware timer is also
available. You can program an interrupt handler to asynchronously handle
interrupts that occur based on this hardware timer; see *Interrupts* on page 165
for more information.

Declaring Timers

A maximum of 15 timer objects (total of both types) can be defined within a single
program. A timer object is declared using one of the following:

> **mtimer** [**repeating**] *timer-name* [**=***initial-value*]**;**
> **stimer** [**repeating**] *timer-name* [**=***initial-value*]**;**

mtimer   Indicates a millisecond timer.

stimer   Indicates a second timer.

repeating An option for the timer to restart itself automatically upon expiration.
With this option, accurate timing intervals can be maintained even if
the application cannot respond immediately to an expiration event.

timer-name A user-supplied name for the timer. Assigning a value to this name
starts the timer for the specified length of time (the specified time is
in seconds for an **stimer** and milliseconds for an **mtimer**). A timer
that is running or has expired can be started over by assigning a new
value to this name. The timer object can be evaluated while the timer
is running, and it indicates the time remaining. Setting the timer to
0 turns the timer off. No timer expiration event occurs for a timer
that has been turned off (see the description of the **timer_expires**
event described in the *Neuron C Reference Guide*).

initial-value   An optional initial value to be loaded into the timer on power-up or reset. Zero is loaded by the Neuron firmware (in other words, the timer is turned off) if no explicit initial-value is supplied.

Examples

An example of declaring a timer object and assigning a value to it is shown below:

```
// start timer with value of 5 sec
stimer led_timer = 5;
```

An example of turning a timer off is shown below:

```
stimer led_timer;
when (some-event)
{
    led_timer = 0;
}
```

An example of evaluating the value of a running timer is shown below:

```
stimer repeating led_timer;
when (some-event)
{
    time_remaining = led_timer;
    ...
}
```

Note: When setting and examining timers in the NodeBuilder debugger, certain inaccuracies could occur. When a timer is set during program execution and is examined while the program is halted (includes single stepping and breakpoints), the timer value can be as much as 200 milliseconds larger than the actual time until expiration. No such inaccuracy exists on a timer that is allowed to run without a debugger halt.

The timer_expires Event

The **timer_expires** event becomes TRUE when a timer expires. The syntax of this event is the following:

timer_expires [**(***timer-name***)**]

timer-name Specifies a specific timer to check.

If *timer-name* is not included, the event is an unqualified **timer_expires** event. An *unqualified event* expression is one that omits the optional qualifier syntax that limits the objects to which the event applies.

A timer event is unique because it can be cleared only by checking for specific (qualified) timer expiration events. Other events can be cleared by checking for either the qualified or unqualified events. For example, the following **when** clause checks for the expiration of the **led_timer**, so the **timer_expires** event for that timer is cleared to FALSE.

Examples:

```
stimer led_timer;
when (timer_expires(led_timer))
{
    io_out(io_led, OFF);  // Turn off the LED
```

```
        }
```

If your program has multiple timers, you must include a specific check for each timer so that the expiration event is cleared, as shown below:

```
mtimer x;
mtimer y;
mtimer z;
when (timer_expires(x))
{
    // task
}
when (timer_expires(y))
{
    // task
}
when (timer_expires(z))
{
    // task
}
```

An alternate style of checking for specific timers is shown below. This example also demonstrates that an event expression is not restricted to use only in a **when** clause.

```
when (timer_expires)
{
    if (timer_expires(x))
      ...
    else if (timer_expires(y))
      ...
    else if (timer_expires(z))
      ...
}
```

Note: Be sure to check for specific timer events while using the unqualified **timer_expires** event. Unlike all other predefined events, which are TRUE only once per pending event, the <u>unqualified</u> **timer_expires** event remains TRUE as long as *any* timer has expired.

Which style you choose to use for checking timer expiration depends on the circumstances in your application. Use the first style of checking for specific timers if you're concerned about code space. Use the second style if you're concerned about speed of execution, performance, or response time.

For an example of a complete program that declares a timer and uses the **timer_expires** event, see *Example 1: Thermostat Interface* on page 33.

Input/Output

Each Neuron Chip and each Smart Transceiver has a variety of built-in electrical interface options for performing input and output (I/O) functions. Before performing I/O, you must first declare the I/O objects that monitor and control the 11 or 12 Neuron Chip or Smart Transceiver I/O pins, named **IO_0, IO_1, ..., IO_11**.

To perform I/O, you normally use the built-in I/O functions: **io_in( )**, **io_out( )**, **io_set_direction( ), io_select( ), io_change_init( )**, and **io_set_clock( )**. The **io_out_request( )** function is used to perform I/O with the **parallel** I/O object.

I/O objects can also be linked to Neuron C events, because changes in I/O often affect task scheduling.

The Neuron C declaration syntax for I/O objects is described in detail in the *I/O Model Reference for Smart Transceivers and Neuron Chips*.

I/O Object Types

Many I/O models are available for Neuron Chips and Smart Transceivers. Certain I/O models are available only for specific chip types, but most are available to all Neuron Chips and Smart Transceivers. A Neuron C application instantiates an I/O model as an I/O object.

The I/O models are grouped into the following categories:

- *Direct I/O Models* are based on a logic level at the I/O pins; none of the Neuron Chip or Smart Transceiver hardware's timer/counters are used in conjunction with these I/O models. These models can be used in multiple, overlapping combinations within the same Neuron Chip or Smart Transceiver. Direct I/O models include the following types:

Input Model Types	*Output Model Types*
bit	**bit**
byte	**byte**
leveldetect	**nibble**
nibble	**touch**
touch	

- *Timer/Counter I/O Models* use a timer/counter circuit in the Neuron Chip or Smart Transceiver. Each Neuron Chip and each Smart Transceiver has two timer/counter circuits: One whose input can be multiplexed, and one with a dedicated input. Timer/counter I/O models include the following types:

Input Model Types	*Output Model Types*
dualslope	**edgedivide**
edgelog	**frequency**
infrared	**infrared_pattern**
ontime	**oneshot**
period	**pulsecount**
pulsecount	**pulsewidth**
quadrature	**stretchedtriac**
totalcount	**triac**
	triggeredcount

- *Serial I/O Models* are used for transferring data serially over a pin or a set of pins. The **neurowire**, **i2c**, **magcard**, **magcard_bitstream**, **magtrack1**, and **serial** I/O models are mutually exclusive within a single Neuron Chip or Smart Transceiver. Both the input and output versions of a serial I/O model can coexist within a single Neuron Chip or Smart Transceiver. Serial I/O models include the following types:

Serial Input Model Types	*Serial Output Model Types*
bitshift	**bitshift**
magcard	**serial**
magcard_bitstream	

magtrack1
serial
wiegand

Serial Input/Output Model Types
i2c
neurowire
sci
spi

- *Parallel I/O Models* are used for high-speed bidirectional I/O. I/O models within this group use all of the Neuron Chip or Smart Transceiver I/O pins. The parallel I/O models include the following types:

  Parallel Input/Output Model Types
  muxbus
  parallel

Declaring I/O Objects in Neuron C

Declaring an I/O object in a Neuron C application performs both of the following tasks:

- Informs the compiler what type of I/O operation is performed, and on which pin or pins. The compiler creates instructions that configure the hardware within the Neuron core as a result of this declaration. The firmware configures the hardware whenever the device or application is reset.

- Associates the name of the I/O object with the hardware.

The general syntax for declaring an I/O objects in the Neuron C language is shown below.

pin direction type [options] io-object-name;

pin

One of the Neuron C keywords that name one of the twelve I/O pins, **IO_0** through **IO_11** (pin **IO_11** is available only on Series 3100 Power Line devices and Series 5000 and 6000 devices). The named pin defines the first pin for multi-pin I/O objects. In general, pins can appear in a single object declaration only. However, a pin can appear in multiple declarations of the **bit**, **nibble**, and **byte** I/O object types. Also, **IO_8** can appear in multiple declarations of **neurowire master** specifying different **select** pins. In this case, it is not required that all declarations have the same sense, that is, input or output; see the *I/O Model Reference*.

direction

Specifies whether the pin is an input or an output. Some I/O objects are bidirectional, and do not require the specification of direction.

type

Specifies the I/O object type.

options

> Optional I/O parameters, dependent on the chosen *type* for the I/O object. The description of each object type includes the type's available options. Except where noted, these options can be listed in any order. All options have default values that are used when you do not include the option in the object declaration.

io-object-name

> A user-supplied name for the I/O object, in the ANSI C format for variable identifiers.

The description for each I/O object includes a detailed explanation of the syntax for each I/O object type.

Example: A logic level needs to be measured at the IO3 input pin of the device, which is named **IO_3** in Neuron C. The pin is connected to a proximity detector, as its programmatic name indicates.

```
IO_3 input bit ioProxDetector;
```

Whenever your program refers to the **ioProxDetector** variable, it is actually referring to the logic level on pin IO3.

Some I/O objects can be combined or multiplexed according to specific rules. In addition, there are a number of Neuron C functions and events predefined for various I/O objects. See the *I/O Model Reference for Smart Transceivers and Neuron Chips* for more information about I/O objects.

Device Self-Documentation

You can include a text string that describes your device in your application. This text string can be accessed by any network management tool, and can be used by a network integrator to verify that they have the correct device when designing in or installing your device. This text string is appended to the device self-documentation (SD) string. The Neuron C compiler automatically generates a portion of the SD string that documents the functional profiles that are implemented by the functional blocks in your application. You can add additional text for the SD string using the following compiler directive as described in the *Compiler Directives* chapter of the *Neuron C Reference Guide*:

> **#pragma set_node_sd_string** *C-string-const*

Examples

This section presents three complete programs that illustrate Neuron C capabilities and good coding style. The examples are:

1 Thermostat interface

2 Simple light dimmer interface

3 Seven-segment LED display interface

Example 1: Thermostat Interface

This thermostat measures the resistance of a thermistor by measuring the pulse-width of a waveform that is input to pin IO6. The I/O object declaration is set up to measure the on-time of the waveform. A simple *T=mx+b* scaling of the on-time yields the temperature.
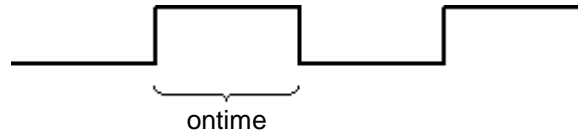
The example also uses a shaft encoder generating a quadrature input as a dial to select a new temperature setting (see **Figure 2**). The **quadrature** input object type is used with the **io_update_occurs** event. The input value of the input object represents the change in rotational offset since the last input. Shaft encoders typically generate offsets of 16 to 256 counts per 360 degrees rotation. The **io_update_occurs** event evaluates to TRUE only when a nonzero offset has been measured. In the following application, the task associated with the **when (io_update_occurs...)** clause is executed only when the quadrature input dial has moved from the previously measured position.
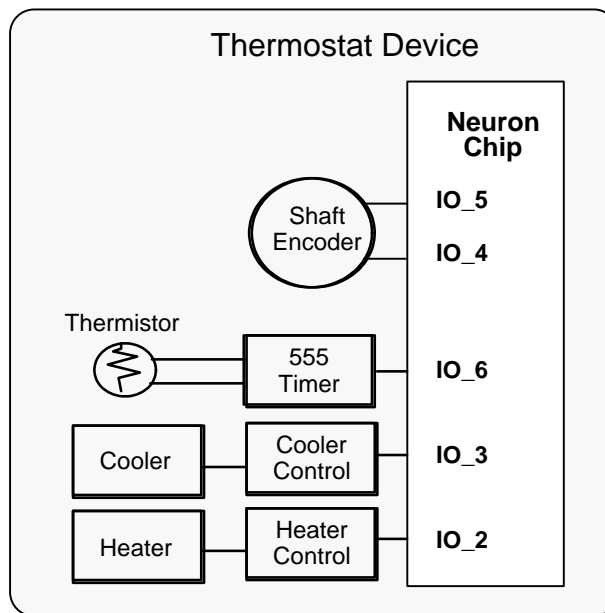
Figure 2. Sample Thermostat Device

The **io_changes** event would rarely be used with the **quadrature** I/O object, because the event would evaluate to TRUE only when a *change* in the measured count occurred. The **io_changes** event would not evaluate to TRUE as long as the input object were moving at a constant rate because the nonzero measurements would be the same. This example is intended to illustrate use of typical I/O objects. Network variable information has been omitted; it is covered in detail in Chapter 3, *How Devices Communicate Using Network Variables*, on page 45.

```
// THERMOS.NC -- LonWorks thermostat device
```

```
// Uses a thermistor to measure temperature, and a
// quadrature encoder to enter setpoint.  Activates either
// heating or cooling equipment via bit outputs.

///////////////// Include Files //////////////////
#include <stdlib.h>
        // for muldiv()
//////////////////////// Timers /////////////////
stimer repeating tmCheckHeatOrCool;
        // Automatically repeating timer

//////////////// Constants /////////////////////////
#define TEMP_DEG_F(t)  (((long)t - 32L) * 50 / 9 + 2740)
      // macro to convert degrees F to SNVT_temp
const SNVT_temp DESIRED_TEMP_MAX = TEMP_DEG_F(84);
const SNVT_temp DESIRED_TEMP_MIN = TEMP_DEG_F(56);
const SNVT_temp BAND_SIZE = 10;
// Guardband of +/- 1 deg C around desired temperature

/////////////// I/O Objects /////////////////////
IO_6 input ontime clock (1) invert ioTempRaw;
IO_4 input quadrature ioShaftIn;
IO_2 output bit ioHeatingOn = FALSE;
IO_3 output bit ioCoolingOn = FALSE;

/////////////// Global Variables ////////////////////
SNVT_temp newTemp     = TEMP_DEG_F(70);// init to 70 deg F
SNVT_temp desiredTemp = TEMP_DEG_F(70);

enum {
    OFF, HEATING, COOLING
} equip = OFF;    // current state of HVAC equipment

////////////////// Tasks /////////////////////
// I/O update task --
// read thermistor voltage-to-frequency converter
when (io_update_occurs(ioTempRaw)) {
    // An update occurs periodically as the ontime is
    // sampled. The new sample is placed in 'input_value.'
    // Calculation is performed using 32-bit intermediate
    // math, then the result stored as a SNVT_temp. The
    // input is scaled based on the temperature coefficient
    // of the thermistor.
    newTemp = muldiv(input_value, 25000, 9216) + 2562;
}

///////////////////////////////////////////////
//  I/O update task -- read quadrature encoder
// A quadrature input is used as a dial to select a new
// temperature setting.

when (io_update_occurs(ioShaftIn)) {
// An update occurs for a quadrature I/O object when the
// accumulated offset is nonzero. The value is placed in
// 'input_value' by the io_update_occurs event.
    desiredTemp += input_value;     // Assumes no overflow
```

Focusing on a Single Device

```
            desiredTemp = min(DESIRED_TEMP_MAX, desiredTemp);
            desiredTemp = max(DESIRED_TEMP_MIN, desiredTemp);
        }
        //////////////////////////////////////////////////////
        //  Timer task -- execute control algorithm
        // A timer is used to decide periodically whether to
        // activate heating or cooling. The temperature comparison
        // is done only every five minutes to prevent cycling the
        // equipment too frequently. There are two digital outputs:
        // one for activating the heating equipment, and one for
        // activating the cooling equipment.
        when (timer_expires(tmCheckHeatOrCool)) {
            switch (equip) {
            case HEATING:
                if (newTemp > desiredTemp) {   // if too hot
                    equip = OFF;                // turn off heater
                    io_out(ioHeatingOn, FALSE);
                }
                break;

            case OFF:
                if (newTemp < desiredTemp - BAND_SIZE) {
                    equip = HEATING;          // if too cold, then
                    io_out(ioHeatingOn, TRUE);  // turn on heater
                } else if (newTemp > desiredTemp + BAND_SIZE) {
                    equip = COOLING;          // if too hot, then
                    io_out(ioCoolingOn, TRUE);  // turn on cooler
                }
                break;

            case COOLING:
                if (newTemp < desiredTemp) {   // if too cold
                    equip = OFF;                // turn off cooler
                    io_out(ioCoolingOn, FALSE);
                }
                break;
            }
        }

        //////////////////////////////////////////////////////
        //  Reset task -- Set the repeating timer to 300 seconds

        when (reset) {
            tmCheckHeatOrCool = 300;  // 5 minutes, repeating
        }
```

Example 2: Simple Light Dimmer Interface

The following example shows Neuron C code for a simple light dimmer. The example uses two I/O objects, a **triac** control circuit to control the lamp brightness and a **quadrature** input to select the light level (see **Figure 3** on page 36). For the **triac** output object, a value of 1 is maximum brightness, and a value of 320 is minimum brightness (OFF) when the line frequency is 60 Hz. The initial value on power-up is full OFF (65535).

The **io_update_occurs** event is used in a **when** clause. An implicit call to **io_in( )** occurs when this event is called. The program can then access the measured value through the built-in variable **input_value**.
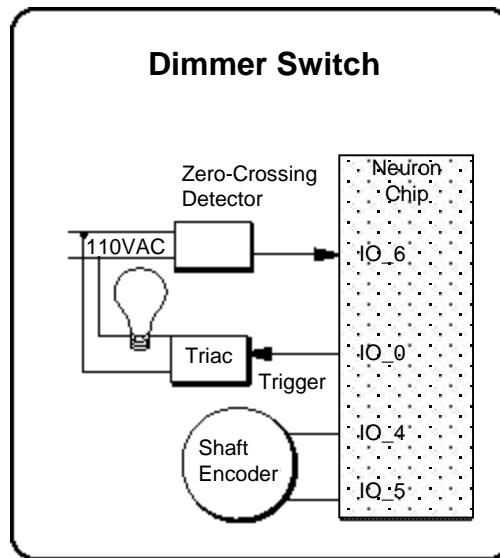
Figure 3. Simple Dimmer Device

```
// DIMMER.NC -- LonWorks triac dimmer control

// Uses a triac output to control an incandescent lamp
// Uses a shaft encoder input to set desired lighting level

//////////////////////// I/O Objects ////////////////////////
IO_0 output triac pulse sync (IO_6) clock (6) ioLampTriac;
IO_4 input quadrature ioShaftIn;

//////////////////////// Constants ////////////////////////
// These constants are appropriate for 60Hz line frequency
const unsigned long MIN_BRIGHTNESS = 320;
const unsigned long MAX_BRIGHTNESS = 1;

//////////////////// Global Variables ////////////////////
signed long currentBrightness;

//////////////////////// Tasks ////////////////////////

//  Reset task -- turn the lamp off
when (reset) {
    io_out(ioLampTriac, MIN_BRIGHTNESS);
    currentBrightness = MIN_BRIGHTNESS;
}

//  I/O update task -- read quadrature input dial
//                     to select the light level
when (io_update_occurs(ioShaftIn)) {
    // An update occurs for a quadrature input
    // object when the accumulated offset is
```

```
                // nonzero.  The sample value is in
                // 'input_value'.  The value is subtracted
                // since a lower value means more light.

                currentBrightness -= input_value;

                // Look for underflow or overflow
                if (currentBrightness < MAX_BRIGHTNESS)
                    currentBrightness = MAX_BRIGHTNESS;
                else if (currentBrightness > MIN_BRIGHTNESS)
                    currentBrightness = MIN_BRIGHTNESS;
                // Change the triac setting to the
                // desired brightness level
                io_out(ioLampTriac, currentBrightness);
            }
```

Example 3: Seven-Segment LED Display Interface

The following example shows how to connect multi-character displays to the
neurowire port. The display has an 8-bit configuration register and a 24-bit
display register. This configuration can be defined as follows:

```
IO_2 output bit ioEnable = 1;
IO_8 neurowire master select(IO_2) ioDisplay;
unsigned char displayReg[3];
unsigned char configReg;
   .
   .
   .
void refreshDisplay() {
  io_out(ioDisplay, &configReg, 8);
  io_out(ioDisplay, displayReg, 24);

}
```
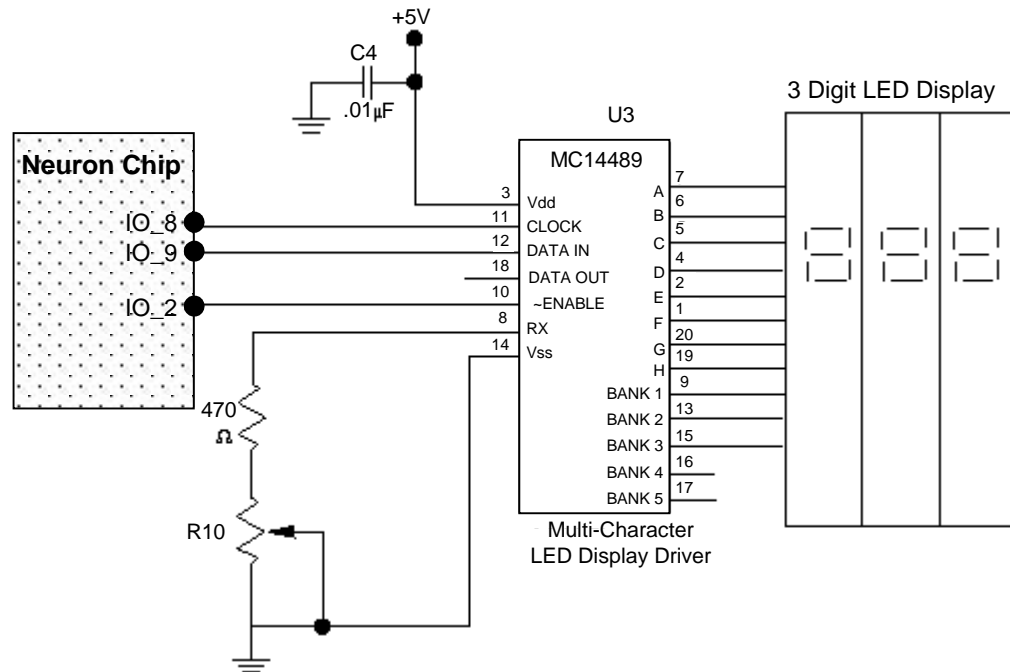
Figure 4. Neurowire Connection to a Display

Note that the figure does not show pull-up resistors for the IO_2, IO_8, or IO_9 pins; for a Series 3100 device, you can use the **#pragma enable_io_pullups** directive to add pull-ups to these pins; for a Series 5000 or 6000 device, you need to add external pull-up resistors for the pins.

Input Clock Frequency and Timer Accuracy

Depending on the manufacturer and version, the Neuron Chip and Smart Transceiver system clock frequencies are 80 MHz, 40 MHz, 20 MHz, 10 MHz, 6.5536 MHz, 5 MHz, 2.5 MHz, 1.25 MHz, and 625 kHz. Certain timers listed below are *fixed timers*; that is, they have the same absolute duration regardless of the input clock selected. However, the slower the system clock, the less accurate the timer. *Scaled timers*, also listed below, scale in proportion to the input clock.

Fixed Timers

In general, timers discussed in this manual are of fixed duration unless noted otherwise. The following timers are implemented in hardware and have periods that are independent of the Neuron Chip or Smart Transceiver input clock frequency. However, the accuracy of these timers is determined by the accuracy and frequency of the input clock for the Neuron Chip or Smart Transceiver.

- Preemption mode timeout timer.

- Pulsecount input timer. Timer used to determine the counting interval for the **pulsecount** input object. The interval is (223)/107 (approximately 0.8388608) seconds.

- Triac pulse timer. Timer used to generate pulses for the **stretchedtriac** and **triac** output objects.

The following timers are implemented in software and have periods that are independent of the Neuron Chip or Smart Transceiver input clock. The accuracy of these timers is discussed in the next section.

- Application second timer (that is, an **stimer** declared in a Neuron C program).

- Application millisecond timer (that is, an **mtimer** declared in a Neuron C program).

Scaled Timers and I/O Objects

Timers and I/O objects that scale with the input clock are directly proportional to the input clock. For example, a serial object configured at 2400 bps would actually run at 600 bps given a 2.5 MHz (1/4 speed) input clock. The following timers scale with the input clock:

- **bitshift** clock

- **neurowire master** clock

- **serial** clock

- watchdog timer (only scales for Series 3100 devices)

Note: The configurable EEPROM write timer accuracy is affected by the input clock. See *EEPROM Write Timer* on page 43 for more information.

Calculating Accuracy for Software Timers

This section describes the accuracy of the millisecond and second timers for various system clock rates.

Accuracy of Millisecond Timers

The following formulas define the range of accuracy for a millisecond timer. Accuracy is expressed as a low and high duration. The low duration *(L)* is the minimum time from when a timer is set to when the system posts an event for the application. The high duration *(H)* is the maximum time from when a timer is set to when an event is posted. *L* and *H* are expressed below as a function of *E*, the expected duration.

The added delay to detect the expiration event, that is, the *latency*, is a function of the application and is *not* included in these formulas. For example, an event posted while the application is executing a task associated with a **when** clause is be detected until the executing task completes and returns control of the application to the scheduler.

Note*:* When an event is posted by the Neuron firmware, it becomes visible to the scheduler and to other events (for example, **io_changes**, **nv_update_occurs**).

With a 10 MHz Input Clock

In the following formula, the *floor( )* function returns the largest integer not greater than the argument, for example, *floor(3.3)* = 3 or *floor*(3.0) = 3. For a

Series 3100 device with a 10 MHz clock, the expected duration of a millisecond timer is:

$$E = 0.8192 * floor((D/0.82) + 1)$$

where D is the specified duration for the timer. For example, for a timeout of 100 ms, E equals 99.94 ms.

For a Series 3100 device with a 10 MHz clock with a 10 MHz clock, the low duration is:

$$L = E - 12 \text{ ms}$$

and the high duration is:

$$H = E + 12 \text{ ms}$$

With Other Clock Speeds

The following formulas allow you to calculate accuracy for millisecond timers when other input clock rates are selected. In these formulas, S depends on the input clock rate or the system clock rate, as shown in **Table 5**.

Table 5. Determining S

S	Input Clock Rate (Series 3100)	System Clock Rate (Series 5000 and 6000)
0.063	—	80 MHz
0.125	—	40 MHz
0.25	40 MHz	20 MHz
0.5	20 MHz	10 MHz
1	10 MHz	5 MHz
1.5259	6.5536 MHz	—
2	5 MHz	—
4	2.5 MHz	—
8	1.25 MHz	—
16	625 kHz	—

$$E = 0.8192 * floor ((floor(D/S)*S)/0.82) +1)$$

Two factors determine E. The first is that the slower the input clock speed, the less granular the input clock. For example, at 1/16 speed, the millisecond granularity is 16 milliseconds (one clock tick every 16 milliseconds). The second factor is that the hardware generates 819.2 microsecond ticks that the software

treats as 820 microsecond ticks. This means that a timer duration is actually 0.999 times the specified duration.

For example, for a Series 3100 device with a 2.5 MHz clock, a specified timeout of 99 ms would result in an expected duration of 96.67 ms.

The complete formulas for calculating the low and high durations are:

$$L = E - (11*S + 1)$$

$$H = E + (11*S + 1)$$

The high duration with a 2.5 MHz clock and a specified timeout of 99 ms would thus equal 141.67 ms; the low duration is 51.67.

Note: The number "11" in the formulas above is based on a typical worst case scenario. In the absolute worst case, that is, the maximum number of timers, network variables, addresses, and so on, this number can be as high as 32.

In addition, the high duration may be increased by *network management delay* (NMD), an additional skew introduced by network management message processing. Normally, this term is 0. But, if a device were to process a network management message, the upper range for any given timeout could be significantly increased. For example, adding a domain to a device would result in an NMD of anywhere from 300 ms to $(300 + 838*S)$ ms. In general, network management operations of this type occur infrequently. It is always good practice to take a device offline, if possible, before sending further network management messages.

To measure an event's duration, a timer can be polled before and after the event, and the difference can be calculated. To measure the duration of events less than 50 milliseconds, use the **get_tick_count( )** function instead of the software timers (see the *Neuron C Reference Guide*).

Repeating Timers

For repeating timers, there is no cumulative drift other than that produced by the difference in D and E. The Nth timeout for repeating timers occurs in the range of LR to HR, where:

$$ER = E * N$$

and

$$LR = ER - (11*S + 1)$$

$$HR = ER + (11*S + 1)$$

For repeating timers, intermediate timeout events are lost if the following is true:

$$\text{abs}(AR - ER) \geq E$$

$$ER - AR > E$$

where AR is the actual duration of the repeating timer.
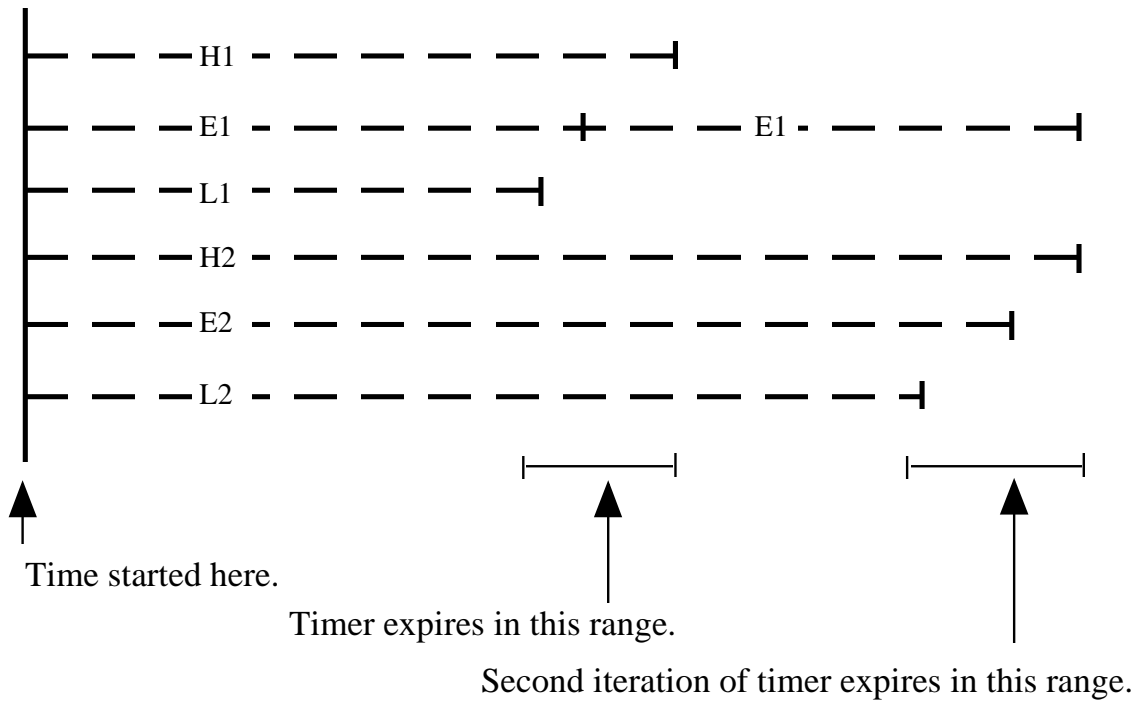
Figure 5. Expected, Low, and High Duration of Timeout Events

Accuracy of Second Timers

The second timers rely on the one-second timer, which is based on the millisecond timer mechanism described earlier. A one-second timer of duration D times out in the range of D-1 to D seconds, where "second" is defined as 1001 milliseconds using the millisecond timer duration formulas for L and H.

For example, at 625 kHz, each "second" is 991.23 milliseconds. Thus a 10-second timer would time out in the range of 8.74 to 10.09 seconds.

For repeating one-second timers, the first timeout occurs in the range of D-1 to D seconds. Subsequent timeouts occur every D seconds. The fifth timeout of a repeating 10-second timer would occur in the range of 48.39 to 49.74 seconds.

Delay Functions

Three functions allow an application to perform timing directly by suspending execution for a given time. These functions provide a concise way to perform timing in-line:

> **delay( )**
>
> **msec_delay( )**
>
> **scaled_delay( )**

The **delay( )** function produces a delay of fixed duration that is independent of input clock speed. This function can be used with the **wink** feature and for I/O debouncing. Its prototype is the following:

> **void delay (unsigned long** *count***);**

count   A value between 1 and 33,333. See the *Neuron C Reference Guide* for the formula used in determining the duration of the delay. Values in the range 33,334 .. 65,535 can be specified, but cause a watchdog timer reset.

Example:

```
when (io_changes(io_switch))
{
   delay(400);  // wait 10msec for debounce
   .
   .
   .
}
```

The **msec_delay( )** function produces a delay of a fixed number of milliseconds independent of the input clock speed. This function can be used with to delay for a more precise, and shorter, period of time than the **delay( )** and **scaled_delay( )** functions. Its prototype is shown below:

void msec_delay (unsigned short *milliseconds***);**

milliseconds   A number of milliseconds to delay (max of 255 ms).

The **scaled_delay( )** function produces a delay with a duration that scales with input clock speed. Its syntax is:

void scaled_delay (unsigned long *count***);**

count   A value between 1 and 33,333. See the *Neuron C Reference Guide* for the formula used in determining the duration of the delay.

EEPROM Write Timer

The accuracy of the configurable EEPROM write timer degrades with the speed of the input clock. To determine the accuracy of an *n* millisecond timeout, use the formula:

duration = *n* * delay(43)

For example, for a Series 3100 device at 625 kHz, a 20 millisecond EEPROM write actually takes 55.2 milliseconds.

3

How Devices Communicate Using Network Variables

This chapter discusses how LONWORKS devices communicate with each other using network variables. It includes a detailed discussion of how to declare network variables and how network variables on different devices are connected to each other. The use of synchronous network variables, the process of polling network variables, authenticated network variables, and network variables that implement a changeable type are also described.

Major Topics

LONWORKS devices communicate with other LONWORKS devices through network variables or application messages. This chapter focuses on network variables, which provide an open interoperable interface, simplify programming and installation, and also reduce program memory requirements. Most Neuron C programs use network variables. Application messages can be used, if required, as described in Chapter 6, *How Devices Communicate Using Application Messages*, on page 123. Although this manual discusses the two methods separately, a single Neuron C program can use both network variables and application messages.

This chapter is divided into the following parts:

- *Overview* on page 50 summarizes the behavior of devices that are readers and writers of a network variable, as well as how network variables are declared. It also describes how network variables on different devices are connected to each other.

- *Declaring Network Variables* on page 53 describes the syntax for declaring network variables, along with related concepts.

- *Connecting Network Variables* on page 55 describes how network variable readers are connected to network variable writers. This process was described in general terms in Chapter 1, *Overview*.

- *Network Variable Events* on page 57 describes the following four scheduling events that are related to network variables: **nv_update_completes**, **nv_update_fails**, **nv_update_occurs**, and **nv_update_succeeds.**

- *Synchronous Network Variables* on page 59 describes the behavior of synchronous network variables.

- *Processing Completion Events for Network Variables* on page 61 describes the two modes of checking for completion events, and the guidelines for use of these different techniques within an application program.

- *Polling Network Variables* on page 62 describes how a reader device can poll the writer device for the latest value of a network variable.

- *Explicit Propagation of Network Variables* on page 66 describes how an application program may exercise explicit control over network variable propagation, instead of permitting the Neuron firmware scheduler to propagate network variable updates automatically.

- *Initial Value Updates for Input Network Variables* on page 68 describes when and how to manage initial values for input network variables.

- *Monitoring Network Variables* on page 71 describes special considerations for implementation of a monitoring device.

- *Authentication* on page 72 describes how to use authenticated network variables to increase network security. Authentication allows a reader to verify the identity of a writer that attempts to update the reader's value of the network variable. Authentication can also prevent unauthorized configuration of a device.

- *Changeable-Type Network Variables* on page 74 describes how to implement network variables that allow their type to be changed at installation time.

Setting Up a LonTalk/IP Network

LonTalk Derived IP Addresses

In order to configure a LonTalk/IP Network, you need to understand how LonTalk/IP uses IP addresses. Each LonTalk device is assigned a LonTalk address and an IP address. Each LonTalk device is assigned a LonTalk address and an IP address. Typically the LonTalk address is simply a compressed form of the device's IP address (or the device's IP address is an inflated from of its LonTalk address). Devices that derive their IP address by inflating their LonTalk address, or derive their LonTalk address by compressing their IP address are said to use "LonTalk Derived IP Addresses".

The first two bytes of a LonTalk derived IPV4 address are derived from the LonTalk domain ID. The third byte and fourth bytes are identical to the LonTalk subnet ID and node ID, respectively.

The first two bytes of the LonTalk derived IP address depend on the LonTalk domain ID length, as shown in the following table (in the table below, S and N represent the LonTalk Subnet ID and Node ID respectively):

Domain Length	Domain ID	IP Address
0		192.168.S.N
1	D1	10.D1.S.N
3	D1D200	D1.D2.S.N
6	Unsupported	Unsupported

Note that certain LonTalk domain ID are invalid for use with LonTalk/IP:

- 6 Byte domain IDs are not supported.
- The 3rd byte of a 3 Byte domain ID must 0.
- The first byte of a 3 byte domain ID must NOT be 10 (0x0A).
- The first two bytes of 3 byte domain ID must NOT be 192,168 (0xC0, 0xA8).

Other domain IDs may not be valid in a particular IP infrastructure as they may conflict with existing IP addresses. Note that the LonTalk subnet 255 is not valid for use with LonTalk/IP networks because the derived LonTalk/IP address would not be a valid unicast address.

Likewise, not all valid unicast IP addresses can be used to derive a LonTalk address. Specifically neither the subnet ID or node ID can be 0, and the node ID must be less than 128.

LonTalk Multicast Addressing

LonTalk devices support three forms of multicast addressing: Domain broadcast, subnet broadcast and group addressing. LonTalk/IP uses IPV4 multicast to implement both domain and subnet broadcast as well as group addressing, as shown in the table below:

LonTalk Address Type	IP Address
Domain Broadcast	239.192.0.0
Subnet Broadcast to subnet S	239.192.0.S
Group addressing to group G	238.192.1.G

LonTalk to Host IP Address Mapping

Some devices support mapping their LonTalk address to a related host IP address. This can be useful on devices that support multiple (virtual) Lontalk devices on a single physical device.

Note that series 6000 devices do not NOT support LonTalk/Host IP address mapping. Each series 6000 device uses a LonTalk derived IP address. However, they can talk to devices that use LonTalk/Host IP mapped addresses provided that they are on the far side of an IzoT router. This is because the IzoT router can perform the LonTalk/Host IP mapping function on behalf of the series 6000 devices.

Routing Considerations

A LonTalk network might consist of a variety of components, including:

- A PC running an OpenLNS Server;

- One or more IzoT routers connecting FT channels to a native IP link;

- One or more Neuron based devices on an FT link;

- One o more EX or Device DX stacks; and

- Non-LonTalk aware IP gateways.

For example, consider the following diagram of a LonTalk/IP network consisting of an ethernet backbone and two FT channels, using LonTalk domain ID length 0. All of the devices use an IP address with the 192.168 prefix. Devices on the ethernet link use a subnet ID of 1, while devices on the two FT channels use subnet 2 and 3.

LNS PC
192.16.1.127

Device Stack
EX
192.168.1.3

IP Gate Way
Route 192.168.2/8 tp 192.168.1.1
Route 192.168.3/8 to 192.168.1.2

Ethernet

IzoT Router
192.168.1.1

IzoT Router
192.168.1.2

FT

FT

FT 6050
192.168.2.1

FT 6050
192.168.2.2

FT 6050
192.168.3.1

FT 6050
192.168.3.2
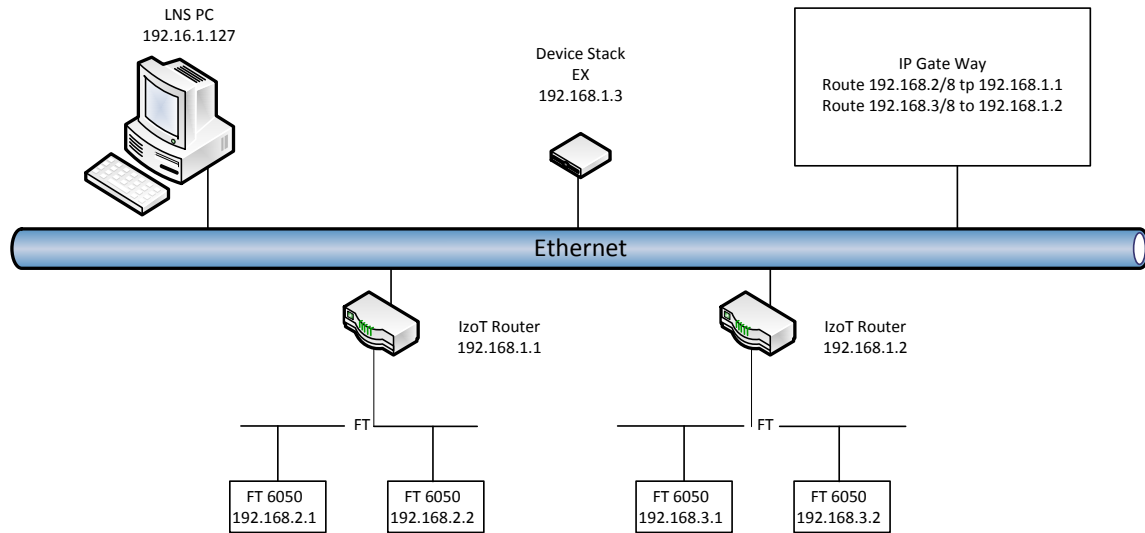
In order for all of these components to be able to properly communicate:

1. IP routes must established. This is done by configuring the IP gateways. In this example, the following routes need to be setup to ensure that IP packets from the two FT channels are first routed to the appropriate IzoT router:

   Route 192.168.2.0 MASK 255.255.255.0 to 192.1.1.1
   Route 192.168.3.0 MASK 255.255.255.0 to 192.1.1.2

2. If the LonTalk/IP network spans multiple IP gateways they must be configured to forward LonTalk/IP multicast groups. The simplest setup is to route all multicast groups with the 3 byte prefix 239.192.0 and 239.192.1. This can be optimized to only route specific groups using the lontalk broadcast prefix 239.192.0, based on which subnets subnets are defined on each side of the IP gateway. For example, the IP gateway shown in the picture above must forward 239.192.0.1, 239.192.0.2, and 239.192.0.3 onto the Ethernet link shown, but does not need to forward these groups on the other side of the gateway. All IP gateways that are part of LonTalk IP network must route the IP multicast address used for domain wide broadcast (239.192.0.0).

   Note that in the illustration above all devices use LonTalk derived IP addressing. However, some components, such as the LNS PC might use LonTalk/Host IP mapped addressing instead. In that case it is presumed that the LNS PC's IP address is already a known routable address.

It is also important to ensure that LonTalk derived IP addresses do not conflict with IP addresses of other IP infrastructure components. This can be avoided, for example, by choosing a LonTalk domain ID on a different IP subnet than other IP components. However, using a unique IP subnet may require configuration of your IP that LonTalk/IP packets are correctly routed. If your IP infrastructure includes multiple independently managed IzoT networks, you must ensure that each of them uses a different LonTalk domain ID.

For example, suppose you are adding your LonTalk/IP network to an existing IP network that uses IP subnets 10.3.12/8 and 10.3.15/8, connected with a single IP gateway. Furthur suppose your LonTalk/IP network has devices on both sides of the IP gateway, and you have chosen domain ID 8 to avoid any conflicts with existing IP addresses. All devices in your LonTalk/IP network that use LonTalk derived IP addresses will have IP addresses of the form 10.8.S.N. First, you must ensure that devices on the two sides of the IP gateway are assigned different LonTalk subnet IDs. This can be done in LNS by defining these devices on different channels and creating a phantom router (that need not be commissioned). Such a network is illustrated below:
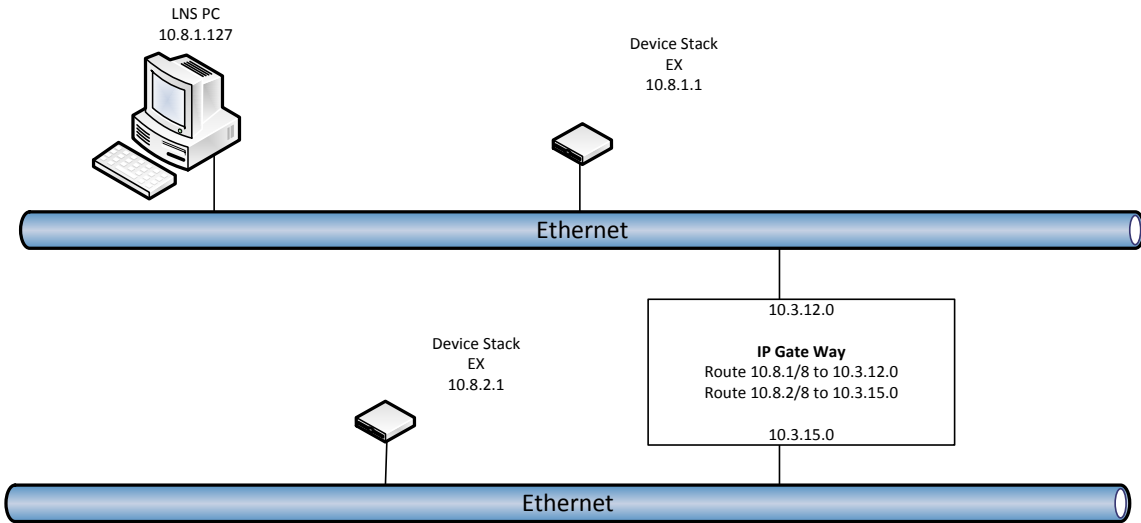
In this diagram the gateway has IP address 10.3.12.0 on one side and 10.3.15.0 on the other side. In order to ensure that LonTalk/IP packets will be correctly routed will need to make sure that the gateway contains the following routes:

    Route 10.8.1.0 MASK 255.255.255.0 to 10.3.12.0
    Route 10.8.2.0 MASK 255.255.255.0 to 10.3.15.0

Network Variable Overview

As described in Chapter 1, *Overview*, a network variable is an object that represents a data value and may be connected to multiple devices on a network. How many network variables a Neuron-hosted device can support depends on the device's memory map, the system firmware, and the development tool version, as shown in **Table 6**.

Table 6. Network Variable Limits

Neuron System Firmware Version	IzoT NodeBuilder Software Version	Maximum Number of Network Variables
Version 15 or earlier	Any	62

How Devices Communicate Using Network Variables

Neuron System Firmware Version	IzoT NodeBuilder Software Version	Maximum Number of Network Variables
Version 16 or later	3.1 or earlier	62
	FX or later	254

Important: The NodeBuilder CodeWizard declares network variables so that they are placed in RAMNEAR memory by default. However, because of the larger number of network variables that are available for devices built with the IzoT NodeBuilder Software, your device could run out of available RAMNEAR memory. In this case, declare as many network variables as necessary to use RAMFAR memory:

1. Within the NodeBuilder CodeWizard, right-click the network variable and select **Properties** to open the NV Properties dialog.

2. Within the NV Properties dialog, click **Advanced** to open the Advanced NV Properties dialog.

3. Within the Advanced NV Properties dialog, select **far**. Click **OK** to close the dialog.

4. Within the NV Properties dialog, click **OK** to close the dialog.

Network variables are defined within the program that runs on an individual Neuron Chip or Smart Transceiver. As an example, consider a lamp program with one network variable, named **nviLamp** (see **Figure 6** on page 52). Also, consider a switch program with one network variable, named **nvoSwitch**. The same lamp program is installed on each of the three lamp devices, and the same switch program is installed on each of the two switch devices in the figure.
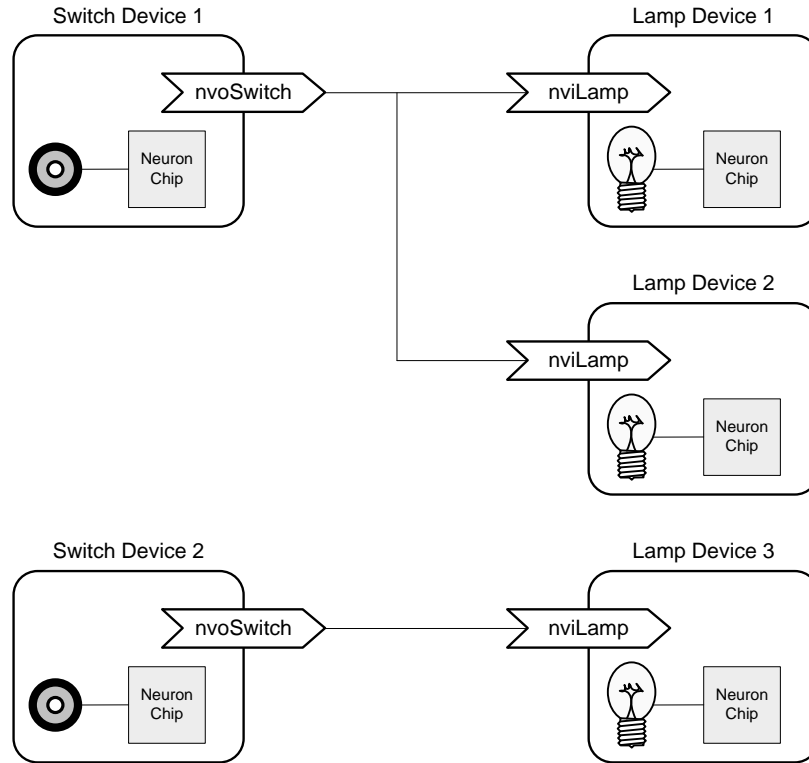
Figure 6. Sample Development Network with Five Devices

The declarations for these two network variables, which appear in different programs, are the following:

```
network output SNVT_switch nvoSwitch;
```

and

```
network input SNVT_switch nviLamp;
```

Behavior of Writer and Reader Devices

A writer device can change the value of a network variable. The connected network variables in all reader devices are then updated to reflect this change. In general, a reader device only reads from its copy of the network variable. One exception is that a reader device can provide an initial value to the network variable when the variable is declared. Another exception is that a reader device can modify its local copy of a network variable in its program. However, in neither case is the new value propagated to any other devices.

A writer device can also read from its last copy of the network variable, but it only sees the value it wrote last. In other words, two writers of the same network variable cannot change each other's value.

When a writer device writes a value to an output network variable, the Neuron firmware causes a LONWORKS message to be sent to all readers of the variable, informing them of the new value. By default, the message is sent using the acknowledged (ACKD) service. Not all readers may receive updates simultaneously. The network application must be designed to handle update failures and delays.

How Devices Communicate Using Network Variables

Note: This discussion uses the terms *writer device* and *reader device*. A writer device is a device that writes to a particular network variable (an output network variable). A reader device is a device that reads a particular network variable (an input network variable). In many cases, a device has both input and output network variables declared in its program, and therefore acts both as a "writer device" and a "reader device," depending on the network variable.

When Updates Occur

The new value of a network variable received by a reader device does not take effect immediately upon reception and processing of the message. Similarly, assignment of a new value to an output network variable does not cause a message to be sent immediately. Rather, updates occur at the end of a critical section in the application program. A *critical section* is defined as a set of application program statements during which network variable updates are not propagated.

A task is an example of a critical section: once begun, each task runs to completion. When network variable updates are received or requested, they are posted by the scheduler at the end of each critical section. An application can use the **post_events( )** function to divide a single task into two or more critical sections. The **post_events( )** function can be used to increase throughput and improve response time since it forms a boundary at which outgoing network variable updates are sent and incoming network variable updates are processed. See Chapter 7, *Additional Features*, on page 157, for further discussion of **post_events( )**.

Declaring Network Variables

The syntax for declaring a network variable is shown below. The first form of the declaration is for a simple network variable, and the second form is for a network variable array.

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

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

Note: The brackets around the term *array-bound* do not, in this case, indicate an optional field. They are required and must be part of the program.

A Neuron-hosted device can declare up to 254 network variables (including array elements), but the device's resource capacity (as defined by its memory map) might not support the maximum number of network variables.

You can declare an array of network variables using the second form of the syntax shown above. The array can only be single dimension. The *array-bound* must be a constant. Each element of the array is treated as a separate network variable for purposes of events, transmissions on the network, and so on. Therefore, each element counts individually towards the maximum number of network variables on a given device. Each element of the array is then a *separately bindable* network variable.

After the device design is complete, you specify connections between network variable outputs and inputs on different devices. This is discussed in *Connecting Network Variables* on page 55. The specification of the desired connections is used by a network tool to generate the appropriate network addresses. When these addresses are downloaded into the devices, they ensure that updates sent by writers reach all of the intended readers.

In the lamp and switch example shown in **Figure 6** on page 52, the output network variables in column 1 are connected to the input network variables in column 2, as listed in **Table 7**.

Table 7. Connecting Output Network Variables

Output (device/variable_name)	Input (device/variable_name)
switch1/nvoSwitch	lamp1/nviLamp lamp2/nviLamp
switch2/nvoSwitch	lamp3/ nviLamp

When a network variable that is a structure is modified by a network variable writer, the entire structure is updated at the next critical section boundary for all network variable readers, regardless of whether the structure was wholly or partially modified.

Network variables can be declared with a single dimension array bound. Each element of the array is then a separately bindable network variable. See the descriptions of the **poll( )** function, the built-in **nv_array_index** variable**,** and the **nv_update_completes**, **nv_update_fails**, **nv_update_occurs**, and **nv_update_succeeds** events in the *Neuron C Reference Guide* for more information.

When an element of a network variable that is an array is modified by a network variable writer, only the modified element is updated at the next critical section.

The maximum size of a network variable is 228 bytes for Series 6000 chips and 31 bytes for other Neuron Chips and Smart Transceivers. Other platforms such as Shortstack generally support 228-byte network variables. In the case of a network variable array, each element is limited to a size of 228 or 31 bytes, respectively.

The Neuron C compiler includes the SNVT indices (numerical identifiers for the standard network variable types) in the application image for all network variables declared as SNVTs, and optionally also includes the network variable names for all network variables. Network variable names are always included in the device interface file for a device, but integrators might find them useful when they lose the device interface file and need to install your device. You can control these options using the following compiler directives as described in the *Compiler Directives* chapter of the *Neuron C Reference Guide*:

#pragma disable_snvt_si

#pragma enable_sd_nv_names

See the *Neuron C Reference Guide* for additional information about the syntax for declaring a network variable.

Examples of Network Variable Declarations

Some sample network variable declarations are the following:

```
network input SNVT_temp nviTemp;
network output SNVT_switch nvoHeater;
network output int nvoCurrentTemp;
```

Examples of priority network variable declarations are shown below:

```
network output SNVT_alarm bind_info(priority)
       nvoFireAlarm;
network input boolean bind_info (priority(nonconfig))
       nviFireAlarm;
```

An example of declaring a network variable using the unacknowledged service is the following:

```
network output SNVT_lev_cont bind_info(unackd)
       nvoFillLevel;
```

The unacknowledged service can be used for this network variable because we can assume that the control dial generates numerous messages as it is being turned, and you probably don't need or want to receive an acknowledgment for each one. In addition, it is probably not critical to this application if a single message out of several is not received.

Connecting Network Variables

Network variable *connections* are independent of the Neuron C application on a device. Network variable connections are created by a portion of a network tool called the *binder*. The binder can be part of the LonMaker Integration Tool or another network tool.

The binder assigns addresses and transport properties, such as timer values, to all appropriate devices to ensure that information flows to and from the right places.

Example: Binding output network variables to input network variables creates a "closed-loop system", also sometimes called a "servo system". By connecting the network variables of two or more devices, you allow the system to use feedback or error-correction signals from one device to set or maintain the control mechanical position or other parameters of the other device. **Figure 7** on page 56 shows a simple closed-loop system for a stairwell lighting system.

In the figure, the dashed line represents the "forward" connection that drives the lamp from both switches. The lamp reports its current on/off state and dimming level through the solid line that represents the feedback connection to the switches, which allows all components of the stairwell lighting system to be aware of current lighting conditions.
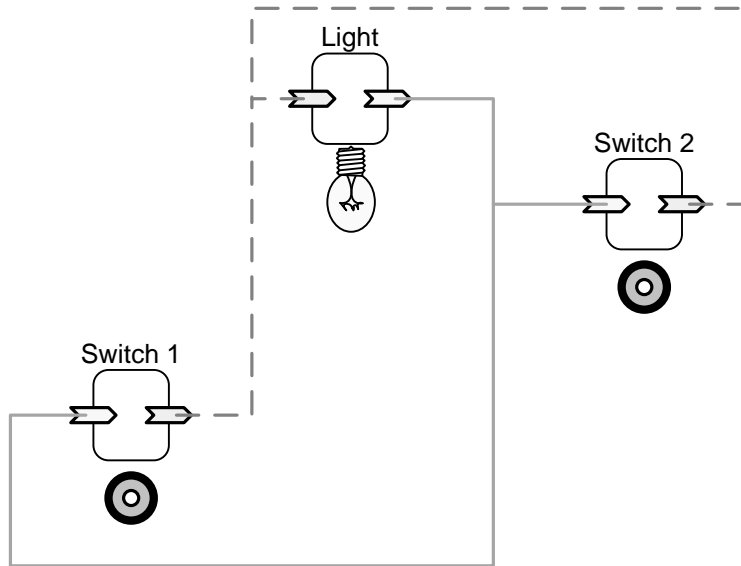
Figure 7. A Simple Closed-Loop System

Use of the is_bound( ) Function

A typical application need not know where local output network variables are connected, and need not know the source of updates to local input network variables. While such detail is available where necessary, typical applications can focus on the semantics of the input network variable update and the local control algorithm.

When necessary, a Neuron C application can determine if a network variable has been connected by a network tool by calling the **is_bound( )** function. For example, a device might expect periodic updates to a local input network variable under normal circumstances, and might enter a failure mode if such updates fail to arrive for a prolonged amount of time. Such an application could then use the **io_bound( )** function as a predicate to disable or enable certain aspects of the application algorithm.

Whenever an unconnected output network variable is updated, an **nv_update_succeeds** event becomes TRUE even though no update actually occurred (see also *Processing Completion Events for Network Variables* on page 61).

The **is_bound( )** function only indicates whether the network variable is bound or unbound. Another device, such as a network tool for monitoring and control, may still attempt to obtain the current value of an unbound output network variable by polling, or may still update an unbound input network variable by setting its value without a bound connection. Thus, reducing the application's processing requirements by conditional processing based on the **is_bound( )** function should be limited to those devices that cannot operate without a bound connection, such as devices implementing a closed-loop system.

Network Variable Events

Chapter 2, *Focusing on a Single Device*, on page 17, introduced the event scheduling mechanism and discussed a number of predefined events. Four predefined events are specifically related to network variables:

> **nv_update_completes** [(*network-var-reference*)]
> **nv_update_fails** [(*network-var-reference*)]
> **nv_update_occurs** [(*network-var-reference*)]
> **nv_update_succeeds** [(*network-var-reference*)]

The **nv_update_occurs** event applies only to input network variables. The other three events (**nv_update_completes**, **nv_update_fails**, **nv_update_succeeds**) apply to output network variables when they are updated, and to input network variables when they are polled.

The event expression may be qualified with a *network-var-reference*, which can be a network variable name, a network variable array element (as in *network-var*[*index*]), a network variable array name, or a range of network variables. If the event is qualified by an array name, the event occurs once for each element for which the event is applicable.

The form of the event that permits a range of network variables has the syntax shown below. The range consists of two network variable or network variable element references, separated by two consecutive dot characters "." indicating the range. This syntax applies to all four *event-names* shown above. Each network variable is assigned a global index by the compiler. An array of network variables is assigned consecutive indices, one for each element. The range event applies to all network variables whose global indices are between the global index for *nvA* and *nvB*, inclusive. The global index of *nvA* must be less than the global index of *nvB*.

> *event-name* [(*nvA* .. *nvB*)]

This section provides an introduction to these events. For convenience, we refer to them as network variable *completion events*, because they all pertain to whether or not a network variable update or poll has completed. Note that completion does not imply success. See also Chapter 6, *How Devices Communicate Using Application Messages*, on page 123, which includes more detailed information on using these completion events.

The nv_update_occurs Event

When a new value has been received for an input network variable, the **nv_update_occurs** event evaluates to TRUE. If a specific network variable is not used to qualify the event, it evaluates to TRUE for any network variable update on that device.

The **nv_update_occurs** event is used in many situations. For example, a lamp program could use this event as follows:

```
// Use the network variable's value
// as the new state for the lamp

network input SNVT_switch nviLampState;

when (nv_update_occurs(nviLampState)) {
```

```
        io_out(ioLED, nviLampState.state);
    }
```

In the following example, when a thermostat device receives a new temperature setpoint, it checks the current temperature and turns the heater on or off if necessary:

```
network input SNVT_temp nviSetpoint;
network output SNVT_switch nvoHeater;
network output SNVT_temp nvoCurrentTemp;
when (nv_update_occurs(nviSetpoint)) {
    nvoHeater.state = nvoCurrentTemp < nviSetpoint;
}
```

Most applications do not need to know the source of an input network variable update, and can focus on implementing behavior in response to receiving the updated network variable value.

However, the **nv_in_addr** built-in variable, described in the *Neuron C Reference Guide*, provides addressing information about the device that originated the update.

The nv_update_succeeds and nv_update_fails Events

When a network variable update or poll fails, the **nv_update_fails** event evaluates to TRUE. If no network variable is specified for the event, it evaluates to TRUE for any network variable update or poll that failed on that device. If multiple network variables are specified, the event can be TRUE once for each network variable update or poll that failed.

Similarly, the **nv_update_succeeds** event evaluates to TRUE whenever an output network variable update has been successfully sent or polled values have been received from all the writers.

You can use the **nv_update_fails** event for any output network variables. The following example illustrates using the **nv_update_fails** event with a single output network variable:

```
network output SNVT_switch nvoSwitch;

when (nv_update_fails(nvoSwitch))
{
    // take some corrective action
}
```

Here is an example of testing for network update failure and success:

```
boolean heater_failed;
network output SNVT_switch nvoHeater;
when (nv_update_fails(nvoHeater))
{
    heater_failed = TRUE;
          // remember update failure
}

when (nv_update_succeeds(nvoHeater))
{
```

```
        heater_failed = FALSE;
            // heater device received update
    }
```

The nv_update_completes Event

The **nv_update_completes** event evaluates to TRUE whenever an output
network variable update or poll either succeeds or fails. An example of testing
for network variable update completion is shown below:

```
#include <io_types.h>
#define C_TO_K 2740
IO_7 input ontime invert clock(2) io_temperature_sensor;
network output SNVT_temp nvoCurrentTemp;
when (nv_update_completes(nvoCurrentTemp))
{  // latest temperature has been sent out
    ontime_t sensor_value = io_in(io_temperature_sensor);

    // send another update     nvoCurrentTemp =
muldiv24(sensor_value, 221, 642)
                  + 211 + C_TO_K;
       // tenths of a degree,C
}
```

If a program checks for **nv_update_completes** or **nv_update_succeeds** for any
network variable, the program is said to use comprehensive completion event
testing. See *Comprehensive Completion Event Testing* on page 61 for the rules
you should follow.

The muldiv() and muldiv24() combine a multiplication and a division with a
higher resolution intermediate result, and are detailed in the *Neuron C Reference
Guide*.

Synchronous Network Variables

When an output network variable is updated, the Neuron firmware ensures that
the most recent value assigned to an output is propagated and received as an
event by any connected input network variables. Thus, if multiple updates are
made to an output network variable within the same critical section, only the last
value assigned is ensured to be propagated and received as an event at the input
network variables.

A critical section is defined by the boundaries of the executing when task. After
the when task completes, the most recent value is scheduled for propagation.

You can specify that *all* updates to an output network variable (within the
critical section) must be propagated and received as events by using the
synchronous subclass of network variables.

However, for a synchronous output network variable, each updated value is
immediately scheduled for propagation, which requires an application output
buffer for each scheduled update. If all output buffers are already in use when an
update to a synchronous output network variable is scheduled, the Neuron
firmware enters preemption mode to attempt to complete pending transactions
and thus free in-use application output buffers.

As a result, updating synchronous output network variables can cause your when task to execute in a non-deterministic fashion. See *Preemption Mode* on page 61 for more information about buffers and preemption mode.

Declaring Synchronous Network Variables

To declare a synchronous network variable, include a **synchronized** or **sync** keyword in its declaration. An example declaration is shown below:

```
network output sync SNVT_temp nvoRelativeTemp;
```

In the following example, the network variable is declared as synchronous so that all the updates are sent. (If more than one alarm goes off, we want to receive notice of all alarms, not just the most recent one.)

```
// ensure multiple alarms are handled serially
network output sync SNVT_alarm nvoAlarm;
```

Synchronous output network variables do not have to be connected to synchronous input network variables. All input network variables operate synchronously regardless of whether the synchronous attribute was assigned.

Synchronous vs. Nonsynchronous Network Variables

For most applications, nonsynchronous network variables are adequate and should be used when possible. Many applications need the most *recent* value, not all of the values, for a given network variable. Widespread use of synchronous network variables that are frequently updated could delay processing if the program frequently runs out of buffers (see *Preemption Mode* on page 61). Depending on the device buffering, channel speed, and congestion of the network, application performance could be adversely affected by extensive use of synchronous network variables.

If a program is required to use *relative* (or *delta*) data values, synchronous network variables might be necessary to preserve the intermediate data values. For programs using *absolute* data values, nonsynchronous network variables are usually sufficient.

A nonsynchronous output network variable goes out on the network when the next output buffer is available. If the program updates the variable again before that time, only the most recent value goes out. A synchronous output network variable causes the application to wait for an output buffer if none is available. In this case, the scheduler enters preemption mode (see *Preemption Mode* on page 61).

For input network variables, an incoming network variable update always results in an event for the application. All input network variables operate synchronously regardless of whether the synchronous attribute was assigned.

Updating Synchronous Network Variables

Synchronous network variables are always updated at the end of each critical section. If a buffer is not available, the scheduler waits for one. Nonsynchronous network variables, on the other hand, are updated at the end of critical sections when the scheduler has application buffers available to do so. Unlike synchronous network variables, they are not always updated at the end of the

next critical section. As already pointed out, where multiple updates occur, the intermediate values might never be propagated across the network.

Preemption Mode

The scheduler enters *preemption mode* when a synchronous output network variable update occurs and there is no application output buffer available. Because the system must send out the synchronous output network variable update, it processes completion events, incoming **msg_arrives** or **nv_update_occurs** events, and response events until an application output buffer becomes available.

Other events are not processed, unless the **when** clause for the event is preceded by the keyword **preempt_safe**. See Chapter 2, *Focusing on a Single Device*, on page 17, for the syntax of a **when** clause. See Chapter 6, *How Devices Communicate Using Application Messages*, on page 123, for a further discussion of preemption mode, and when to use the **preempt_safe** keyword.

A delay in application processing thus occurs when the system enters preemption mode. The length of the delay depends on how long it takes for an application output buffer to become free. This delay depends on network traffic, channel bit rate, and other factors.

Processing Completion Events for Network Variables

For network variables, there are two modes of checking for completion events: partial completion event testing, and comprehensive completion event testing. For message tags (see Chapter Chapter 6, *How Devices Communicate Using Application Messages*, on page 123), only comprehensive completion event testing is available.

Partial Completion Event Testing

If you choose to use partial completion event testing in your program, you then have two choices of how to process completion events for each network variable:

1   Do not check for any completion events.

2   Check for only the failure event (**nv_update_fails**).

For example, within a program containing two network variables:

- Network Variable 1:   Program checks for no completion events.

- Network Variable 2:   Program checks for failure only.

Comprehensive Completion Event Testing

Comprehensive completion event testing offers the same set of choices for network variable completion events that is available for processing message tag completion events (see Chapter 6, *How Devices Communicate Using Application Messages*, on page 123). If you choose to use comprehensive completion event testing in your program, you then have three choices of how to process completion events for each network variable:

1   Do not check for any completion events.

2   Check for the failure and the success events
(**nv_update_fails, nv_update_succeeds**).

3   Check for the update completion event (**nv_update_completes**).

For example, the following is an acceptable strategy within a program containing three network variables:

- Network Variable 1:   Program checks for no completion events.

- Network Variable 2:   Program checks for failure and success.

- Network Variable 3:   Program checks for update completion only.

Note: If you choose to use comprehensive completion event testing features (with network variables), all completion code processing for network variables must be comprehensive completion event testing. This restriction does *not* mean that events must be checked for all network variables; it only means that a single program can use *either* partial *or* comprehensive completion event testing, but cannot intermix both techniques. The Neuron C compiler detects use of the comprehensive event feature on a *per-program* basis.

Tradeoffs

Using comprehensive completion event testing for processing network variable completion events within a program requires more code space and is less efficient than using partial completion event testing. If you choose a comprehensive completion event testing feature, such as checking **nv_update_completes**, you are limited to comprehensive completion event testing features for whichever network variable's events in which you are interested. For example, within a program using comprehensive completion event testing, you cannot simply check for **nv_update_fails**, because that feature applies only to partial completion event testing.

Polling Network Variables

As described earlier in this chapter, a network variable update is normally initiated when a device assigns a value to an output network variable. Another device can request that the first device send its latest value for a network variable. The process of requesting current network variable values from a device is called *polling*.

A device's program can poll any input network variables at any time, including initial power-up and when transitioning from offline to online. Polling on initial power-up can cause network congestion if many devices are powered-up at the same time, and they all do power-up polling. See *Initial Value Updates for Input Network Variables* on page 68 for more information about polling during power-up and reset processing.

Polling an input network variable from your program requires the network binder to apply a different scheme when connecting output network variables between writer and reader devices, requiring additional address table entries to be used on the reader device. If you add polling to an existing application that did not previously use polling, you must create a new device interface (XIF) file

for the device, and import the new device interface file into any network tools that used the previous version.

The reader device makes its request through the **poll( )** function. The syntax is shown below:

poll ([*network-var*]**);**

network-var   is an input network variable identifier.

If no network variable is specified, all input network variables for the device are polled. For Neuron-hosted applications, an explicit **polled** declaration is not allowed for an input network variable; see *Declaring an Input Network Variable as Polled* on page 64.

The *network-var* identifier can also be a network variable array identifier, or an element of a network variable array, as in *network_var*[*index*]. If a network variable array name is used without an index, all elements of the array are polled.

The new value resulting from the poll is not immediately available after the **poll( )** function call. Use a qualified **nv_update_occurs** event in a **when** clause, or some other conditional statement, to obtain the new, polled value.

Example:

```
mtimer tDelayedPolling;

network input SNVT_switch nviCooling;

when (reset) {
   // set up timer for delayed power-up polling:
   tDelayedPolling = 4ul * random();  // >= 1 second
   ... // other reset processing
}


when (timer_expires(tDelayedPolling)) {
   poll(nviCooling);
   ...
}

when (nv_update_occurs(nviCooling)) {
   ...
}
```

Here is a lamp program that includes a poll of the input network variable **nviLampState** after a reset event. The device obtains the most recent value of **nviLampState,** and then uses that value after reset.

```
// LAMP.NC -- Sample lamp actuator program,
// polls the switch on reset

//////////////// Network Variables ///////////////////
network input SNVT_switch nviLampState;

///////////////////////// Constants ///////////////////
#define LED_ON      0
#define LED_OFF     1
```

```
////////////////////////// I/O Objects /////////////////////
IO_0 output bit ioLED = LED_OFF;

mtimer tDelayedPolling;

////////////////////////// Tasks /////////////////////////
// NV update task -- handle update to lamp state
// Use the network variable's value as the new state
// for the lamp
when (nv_update_occurs(nviLampState)) {
   io_out(ioLED,
      nviLampState.value && nviLampState.state
         ? LED_ON : LED_OFF);
}

when (reset) {
   // set up timer for delayed power-up polling:
   tDelayedPolling = 4ul * random();   // >= 1 second
   ... // other reset processing
}


when (timer_expires(tDelayedPolling)) {
   poll(nviLampState);
}
```

Declaring an Input Network Variable as Polled

For Neuron-hosted applications, you cannot declare in input network variable as **polled**; the Neuron C compiler issues an error message for polled input network variables.

However, within a model file for a host-based application (such as for a ShortStack, FTXL, or *i*.LON SmartServer application), you can declare an input network variable as polled:

> **network input polled** *type netvar***;**

For host-based applications, the application code is written in a language other than Neuron C, and therefore the presence of the **poll( )** function within the application would not cause the compiler to adjust the device's self-identification data to allow the network tool to create the network variable binding appropriately. Thus, within a model file, the declaration of the input network variable is allowed to be declared as **polled** so that the ShortStack Micro Server, FTXL LonTalk protocol stack, or *i*.LON SmartServer can allow the network variable to be polled.

Declaring an Output Network Variable as Polled

To allow devices to receive network variable updates from other devices only at specified times, declare the output network variable as **polled**:

> **network output polled** *type netvar***;**

In this special case, the output network variable's value is never propagated as a result of its value changing. Instead, the output network variable's value is sent

only in response to a poll request from a reader device, or if the **propagate( )** function is called for that network variable.

Example:

A lamp and switch example could also be written to use explicit polling of the switch network variable. Complete programs illustrating polling are shown below.

Listing 1. Lamp Program Using Polling

```
// LAMP.NC -- Sample lamp actuator program,
// polls the switch periodically

//////////////////// Network Variables ///////////////
network input SNVT_switch nviLampState;

/////////////////////// Constants ////////////////////
#define LED_ON      0
#define LED_OFF     1

/////////////////////// I/O Objects ///////////////////
IO_0 output bit ioLED = LED_OFF;

/////////////////////// Timers ////////////////////////
mtimer tmPoll;

///////////////////////// Tasks //////////////////////
//   NV update task -- handle update to lamp state
//   Use the network variable's value as the new
// state for the lamp
when (nv_update_occurs(nviLampState)) {
    io_out(ioLED,
        nviLampState.value && nviLampState.state
            ? LED_ON : LED_OFF);
    tmPoll = 500;  // Wait 500 msec before polling again
}

///////////////////////////////////////////////////////
// Reset and timer task
// request last value from any switch attached
when (reset) {
    tmPoll = 4ul * random();  // >= 1 second
}

when (timer_expires(tmPoll) ) {
    poll(nviLampState);
}
```

Listing 2. Switch Program Using Polling

```
// SWITCH.NC -- Sample switch sensor program
// Only transmits switch state when polled by the lamp

//////////////////// Network Variables  ///////////////
network output polled SNVT_switch nvoSwitchState;

/////////////////////// Constants  ///////////////////
#define BUTTON_DOWN 0
```

```
#define BUTTON_UP   1

///////////////////// I/O Objects ///////////////////
IO_4 input bit ioButton = BUTTON_UP;

/////////////////////////// Tasks ////////////////////
// I/O task -- handle pushbutton down event
// Just toggle the network variable (nvoSwitchState).
// In this case, no message is sent until a poll request
// is received from a reader device
when (io_changes(ioButton) to BUTTON_DOWN)
{
    // button pressed, so toggle state
    nvoSwitchState.state = !nvoSwitchState.state;
    nvoSwitchState.value = nvoSwitchState.state ? 200 : 0;
}

when (reset) {
    io_change_init(ioButton);
    ... // other reset processing
}
```

Explicit Propagation of Network Variables

As described earlier in this chapter, a network variable update is normally initiated when a device assigns a value to an output network variable. In this case, the network variable update is initiated implicitly by code that is generated by the compiler to handle the variable's modification.

An application can also explicitly request propagation for the output network variable. Such explicit propagation is commonly used in the implementation of "heartbeats," regularly scheduled repeated propagation of the most recent value, as supported by many interoperable devices. This technique could also be useful in situations where the variable is not directly modifiable, or it might also result from using pointers to network variables.

A device's program can propagate any output network variables at any time, including during initial power up and when transitioning from offline to online. Network variable propagation on initial power-up can cause network congestion if many devices are powered-up at the same time, and they all do power-up propagation.

The application explicitly propagates an output network variable with the **propagate( )** function. The syntax is shown below:

> **propagate (**[*network-var*]**);**

network-var   Specifies an output network variable identifier.

If no network variable is specified, all output network variables for the device are propagated. The *network-var* identifier can also be a network variable array identifier, or an element of a network variable array, as in *nv_array*[*index*]. If a network variable array name is used without an array index, all elements of the array are propagated.

The **propagate( )** function can be used to send the value of an output network variable that is declared **const**, and thus cannot be assigned to. Because a value assignment triggers implicit network variable propagation, and because a **const**

variable cannot be assigned to, an explicit mechanism for propagation is required. See the **propagate( )** function in the *Neuron C Reference Guide* for additional information.

Example:

```
network output SNVT_temp nvoTemp;

when (timer_expires(heartbeat))
{
    propagate(nvoTemp);
}
```

The **propagate( )** function can also be useful where pointers are used to update output network variables. For example, assume that some function, **f( )**, calculates a complicated set of values and places them in a network variable structure. Assume the function is designed to operate on several similar such variables within a device, thus the function is passed a pointer to each variable.

For efficiency, it might be best to code this function to operate on the variables through a pointer reference. However, the Neuron C compiler cannot distinguish between a pointer to a regular application variable, and a pointer to a network variable. Thus, updates to a network variable through a pointer do not trigger an implicit propagation, and an explicit propagation is required.

Furthermore, because of the inability to distinguish pointers to network variables, Neuron C treats pointers to network variables as pointers to **const** data, thus avoiding the problem of a modification to the variable through the pointer. In Neuron C, removal of the **const** attribute is not normally permitted. However, the **#pragma relaxed_casting_on** directive directs the compiler to permit this cast. Casting can either be explicit, or implicit by variable assignment or function parameter passing.

Example:

```
network output SNVT_temp_f nvoTemp;

//
// the hypothetical function f() could be supplied with a
// binary (pre-compiled) function library for easy re-use,
// of for protection of intellectual property (function
// f's algorithm).
//
extern void f(SNVT_temp_f* pNv);

when (some-event)
{
#pragma relaxed_casting_on
    // Without pragma above, this would result in
    // an error, because the address of a network
    // variable is treated as 'const <type> *'.
    // Passing such a type as the function parameter
    // results in an implicit cast, since the function
    // prototype defines the variable as '<type> *'.

    f((SNVT_temp_f*)&nvoTemp);
    propagate(nvoTemp);  // Explicit propagation needed
            // because f() modified nvoTemp by pointer.
}
```

Initial Value Updates for Input Network Variables

Many applications react not only to physical inputs received through I/O models and peripheral hardware, but also respond to input network variable values. For those devices, power-up (and post-reset) behavior must be carefully considered.

Example: A simple room temperature controller might accept a temperature sensor's current reading through an input network variable, and might compute and output a control value that drives the valve:

```
network input  SNVT_temp nviCurrent;
network output SNVT_volt nvoValve;

when(nv_update_occurs(nviCurrent)) {
    nvoValve = control_algorithm(nviCurrent);
}
```

When this device powers up, it simply waits for the current temperature input value to change. After a new value is received, the device updates the **nvoValveDriver** output network variable, and the valve moves to a new position.

For many devices, this model is sufficient. The temperature sensor's reading changes over time, and most sensors also implement periodic heartbeats at a configurable interval. When the heartbeat interval expires, or the actual temperature reading changes over the the threshold of a hysteresis, the sensor updates its output network variable, the room temperature controller takes appropriate action, and the heating valve is adjusted accordingly. The devices in this network respond to external events and act as soon as necessary.

Some devices, however, need to produce up-to-date control values sooner after power-up or reset, or need to ensure a consistent and up-to-date set of input network variables. Consider, for example, an enhanced room temperature controller. This new controller also accepts a temperature setpoint value through an input network variable.

Example:

```
network input  SNVT_temp nviCurrent;
network input  SNVT_temp nviSetpoint;
network output SNVT_volt nvoValve;

when(nv_update_occurs(nviSetpoint))
when(nv_update_occurs(nviCurrent)) {
    nvoValve = control_algorithm(nviCurrent, nviSetpoint);
}
```

However, this example is flawed, because it cannot successfully compute a new control value unless both input network variables have been updated. There are several ways to solve this problem:

- Track the last-known good value using a suitable type of non-volatile memory or a battery back-up device design.

  Most low-cost non-volatile memory parts, such as EEPROM and flash memory, have a limited number of write cycles. Therefore, this approach only works with infrequently updated input network variables, and also assumes that the actual setpoint and current temperature do not change

much compared to that last-known good value.

Example:

```
network input  SNVT_temp nviCurrent;
network input  SNVT_temp nviSetpoint;
network output SNVT_volt nvoValve;

eeprom uninit SNVT_temp lastGoodCurrent, lastGoodSetpoint;

when(nv_update_occurs(nviCurrent)) {
    lastGoodCurrent = nviCurrent;
    nvoValve = algorithm(lastGoodCurrent, lastGoodSetpoint);
}

when(nv_update_occurs(nviSetpoint)) {
    lastGoodSetpoint = nviSetpoint;
    nvoValve = algorithm(lastGoodCurrent, lastGoodSetpoint);
}
```

• After power-up or reset, track which network variable has been updated, and execute the algorithm only after all required input data has arrived.

Example:

```
#define CURRENT_OK 0x01
#define SETPOINT_OK 0x02
#define ALL_OK (CURRENT_OK | SETPOINT_OK)

unsigned received;

network input  SNVT_temp nviCurrent;
network input  SNVT_temp nviSetpoint;
network output SNVT_volt nvoValve;

when(nv_update_occurs(nviCurrent)) {
    received |= CURRENT_OK;
    if (received & ALL_OK == ALL_OK) {
        nvoValve = algorithm(lastGoodCurrent, lastGoodSetpoint);
    }
}

when(nv_update_occurs(nviSetpoint)) {
    received |= SETPOINT_OK;
    if (received & ALL_OK == ALL_OK) {
        nvoValve = algorithm(lastGoodCurrent, lastGoodSetpoint);
    }
}
```

After power-up, poll both input network variables, thus explicitly requesting that the data sources resend their most recent value. This solution is popular, simple, and straight-forward.

Example:

```
network input   SNVT_temp nviCurrent;
network input   SNVT_temp nviSetpoint;
network output SNVT_volt nvoValve;

when(reset) {
```

```
    poll(nviCurrent);
    poll(nviSetpoint);
}
when(nv_update_occurs(nviSetpoint))
when(nv_update_occurs(nviCurrent)) {
    nvoValve = control_algorithm(nviCurrent, nviSetpoint);
}
```

While this straight-forward solution is adequate for many devices, it is not without problems:

- The multi-NV polling approach still needs to ensure that all input network variables have been updated prior to computing a new output value. The last example also needs to track arrival of all input network variable updates, as demonstrated in the previous example.

- Polling network variables immediately after reset (or power-up) can lead to network congestion for a site-wide power-up, and can cause the network stabilization to take some time after power has been applied.

While reset-time polling is not a problem if only a small number of devices issue only a small number of poll requests, a single device (or its developer) does not normally know the nature and application algorithm of the other devices in the network.

When using power-up polls, combined with a suitable technique to monitor input network variable updates, it is strongly recommended that you insert a random delay of a few seconds between the reset and the initiation of these poll requests. This delay helps to spread the power-up peak traffic demand in the network, and improves overall startup performance.

Example:

```
network input  SNVT_temp nviCurrent;
network input  SNVT_temp nviSetpoint;
network output SNVT_volt nvoValve;

mtimer powerupDelay;

when(reset) {
    powerupDelay = 16ul * random() + 500ul;
    // 0.500 to 4.5s delay
}

when(timer_expires(powerupDelay)) {
    poll(nviCurrent);
    poll(nviSetpoint);
}
when(nv_update_occurs(nviSetpoint))
when(nv_update_occurs(nviCurrent)) {
    nvoValve = control_algorithm(nviCurrent, nviSetpoint);
}
```

The best option, where appropriate, is to use poll-free techniques. Tracking of incoming updates, keeping records of last known-good-values or use of heartbeating sensors are all good tools to solve the problem of initial network variable updates for many devices.

Monitoring Network Variables

A monitoring device is a LONWORKS device that receives data from many other devices. For example, an alarm display device may monitor many alarm sensor devices. The sensor devices could all have a network variable output declared as a **SNVT_alarm** output, and the monitor device could have a network variable input, declared as a **SNVT_alarm** input.

Typically, the monitor device waits for a change to its input network variable. When a change occurs, it must identify which device originated the change. The method of determining the source of a change depends on the method used to connect the sensor outputs to the monitor input.

Following are a few options for the network monitor device; in the examples, the sensor devices all have a single **SNVT_alarm** output network variable that must be monitored by the network monitor device:

- Declare the network variable input as an array, and connect each element of the array to a different sensor. Wait for an **nv_update_occurs** event for the entire array, and then use the **nv_array_index** built-in variable to determine which device originated the change.

  Example:
  ```
  network input SNVT_alarm nviAlarmArray[50];
  SNVT_alarm alarm_value;
  unsigned int  alarm_device;

  when (nv_update_occurs(nviAlarmArray))
  {
      alarm_device =  nv_array_index;
      alarm_value = nviAlarmArray[alarm_device];

      // Process alarm_device and alarm_value
  }
  ```

  This method is appropriate when the number of devices to be monitored does not exceed the network variable limits of the monitoring device.

- Declare the network variable input as a single input on the monitor device, and declare the network variable outputs as polled outputs on the sensor devices. Create a single connection with all the sensor outputs and the monitor input. Explicitly poll each of the sensors using explicit addressing and explicit messages as described in the next chapter. Because the devices are explicitly polled, the monitor device always knows the source of a network variable update.

  This method is appropriate for any number of devices, as long as the delays introduced by the polling loop are acceptable for the application.

- Declare the network variable input as a single input and create a single connection with all the sensor outputs sending their values to the single monitor input. This configuration is called a *fan-in connection*. Wait for an **nv_update_occurs** event for the network variable input, and then use the **nv_in_addr** built-in variable to determine the source address of the device that originated the change. Implement a configuration

property array that is set by the device plug-in to identify the fanned-in devices. Following is an example for the code on a network monitor device:

Example:
```
network input SNVT_alarm nviAlarm;
SNVT_alarm alarm_value;
nv_in_addr_t alarm_device_addr;

when (nv_update_occurs(nviAlarm)) {
    alarm_device_addr = nv_in_addr;
    alarm_value = nviAlarm;
    // Process alarm_device_addr and alarm_value
    // Look up alarm_device_addr in a configuration
    // property set by a plug-in at installation time
}
```

This method is appropriate for any number of devices.

The *Neuron C Reference Guide* describes the contents of the **nv_in_addr** built-in variable.

Authentication

Authentication is a special form of an acknowledged service between one writer device and from 1 to 63 reader devices. Authentication is used by the reader devices to verify the identity of the writer device. This type of service is useful, for example, if a device containing an electronic lock receives a message to open the lock. By using authentication, the electronic lock device can verify that the "open" message comes from the owner, not from someone attempting to break into the system.

Authentication doubles the number of messages per transaction. Authentication can be used with acknowledged updates or network variable polls. It cannot be used with unacknowledged or repeated updates. An acknowledged message normally requires two messages, an update and an acknowledgment. An authenticated message requires four messages, as shown in **Figure 8** on page 74. This could affect system response time and capacity.

The following sections describe how to set up devices to use authentication and how authentication works.

Setting Up Devices to Use Authentication

To set up a device to use authenticated network variables or send authenticated messages, follow these steps:

1   Declare the network variable as authenticated. For application messages to be authenticated, specify TRUE in the **authenticated** field of the **msg_out** object.

2   Specify the authentication key to be used for this device using a network tool. The LonMaker tool can be used to install a key during development.

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

Declaring Authenticated Variables and Messages

For network variables, include the **authenticated** (or **auth**) keyword as part of the connection information. The partial syntax is shown below. For complete syntax of the **bind-info** clause, see the *Neuron C Reference Guide*.

bind_info (authenticated [(config | nonconfig)])

Note: The **authenticated** keyword can be abbreviated as **auth**. Likewise, the **nonauthenticated** keyword can be abbreviated as **nonauth**.

If you also include the **config** keyword in the declaration, network tools can change the authentication status of this network variable after the device has been installed. Include the **nonconfig** keyword to prevent the authentication status from being changed for this network variable.

Example:

```
network output UNVT_boolean
    bind_info(auth(nonconfig)) nvoSafeLock;
```

With this declaration, authentication can never be turned off for updates of the **nvoSafeLock** network variable, because the declaration includes the **nonconfig** keyword.

Specifying the Authentication Key

All devices that read or write a given authenticated network variable connection must have the same authentication key. This 48-bit authentication key is used in a special way for authentication, as described below.

The key itself is transmitted to the device only during the initial configuration. All subsequent changes to the key do not involve sending it over the network. The network tool can modify a device's key over the network, in a secure fashion, with a network management message.

How Authentication Works

The following sequence describes an example of authentication (**Figure 8** on page 74 illustrates the process):

1   Device A sends an update to a network variable declared as authenticated on Device B using the acknowledged service. If Device A does not receive the challenge, it sends a retry of the initial update.

2   Device B generates a 64-bit random number and returns, to Device A, a challenge packet that includes the 64-bit random number. Device B then uses the encryption algorithm (built into the Neuron firmware) to compute a transformation on that random number using its 48-bit authentication key and the message data. The transformation is stored in Device B.

3   Device A then also uses the encryption algorithm (built in to the Neuron firmware) to compute a transformation on the random number (returned to it by Device B) using its 48-bit authentication key and the message data. Device A then sends this computed transformation to Device B.

4   Device B compares its computed transformation with the number it
receives from Device A. If the two numbers match, the identity of the
sender is verified, and Device B can perform the requested action and
send its acknowledgment to Device A. If the two numbers do not match,
Device B does not perform the requested action and an error is logged in
the error table.

If the acknowledgment is lost and Device A tries to send the same message again,
Device B remembers that the authentication was successfully completed, and
acknowledges it again.
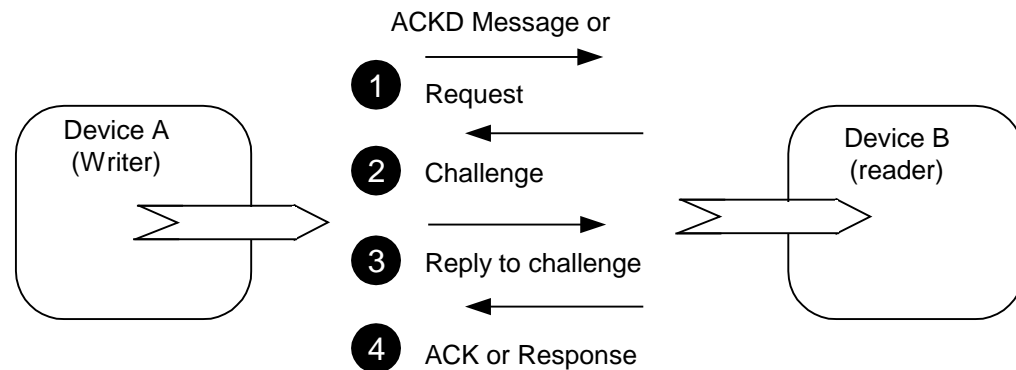
Figure 8. Authentication Process

If Device A attempts to update an output network variable connected to multiple
readers, each receiver device generates a different 64-bit random number and
sends it in a challenge packet to Device A. Device A must then transform each of
these numbers and send a reply to each receiver device.

The principal strength of authentication is that it cannot be defeated by simple
record and playback of commands that implement the desired functions (for
example, unlocking the lock). Authentication does not require that the specific
messages and commands be secret, because they are sent unencrypted over the
network, and anyone who is determined can read those messages.

It is good practice to connect a device directly to a network tool with no other
devices on the same network when installing its authentication key the first
time. This prevents the key from being sent over a large network where an
intruder might detect it. Once a device has its authentication key, a network tool
can modify the key, over the network, by sending an increment to be added to the
existing key.

Alternatively, your development tool might support exporting your device's
application image in a pre-configured state including your initial authentication
key. See your development tool's documentation for information about exporting
pre-configured application images.

Changeable-Type Network Variables

You can create network variables that support their type and size being changed
during installation. This kind of network variable is called a *changeable-type
network variable*.

You can use a changeable-type network variable to implement a generic functional block that works with different types of inputs and outputs. For example, you can create a general-purpose device that can be used with a variety of sensors or actuators, and then create a functional block that allows the integrator to select the network variable type depending on the physical sensor or actuator attached to the device during installation.

You can support type changing to any network variable type defined in a resource file (that is, any SNVT or UNVT in a resource file). You can only create a changeable-type network variable if the network variable is a member of a functional block, and if it is not a configuration network variable. An integrator typically uses a plug-in that you create to change network variable types. A network variable cannot change its type or size while it is connected (because the change would make the connection invalid).

The NodeBuilder Code Wizard generates code that contains a framework for supporting changeable-type network variables; see *Using a Changeable-Type Network Variable* in the *IzoT NodeBuilder User's Guide* for details.

The following details all that is required to create a changeable-type network variable without the use of the NodeBuilder Code Wizard, followed by a detailed discussion of the requirements that the application must meet to support the changeable-type network variables. The section completes with a commented source code example.

To create a changeable-type network variable, follow these steps:

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

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

network output changeable_type SNVT_volt_f nvoValue;

2   Set the changeable-interface bit in the program ID for the device template. You can set this bit by setting **Has Changeable Interface** in the standard program ID calculator when you create the device template, as described in the *IzoT NodeBuilder User's Guide*.

3   Implement a **SCPTnvType** configuration property that applies to the changeable-type network variable. See Chapter 4, *Using Configuration Properties to Configure Device Behavior*, on page 89, for more information about configuration properties. This configuration property is used by network tools to notify your application of changes to the network variable type.

Your application requires notification of changes to this configuration property. You can provide this notification by declaring the configuration property with the **reset_required** or **object_disabled** modifier and checking the **SCPTnvType** value in the director function, or you can

implement configuration property access through LW-FTP and check, in the **stop_transfer( )** function, whether the **SCPTnvType** value has been modified. Alternatively, you can implement the **SCPTnvType** configuration property as a configuration network variable and check the current type in the task for the **nv_update_occurs(***cpnv-name***)** event.

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

Example:

```
SCPTnvType cp_family cp_info(reset_required) nvType;

network output changeable_type SNVT_volt_f nvoValue
    nv_properties { nvType };
```

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

Example:

```
SCPTnvType cp_family cp_info(reset_required) nvType;
const SCPTmaxNVLength cp_family nvMaxLength;

network output changeable_type SNVT_volt_f nvoValue
    nv_properties { nvType,
    nvMaxLength=sizeof(SNVT_volt_f)  };
```

5   Implement code in your Neuron C application to process changes to the **SCPTnvType** value. The required code is described in the following section.

Implement code to provide information about the current length of the network variable to the Neuron firmware. This is described in *Processing a Size Change* on page 80.

Implement your application's algorithm such that it can process all possible types the changeable-type network variable might use at runtime. An example and fragment for such code is shown in *Changeable-Type Example* on page 82.

6   The LonMaker browser provides your integrators with a user interface to change network variable types. You typically want a custom interface for integrators to change network variable types on your device. For example, the custom interface could restrict the available types to the types supported by your application, thus preventing configuration errors. To provide a custom interface, implement code in your plug-in to provide an interface for users to change the network variable type. The required plug-in code is discussed in the *LNS Plug-in Programmer's Guide*.

You cannot change the type of a configuration property unless it inherits its type from a changeable-type network variable. In this case, the configuration

property automatically assumes the size and type of the network variable it applies to, and is governed by the same initial type and maximum size.

Processing Changes to a SCPTnvType CP

When a plug-in or the LonMaker browser changes the type of a network variable, it informs your application of the change by writing a new value to the **SCPTnvType** configuration property associated with the network variable. The definition of the **SCPTnvType** type is provided below:

```
typedef struct {
    unsigned short  type_program_ID[8];
    unsigned short  type_scope;
    unsigned long   type_index;
    nv_type_category_t type_category;
    unsigned short  type_length;
    signed long     scaling_factor_a;
    signed long     scaling_factor_b;
    signed long     scaling_factor_c;
} SCPTnvType;
```

By default, a **SCPTnvType** configuration property is initialized to the following values:

- Program ID = {0, 0, 0, 0, 0, 0, 0, 0}

- Scope = 0

- Index = 1

- Category = NVT_CAT_INITIAL (see *Validating a Type Change* on page 78)

- Type length = 1 byte

- Scaling factor A = 0

- Scaling factor B = 0

- Scaling factor C = 0

Important: Because LNS sets network-variable values through the **SCPTnvType** configuration property, your application must initialize the **SCPTnvType** configuration property to a meaningful value for the application. This initialized value also becomes the "last-known good value" for the application, in case a type-change request must be rejected (see *Rejecting a Type Change* on page 81).

When your application detects a change to the **SCPTnvType** value, it must determine if the change is valid, as described in *Validating a Type Change* on page 78. If it is valid, the application must process the change, as described in *Processing a Type Change* on page 79. On the other hand, if the application determines that the change is not valid or supported, it must report an error, as described in *Rejecting a Type Change* on page 81. If the change is valid and supported by your application, and the change also changes the size of the network variable, your application must implement the size change, as described in *Processing a Size Change* on page 80.

Validating a Type Change

There are several ways that your application can determine if it supports a particular **SCPTnvType** value. It can look for specific types, as specified by the **type_program_ID**, **type_scope**, and **type_index** fields. Alternatively, it can look for specific type categories, as defined by the **type_category** and **type_length** fields.

The **type_program_ID** and **type_scope** values specify a program ID template and a resource scope that together uniquely identify a resource file set. The **type_index** value identifies the network variable type within that resource file set. If the **type_scope** value is 0, the **type_index** value is a SNVT index. The **type_program_ID**, **type_scope**, and **type_index** values uniquely identify a type to your application as well as to any network tools that need to determine the current type, or modify the type, of the network variable to which the property applies. Your application can ignore these values if the remaining fields in the **SCPTnvType** structure provide sufficient information for the application.

The **type_category** enumeration is defined in the **<snvt_nvt.h>** include file as follows:

```
typedef enum nv_type_category_t {
    /*  0 */   NVT_CAT_INITIAL = 0,  // Initial (default)
type
    /*  1 */ NVT_CAT_SIGNED_CHAR,    // Signed Char
    /*  2 */ NVT_CAT_UNSIGNED_CHAR,  // Unsigned Char
    /*  3 */ NVT_CAT_SIGNED_SHORT,   // 8-bit Signed Short
    /*  4 */ NVT_CAT_UNSIGNED_SHORT, // 8-bit Unsigned Short
    /*  5 */ NVT_CAT_SIGNED_LONG,    // 16-bit Signed Long
    /*  6 */ NVT_CAT_UNSIGNED_LONG,  // 16-bit Unsigned Long
    /*  7 */ NVT_CAT_ENUM,      // Enumeration
    /*  8 */ NVT_CAT_ARRAY,     // Array
    /*  9 */ NVT_CAT_STRUCT,    // Structure
    /* 10 */ NVT_CAT_UNION,     // Union
    /* 11 */ NVT_CAT_BITFIELD,  // Bitfield
    /* 12 */ NVT_CAT_FLOAT,     // 32-bit Floating Point
    /* 13 */ NVT_CAT_SIGNED_QUAD,    // 32-bit Signed Quad
    /* 14 */ NVT_CAT_REFERENCE,      // Reference
    /* -1 */ NVT_CAT_NUL = -1  // Invalid Value
} nv_type_category_t;
```

This enumeration describes the type, stating whether it is a signed short, or floating-point, or structure, for example, but not providing information about structure or union fields or other similar details. The **type_length** field is necessary to provide the number of bytes of a structure or union type, though it is set for all types. To support all scalar types, test for a **type_category** value between **NVT_CAT_SIGNED_CHAR** and **NVT_UNSIGNED_LONG**, plus **NVT_CAT_SIGNED_QUAD**. To also support floating point types, also test for a **type_category** value of **NVT_FLOAT**.

The **SCPTnvType** configuration property can be shared between multiple changeable-type network variables. In this case, the application must make sure to process all network variables from the property's application set — **SCTPnvType** applies to all these network variables, and so does the type change request. The type change can only be accepted if all related network variables can perform the required change.

If one or more type-inheriting configuration properties apply to changing configuration network variables (CPNVs), these type-inheriting CPNVs also change their type at the same time. If this type-inheriting CPNV is shared among multiple network variables, all related network variables must change to the new type. Sharing a type-inheriting configuration property among both changeable and non-changeable network variables is not supported.

Processing a Type Change

When the application detects a type change request and recognizes the type detailed in the related **SCPTnvType** property as a supported type, and also confirms that all affected network variables can perform the change, the application performs the type change.

To evaluate the type of a given network variable, you can read the network variable's ___type_scope and __type_index properties to obtain the scope identifier (0..6) and index (1..65535) of the type's definition in the resource file in which it is defined.

Note, however, that the __type_scope and __type_index properties supply compile-time constant values, and do not reflect the current type for a changeable type network variable. __type_scope and __type_index properties always provide scope and index data matching the initial network variable definition.To perform a type change that does not change the size of the network variable, your application need do nothing but memorize the current type details. A different part of the application, the type-independent implementation of your application's algorithm, queries these details as and when required, and processes the network variable data accordingly. The processing required in the type-independent implementation of the application depends on the range of types supported by your application. For example, if your application only supports changing between different floating-point types, no additional processing might be required. If your application supports changing between different scalar types, it might require the use of scaling factors and network variable type length to convert the raw network variable value to a scaled value. For example, the **SNVT_lev_cont** type is an unsigned short value that represents percentages from 0 to 100 percent, with a resolution of 0.5%. The actual data values (also called *raw* values) are in the variable range from 0 to 200. The scaling factors for **SNVT_lev_cont** are defined as a=5, b= -1, c=0. To convert from raw data to scaled fixed-point data, use the following formula:

$$scaled = \left(a*10^b*\left(raw+c\right)\right)$$

Your application can convert the raw data of a changeable type input network variable, internally, to an actual scaled value for use as a floating-point data item, for example, using the above formula. To convert the data back to a raw value for an output network variable, use the following inverted scaling formula:

$$raw = \left(\frac{scaled}{a*10^b}\right) - c$$

You can use cast operations and pointer manipulations to handle type changes. See *Changeable-Type Example* on page 82 for an example.

If a network variable type or size is changed and that network variable is a member of an inheriting configuration property's application set, and that

property is implemented as a configuration network variable, then the application must process the same type or length changes that were performed on the network variable for the configuration network variable.

However, if the configuration property is implemented within a configuration file, no change to the configuration file is required. The configuration file states the configuration property's initial and maximum size (in the CP documentation-string *length* field), and LNS derives the current and actual type for type-inheriting CPs from the associated network variable.

Your application must always support the **NVT_CAT_INITIAL** type category. If the requested type is of that category, your application must ignore all other content of the **SCPTnvType** configuration property and change the related network variable's type back to its initial type. The network variable's initial type is the type the network variable was declared with in Neuron C (**SNVT_volt_f** in the earlier example).

Processing a Size Change

If a supported change to the **SCPTnvType** configuration property results in a change in the size of a network variable type, your application must provide code to memorize the current length of the network variable. It must further provide code to inform the Neuron firmware about the current length of the changeable-type network variable. The current length information must be kept in non-volatile storage, but details of the required implementation depend on the chosen mechanism for supporting the Neuron firmware.

Two such mechanisms are supported, a legacy one called the *nv_len method*, and a more robust *NV length override system image extension* method.

You can explicitly set and maintain the new length of the network variable using the built-in **nv_len** property of the network variable. You can access and modify the built-in **nv_len** property as shown below:

Example of legacy nv_len property:

```
size_t oldNVLen, newNVLen;
oldNVLen = nv-name::nv_len;
nv-name::nv_len = newNVLen;  // not recommended (see text)
```

Important: When the Neuron C compiler detects use of the **nv_len** property to modify a network variable's length, it requests that the linker place the network variable fixed configuration table in writeable memory. This could make it difficult to fit such an application into the memory of a device based on a Neuron 3150 Chip or an FT 3150 Smart Transceiver if the device has no writeable external memory for the application, such as EEPROM or flash memory.

Starting with version 14, the Neuron firmware implements an *NV length override system image extension* that is managed by the application. Whenever the firmware needs the length of a network variable, it calls the **get_nv_length_override( )** system image extension to get it. Compared to writing to the **nv_len** property, this new method provides more reliable updates to network variable sizes, because the old method could cause a device to go applicationless if a power failure occurred in the middle of a network variable size update.

You can enable the NV length override system image extension with the following compiler directive:

#pragma system_image_extensions nv_length_override

Using this compiler directive with a version of the Neuron firmware that does not support system extensions causes an **NLD#477** linker error.

To implement a NV length override system image extension, provide a function with the following prototype:

unsigned _RESIDENT get_nv_length_override(unsigned uNvIndex);

Note that the *get_nv_length_override()* must always be implemented as a resident function.

The **get_nv_length_override( )** function returns the current length of the network variable with the index specified in the argument, or the result of the *get_declared_nv_length()* API to indicate that the type has not been changed and the network variable's initial length is still valid.

The get_declared_nv_length() API has been added with version 2.3 of the Neuron C language. Applications designed for backward compatibility with earlier versions of the Neuron C compiler can use the *_SUPPORT_LARGE_NV* preprocessor symbol to control conditional compilation, as shown in the changeable-type example below.

The system image extension method only works with version 14 firmware, or newer. To support development of applications that use the best possible method depending on the target hardware, you can use conditional compilation to support both methods. This is, for example, used by the NodeBuilder Code Wizard to allow for the LTM-10A device to exercise and implement support for changeable-type network variables. The changeable-type example, later in this chapter, implements such a strategy.

Whenever possible, the system image extension technique should be used, because of its more robust implementation. However, a compiler directive is provided to permit the use of the **system_image_extensions nv_length_override** directive with targets that do not support system extensions. You can turn the **NLD#477** linker error, which would normally occur in such a condition, into a linker warning by using the following directive:

#pragma unknown_system_image_extension_isa_warning

Rejecting a Type Change

If a network tool attempts to change the type of a changeable-type network variable to a type that is not supported by the application, your application must do the following:

- Report the error within a maximum of 30 seconds from the receipt of the type change request. To report the error, the application should signal an **invalid_request** through the Node Object functional block and optionally disable the related functional block. If the application does not include a Node Object functional block, the application can set an application-specific error code using the **error_log( )** function and take the device offline (use **go_offline( )**).

  By setting the functional block status, the rest of the functional blocks on your device can continue to function normally. You can use both methods to provide a more precise indication of the error to a network integrator.

See Chapter 5, *Using Functional Blocks to Implement a Device Interface*, on page 107, for more information on using functional blocks.

- Reset the **SCPTnvType** value to the last known good value.

- Reset all other housekeeping data, if any, so that the last known good type is re-established.

In the interest of future-proof implementations, the application should be sure to reject all change requests to unknown types, as shown in the changeable-type example below.

Changeable-Type Example

The following code sample shows a typical implementation of a changeable-type network variable. It implements **nvoVolt** as a changeable-type output network variable. This example uses utility functions, such as **getObjStatus( )**, **updateNode_Status( )**, and **setFblockDisable( )**. These utility functions are part of the framework provided by NodeBuilder Code Wizard. Your application might not contain those functions, and you should consider providing equivalent functionality in that case.

Parts of the example below are shown in **boldface type**. This indicates the most important parts of the example. The rest of the code (non-boldface type) can be considered more detail-oriented on first read-through.

```
#include <control.h>
#include <float.h>
#include <mem.h>
#include <snvt_nvt.h>

#pragma relaxed_casting_on
#define TYPE_ERROR 1
#define NV_LENGTH_MISMATCH 2


// Forward-declaration of the fblock's director function:
void fbSensorDirector(unsigned uFbIndex, int iCommand);


// Declare the SCPTnvType family.  Note the use of the
// cp_info modifier; the application must have some
// mechanism to become aware of a type change request
// so that it can validate and honor or reject that
// request.  Other possibilities for such a notification
// include the object_disable or offline CP flags, or
// the implementation of this CP as a configuration
// network variable.
SCPTnvType cp_family cp_info(reset_required) nvType;

// SCPTmaxNVLength is optional, but allows for a
// network tool to filter out those types that will
// not be acceptable due to excessive length.  The
// type change routine, below, still must verify that
// the requested type is within supported limits.
const SCPTmaxNVLength cp_family nvMaxLength;

// Declare the changeable-type network variable.
// The network variable's initial type also determines
// its maximum length, hence the initialization of the
// nvMaxLength property using the sizeof() operator
// applies to the NV's inital type.
network output changeable_type SNVT_volt_f nvoVolt
    nv_properties {
        nvType,
```

```
                nvMaxLength=sizeof(SNVT_volt_f)
};

// A functional block that contains
// the changeable-type network variable:
fblock SFPTopenLoopSensor {
    nvoVolt implements nvoValue;
    director fbSensorDirector;
} fbSensor external_name("Sensor");

// nvTypeLastGood memorizes the last known good type of the
// changeable-type network variable.  This is not a
// configuration property, but a simple (EEPROM) variable
// using the same type.
// Note this variable must be initialized correctly, to
// allow the device to come out of the initial reset cycle
// without an accidental type change, and to allow the
// changeable-type NV to function correctly even without
// an explicit type change:
eeprom SCPTnvType nvTypeLastGood =
    {{0, 0, 0, 0, 0, 0, 0, 0}, 0, 1, NVT_CAT_INITIAL, 1, 0, 0, 0};

// The following two compiler directives enable the system extension,
// and allow for its use even if the target device doesn't support
// system extensions.  See text for details, and see the Neuron C
// Reference Guide, Compiler Directives, for details about these
// directives.
#pragma system_image_extensions nv_length_override
#pragma unknown_system_image_extension_isa_warning   // see text!

// changeLength() performs or rejects the type change request.
// It is called from the director function in response to a
// device reset because the SCPTnvType has been declared with
// "cp_info(reset_required)."  Other CP flags, such as
// object_disabled, require a different invocation.  SCPTnvType
// may also be implemented as a configuration network variable,
// allowing for invocation of the changeLength() function from
// a "when(nv_update_occurs(...))" task.

void changeLength(void) {

    // First, check to see if there is anything to do at all:
    // is there a real type change request pending?  The
    // changeLength() function could have been invoked as a
    // result of a regular device reset (or whichever other
    // update notification event is associated with the nvType CP).

    if ((nvoVolt::nvType.type_category != NVT_CAT_NUL)
      && (memcmp((void*)&nvTypeLastGood,(void*)&nvoVolt::nvType,
            sizeof(SCPTnvType)) != 0)) {

        // In case multiple network variables share the same
        // SCPTnvType configuration property, make sure all
        // affected network variables are unbound.  Use is_bound()
        // for all these network variables and reject the type change
        // if any reports being bound.
        // Check if requested type is within acceptable size
        // limits.  The sizeof(nvoVolt) function always returns the
        // initial size of the network variable, which equals
        // its maximum size.

        if (nvoVolt::nvType.type_length > sizeof(nvoVolt)) {

            // Reject: set the nvType CP back to the last known
            // good value, log the error, and notify the
            // network tool.  In addition to the minimum
            // requirements, this example implementation
            // also automatically disables the fblock

            nvoVolt::nvType = nvTypeLastGood;
            error_log(TYPE_ERROR);
```

```
                getObjStatus(fbSensor::global_index)->invalid_request
                        = TRUE;
                updateNode_Status();
                setFblockDisable(fbSensor::global_index, TRUE);

        } else switch (nvoVolt::nvType.type_category) {

                case NVT_CAT_SIGNED_LONG:
                case NVT_CAT_UNSIGNED_LONG:
                case NVT_CAT_FLOAT:

                        // Accept long and float.
                        // Store the current type information and, for
                        // debugging purpose only, also change the length of
                        // the network variable via its nv_len property.  See
                        // further below for an example implementation of the
                        // recommended get_nv_length_override technique for
                        // this network variable.
                        // Writing the nv_len property is not recommended, but
                        // the technique demonstrated in this example can be
                        // useful when the debugging platform does not support
                        // the recommended get_nv_length_override technique.

                        nvTypeLastGood = nvoVolt::nvType;

#ifdef _DEBUG  // see text!
                        nvoVolt::nv_len = nvoVolt::nvType.type_length;
#endif

                        // For all inheriting configuration properties that
                        // apply to this network variable and that are
                        // implemented as configuration network variables,
                        // repeat this type change.

                        break;

                case NVT_CAT_INITIAL:

                        // This is a request to change the type back to its
                        // initial type (whichever is the initial type).
                        // For cardinal types with significant scaling
                        // factors A, B, or C, the application may need to
                        // restore those scaling factors or to preserve
                        // that knowledge otherwise; see GetCurrent() or
                        // SetCurrent() functions, below, for details.
                        // The sizeof() function always returns the size of
                        // the initial type.

                        nvoVolt::nvType.type_length = sizeof(nvoVolt);
                        nvTypeLastGood = nvoVolt::nvType;

#ifdef _DEBUG  // see text!
                        nvoVolt::nv_len = nvoVolt::nvType.type_length;
#endif
                        // For all inheriting configuration properties that
                        // apply to this network variable and that are
                        // implemented as configuration network variables,
                        // repeat this type change.

                        break;

                // Reject all other types.  This example implementation
                // just refuses the change request and continues to
                // operate on the last known good type:

                default:
                    nvoVolt::nvType = nvTypeLastGood;
                    error_log(TYPE_ERROR);
                    getObjStatus(fbSensor::global_index)->invalid_request
                            = TRUE;
                    updateNode_Status();
```

```
              } // end of switch
         } // any change at all
   } // function changeLength()

   // The fbSensorDirector() function manages this functional block.
   // Because the nvType CP has been declared with the reset_required
   // flag, the director must call the changeLength() function as part
   // of the reset processing to allow for the type change request to
   // be executed.
   // The director function is not called automatically, but
   // requires a framework that explicitly calls the director.
   // The director implementation shown here is incomplete, as it
   // ignores all other commands and duties.  See the director
   // implementation generated by the NodeBuilder Code Wizard
   // for a more comprehensive example of a director function, and
   // for a complete framework that issues director invocations.

   void fbSensorDirector(unsigned uFbIndex, int iCommand) {
       if ((TFblock_command)iCommand == FBC_WHEN_RESET) {
           changeLength();
           setLockedOutBit(uFbIndex, FALSE);
       } // FBC_WHEN_RESET
   } // fbSensorDirector()

   // Whenever the current length of the changeable network-type variable is
   // required by the Neuron firmware, the firmware calls the
   // get_nv_length_override() system image extension.  This function
   // returns the current length of the given NV (in bytes) or the result
   // of the get_declared_nv_length() API (or 0xFF for earlier versions) to
   // indicate that the initial type is still unchanged.

   unsigned _RESIDENT get_nv_length_override(unsigned uNvIndex) {
   #if defined(_SUPPORT_LARGE_NV)
       unsigned uResult = get_declared_nv_length(uNvIndex);
   #elseunsigned uResult = 0xFF;
   #endif

       if (uNvIndex == fbSensor::nvoValue::global_index) {
           // Return current length for our example NV, or return
           // 0xFF to indicate the NV has the initial length:

           if (nvTypeLastGood.type_category != NVT_CAT_INITIAL
            && nvTypeLastGood.type_category != NVT_CAT_NUL) {
               // this is a distinct current length:
               uResult = nvTypeLastGood.type_length;
           }
       }

       return uResult;
   }
   // Triggered by some appropriate I/O event, timer, or network event,
   // the application will need to process data for the changeable-type
   // network variable.  This example does not include an algorithm that
   // performs numeric operations using the changeable-type data, but two
   // conversion routines are shown that convert the current type of
   // the changeable network variable into a float_type variable for
   // internal use in such numeric operations, and vice versa.

   void GetCurrent(float_type* const pFloat) {
       // One union to hold all possible current types, plus the initial
       // type of the changeable type NV
       union {
           unsigned long uLong;
           signed long   sLong;
           SNVT_volt_f   xInitial;
       } nvLocal;

       // bProcessABC: a flag to indicate whether the scaling factors
       // A,B,C must be honored and used
       boolean bProcessABC = FALSE;
```

```
        nvLocal.xInitial = nvoVolt;

        switch (nvoVolt::nvType.type_category) {
            case NVT_CAT_SIGNED_LONG:
                // Current type is signed long. Convert to float.
                fl_from_slong(nvLocal.sLong,pFloat);
                bProcessABC = TRUE;
                break;
            case NVT_CAT_UNSIGNED_LONG:
                // Current type is unsigned long. Convert to float.
                fl_from_ulong(nvLocal.uLong,pFloat);
                bProcessABC = TRUE;
                break;
            case    NVT_CAT_INITIAL:
                // Fall through to float.
            case NVT_CAT_FLOAT:
                // Float is current. No conversion is required, just
                // copy data into local variable.
                *pFloat = nvLocal.xInitial;
                break;
            default:
                // Unsupported type.  The changeLength() handler should
                // have recognized this and rejected the type earlier.
                // Log this application error and set the device offline:
                error_log(TYPE_ERROR);
                go_offline();
        } // switch

        if (bProcessABC) {
            // TODO: If needed by the application algorithm, transform
            // the raw *pFloat NV value into the scaled float equivalent
            // using the following formula:
            //   scaled = A * 10**B * (*pFloat + C)
            // Scaling factors are accessible via the scaling_factor_X
            // members of the SCPTnvType CP, for example
            // nvo1::nvType.scaling_factor_a.  This transformation is a
            // costly operation and it is recommended to design
            // the application algorithm such that this conversion
            // is not required at all, if possible.
        }
} // GetCurrent()
void SetCurrent(float_type* pFloat) {
    // One union to hold all possible current types, plus the initial
    // type of the changeable NV.
    union {
        unsigned long uLong;
        signed long   sLong;
        SNVT_volt_f   xInitial;
    } nvLocal;

    boolean bConversionOK = TRUE;
    boolean bProcessABC = nvoVolt::nvType.type_category ==
NVT_CAT_SIGNED_LONG
            || nvoVolt::nvType.type_category == NVT_CAT_UNSIGNED_LONG;

    if (bProcessABC) {
        // TODO: if needed by the application algorithm, revert the
        // conversion done in GetCurrent() by using the following
        // formula:
        //   raw = (*pFloat / (A * 10**B)) - C
        // See GetCurrent(), above, for more details.
    }

    switch (nvoVolt::nvType.type_category) {
        case NVT_CAT_SIGNED_LONG:
            // Current type is signed long. Convert from float.
            nvLocal.sLong = fl_to_slong(pFloat);
            break;
        case NVT_CAT_UNSIGNED_LONG:
            // Current type is unsigned long. Convert from float.
            nvLocal.uLong = fl_to_ulong(pFloat);
```

```
            break;
        case    NVT_CAT_INITIAL:
            // Fall through to float.
        case NVT_CAT_FLOAT:
            // Float is current. No conversion is required, just
            // copy data into local variable.
            nvLocal.xInitial = *pFloat;
            break;
        default:
            // Unsupported type.  The changeLength() handler should
            // have recognized this and rejected the type earlier.
            // Log this application error and set the device offline:
            error_log(TYPE_ERROR);
            go_offline();
            bConversionOK = FALSE;
    } // switch

    if (bConversionOK) {
        // Update the actual network variable in case the conversion
        // was OK (current type is in fact supported).
        // A more generic implementation of these conversion functions
        // is likely to use a pointer to the changeable type network
        // variable's initial type as a second argument, thus allowing
        // the SetCurrent() and GetCurrent() functions to be used for
        // all changeable type NVs of the same initial type.
        // This approach is likely to require explicit calls to the
        // propagate() function; see the Neuron C Reference Guide
        // for details.

        nvoVolt = nvLocal.xInitial;
    } // bConversionOK
} // SetCurrent()
```

4

Using Configuration Properties to Configure Device Behavior

This chapter discusses the declaration and use of configuration properties. Configuration properties are part of the device interface, and are used by network tools to configure device behavior during and after network installation.

Overview

A configuration property is a data item that, like a network variable, is part of the device's interoperable interface. A configuration property can be modified by a network tool. Configuration properties facilitate interoperable installation and configuration tools by providing a standardized network interface for device configuration data. Like network variables, configuration properties also provide a well-defined interface.

Each configuration property type is defined in a resource file that specifies the data encoding, scaling, units, default value, invalid value, range, and behavior for configuration properties based on the type. A rich variety of standard configuration property types (SCPTs) are defined in the standard resource file set. You can view all currently defined SCPTs online at types.lonmark.org. You can also create your own user configuration property types (UCPTs) that are defined in resource files that you create with the IzoT Resource Editor.

Declaring Configuration Properties

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

The disadvantages of configuration network variables are that they are limited to a maximum size of a network variable (228 bytes for Series 6000 chips, 31 bytes for earlier Neuron chips and Smart transceivers, and 228 bytes for most other platforms) of 31 bytes each, and that the number of configuration network variables is determined by the maximum number of network variables for the target platform.

To implement a configuration property as a configuration network variable, declare it using the **network … config_prop** syntax described in *Declaration of Configuration Network Variables* on page 91.

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

The advantages of implementing configuration properties as configuration files is that there are no limits on configuration property size or the number of configuration properties, except as constrained by the available memory space on the device. The disadvantages are that other devices cannot connect to or poll a

configuration property implemented as a configuration file, instead requiring a network tool to modify a configuration property implemented within a configuration file, and no events are automatically generated when a configuration property implemented within a configuration file is updated. The application can force notification of updates by requiring network tools to reset the device, disable the functional block, or take the device offline when a configuration property is updated (though the reset or online notification is the only type of notification that occurs after the configuration property has been modified). Alternatively, the application can also force notification by implementing configuration file access through the LONWORKS file transfer protocol (LW-FTP) and monitoring the **stop_transfer( )** function. This option requires additional code space for the LW-FTP server code.

To implement a configuration property as a part of a configuration file, declare it with the **cp_family** syntax described in *Declaring Configuration Properties within Files* on page 92.

Declaration of Configuration Network Variables

The configuration network variable declaration syntax is similar to the declaration syntax of a non-configuration network variable as already discussed in Chapter 3, *How Devices Communicate Using Network Variables*, on page 45.

The complete syntax for declaring a configuration network variable is shown below. The declaration is made distinct from other network variable declarations by the inclusion of the **config_prop** keyword following the type of the network variable declaration. The **config_prop** keyword can be abbreviated as **cp**.

> **network input** [*netvar-modifier*] [*class*] *type* **config_prop** [*cp-modifiers*]
> [*connection-info*] *identifier* [= *initial-value*] **;**

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

Examples:

```
network input SCPTupdateRate config_prop nciUpdateRate;

network input SCPTbypassTime cp nciBypassTime = ...
```

The *netvar-modifier* and *class* portions of this syntax were discussed in depth in the previous chapter, and they apply equally to a configuration network variable as they do to any other network variable, except that the *class* cannot be **config**. A **config** network variable is not a fully managed configuration property, it is a manually managed one. The **config** keyword is obsolete and not recommended for use in new development, but is provided to allow legacy applications to be used with the Neuron C Version 2 compiler.

Similar to the configuration CP family members, configuration network variables must be declared with a *type* that is defined by a configuration property type within a resource file. The type can be a standard (SCPT) or user (UCPT) type. The *cp-modifiers* clause that can optionally follow the **config_prop** keyword is also identical with the CP family declaration discussed earlier in this chapter (see the *Neuron C Reference Guide* for a discussion of the *cp-modifiers* syntax and semantics).

The *connection-info* for a configuration network variable is no different than the connection info for any other input network variable, as discussed in the previous

chapter. Like any other network variable, a configuration network variable can be an array, with each element of the array being a separately handled configuration property, or with the entire configuration network variable array being handled as a single configuration property. See *Instantiation of Configuration Properties* on page 94 for details.

A configuration network variable's declaration can contain an *initial-value* or an *initializer-list*, like any other network variable declaration, as discussed in the previous chapter. <u>Unlike</u> any other network variable, a configuration network variable cannot, itself, also have a network variable property list. That is, you cannot define configuration properties that apply to other configuration properties.

When using a network variable array as a configuration property or properties, particular care should be given to the compiler's rules of initialization for that network variable array. The array elements can be initialized in the declaration, as is the case with any variable or array variable declaration. If some or all of the array elements are not initialized, the uninitialized elements default to a zero initialization. However, each array element can be initialized when it appears as a property in a properties clause, and this declaration overrides the initialization in the declaration, but only for the element that appears in that property clause. Similarly, if the entire network variable array is used as a single configuration property, the entire array can be initialized when it appears as a property in a properties clause.

You cannot change the type of a configuration property unless it inherits its type from a changeable-type network variable.

Declaring Configuration Properties within Files

You can declare a configuration property that is to be implemented within a configuration file using a CP family declaration. A CP family declaration can be thought of as a *meta*-declaration, defining a type construct for later use in the program. It can be used to declare a collection of many configuration properties, identical in type and certain other settings, but individually applying to one or more different network variables, functional blocks (as described in Chapter 5, *Using Functional Blocks to Implement a Device Interface*, on page 107), or the device itself. A CP family can have zero members, one member, or many members. No code or data is generated until you declare members of the CP family, as described later. In this regard, the CP family is similar to a C language **typedef**.

The syntax for declaring a CP family is shown below:

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

family-ident : identifier **[** *array-bound* **]**
                    identifier

Example:

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

The *type* for a CP family cannot be just a standard C type, such as **int** or **char**. Instead, the declaration must use a configuration property type (CPT) from a resource file. The configuration property type can either be a standard configuration property type (SCPT) or a user configuration property type (UCPT).

There are over 300 SCPT definitions available today, and you can create your own manufacturer-specific types using UCPTs. The SCPT definitions are stored in the **standard.typ** file, which is part of the standard resource file set included with the NodeBuilder tool. There can be many similar resource files containing UCPT definitions, and these are managed by the IzoT Resource Editor as described in the *IzoT NodeBuilder User's Guide*.

A configuration property type is similar to an ANSI C **typedef**, but it is also much more. The configuration property type also defines standardized semantics for the type. The configuration property definition in a resource file contains information about the default value, minimum and maximum valid values, a designated (optional) invalid value, and language string references that permit localized descriptive information, additional comments, and units strings to be associated with the configuration property type.

The *cp-modifiers* begin with the **cp_info** keyword followed by a parenthesized list of option keywords. The keywords and their meanings are discussed in the *Configuration Property and Network Variable Declarations* chapter of the *Neuron C Reference Guide*.

If the declaration of the CP family contains an *array-bound* expression following the family identifier name, each member of the CP family is declared to be a separate array. For example, a family can consist of three members: an array property for some network variable *A*, another array property for another network variable *B*, and a third array property for a functional block *C*.

Example:

```
SCPTgain cp_family cpGain[3] = { { 2, 3 },
                                 { 1, 5 },
                                 { 2, 1 }
                               };
```

The *initial-value* in the declaration of a CP family is optional. If *initial-value* is not provided in the declaration, the default value specified by the resource file is used. The *initial-value* given is an initial value for a single member of the family, but the compiler replicates the initial value for each instantiated family member.

The initialization rules for a CP family member are shown below. The initialization rules are used to set the initial value that is to be loaded in the value file from the linked image, as well as the value file stored in the device interface file. A network tool can use the initial value as a *default value*, and might, at times, reset the configuration properties (or a subset of them) back to the default values. Consult the documentation of the particular network tool, for example, the *LonMaker User's Guide*, for more information on the tool's use of configuration property default values.

In the initialization rules that follow, the compiler uses the first rule that applies to the configuration property.

1   If the configuration property is initialized explicitly in its instantiation, then this is the initial value that is used.

2   If the configuration property is initialized explicitly in the CP family declaration, then the family initializer is used.

3   If the configuration property applies to a functional block, and the functional profile that defines the functional block specifies a default

value for the associated configuration property member, then the functional profile default is used.

4   If the configuration property type for the configuration property defines a default value, then that default value is used as the initial value. This rule does not apply for a configuration property type that is type-inheriting; see *Type-Inheriting Configuration Properties* on page 104.

5   If no initial value is available from any of the preceding rules, a value of all zeros is used.

The **cp_family** declaration is repeatable. The declaration can be repeated two or more times, and, as long as the duplicated declarations match in every regard, the compiler treats them as a single declaration.

The following example shows a valid repetition, two invalid repetitions, and a non-repeating case.

Example:

```
// INITIAL declaration of family:
SCPTgain cp_family cpGain = { 2, 3 };

// VALID repetition: families are identical
SCPTgain cp_family cpGain = { 2, 3 };

// INVALID repetition: different initializer!
SCPTgain cp_family cpGain = { 1, 10 };

// INVALID repetition: different cp_info
SCPTgain cp_family cp_info(offline) cpGain = { 2, 3 };

// NO REPETITION, but creation of a valid second family:
SCPTgain cp_family cp_info(offline) cpLowGain = { 1, 8 };
```

Instantiation of Configuration Properties

Configuration properties can apply to a device, one or more functional blocks, or one or more network variables. In each case, a configuration property is made to apply to its respective objects through a *property list*. Property lists for the device and network variables are explained in the following sections; property lists for functional blocks are explained in Chapter 5, *Using Functional Blocks to Implement a Device Interface*, on page 107. You cannot have more than one configuration property of any given SCPT or UCPT type that applies to an object, where that object is a network variable, a functional block, or the entire device.

As discussed above, the **cp_family** declaration is similar to a C language **typedef** because no actual variables are created as a result of the declaration. In the case of a type definition, variables are instantiated when the type definition is used in a later declaration that is not, itself, another **typedef**. At that time, variables are *instantiated*, which means that variables are declared and memory is allocated for and assigned to the variables. The variables can then be used in later expressions in the executable code of the program.

The instantiation of a CP family member occurs each time the CP family declaration's identifier is used in a property list. For exceptions to this rule, see *Sharing of Configuration Properties* on page 102.

However, a configuration network variable is already instantiated at the time it is declared. For a configuration network variable, the property list serves only to identify the association between the configuration property and the object or objects to which it applies.

Device Property Lists

A device property list declares instances of configuration properties defined by CP family declarations and configuration network variable declarations that apply to a device. See the *Neuron C Reference Guide* for the device-property syntax.

Example:

```
SCPTlocation cp_family cpLocation;

device_properties {
    cpLocation = { "Unknown" }
};
```

The device property list appears at file scope. This is the same level as a function declaration, a task declaration, or a global data declaration. The device property list begins with the **device_properties** keyword. It then contains a list of property references, separated by commas. Each property reference must be the name of a previously declared CP family or the name of a previously declared configuration network variable. If the network variable is an array, a single array element can be chosen as the device property, so an array index must be given as part of the property reference in that case to identify the element. Alternatively, the entire network variable array can be chosen as the device property, so no array index is given in the property reference in that case.

Example of a CP network variable array element:

```
network input SCPTlocation cp cpLocation[5];

device_properties {
    cpLocation[0] = { "Unknown" }
};
```

The example above implements a single device property of type **SCPTlocation**, which is implemented by the first element of the configuration property network variable array. The remaining four elements of that array are unused in the above example.

In contrast, the following example illustrates the use of a configuration network variable array as a single device property. The device property with internal name **cpOemType** is a single-dimensional array of three elements, each of type SCPToemType.

Example of entire CP network variable array as a single property:

```
network input SCPToemType cp cpOemType[3];

device_properties {
    cpOemType = { "Label 1", "Label 2", "Label 3" }
};
```

Following the *property-identifier*, there can be an optional *initializer*, and an optional *range-mod*. These elements are discussed in detail in the *Neuron C Reference Guide* chapter on *Configuration Properties and Network Variables*.

A Neuron C program can have multiple device property lists. These lists are merged together by the Neuron C compiler to create one combined device property list. However, you cannot have more than one configuration property of any given SCPT or UCPT type that applies to the device.

If two separate modules specify a particular configuration property of the same type in the device property lists, this situation causes a compilation error.

Example of incorrect CP conflict:

```
UCPTsomeDeviceCp cp_family cpSomeDeviceCp;
SCPTlocation cp_family cpLocation;
SCPTlocation cp_family cpPlacement;

device_properties {
    cpSomeDeviceCp,
    cpLocation,
    cpPlacement // Conflicts with cpLocation
};
```

Network Variable Property Lists

A network variable property list declares instances of configuration properties defined by CP family declarations and configuration network variable declarations that apply to a network variable. See the *Neuron C Reference Guide* for the configuration network variable's property list syntax.

Example:

```
// CP for heartbeat and throttle (default 1 min each)
SCPTmaxSndT cp_family cpMaxSendT = { 0, 0, 1, 0, 0 };
SCPTminSndT cp_family cpMinSendT = { 0, 0, 1, 0, 0 };

// NV with heartbeat and throttle:
network output SNVT_lev_percent nvoValue
    nv_properties {
        cpMaxSendT,
        // override default for minSendT to 30 seconds:
        cpMinSendT = { 0, 0, 0, 30, 0 }
};
```

The network variable property list begins with the **nv_properties** keyword. It then contains a list of property references, separated by commas, exactly like the device property list. Each property reference must be the name of a previously declared CP family or the name of a previously declared configuration network variable. The rest of the syntax is very similar to the device property list syntax discussed above.

Following the *property-identifier*, there can be an optional *initializer*, and an optional *range-mod*. These optional elements are discussed in more detail in the *Neuron C Reference Guide*.

You cannot have more than one configuration property of any given SCPT or UCPT type that applies to the same network variable. A compilation error occurs

when a particular configuration property type is used for more than one property in the network variable's property list.

Unlike device properties, network variable properties can be shared between two or more network variables. The use of the **global** keyword creates a CP family member that is shared between two or more network variables. The use of the **static** keyword creates a CP family member that is shared between all the members of a network variable array, but not with any other network variables outside the array. See *Sharing of Configuration Properties* on page 102 for more information on this topic.

Accessing Property Values from a Program

You can access configuration properties from a program just as you can access any other variable. For example, you can use configuration properties as function parameters and you can use addresses of configuration properties.

However, to use a CP family member in an expression, you must specify *which* family member to access, because there could be more than one member of the same CP family with the same name applying to different network variables. The syntax for accessing a configuration property from a network variable's property list uses the Neuron C context operator, a double colon, as shown below:

> *nv-context* **::** *property-reference*

> *nv-context* : *identifier* **[** *index-expr* **]**
>   *identifier*

> *property-reference : property-identifier* **[** *index-expr* **]**
>   *property-identifier*

Example:

```
// CP for heartbeat and throttle (default 1 min each)
SCPTmaxSndT cp_family cpMaxSendT = { 0, 0, 1, 0, 0 };
SCPTminSndT cp_family cpMinSendT = { 0, 0, 1, 0, 0 };

// NV with heartbeat and throttle:
network output SNVT_lev_percent nvoValue
   nv_properties {
       cpMaxSendT,
       cpMinSendT = { 0, 0, 0, 30, 0 }
};

void f(void)
{
   ...
   if (nvoValue::cpMaxSendT.seconds > 0) {
       ...
   }
}
```

The particular CP family member is identified by a qualifier that precedes it. This qualifier is called the *context*. The context is followed by two consecutive colon characters, called the *context operator*, and then the name of the property. Because there cannot be two or more properties with the same configuration property type that apply to the same network variable, each property is unique within a particular context. The context therefore uniquely identifies the

property. For example, a network variable array, **nvoArray**, with 10 elements, could be declared with a property list referencing a CP family named **cpXyz**. There would then be 10 different members of the **cpXyz** CP family, all with the same name. However, adding the context, such as **nvoArray[4]::cpXyz**, or **nvoArray[j]::cpXyz**, uniquely identifies the CP family member.

Since the same CP family could also be used as a device property, there is a special context defined for the device. The device's context is a context operator (two consecutive colon characters) without a preceding context identifier.

Example, Accessing a Device Property:

```
network input SCPToemType cp cpOemType[3];

device_properties {
    cpOemType = { "Label 1", "Label 2", "Label 3" }
};

void f(void)
{
    if (strcmp(::cpOemType[0].ascii, "Demo") == 0) {
        ... // special demo mode
    } else {
        ... // normal operation
    }
}
```

Even though a configuration network variable can be uniquely accessed through its variable identifier, it can also be accessed equally well through the context expression, just like the CP family members.

When more than one syntactically correct method exists for accessing a particular CP, these methods are equivalent. The compiler translates all alternatives into the same, equally efficient, code. In such a case, you should choose the syntax that best documents your application algorithm.

Advanced Configuration Property Features

Configuration properties support a few advanced features that are described in this section. The first of these features is the use of configuration properties with network variable arrays. Second is the initialization of configuration properties at time of instantiation.

Another advanced feature is sharing of configuration properties, where a single configuration property can apply to two or more network variables, or two or more functional blocks (see Chapter 5, *Using Functional Blocks to Implement a Device Interface*, on page 107, for information on functional blocks). However, no single configuration property (or configuration property family member) can apply to both network variables *and* functional blocks.

The last advanced feature discussed in this section is configuration properties with type-inheritance. Some configuration property types (CPTs) indicate that the type of the configuration property is actually defined by the network variable to which it applies. Type-inheriting configuration properties are discussed further at the end of this chapter.

Configuration Properties Applying to Arrays

When configuration properties apply to network variable arrays, the compiler provides replication of the configuration properties for each member of the array (unless the property is shared, as discussed in *Sharing of Configuration Properties* on page 102). Consider a network variable array with four elements (each corresponds to some sensor, perhaps):

```
network output SNVT_volt nvoVoltage[4];
```

Now, suppose that we want to provide a **SCPTmaxSendTime** configuration property for each sensor output that is used to configure the maximum amount of time (in seconds) between consecutive updates of the output network variable. If we use a configuration property family, we can use the following declarations. When using a configuration property in this manner, the Neuron C compiler automatically creates a distinct family member for each element of the network variable array.

Example with a CP family property:

```
SCPTmaxSendTime cp_family cpMaxSendTime;

network output SNVT_volt nvoVoltage[4]
    nv_properties { cpMaxSendTime };
```

Another approach is to use a separate network variable array for the **SCPTmaxSendTime** configuration properties. For example, the network variable array declaration shown below provides four elements in the **cpMaxSendTime** array, each of them a configuration property corresponding to the respective element in the **nvoVoltage** array.

Example with a network variable property:

```
network input cp SCPTmaxSendTime cpMaxSendTime[4];

network output SNVT_volt nvoVoltage[4]
    nv_properties { cpMaxSendTime[0] };
```

When using a distributed array of network variables for the configuration properties as shown in the example above, the configuration property reference in the **nv_properties** clause must contain an index. This index is used by the compiler as a *starting index*; the compiler automatically assigns the subsequent elements of the configuration property network variable array to the elements of the underlying network variable array to which the property applies.

The compiler also checks for existence of a sufficient number of elements. The following declarations would give errors:

Example of insufficient array size:

```
network input cp SCPTmaxSendTime cpMaxSendTime[3];
            // Insufficient # of elements

network output SNVT_volt nvoVoltage[4]
    nv_properties { cpMaxSendTime[0] };
```

Example of insufficient array elements in reference:

```
network input cp SCPTmaxSendTime cpMaxSendTime[4];

network output SNVT_volt nvoVoltage[4]
```

```
        nv_properties {
                cpMaxSendTime[1]   // insufficient members left
        };
```

The index of the configuration property reference in the **nv_properties** clause is a *starting* index. This index need not be zero. For example if there were two network variable arrays named **nvoVolt1** and **nvoVolt2**, and each were to have a **SCPTmaxSendTime** property, the following declarations could accomplish this scenario, where part of the configuration property network variable array is used for one array of output network variables, and the other is used for another array of output network variables. Although this case shows all members of the **cpMaxSendTime** array being used, that is not a requirement.

Example of an array split between different property clauses:

```
network input cp SCPTmaxSendTime cpMaxSendTime[7];

network output SNVT_volt nvoVolt1[4]
    nv_properties { cpMaxSendTime[3] };

network output SNVT_vold nvoVolt2[3]
    nv_properties { cpMaxSendTime[0] };
```

The examples above focused on applying single configuration properties to arrays of network variables. However, a second case exists where the configuration property itself is an entire array, rather than an element, as shown below.

Example of CP family array applying to network variable array:

```
UCPTeventData cp_family cpEventData[100];

network output SNVT_volt nvoVolt[4]
    nv_properties { cpEventData };
```

This example implements four output network variables **nvoVolt[0]**..**nvoVolt[3]**. Each of these four network variables implements a **cpEventData** configuration property, which itself is an array of 100 elements. Each array element is of the hypothetical type **UCPTeventData**.

A similar construct cannot be achieved with configuration network variables, however, because of a crucial difference in the two configuration property implementation techniques. That difference is, as discussed earlier, that the declaration of a **cp_family** alone does not create any configuration properties, whereas configuration network variable declaration actually creates the (configuration) network variable. Consider the following, incorrect, example for illustration:

Example of CP-NV array applying to network variable array:

```
network input UCPTeventData cp cpEventData[10];

network output SNVT_volt nvoVolt[4]
    nv_properties {
            // causes compilation error:
            cpEventData
    };
```

This last example does not compile. Because a configuration network variable declaration actually creates the (configuration) network variable, the compiler cannot multiply the number of configuration network variables by four as

required to implement one configuration property array for each of the network variable elements in the array **nvoVolt**.

Consequently, a configuration property network variable array that applies to an array of network variables (or functional blocks) must be shared using either the **static** keyword or the **global** keyword. See *Sharing of Configuration Properties* on page 102 for more information on these keywords.

Initialization of Configuration Properties at Instantiation

You can initialize a configuration property of fixed type in its declaration. When a network variable array is used as an array of configuration properties, the following example could occur. Each of the four configuration properties shown below is initialized to the value '10' (a power-up delay value is a number of seconds).

```
network input cp SCPTpwrUpDelay nvcp[4] = {10, 10, 10, 10};
```

It is *not required* to initialize the configuration property at instantiation, but this can be useful, as explained in the following example. Assume that we want to declare two network variables **nvoA** and **nvoB**, and we want to associate the **nvcp[0]** configuration property with **nvoA**, and **nvcp[1]** with **nvoB**. Furthermore, in these two instances, we want the power-up delay properties to be 5 seconds, and 10 seconds, respectively.

We can then *override* the initial value in the declaration with a new initial value in the instantiation of the property for **nvoA**, but take advantage of the previous initialization of **nvcp[1]** to 10.

Example of Network Variable CP Initialization

```
network input cp SCPTpwrUpDelay nvcp[4] = {10, 10, 10, 10};

network output SNVT_volt nvoA
    nv_properties { nvcp[0] = 5 };

network output SNVT_amp nvoB
    nv_properties { nvcp[1] };
```

Extending the above example, consider another network variable array **nvoC** of two members, where we use **nvcp[2]** and **nvcp[3]** as configuration properties of **nvoC[0]** and **nvoC[1]**, respectively. Also, we want these configuration properties each initialized to 60 seconds. We can use the following declaration:

```
network output SNVT_count nvoC[2] = {100, 100}
    nv_properties { nvcp[2] = 60 };
```

The **nvoC** network variable is an array, so the **nvcp[2]** property reference is treated as a *starting point* for the compiler to perform the automatic assignment of properties, as discussed *Configuration Properties Applying to Arrays* on page 99. The compiler automatically replicates the reference to **nvcp[2]**, which applies to **nvoC[0]**, and the replication occurs for each subsequent element of the **nvoC** array (**nvcp[3]** to **nvoC[1]**, and so on). In this replication, the compiler also replicates the initialization (in this case, **nvcp[3]** is therefore also initialized to 60). It is therefore *not possible* to have *different* initial values for each

element's configuration property, unless these initial values are provided with the declaration of the configuration network variable array as shown here.

Example of Network Variable CP Initialization

```
network input cp SCPTpwrUpDelay nvcp[4] = {10, 20, 30, 40};

network output SNVT_volt nvoA
    nv_properties { nciPwrUpDly[0] };

network output SNVT_amp nvoB
    nv_properties { nciPwrUpDly[1] };

network output SNVT_count nvoC[2] = {100, 100}
    nv_properties { nciPwrUpDly[2] };
```

Some configuration property types (for example, **SCPTdefOutput**) are *type-inheriting*. This means that the SCPT definition does not, itself, specify the data type for the configuration property. Instead, the configuration property's data type is inherited from the network variable to which it applies. In this case, the only explicit initialization that is permitted is in the instantiation in the property list, not in the declaration. This situation is explained further in *Type-Inheriting Configuration Properties* on page 104.

Sharing of Configuration Properties

The typical instantiation of a configuration property is unique to a single device, functional block, or network variable. For example, a CP family whose name appears in the property list of five separate network variables has five instantiations, and each instantiation is specific to a single network variable. Similarly, a network variable array of five elements that includes the same CP family name in its property list instantiates five members of the CP family, each applying to one of the network variable array elements.

You can change the instantiation behavior using the **static** and **global** keywords. The **global** keyword causes a single CP family member to be shared among all network variables whose property list contains that CP family name. There can only be one such global member in a CP family, and that member is shared among all network variables that instantiate it in their property lists.

The same sharing considerations apply to configuration properties that apply to functional blocks; see Chapter 5, *Using Functional Blocks to Implement a Device Interface*, on page 107, for more information about functional blocks and configuration properties, but note that a CP can only be shared among network variables *or* functional blocks.

The **static** keyword causes a single CP family member to be shared among all elements of a network variable array, but the sharing of the **static** member does not extend outside of the array.

Example:

```
// CP for throttle (default 1 minute)
SCPTmaxSndT cp_family cpMaxSendT = { 0, 0, 1, 0, 0 };
    // This family will have a total of 2 members

// NVs with shared throttle:
network output SNVT_lev_percent nvoValue1
```

```
        nv_properties {
            global cpMaxSendT
        };
    network output SNVT_lev_percent nvoValue2
        nv_properties {
            global cpMaxSendT // The same as the one above
        };
    network output SNVT_lev_percent nvoValueArray[10]
        nv_properties {
            static cpMaxSendT // Shared among the array
                        // elements only
        };
```

Although the discussion above concerns instantiation and shared CP family members, configuration network variables can also be shared using a similar method. Use the **static** keyword in the array's property list to share a configuration network variable among members of a network variable array. Use the **global** keyword in the configuration network variable's property list to share the property among two or more network variables.

The only difference between the configuration network variable and the CP family member, in this regard, is that the configuration network variable cannot appear in two or more property lists *without* the **global** keyword because there is only one instance of the network variable (whereas CP families can have multiple instances).

A configuration property that applies to a device cannot be shared because there is only one device per application.

Configuration Property Sharing Rules

The following rules apply to configuration property sharing. This list summarizes the rules described elsewhere in this chapter.

1   A configuration property can only be shared between multiple network variables, or between multiple functional blocks, but not between a combination of network variables and functional blocks at the same time.

2   All configuration property types can be shared.

3   A configuration property that applies to the entire device cannot be shared.

4   Multiple functional blocks or network variables can share a configuration property. A shared configuration property can apply to multiple singular functional blocks or network variables, a functional block or network variable array, a number of functional block or network variable arrays, or any combination thereof.

5   A configuration property that is shared among the members of a functional block or network variable array must always be shared among all members of that array.

6   A configuration property can be shared between network variables on different functional blocks.

7   A configuration property that inherits its type from a network variable can only be shared between network variables that are all of the same type. Therefore, all changeable type network variables that share an

inheriting configuration property must also share an instantiation of **SCPTnvType**, so that the set of changeable network variables always have the same, single, type and so that type changes occur at the same time.

8   Two (or more) mandatory functional profile template configuration properties can be implemented using a single, shared, configuration property provided the shared configuration property meets the requirements of all individually listed FPT members (for example, same type, same array size, and so on).

9   A single configuration property that inherits its type from a network variable cannot be shared simultaneously by both changeable and non-changeable network variables.

Type-Inheriting Configuration Properties

You can define a configuration property type that does not include a complete type definition, but instead uses the type definition of the network variable to which it applies. A configuration property type that uses another variable's type is called a *type-inheriting configuration property*.

A typical use for type-inheriting configuration properties is for value limiting configuration properties. Configuration properties that describe a network variable's minimum or maximum values, or threshold or hysteresis levels, for example, need not be defined for every possible network variable type. Instead, such a configuration property is fully described by its semantics, and through the item to which it applies.

When the CP family member for a type-inheriting configuration property appears in a property list, the instantiation of the CP family member uses the type of the network variable. Likewise, a configuration network variable can be type-inheriting, but each element of a configuration network variable array must inherit the *same* type.

Because the type of a type-inheriting configuration property is not known until the instantiation, the configuration property initializer option can only be provided in the property list rather than in the declaration. Likewise, different *range-mod* strings may apply to different instantiations of the property, and therefore, for a type-inheriting configuration property, the *range-mod* option can only be provided in the property list, rather than in the declaration.

Shared configuration network variables (see the preceding section on that topic) that are also type-inheriting can only be shared among network variables of identical type.

A type-inheriting configuration property cannot be used as a device property, because the device has no type from which to inherit. It should be noted that type-inheriting configuration properties can apply to network variables or to functional blocks, but not to the device. In the case that a type-inheriting configuration property applies to a network variable explicitly, it derives its type from the network variable. In the event that the type-inheriting configuration property applies to the entire functional block, the property derives its type from the functional block's principal member network variable.

Each functional profile should have one member network variable designated as the principal member network variable. The profile must define a principal

network variable if type-inheriting configuration property members exist that apply to the entire profile. Type-inheriting configuration properties can be shared among multiple network variables or multiple functional blocks, but the types of the network variables to which the type-inheriting configuration property explicitly or implicitly applies must all be the same.

A typical example of a type-inheriting configuration property is the **SCPTdefOutput** configuration property type. The **SFPTopenLoopSensor** functional profile is an example of a functional profile that lists the **SCPTdefOutput** configuration property as an optional configuration property, and it is used to define the default value for the sensor's principal network variable. The **SFPTopenLoopSensor** functional profile itself, however, does not define the type for the principal network variable.

The following example implements a **SFPTopenLoopSensor** functional block with an optional **SCPTdefOutput** configuration property. The configuration property inherits the type, **SNVT_amp** in this case, from the network variable it applies to as shown below.

Example:

```
SCPTdefOutput cp_family cpDefaultOutput;

network output SNVT_amp nvoAmpere nv_properties {
   cpDefaultOutput = 123
};

fblock SFPTopenLoopSensor {
   nvoAmpere implements nvoValue;
} fbAmpereMeter;
```

The initial value (**123** in the example) can only be provided in the instantiation of the configuration property as discussed above, and not in the declaration, because the type for **cpDefaultOutput** is not known until it is instantiated and because it is a type-inheriting configuration property.

Type-Inheriting Configuration Properties for Network Variables of Changeable Type

Type-inheriting configuration properties can also be combined with changeable-type network variables; see *Changeable-Type Network Variables* on page 74. The type of such a network variable can be changed by a network integrator when the device is installed in a network.

Example:

```
SCPTdefOutput cp_family cpDefaultOutput;
SCPTnvType cp_family    cpNvType;
network output changeable_type SNVT_amp nvoValue
   nv_properties {
       cpDefaultOutput = 123,
       cpNvType
   };
fblock SFPTopenLoopSensor {
   nvoValue implements nvoValue;
} fbGenericMeter;
```

The **nvoValue** principal network variable, although it is of changeable type, must still implement a default type (**SNVT_amp** in the example above). Because the **SCPTdefOutput** type-inheriting configuration property inherits the type information from this initial type, the initializer for **cpDefaultOutput** must therefore be specific to this instantiation. Furthermore, the initializer must be valid for this initial type.

Should the network integrator decide to change the type of the underlying network variable at runtime to, for example, **SNVT_volt**, then it is the responsibility of the network tool to apply the formatting rules that apply to the new type when reading or writing this configuration property. The network tool must also set any type-inheriting configuration properties to reasonable initial values that correspond to the new type of the network variable (and thus, the newly inherited type of the configuration property).

If a type-inheriting configuration properties applies to more than one changeable-type network variable (implicitly or explicitly), all the network variables in the configuration property's application set must share one **SCPTnvType** configuration property, to ensure that all those network variables always have the same type.

A type-inheriting configuration property cannot be shared among changeable-type network variables and fixed-type network variables, even if all have the same (initial) type.

5

Using Functional Blocks to Implement a Device Interface

This chapter discusses the use of functional blocks to provide a task-oriented interface for a device. You can use functional blocks to group network variables and configuration properties that perform a task together.

Overview

The device interface for a LONWORKS device consists of its functional blocks, network variables, and configuration properties. A *functional block* is a collection of network variables and configuration properties, used together to perform one task. The network variables and configuration properties contained within a functional block are called the *functional block members*. Using functional blocks promotes modular device design, and focuses the integrators on the control algorithm.

Functional blocks simplify the installation of your devices by network integrators. Integrators are able to view your devices as collections of task-oriented functional blocks rather than monolithic device applications. When network integrators combine your functional blocks with other functional blocks within a network, they can easily see the relation of the different functional blocks within the network. For example, **Figure 9** illustrates the integrator view of a temperature controller implemented with functional blocks. The example illustrates seven functional blocks and their network connections. These functional blocks are actually implemented on two devices, but the figure clearly shows how the input sensors provide data to a temperature controller that in turn provides outputs to actuators and a process monitor.
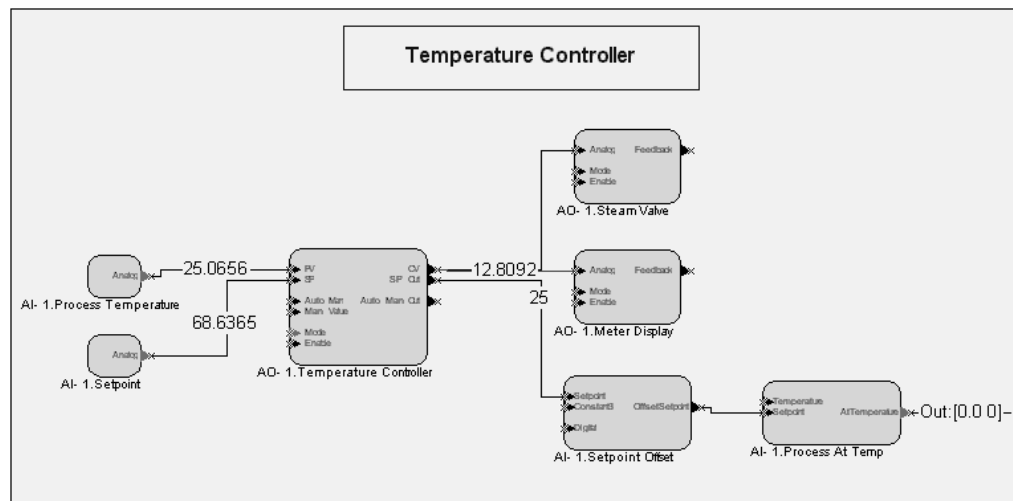
Figure 9. Network Design with Functional Blocks

Functional blocks are defined by *functional profiles*. A functional profile is used to describe common units of functional behavior. All four member categories are optional, for example, there might be no mandatory member network variables defined in a profile, or there might not be any optional configuration property members defined.

Each functional profile defines mandatory and optional network variables and configuration properties. Each functional block implements an instance of a functional profile. A functional block must implement all of the mandatory network variables and mandatory configuration properties defined by the functional profile, and can implement any of the optional network variables and optional configuration properties defined by the functional profile. A functional block can also implement network variables and configuration properties not

defined by the functional profile – these are called *implementation-specific* network variables and configuration properties.

For example, **Figure 10** illustrates standard functional profile number 3050, the Constant Light Controller profile. This profile defines two mandatory inputs, one mandatory output, and one optional input. It also defines one mandatory configuration property and eight optional configuration properties.
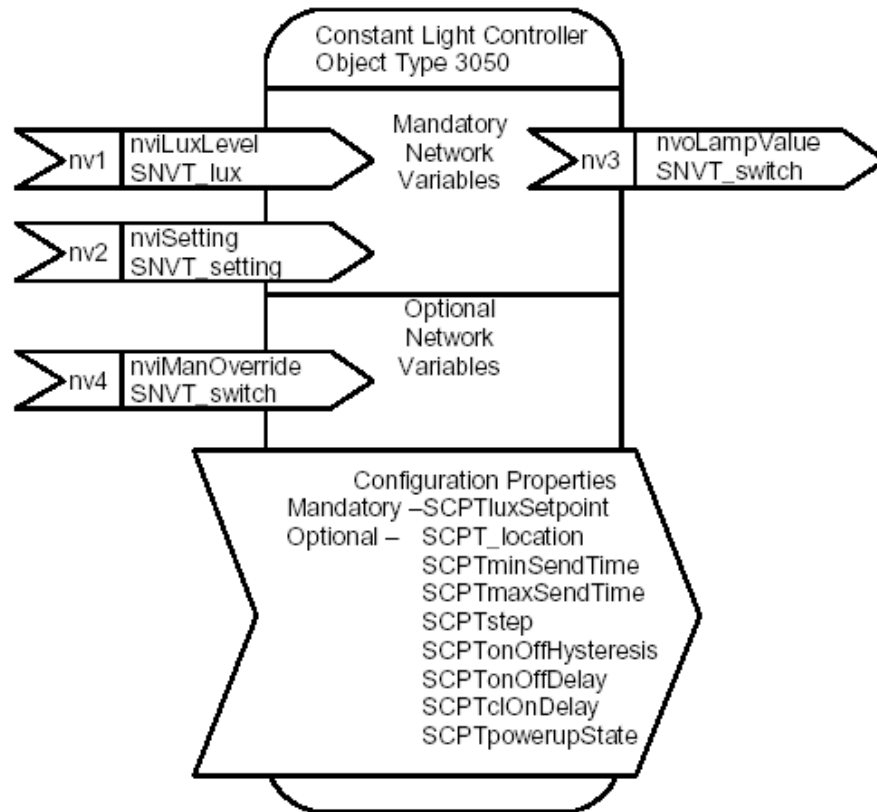
Figure 10. Constant Light Controller Functional Profile

Functional profiles are defined in *resource files*. You can use standard functional profiles defined in the standard resource file set, or you can define your own functional profiles in your own resource file sets. A functional profile defined in a resource file is also called a *functional profile template*.

Standard functional profiles (also called standard functional profile templates, or SFPTs) are defined by LONMARK International. You can view all currently defined standard functional profiles online at types.lonmark.org. Additional documentation for the standard functional profiles is available under *Technical Resources* at www.lonmark.org. You can create your own user-defined functional profiles (also called user functional profile templates, or UFPTs) using the IzoT Resource Editor. You declare functional blocks in your Neuron C applications using **fblock** declarations. These declarations are described in this chapter.

You can declare up to 254 functional blocks in a Neuron C application or model file, resources permitting.

A functional block declaration does not cause the compiler to generate executable code, but the compiler generates self-identification and self-documentation data

and some required data structures that implement a functional block. Principally, the functional block creates associations among network variables and configuration properties. The compiler then uses these associations to create the self-documentation (SD) data and self-identification (SI) data in the device and in its associated device interface file (.xif extension).

Functional Block Declarations

The complete syntax for declaring a functional block is the following:

> **fblock** *FPT-identifier* **{** *fblock-body* **}** *identifier* [*array-bounds*] [*ext-name*] [*fb-property-list*] **;**

> *ext-name* : **external_name (** *C-string-const* **)**
>   **external_resource_name (** *C-string-const* **)**
>   **external_resource_name (** *const-expr* **:** *const-expr* **)**

> *array-bounds* : **[** *const-expr* **]**

> *fblock-body* : [*fblock-member-list*] [**;** *director-function*]

> *fblock-member-list* : *fblock-member-list* **;** *fblock-member*
>   *fblock-member*

> *fblock-member* : *nv-reference* **implements** *member-name*
> *nv-reference impl-specific*

> *impl-specific* : **implementation_specific (** *const-expr* **)** *member-name*

> *nv-reference* : *nv-identifier array-index*
>   *nv-identifier*

> *array-index* : **[** *const-expr* **]**

> *director-function* :   **director** *identifier* **;**

The functional block declaration begins with the **fblock** keyword, followed by the name of a functional profile from a resource file. The functional block is an implementation of the functional profile (that is, it *instantiates* the profile). The functional profile defines the abstract network variable and configuration property members, a unique key called the *functional profile number* or *functional profile key*, and other information. The network variable and configuration property members are divided into mandatory members and optional members. Mandatory members must be implemented, and optional members can be implemented as needed.

The functional block declaration then proceeds with a member list. In this member list, you associate network variables that you declared previously in the application with the network variable members of the profile. The **implements** keyword associates your application network variables with the profile network variable members. The member list can be omitted if the functional block is used only as a collection of related configuration properties.

At a minimum, every mandatory network variable member of the profile must be implemented by an actual network variable in the Neuron C program. Each network variable (or, in the case of a network variable array, each array element) can implement no more than one profile member, and can be associated with at most one functional block.

Example:

```
network output SNVT_amp nvoAmpere;

fblock SFPTopenLoopSensor {
    nvoAmpere implements nvoValue;
} fbAmpereMeter;
```

A Neuron C program can also implement *additional* network variables in the functional block that are not in the lists of mandatory or optional members of the profile. Such additional network variable members beyond the profile are called *implementation-specific* members. Declare these extra members in the member list using the **implementation_specific** keyword, followed by a unique index number, and a unique name.

Each network variable in a functional profile assigns an index number and a member name to each profile network variable member, and the implementation-specific member cannot use any of the index numbers or member names that the profile has already used.

Example:

```
network output SNVT_amp nvoAmpere;
network output polled SNVT_time_stamp nvoInstallDate;

fblock SFPTopenLoopSensor {
    nvoAmpere implements nvoValue;
    nvoInstallDate implementation_specific(128)
            nvoInstall;
} fbAmpereMeter;
```

The above example implements the **nvoValue** mandatory network variable of the **SFPTopenLoopSensor** functional profile, and adds an implementation-specific **SNVT_time_stamp** network variable with a member name of **nvoInstall**. The member name, **nvoInstall** in this example, is typically used to refer to the member network variable, as discussed in *Accessing Members and Properties of a Functional Block from a Program* on page 116.

The name of the network variable, **nvoInstallDate**, however, is the name that is exposed to the network integrator by means of network variable self-documentation (SD) data and device interface files. In a network tool, the name **nvoInstall** appears as the member of the functional block, wherever the network tool uses the profile definition.

Important: Implementation-specific network variable or configuration properties are no longer acceptable according to the rules of the interoperability application layer guidelines, starting with version 3.4.

If you plan to submit your device for certification, you must perform one of the following tasks:

- Remove the implementation-specific items from the interoperable interface (for example, declare them as device network variables).

- Create a user-defined functional profile that includes the desired additions, listed as mandatory or optional member NV or CP.

The implementation-specific NV member feature can also be used repeatedly to add each element of an entire NV array to the functional block. One element is

added per line. In this way, a functional block can contain an NV array with the elements declared as consecutive NV members.

At the end of the member list there is an optional item that permits the specification of a director function. The director function specification begins with the **director** keyword, followed by the identifier that is the name of the function, and ends with a semicolon.

Example:

```
network output SNVT_amp nvoAmpere;

extern void MeterDirector(unsigned fbIdx, unsigned cmd);

fblock SFPTopenLoopSensor {
    nvoAmpere implements nvoValue;
    director MeterDirector;
} fbAmpereMeter;
```

See *The Director Function* on page 119 for more details about directors.

After the member list, the functional block declaration continues with the name of the functional block itself. A functional block can be a single declaration, or it can be a singly dimensioned array.

If the functional block is implemented as an array as shown in the example below, then each network variable that implements a member of that functional block must be declared as an array of at least the same size. When implementing the **fblock** array's member with an array network variable element, the starting index of the first network variable array element in the range of array elements must be provided in the implements statement. The Neuron C compiler automatically adds the following network variable array elements to the **fblock** array elements, distributing the elements consecutively.

Example:

```
network output SNVT_lev_percent nvoValue[6];

// The following declares an array of four fblocks,
// with "nvoAnalog" members implemented by the
// network variables nvoValue[2] .. nvoValue[5],
// respectively.

fblock SFPTanalogInput {
    nvoValue[2] implements nvoAnalog;
} myFb[4];
```

You can provide an optional external name for each functional block. To specify an external name, use the **external_name** keyword, followed by a string of up to 16 characters in parentheses. The string becomes part of the device interface which is exposed to network tools.

Alternatively, you can provide an optional external name that is specified by a language string in a resource file using the **external_resource_name** keyword. In this case, the device interface information contains a scope and index pair (the first number is a scope, then a colon character, then the second number is an index). The scope and index pair identifies a language string in a resource file, which a network tool can access for a language-dependent name of the functional block. You can use the scope and index pair to reduce memory requirements and

to provide language-dependent names for your functional blocks. The external name is discussed in more detail in the *Neuron C Reference Guide*.

Example:

```
#define  NUM_AMMETERS   4

network output SNVT_amp nvoAmpere[NUM_AMMETERS];

fblock SFPTopenLoopSensor {
    nvoAmpere[0] implements nvoValue;
} fbAmpereMeter[NUM_AMMETERS] external_name("AmpereMeter");
```

Functional Block Property Lists

At the end of the functional block declaration is a property list, similar to the device property lists and the network variable property lists discussed in the previous chapter. The functional block's property list, at a minimum, must include all of the mandatory properties defined by the functional profile that apply to the functional block. You may add implementation-specific properties to the list without any special keywords. You cannot implement more than one property of any particular SCPT or UCPT type for the same functional block.

The functional block's property list must only contain the mandatory and optional properties that apply to the functional block as a whole. Properties that apply specifically to an individual abstract network variable member of the profile must appear in the *nv-property-list* of the network variable that implements the member, rather than in the *fb-property-list*.

See the *Neuron C Reference Guide* for a description of the syntax for the functional block's property list.

Example:

```
SCPTdefOutput  cp_family cpDefaultOutput;
SCPTbrightness cp_family cpDisplayBrightness;

network output SNVT_amp nvoAmpere;
network output polled SNVT_time_stamp nvoInstallDate;

fblock SFPTopenLoopSensor {
    nvoAmpere implements nvoValue;
} fbAmpereMeter external_name("AmpereMeter")
    fb_properties {
        cpDefaultOutput,        // optional CP
        cpDisplayBrightness     // implementation-spec.
    };
```

The example implements an open-loop sensor as an ampere meter. The **nvoValue** mandatory network variable is implemented, but no optional network variables are. The **SCPTdefOutput** optional configuration property is also implemented.

The names in the above example for the CP families (**cpDefaultOutput** and **cpDisplayBrightness**) have no external relevance; these names are only used within the device's source code to reference the configuration property. See *Accessing Members and Properties of a Functional Block from a Program* on page

116 and *Accessing Members and Properties of a Functional Block from a Network Tool* on page 118 for more details.

Shared Functional Block Properties

Just as network variable properties can be shared, functional block properties can be shared between two or more functional blocks. The **global** keyword creates a configuration property member that is shared among two or more functional blocks. This global member is a *different* member than a global member shared among network variables. The **static** keyword creates a configuration property member that is shared among all the members of a functional block array, but not with any other functional blocks or network variables outside the array.

For example, consider a three-phase ampere meter, implemented with an array of three **SFPTopenLoopSensor** functional blocks. Assume the hardware contains a separate amplifier for each phase, but a common analog-to-digital converter for all three phases. Each phase thus has individual gains factors, but might have to share one property to specify the sample rate for all three phases:

Example:

```
#define   NUM_AMMETERS    3

SCPTgain cp_family cpGain;
SCPTupdateRate cp_family cpUpdateRate;

network output SNVT_amp nvoAmpere[NUM_AMMETERS];

fblock SFPTopenLoopSensor {
    nvoAmpere[0] implements nvoValue;
} fbAmpereMeter[NUM_AMMETERS]  external_name("AmpereMeter")
    fb_properties {
        cpGain,
        static cpUpdateRate
};
```

Assume, furthermore, that the same device also contains a three-phase voltage meter with an implementation that mirrors the one from the ampere meter. And, assume there is a **SCPTbypassTime** configuration property that limits the duration of a locally initiated bypass mode for all six meters.

The following example implements all six meters, implementing a global **SCPTbypassTime** configuration property that is shared between all **fblock**s that refer to it, and implementing two static **SCPTupdateRate** configuration properties, shared among the members of the respective **fblock** array:

Example:

```
#define   NUM_PHASES      3

SCPTgain     cp_family cpGain;
SCPTupdateRate cp_family cpUpdateRate;
SCPTbypassTime cp_family cpBypassTime;

network output SNVT_amp nvoAmpere[NUM_PHASES];
network output SNVT_volt nvoVolt[NUM_PHASES];
```

```
fblock SFPTopenLoopSensor {
    nvoAmpere[0] implements nvoValue;
} fbAmpereMeter[NUM_PHASES]  external_name("AmpereMeter")
    fb_properties {
        cpGain,
        static cpUpdateRate,
        global cpBypassTime
};

fblock SFPTopenLoopSensor {
    nvoVolt[0] implements nvoValue;
} fbVoltMeter[NUM_PHASES]  external_name("AmpereMeter")
    fb_properties {
        cpGain,
        static cpUpdateRate,
        global cpBypassTime
};
```

Scope Rules

When adding implementation-specific network variables or configuration properties to a standard or user functional profile, you must ensure that the scope of the resource definition for the additional item is numerically less than or equal to the scope of the functional profile.

For example, if you add an implementation-specific network variable or configuration property to a standard functional block (SFPT, scope 0), you must define that configuration property with a standard type (SCPT), and use a standard network variable type (SNVT) for the implementation-specific network variable.

A second example: if you implement a functional block based on a manufacturer scope (scope 3) resource file, you can add an implementation-specific network variable or configuration property that is defined in the same scope 3 resource file, and you can also add an implementation-specific network variable or configuration property defined by a SNVT or SCPT.

You can add implementation-specific members to standard functional profiles using inheritance by performing the following steps:

1   Use the IzoT Resource Editor to create a user functional profile with the same functional profile key as the standard functional profile you wish to inherit from.

2   Set **Inherit Members from Scope 0** in the functional profile definition. This setting makes all members of the standard functional profile part of your user functional profile.

3   Declare a functional block based on the new user functional profile.

4   Add implementation-specific members to the functional block. These members can be implemented using user-defined UNVT or UCPT types, themselves defined at the same scope as the inheriting functional profile.

Important: Implementation-specific network variable or configuration properties are no longer acceptable according to the rules of the interoperability application layer guidelines, starting with version 3.4.

Alternatively, you can create a functional profile that inherits members from a standard functional profile, and add your own profile-specific members to the functional profile, by performing the following steps:

1.   Use the IzoT Resource Editor to create a user functional profile with the same functional profile key as the standard functional profile that you want to inherit from.

2.   Set **Inherit Members from Scope 0** in the functional profile definition. This setting makes all members of the standard functional profile part of your user functional profile.

3.   Add your additional members to the new user functional profile.

4.   Declare a functional block based on the new user functional profile.

This method provides better documentation and easier reusability than using implementation-specific members.

Accessing Members and Properties of a Functional Block from a Program

You can access the network variable and configuration property members of a functional block from a program just as you can access any other variable. For example, members can be used in expressions, as function parameters, or as operands of the address operator or the increment operator. To access a network variable member of a functional block, or to access a network variable configuration property of a functional block, the network variable reference can be used in the program just as any other variable would be.

However, to use a CP family member, you must specify which family member is being accessed, because more than one functional block could have a member from the same CP family. The syntax for accessing a configuration property from a functional block's property list uses the Neuron C context operator, a double colon, as follows:

fb-context **::** *property-identifier* [**[** *index-expr* **]**]

fb-context : identifier **[** *index-expr* **]**
             identifier

The particular CP family member is identified by a qualifier that precedes it. This qualifier is called the *context*. The context is followed by two consecutive colon characters, and then the name of the property. The context uniquely identifies the property. For example, a functional block array, **fba**, with 10 elements, could be declared with a property list referencing a CP family named **cpXyz**. There would then be 10 different members of the CP family **cpXyz**, all with the same name. However, adding the context, such as **fba[4]:: cpXyz**, or **fba[j]:: cpXyz**, would uniquely identify the CP family member.

Just like for network variable properties, even though a configuration network variable can be uniquely accessed through its variable identifier, it can also be accessed equally well through the context expression, just like the CP family members.

Also, the network variable members of the functional block can be accessed through a similar syntax. The syntax for accessing a functional block member is shown below (the *fb-context* syntactical element is defined above):

fb-context **::** *member-identifier*

This expression uses the network variable's member identifier, not the network variable's unique name. Using the context expression to identify a member network variable therefore promotes modular device design and reuse of code – multiple functional blocks implementing the same functional profile can all implement the same network variable members, although each block's members are mapped to a different network variable.

Finally, the properties of the functional block's network variable members can also be accessed through an extension of this syntax. The syntax for accessing a functional block's member's property is shown below (the *fb-context* syntactical element is defined above):

fb-context **::** *member-identifier* **::** *property-identifier* [**[** *index-expr* **]**]

Example:

```
#define   NUM_AMMETERS    3

SCPTmaxSndT     cp_family cpMaxSendTime;
SCPTminSndT     cp_family cpMinSendTime;
SCPTgain        cp_family cpGain[4];
SCPTupdateRate  cp_family cpUpdateRate;

network output SNVT_amp nvoAmpere[NUM_AMMETERS]
    nv_properties {
        cpMaxSendTime,
        cpMinSendTime
    };

fblock SFPTopenLoopSensor {
    nvoAmpere[0] implements nvoValue;
} fbAmpereMeter[NUM_AMMETERS]  external_name("AmpereMeter")
    fb_properties {
        cpGain,  // Each property is an array [4]
        static cpUpdateRate
    };
```

All of the following constructs are examples for valid code:

```
nvoAmpere[2] = 123;
fbAmpereMeter[2]::nvoValue = 123;
fbAmpereMeter[0]::cpGain[i].multiplier = 2L;
nvoAmpere[2]::cpMaxSendTime.seconds = 30;
fbAmpereMeter[2]::nvoValue::cpMaxSendTime.hour = 0;
z = ((SCPTmaxSndT *)&nvoAmpere[2]::cpMaxSendTime)->day;
```

Pointers can be used with CP family members as shown; however, the configuration properties are stored in EEPROM. This causes the compiler to apply special rules as described for the **#pragma relaxed_casting_on** directive in the *Neuron C Reference Guide*.

Because **cpGain** is a static configuration property, the following expression is always true:

```
fbAmpereMeter[0]::cpGain[i].multiplier ==
    fbAmpereMeter[1]::cpGain[i].multiplier
```

The following expressions are *incorrect* and cause a compiler error:

```
        // '.' instead of '::'
        fbAmpereMeter[0].cpGain[i].multiplier = 123;

        // reference of CP family, not CP family member
        cpGain.multiplier = 123;

        //    '::' instead of '.'
        fbAmpereMeter[0]::cpGain[i]::multiplier = 123;
```

Neuron C also provides some built-in properties for a functional block. The built-in properties are shown below (the *fb-context* syntactical element is defined above):

> *fb-context* **:: global_index**

> *fb-context* **:: director (** *expr* **)**

The **global_index** property is an **unsigned short** value that corresponds to the global index assigned by the compiler. The global index is a read-only value. The global index ranges from 0 (zero) to 253, with each **fblock** and element of an **fblock** array having a unique index. The order of the **fblock** index follows the order in which the **fblock** declarations are compiled.

Use of the **director** property as shown calls the director function that appears in the declaration of the functional block. The compiler provides the first parameter to the actual director function automatically (the first argument is the global index of the functional block), and the *expr* shown in the syntax above becomes the director function's second parameter. This second parameter is usually referred to as **unsigned uCommand**, however, the compiler passes any value of type **unsigned** without imposing any special interpretation.

The **director** property can be used in any case, no matter whether a director function is defined for this individual **fblock**, shared among various **fblock**s, or not defined at all. In case no actual director function is defined, use of the **director** property does not cause a compilation or run-time error. The firmware support for the **director** property handles the case of an undefined director function by taking no action other than just returning to the calling program.

For more about the director property, the **global_index** property, and for examples showing their use, see *The Director Function* on page 119.

Accessing Members and Properties of a Functional Block from a Network Tool

Network tools can implement whatever representation suits the tool's user interface and purpose best. The LonMaker Integration Tool, for example, focuses on a graphical representation of functional blocks and member network variables. Configuration properties are typically represented by custom controls in specialized configuration software, such as an LNS device plug-in.

For example, the **cpGain** property from the example above might be presented as a slider for graphical adjustment of the gain factor with the mouse or cursor keys.

However, most tools also supply some textual reference to configuration properties. When listing configuration property members, those members are

typically listed using or including their type name, for example, *"SCPTupdateRate"* or *"UCPTboosterControl"*.

Because a functional block can only implement one configuration property of a given type, this naming scheme provides unique names. To avoid confusion between the internal and external names of configuration properties, you should preserve some degree of similarity between the internal and external names.

Example:

```
SCPTbrightness cp_family cpBrightness;
```

This example above implements a configuration property family with the internal name *cpBrightness* of type name *SCPTbrightness*. The type name is likely to appear as an external, textual, reference to that property, depending on the implementation of the network tool.

The Director Function

You can create a *director function* for each functional block. A director function is a function that can provide actions associated with the functional block such as enable, disable, reset, or test. The association with the functional block enables easy implementation of the standard request functions defined by Node Object functional block. These request functions allow a network tool to send a request to a device to enable, disable, reset, or test any functional block on the device. The Node Object implementation can vector these requests to the appropriate functional block function through the use of the director function. The Node Object implementation generated by the NodeBuilder Code Wizard includes code to call the functional block director functions based on inputs from the Node Object Request input.

A director function must match the function prototype shown below. The first parameter is the global index of the functional block for which the director is being called, and the second parameter is a command code upon which the director is to act.

> **void** *director-name* **(unsigned** *fbIndex***, unsigned** *command***);**

You attach a director function to a functional block with an optional declaration statement at the end of the member list of the functional block. You do not need to declare a director function. You can also share one director function among multiple functional blocks, but all elements of a functional block array must share the same director function.

Example:

```
void myDirector (unsigned fbIndex, unsigned command);

fblock . . . {
    /* Member NVs, "implements" . . . */

    director myDirector;
} myFB;

void myDirector (unsigned fbIndex, unsigned command) {
    . . . /* whatever */
}
```

The director function simplifies implementation of functional block commands received through the Node Object functional block by promoting modular development. Each functional block is a functional unit, a collection of network variables and properties. A network tool can send a request to a device's Node Object to enable, disable, reset, or test any functional block on the device. The Node Object implementation can then direct this request to code specific to the requested functional block by calling the functional block's director function. The director function provides an easy way for the device to manage its functional blocks and make sure that events and commands are directed to the proper functional block.

Example:

An implementation of the **SFPTnodeObject** functional block receives requests through the **nviRequest** mandatory member network variable input. Examples for these requests are the **RQ_DISABLED** and **RQ_ENABLED** requests, which requests one or more objects to enter the disabled or enabled state, respectively. These requests can apply to the Node Object functional block, to an individual functional block other than the Node Object functional block, or to all functional blocks implemented on the device. A **SFPTnodeObject** implementation can inspect the scope of the command received, and route the command to the right director function as follows:

```
when (nv_update_occurs(nviRequest))
{
    if (nviRequest.object_id == MyNodeObj::global_index) {
        // NodeObject must handle this:
        MyNodeObj::director(nviRequest.object_request);
    } else {
        // route the command to the best director:
        ...    (see below)
    }
}
```

When a network variable update is received, you can determine the functional block containing the network variable by using the built-in **fblock_index_map** variable. This mapping array has an element for each network variable, and each corresponding element contains the global index of the functional block of which the network variable is a member. If the network variable is not a member of a functional block, the corresponding element contains 0xFF, meaning "no functional block".

You can also directly call the director for a functional block by specifying the functional block index. To do this, call the **fblock_director( )** system function. This function has the same prototype as an individual director, and allows calling any functional block's director function through FB global index. The **fblock_director( )** function automatically selects which of the actual director functions to call, and returns when that director function completes. If the functional block does not have a director, the **fblock_director( )** function returns and does nothing.

Using this structure, the above example can be completed with code that routes the command received to the most appropriate director:

Example:

```
when (nv_update_occurs(nviRequest))
{
```

```
        if (nviRequest.object_id == MyNodeObj::global_index) {
            // NodeObject must handle this:
            MyNodeObj::director(nviRequest.object_request);
        } else {
            // route the command to the best director:
            fblock_director(nviRequest.object_id,
                    nviRequest.object_request);
        }
    }
```

Likewise, a single task can handle all network variable updates by notifying the director function that is in charge of the functional block to which the network variable update applies:

```
    #define   CMD_NV_UPDATE   17

    when (nv_update_occurs)
    {
        fblock_director(fblock_index_map[nv_in_index],
                CMD_NV_UPDATE);
    }
```

There are no limitations on how you use a director function or how you interpret the second parameter to the director function. The director function is a useful means to create Node Object implementations, but you can extend its usage as well.

Sharing of Configuration Properties

Elements of a functional block array or multiple distinct functional blocks can share configuration properties using the **static** and **global** modifiers. The following example implements two configuration properties, one being shared among all members of the functional block array, and one being shared by the other two functional blocks.

Example:

```
    cp_family SCPTgain cpGain = { 1, 0 };
        // This family will have a total of 2 members

    // FBs with shared gain factor:
    fblock ... {
        ...
    } fbA fb_properties {
        global cpGain   // shared by fbA and fbB
    };

    fblock ... {
        ...
    } fbB fb_properties {
        global cpGain   // shared by fbA and fbB
    };

    fblock ... {
        ...
    } fbC[5] fb_properties {
        static cpGain   // shared among fbC[0]..fbC[4]
    };
```

The rules and considerations for shared configuration properties that are outlined in Chapter 4, *Using Configuration Properties to Configure Device Behavior*, on page 89, apply to functional blocks as well as network variables. See *Sharing of Configuration Properties* on page 102 for more details.

6

How Devices Communicate Using Application Messages

This chapter describes the use of application messages, which can be used in place of or in addition to network variables. The request/response mechanism, a special use of application messages, is also described. Other topics covered include preemption mode, asynchronous and direct event processing, the use of completion events with messages and with network variables, and authentication for messages.

Application messages are used for creating a proprietary interface (that is, non-interoperable) to a device. The same mechanism used for application messaging can also be used to create foreign-frame messages (for proprietary gateways) and explicitly addressed network variable messages.

Introduction to Application Messages

Application messages are used for creating a proprietary interface (that is, non-interoperable) to a device. The same mechanism used for application messaging can also be used to create foreign-frame messages (for proprietary gateways) and explicitly addressed network variable messages.

There is one interoperable use for application messages, and that is the LONWORKS file transfer protocol (LW-FTP). This protocol is used to exchange large blocks of data between devices or between devices and tools, and can also be used to implement configuration files.

As described in previous chapters, functional blocks, network variables, and configuration properties are used for creating an open interoperable interface to a device. A device interface can include both an interoperable portion and a proprietary portion. For example, a device can implement a proprietary interface for use solely during manufacturing, and an interoperable interface for use in the field.

The content of an application message is defined by a proprietary message code that is sent as part of the message. This code is followed by a variable-sized data field. The same message code can have one byte of data in one instance and 25 bytes of data in another instance.

You can use a request/response service with application messages to enable an application on one device to cause an application on another device to respond to it. The request/response mechanism is similar to a network variable poll. When a network variable is polled, the application scheduler on the polled device provides the most recent value for that network variable, without intervention of (or knowledge by) the application program. When an application message is sent with the request service, the application program on the remote device takes some action as a result of receiving the request message, and then provides a new value for its response. The request/response service can also be used to implement remote procedure calls, because it provides a way for an application on one device to cause an action on another device.

Application messages use less EEPROM table space than network variables, but performing the equivalent tasks using application messages always consumes more code space than using network variables because of the amount of support code built-in to the Neuron firmware for network variables. In addition, using application messages is a more complicated way of accomplishing such a task. You must explicitly build, send, and receive application messages. Message attributes such as service type, authentication, and priority are defined during compilation or at run-time, and are not configurable by a network tool after device installation (however, these attributes can be set on a message-by-message basis).

Layers of Neuron Software

When you use network variables in a program, the actual building and sending of messages takes place behind the scenes. This is called *implicit* messaging. As shown in **Figure 11**, three layers of software are involved: the application layer (which includes the scheduler), the network layer, and the Media Access Control (MAC) layer. Each of these layers of software corresponds to one or more layers

How Devices Communicate Using Application Messages

of the LonTalk protocol and is handled by a separate processor on a Neuron Chip or Smart Transceiver.

Only one of these layers, the application layer, can be programmed. Your program also has access to some of the information provided by the network layer through the services of the scheduler, as described later in this chapter.
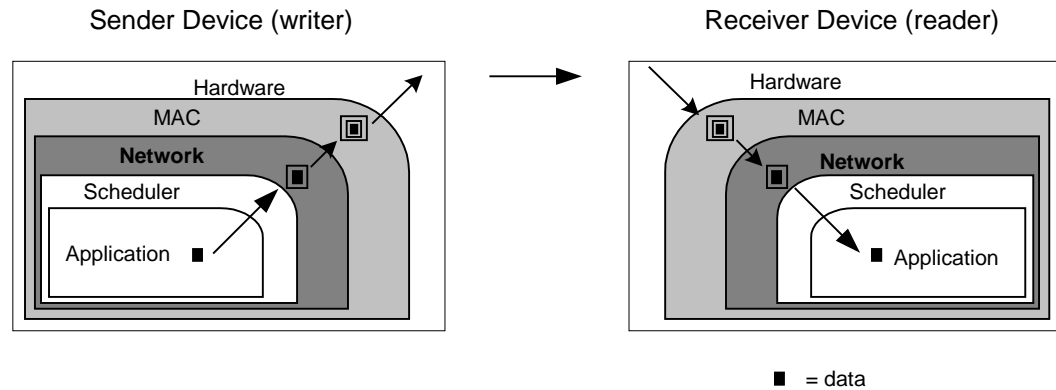
Sender Device (writer)   Receiver Device (reader)

■ = data

Figure 11. Sending a Message

Implicit Messages: Network Variables

Figure 11 illustrates what happens when a device assigns a value to an output network variable. First, the application program assigns a value to the network variable. The scheduler then builds a network variable message and passes the message to the network layer. The network layer adds addressing information to the network variable message and then passes the message to the MAC layer. The MAC layer adds more information to the network variable message, and then sends the message over the communications channel.

When a device receives the network variable message, the message is unpackaged, as follows. First, the MAC layer validates the message. The network layer then checks the addressing information contained in the message to see if it is intended for this device. If it is, it passes the network variable information to the scheduler. The scheduler then makes the new value available to the appropriate task within the application program.

These messages are referred to as *implicit messages* because they are sent and received automatically. Application messages are also referred to as *explicit messages*.

Application Messages

You can explicitly create a message using Neuron C. Rather than using the implicit messaging capability provided by network variables, you can manually construct and send a message. This type of message is called an *explicit* message. You must identify the type of this explicit message using a message code. The *message code* identifies the message as an application message, foreign-frame message, or network variable message. The following sections describe how to use the objects, functions, and events used with application messages. The request/response mechanism, a special use of application messages, is described

following the generalized description of application messages. The same mechanisms used for application messaging can also be used to create and send foreign-frame messages (for proprietary gateways) and explicitly addressed network variable messages.

You must construct an application message using a predefined message object, and then process it using function calls and predefined events. Following is a brief list of the steps described in the following sections. Objects, functions, and events are itemized for each section.

	Functional Step	*Neuron C Feature*
1	Constructing a message	**msg_out** object
2	Sending a message	**msg_send( )** function **msg_cancel( )** function
3	Receiving a message	**msg_arrives** event **msg_receive( )** function **msg_in** object
4	After sending a message with the acknowledged service	**msg_completes** event **msg_succeeds** event **msg_fails** event
5	Sending a response to a message with the request/response service	**resp_out** object **resp_send( )** function **resp_cancel( )** function **resp_arrives** event **resp_receive( )** function **resp_in** object
6	Allocating buffers explicitly	**msg_alloc( )** function **msg_alloc_priority( )** function **msg_free( )** function **resp_alloc( )** function **resp_free( )** function

Constructing a Message

You can construct an application message using the **msg_out** outgoing message object. This definition is built into Neuron C. Use the **msg_send( )** function to send the message. You can only construct one outgoing message (or response) and one incoming message (or response) at any one time. For example, you cannot build up two messages in parallel and send them both. Nor can you parse two input messages at the same time.

The msg_out Object Definition

An outgoing message is predefined as shown below:

```
typedef enum {FALSE, TRUE} boolean;
typedef enum {ACKD, UNACKD_RPT,
```

```
            UNACKD, REQUEST}  service_type;

    struct {
       boolean priority_on;    // TRUE if a priority message
                  // (default:FALSE)
       msg_tag tag;            // message tag (required)
       int     code;           // message code (required)
       int     data[MAXDATA]   // message data (default:none)
       boolean authenticated;  // TRUE if to be authenticated
                  // (default:FALSE)
       service_type service;   // service type (default:ACKD)
       msg_out_addr dest_addr; // see include file msg_addr.h
                  // (optional field)
    } msg_out;
```

priority_on   When set to TRUE, sends the message as a priority message. Specify FALSE, or do not assign to this field, if the message is not a priority message. If used, this field must be the first field set in the message object, even before the tag. The default is FALSE (that is, nonpriority).

tag   A message tag identifier for the message. This field is required. See *Message Tags* on page 128.

code   A numeric message code. This field is required. See *Message Codes* on page 129.

data   The application's data. This field is optional; a message can consist of only a message tag and message code. Because of network buffer overhead, MAXDATA must never exceed 228. MAXDATA is a function of the **app_buf_out_size** pragma (see Chapter 8, *Memory Management*, on page 185):

MAXDATA = app_buf_out_size – 6

or

MAXDATA = app_buf_out_size – 17

(if explicit addressing is used for messages or network variables in this program)

Note: The Neuron firmware observes which locations in the data array have assignments and automatically sets the length of the outgoing message accordingly.

authenticated   A TRUE value specifies that the message is to be authenticated. You can specify FALSE, or not assign to this field, if the message does not need to be authenticated. The default is FALSE (that is, not authenticated).

service   Specifies one of the following:

ACKD (the default) - acknowledged service with retries

REQUEST – request/response protocol

UNACKD - unacknowledged service

UNACKD_RPT - repeated service (message sent multiple times)

> **Note**: Do *not* use **unackd** or **unackd_rpt** in combination with authenticated messages. Use only the **ACKD** or **REQUEST** service type.

dest_addr An optional field in **msg_out** that explicitly specifies a destination address. If **dest_addr** is not set, then the message is sent to the implicit address associated with the tag, if the tag is bound. See *Explicit Addressing* on page 136 for more information.

> **Note**: To use this field, you must include the **<addrdefs.h>** and **<msg_addr.h>** files.

Message Tags

A *message tag* is a connection point for application messages. Incoming application messages are always received on a common message tag called **msg_in**, but you must declare one or more message tags if *outgoing* explicit messages are used. The incoming tag and each outgoing tag or tags can be assigned a unique network address by a network tool.

A message tag declaration can optionally include connection information. The syntax for declaring a message tag is as follows:

> **msg_tag** [*connection-info*] *tag-identifier* [**,** *tag-identifier* ...]**;**

The *connection-info* field is an optional specification for connection options, in the following form:

> **bind_info** (*options*)

The following connection options apply to message tags:

nonbind Denotes a message tag that carries no implicit addressing information and does not consume an address table entry. It is used as a destination tag when creating explicitly addressed messages.

rate_est (*const-expr***)** The estimated sustained message rate, in tenths of messages per second, that the associated message tag is expected to transmit. The allowable value range is from 0 to 18780 (0 to 1878.0 messages/second).

max_rate_est (*const-expr***)** The estimated maximum message rate, in tenths of messages per second, that the associated message tag is expected to transmit. The allowable value range is from 0 to 18780 (0 to 1878.0 messages/second).

tag-identifier A Neuron C identifier for the message tag.

It might not always be possible to determine **rate_est** and **max_rate_est**. For example, message output rates are often a function of the particular network where the device is installed. These optional values can be used by a network tool to perform network device analysis. Although any value in the range 0-18780 can be specified, not all values are used. The values are mapped into encoded values n in the range 0-127. Only the encoded values are stored in the device's self-identification (SI) data. The actual value can be reconstructed from the encoded value. If the encoded value is zero, the actual value is undefined. If the encoded value is in the range 1-127, the actual value is:

$$a = 2^{(n/8)-5}$$

rounded to the nearest tenth. The actual value, a, produced by the formula, is in units of messages per second.

You must assign a message tag to the **msg_out.tag** field for each outgoing message. This specifies which connection point (corresponds to an address table entry) to use for the outgoing message. After the tag field has been assigned, the message must be either sent or cancelled.

Besides addressing, message tags are also used for correlating completion events and responses with outgoing messages. For example, the following **when** clause correlates a message completion event with a message sent by means of the **tag1** message tag:

when (msg_completes(tag1))

By qualifying an event with a message tag, the event becomes TRUE only when an event corresponding to that particular outgoing message occurs.

Message Codes

A *message code* is a numeric identifier for a message. Each application message must include a message code that the receiving applications can use to interpret the contents of the message.

Message codes are used by all LonTalk messages, not just application messages. They fall into the ranges shown in **Table 8**. Codes 0-62 and 64-78 are for use by applications. The lower range is used for proprietary application-specific messages, and the upper range is used for proprietary application-level gateways to other networks.

Table 8. Ranges for Message Codes

Type of Message	Message Code	Description
User Application Messages	**0 to 47 (0x00..0x2F)**	Generic application messages. The interpretation of the message code is left to your application.
Standard Application Messages	**48 to 62 (0x30..0x3E)**	Standard application messages defined by LONMARK International.
Responder Offline	**63 (0x3F)**	Used by application message responses. Indicates that the sender of the response was in an offline state and could not process the request.
Foreign Frames	**64 to 78 (0x40..0x4E)**	Used by application-level gateways to other networks. The interpretation of the message code is left to the application.

Type of Message	Message Code	Description
Foreign Responder Offline	**79 (0x4F)**	Used by foreign frame responses. Indicates that the sender of the response was in an offline state and could not process the request.
Network Diagnostic Messages	**80 to 95 (0x50..0x5F)**	Used by network tools for network diagnostics.
Network Management Messages	**96 to 127 (0x60..0x7F)**	Used by network tools for network installation and maintenance.
Network Variables	**128 to 255 (0x80..0xFF)**	The lower six bits of the message code contain the upper six bits of the (14-bit) network variable selector. The first data byte contains the lower eight bits of the selector.

Example of building an application message:

```
msg_tag motor;

#define MOTOR_ON 0
#define ON_FULL 100

msg_out.tag = motor;
msg_out.code = MOTOR_ON;
msg_out.data[0] = ON_FULL;
```

Block Transfers of Data

You can use the C **memcpy( )** function to transfer blocks of message data into the **msg_out** or **resp_out** objects (see *Using the Request/Response Mechanism* on page 142). You can also use **memcpy( )** to copy a block of data from the **msg_in** or **resp_in** objects. Using this function is the only case where you can use the address of the **msg_out**, **resp_out**, **msg_in**, or **resp_in** objects.

For messages of an unknown or variable length, use **msg_in.len** or **resp_in.len**, limited by the **sizeof(**s**)** to prevent writing past the end of s.

Example block transfer of data:

```
msg_tag motor;
#define MOTOR_ON 0

typedef enum {
    MOTOR_FWD,
    MOTOR_REV
} motor_dir;

struct {
    long      motor_speed;
```

How Devices Communicate Using Application Messages

```
        motor_dir   motor_direction;
        int         motor_ramp_up_rate;
    }   motor_on_message;

    when(some_event) {
        msg_out.tag = motor;
        msg_out.code = MOTOR_ON;
        motor_on_message.motor_direction = MOTOR_FWD;
        motor_on_message.motor_speed = 500;
        motor_on_message.motor_ramp_up_rate = 100;
        memcpy(msg_out.data, &motor_on_message,
            sizeof (motor_on_message));
        msg_send();
    }
```

Sending a Message

You can send and cancel sending a message using the following functions:

msg_send( )

msg_cancel( )

The **msg_send( )** function has the following syntax:

void msg_send(void);

This function sends a message using the **msg_out** object (which must have already been constructed prior to the call to the **msg_send( )** function). It has no parameters, and has no return value.

The following code fragment illustrates sending a message:

```
    msg_tag motor;
    #define MOTOR_ON 0
    #define ON_FULL 100      // (100 percent)

    when (io_changes(switch1) to ON)
    {
        // Send a message to the motor
        msg_out.tag = motor;
        msg_out.code = MOTOR_ON;
        msg_out.data[0] = ON_FULL;
        msg_send();
    }
```

The **msg_cancel( )** function cancels an outgoing message. It has the following syntax:

void msg_cancel(void);

This function cancels the message being built for the **msg_out** object and frees the associated buffer, allowing another message to be constructed. It has no parameters, and has no return value.

If a message is constructed but not sent before the task is exited, the message is automatically canceled.

The first write operation to the **msg_out** object triggers automatic, implicit, buffer allocation. Because all buffers could be busy at that moment, the time taken to complete the first assignment is non-deterministic. For applications

where a non-deterministic waiting period is not acceptable, the **msg_alloc( )** function is supported and allows for explicit message allocation. The **msg_alloc( )** function does not wait for any pending transaction to complete, but returns immediately with success or failure. See the *Neuron C Reference Guide* for more information about this function.

Receiving a Message

You typically receive a message using the **msg_arrives** predefined event. You can also use the **msg_receive( )** function to receive a message.

The msg_arrives Event

The predefined event for receiving a message is **msg_arrives**. Its syntax is:

> **msg_arrives** [**(***message-code***)**]

If a message arrives, this event evaluates to TRUE. You can optionally qualify the event using a message code. In this case, the event is TRUE only when a message arrives containing the specified code.

When mixing unqualified **msg_arrives** events with qualified **msg_arrives** events, the **#pragma scheduler_reset** directive must be specified so that the unqualified event **when** clause is processed after all the qualified event **when** clauses.

It is essential that your program contain a default case as shown in the example below, to prevent an event queue lockup. This issue is explained in detail in *Importance of a Default When Clause* on page 134.

A sample use of this event is shown in Listing 3 below.

Listing 3. Use of **msg_arrives** Event

```
#pragma scheduler_reset

when (msg_arrives(1))
{
    io_out(sprinkler, ON);
}

when (msg_arrives(2))
{
    io_out(sprinkler, OFF);
}

// default case for handling unexpected message codes
when (msg_arrives)
{
    // Do nothing, just discard it
}
```

To prevent the incoming message queue from becoming blocked, a program that receives application messages, such as that shown in Listing 3, should contain a default **when** clause with an unqualified **msg_arrives** event as shown in the example. This is explained further in *Importance of a Default When Clause* on page 134.

The msg_receive( ) Function

The **msg_receive( )** function has the following syntax:

> **boolean msg_receive(void);**

This function receives a message into the **msg_in** object. The function returns TRUE if a new message is received, otherwise it returns FALSE.

If no message is received, this function does not wait for one. You might need to use this function to receive more than one message in a single task, for example, in bypass mode (bypass mode is also called direct event processing). If there already is a received message, it is discarded (that is, its buffer space is freed).

Calling **msg_receive( )** or **resp_receive( )** has the side-effect of calling **post_events( )**. Thus, a call to **msg_receive( )** or **resp_receive( )** defines a critical section boundary (see *Receiving a Response* on page 144).

When you use the **msg_receive( )** function, all messages are received in "raw" form, and special events such as **online**, **offline**, and **wink** can be used, but you must check for these events explicitly through message code checks. For these reasons, you cannot use **msg_receive( )** if the application program handles any special events (that is, **wink**, **online**, and **offline**).

Format of an Incoming Message

The name for the incoming message object is **msg_in**. This definition is built into Neuron C. A message is read by examining the appropriate fields in the object.

The fields of the **msg_in** object are read-only; you cannot assign values to them. An incoming message is predefined as follows:

```
typedef enum {FALSE, TRUE} boolean;
typedef enum {ACKD, UNACKD_RPT,
        UNACKD, REQUEST}   service_type;

struct  {
   int code;                 // message code
   int len;                  // length of message data
   int data[MAXDATA];        // message data
   boolean authenticated;    // TRUE if message was
                             // authenticated
   service_type service;     // service type used by sender
   msg_in_addr addr;         // see <msg_addr.h> include file
   boolean duplicate;        // the message is a duplicate
   unsigned rcvtx;           // the message's receive tx ID
} msg_in;
```

Important: Assigning values to the **msg_out** object can invalidate fields in the **msg_in** object. After receiving a message, you must examine or save any necessary fields in the **msg_in** object *before starting* to send a message.

code   A numeric message code. See *Message Codes* on page 129.

len   The length of the message data.

data   The application data. This field is valid only if **len** is greater than 0. MAXDATA is a function of the **#pragma app_buf_in_size** directive (see Chapter 8, *Memory Management*, on page 185):

MAXDATA = app_buf_in_size – 6

or,

MAXDATA = app_buf_in_size – 17

(if explicit addressing is used by any message or network variable in the program)

authenticated   This field has the value TRUE if the message was authenticated, and FALSE if the message was not authenticated.

service   The message service type; one of the following values:

ACKD - acknowledged service with retries

UNACKD - unacknowledged service

UNACKD_RPT- repeated service (message sent multiple times)

REQUEST- request/response service. When a message is sent using this service, the receiver device returns a response to the sender device, and the sender processes the response. The request/response mechanism is described later in this chapter.

addr   An optional field in the incoming message that an application program can look at to determine the source and destination of the message. You can find the definition of the **msg_in_addr** type in the **<msg_addr.h>** include file.

To use this field, you must include the **<msg_addr.h>** file.

duplicate When this boolean flag is TRUE, it indicates the message is a duplicate request message passed to the application. Duplicate request messages are passed to the application if the application response contains data beyond the one-byte message code.

rcvtx   The receive-transaction ID that the message used in the device's transaction database.

Importance of a Default When Clause

Listing 3 (in *The msg_arrives Event* on page 132) illustrates an important technique to be used with messages: *Any program that receives application messages must be prepared to receive unwanted messages and discard them.* Discards can take the form shown in Listing 3, or can be a **default** case in a **switch** statement.

If a message were to arrive and the application fail to process it, that message would remain at the head of the queue forever, blocking the arrival of any other messages or network variable events and locking up the device until it is reset. One example of a message that would be sent to all devices, most of which are not interested in the message, is the service pin message. Probably only a network tool would want to process the service pin message; all other devices need to discard the message.

If a program does not process messages (either implicitly through the use of **when(msg_arrives)** or explicitly through the use of **msg_receive( )**), the scheduler automatically discards all incoming messages.

A device that uses *only* network variables need not be concerned with this phenomenon, because the scheduler then handles *all* incoming messages.

Application Message Examples

The following example shows how lamp and switch devices could be written using application messages instead of network variables.

Lamp Program

First, here is the program for the lamp devices:

```
// lamp.nc - Generic program for a lamp
// The lamp's state is governed by an incoming
// application message

#define LAMP_ON 1
#define LAMP_OFF 2
#define OFF 0
#define ON 1

// I/O declaration
IO_0 output bit io_lamp_control;

when (msg_arrives) {
    switch (msg_in.code) {
    case LAMP_ON:
        io_out(io_lamp_control, ON);
      break;
    case LAMP_OFF:
        io_out(io_lamp_control, OFF);
      break;
    } //end switch
} //end when
```

Switch Program

Here is the program for the switch devices:

```
// switch.nc - Generic program for a switch
// Send a message when the switch changes state

#define LAMP_ON  1
#define LAMP_OFF 2
#define OFF 0
#define ON 1

// I/O Declaration
IO_4 input bit io_switch_in;

// Message tag declaration
msg_tag TAG_OUT;

// Event-driven code
when (reset) {
    io_change_init(io_switch_in);
```

```
    }

when (io_changes(io_switch_in)) {
   // Set up message code based on the switch state
   msg_out.code = (input_value == ON) ? LAMP_ON : LAMP_OFF;

   // Set up message tag and send message
   msg_out.tag = TAG_OUT;
   msg_send();
}
```

Connecting Message Tags

Every device has a default **msg_in** input message tag. Network integrators use a
network tool to connect message tags for outgoing messages to the **msg_in** input
message tag. For example, message tags for the two example devices are
connected as follows:

TAG_OUT	*connects to*	*msg_in*
on the switch device		on the lamp device

Explicit Addressing

You can explicitly specify a destination address for application messages and
network variables using the data structures in the **<msg_addr.h>** and
<addrdefs.h> include files. To use explicit addressing for outgoing messages,
you must assign appropriate values to all applicable fields of one of the elements
of the **dest_addr** union in the **msg_out** object, prior to calling **msg_send( )**. The
message still needs a message tag, although no addressing information is derived
from the message tag. Thus, no matter how the message tag is bound, explicit
addressing overrides the address specified by the tag.

When you assign an explicit destination address, the message tag is only relevant
for correlation with response and completion event processing. However, if you
use a standard message tag, you still consume an address table entry, even if you
only use the message tag for explicitly addressed messages. To permit a more
optimal use of Neuron resources, use *non-bindable* message tags that carry no
addressing information and do not consume an address table entry. Use the
following syntax to declare a non-bindable message tag:

> **msg_tag bind_info(nonbind** [**,** *other-info*]**)** *tag-name***;**

See *Message Tags* on page 128 for a more detailed discussion of the nonbind
option.

The use of explicit addressing has an effect on the buffer sizes needed by the
Neuron firmware. See **Table 14** on page 210 for more detailed information.

You can send network variable updates using explicit addressing by creating an
explicit message that corresponds to a network variable update and explicitly
setting the destination address. See the *ISO/IEC 14908-1 Control Network
Protocol* standard for a description of the explicit message format of a network
variable update and for more information on addressing.

Sending a Message with the Acknowledged Service

When a device sends a message using the acknowledged service (the default), all receiver devices must acknowledge receipt of the message to the sender device. As shown in **Figure 12**, the network processor is responsible for sending the acknowledgment. This acknowledgment message contains no data and is sent to the network processor on the device where the message originated.

The application layer plays no part in the acknowledgment of a message. How then does a program ever learn whether a message has succeeded or failed? The following section answers this question.
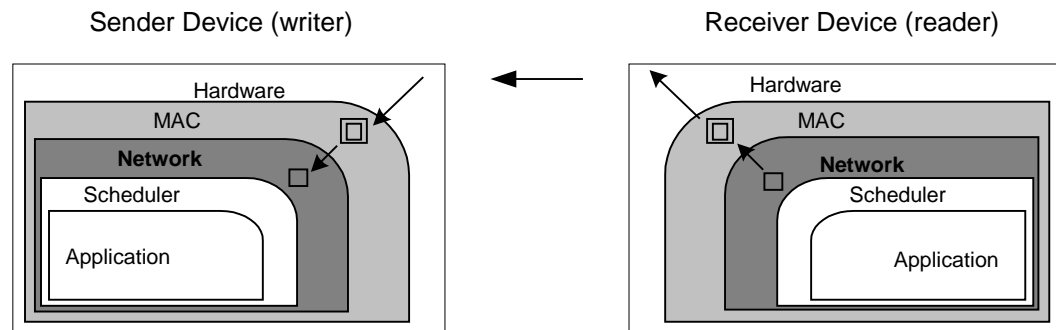
Sender Device (writer)   Receiver Device (reader)

Figure 12. Acknowledging a Message

Message Completion Events

The following events can be used by the sender device to check message completion status:

> **msg_completes** [**(***msg-tag-name***)**]
>
> **msg_succeeds** [**(***msg-tag-name***)**]
>
> **msg_fails** [**(***msg-tag-name***)**]

All three events can be qualified by a message tag name. If unqualified, the event applies to any message.

When using an unqualified message completion event, the built-in variable **msg_tag_index** can be used to determine which message tag was responsible for the event. See the *Predefined Events* chapter in the *Neuron C Reference Guide* for more information.

The **msg_completes** event is the most general event. When an outgoing message completes (that is, succeeds *or* fails), this event evaluates to TRUE.

The **msg_succeeds** event evaluates to TRUE when a message is successfully sent. The **msg_fails** event evaluates to TRUE when a message fails to be sent (after all retry attempts). See **Table 9** for a more precise breakdown of what "success" and "failure" mean for each service type. For a given message, only *one* of these events evaluates to TRUE. The order of processing is thus important. If a **msg_completes** event is processed before the **msg_succeeds** and **msg_fails** events, the **msg_succeeds** and **msg_fails** events never evaluate to TRUE.

Note: See also *Comparison of resp_arrives and msg_succeeds* on page 147.

These events are primarily of interest when you send a message with either the acknowledged service or the request/response service (see *Using the Request/Response Mechanism* on page 142). If you send a message with the unacknowledged or repeated service, the **msg_succeeds** and **msg_completes** events are *always* TRUE as soon as the message is transferred from the network processor to the Media Access Control (MAC) processor on the sender device.

Table 9. Success/Failure Completion Events

Service Used	SUCCESS =	FAIL=
Unacknowledged	Message is transmitted to MAC processor.	See Note.
Repeated	*N* messages are transmitted to MAC processor. (*N* is the number of repeats.)	See Note.
Acknowledged	All acknowledgments have been received by the network processor on the sender device.	One or more acknowledgments are not received. This applies to both messages and network variables.
Request/Response	All responses have been received by the application processor on the sender device.	*For a message:* One or more of the responses did not arrive. *For a network variable poll:* (a) One or more of the responses did not arrive. (b) None of the responses had valid data.

Note: In all cases, if the Neuron firmware encounters an addressing error, a failure event occurs (see the *Neuron C Reference Guide*). If a network variable or message is unbound, a success event occurs.

Processing Completion Events for Messages

When you send a message, you can optionally check the completion event. Several restrictions apply, however, if you do check the completion event.

First, if you check for either **msg_succeeds** *or* **msg_fails**, you must check for *both* events. The alternative is simply to check for **msg_completes**.

Second, if you qualify a completion event with a particular message tag, then you must *always* process completion events for that message tag. A program can thus process completion events for some of its message tags, and ignore completion events for other message tags. In the following example, completion

events for TAG1 are processed, and completion events for TAG2 are *not* processed:

```
when (io_changes(dev1))
{
    ...
    msg_out.tag = TAG1;
    ...
    msg_send();
}

when (msg_completes(TAG1))
{
    ...
}

when (io_changes(dev2))
{
    ...
    msg_out.tag = TAG2;
    ...
    msg_send();
}
```

A third restriction applies to use of the *unqualified* completion event, which implicitly refers to *all* messages. When you use the unqualified completion event, you must process all acknowledged messages, either explicitly for each message tag, or implicitly through use of an unqualified event each time a message is sent.

The following code shows correct processing of completion events by message tag:

```
int failures[2], success;
msg_tag TAG1, TAG2;

when (io_changes(toggle))
{
    msg_out.tag = TAG1;
    msg_out.code = TOGGLE_STATE;
    msg_out.data[0] = input_value;
    msg_send();

    msg_out.tag = TAG2;
    msg_out.code = TOGGLE_STATE;
    msg_out.data[0] = input_value;
    msg_send();
}

when (msg_fails(TAG1))
{
    failures[0]++;
}

when (msg_fails(TAG2))
{
    failures[1]++;
}

when (msg_succeeds)      // any message qualifies
```

```
{
    success++;
}
```

Preemption Mode and Messages

The Neuron firmware enters *preemption mode* when there is no application
buffer available for an outgoing message. If the system needs a free application
buffer, it causes the application program to wait and processes only completion
events, responses, and incoming network variables and messages to facilitate
application buffers' becoming free.

No other predefined or user-defined events are processed unless the
preempt_safe keyword is used in conjunction with a **when** clause containing an
event expression. The syntax for the **when** clause is explained in Chapter 2,
Focusing on a Single Device, on page 17.

The watchdog timer is automatically updated during this wait. If the program
waits for more than a configurable number of seconds, the device is reset. This
configurable timer is controlled by the **preemption_timeout** field of the
configuration structure (**config_data_struct**), which you can modify using the
update_config_data() function; see appendix B of the ISO/IEC 14908-1 *Control
Network Protocol* specification for information about this structure. A *buffer wait
timeout* (also known as *preemption mode timeout*) should only occur if a device is
totally blocked from transmitting. This situation could occur under extreme
network congestion or with certain network failures.

A buffer wait timeout could also occur if a program is not properly freeing
completion events. The most common error is to check for completion events in
bypass mode (for example, **if (nv_update_completes)**) and not to have a
corresponding completion event check in a **when** clause.

With network variables, the system can only enter preemption mode if:

- synchronous output network variables are updated, or

- **flush_wait( )** is called.

When the system is in preemption mode, further attempts to send a message
from a task associated with a message completion event **when** clause cause a
device reset if no buffer is available for the new message.

The following sequence is therefore *not* recommended:

```
when (TOGGLE_ON)
{
    // build a message
    // send the message
}

when (msg_completes)
{
    msg_out.tag = t;      // This sequence is not
                          // recommended.
    msg_out.code = 1;     // Causes a device reset
                          // if the system is
                          // already in preemption
                          // mode
}
```

Instead of using this sequence, build messages and call **msg_send( )** in a task with a **when** clause that does not use the **msg_completes** event. When you update synchronous output network variables, preemption mode is entered at the critical section boundary if there are insufficient application output buffers to accommodate the updates. For example, if you update three synchronous output network variables in a critical section and only two application output buffers are available, preemption mode is entered upon leaving the critical section. The application leaves preemption mode and returns to normal operation after all the outstanding network variable updates are buffered.

When implicit buffer allocation is used (that is, building an explicit message without calling **msg_alloc( )** first), then preemption mode is entered upon the first assignment to **msg_out** if no application output buffer is available. Preemption mode ends as soon as a buffer becomes available (that is, when a completion event is processed). While a device is in preemption mode, <u>no</u> outgoing network variable updates occur, priority or otherwise. Thus, a program that expects priority updates to occur within a bounded amount of time should <u>not</u> use nonpriority synchronous network variables or messages with implicit buffer allocation.

To allocate and free buffers explicitly, use the functions described in *Allocating Application Buffers* on page 149.

You can detect whether or not a program is already in preemption mode with use of the following function:

> **boolean preemption_mode (void);**

This function returns TRUE if the device is in preemption mode. The application must include the **<status.h>** file to use this function.

Asynchronous and Direct Event Processing

You can check events using **when** clauses and events such as **when (msg_completes)**, **when (msg_fails)**, and **when (msg_succeeds)**. This type of event processing is referred to as *asynchronous processing*, because the scheduler handles the exact order of execution. An alternate technique is *direct event processing*, in which you check completion events inside tasks, with **if** and **while** statements.

The following example indicates one way asynchronous and direct processing *cannot* be combined. Do not include message completion events in a task associated with a message completion event clause:

```
when (msg_completes)
{
    post_events();
    if (msg_completes)    // not recommended
        x = 4;
}
```

You can use asynchronous event processing in programs that also do direct event processing. Asynchronous event processing is the typical method for processing events. This method results in smaller application programs. You should call the **flush_wait( )** function before the transition from asynchronous to direct event processing. The **flush_wait( )** function ensures that all outstanding

completion events and response events are processed before switching to direct event processing.

Here is an example of sending a message and processing the completion event directly (that is, checking the event inside a task rather than inside a **when** clause):

```
msg_tag motor;
#define MOTOR_ON 0

when (x==3)
{
    // send a message
    flush_wait();
    msg_out.tag = motor;
    msg_out.code = MOTOR_ON;
    msg_send();

    // check completion status
    while (!msg_succeeds(motor)) {
        post_events();
        if (msg_fails(motor))
            node_reset();
    }
}
```

Using the Request/Response Mechanism

Request/response messages provide a mechanism for an application running on one device to request data from an application running on another device. The request/response mechanism is used automatically by the Neuron firmware to poll input network variables and can also be used by application programs that use application messaging.

A *request* is a message that uses the request service. Sending a request message is similar to polling a network variable. A poll receives the most recent value from the scheduler for a particular network variable. A request, in contrast, can force the application on the responding device to evaluate the request *at the time of the request* and then send back a response.

Important: Because an Interoperable Self-Installation (ISI) network uses unbounded groups (group size 0), your ISI-enabled application should not poll network variable values. Using a request/response service with unbounded groups can significantly degrade network performance.

The functions, events, and objects for constructing, sending, and receiving responses are analogous to those for constructing, sending, and receiving messages, described in the previous section. They are also summarized in the following paragraphs.

An example of sending a request is the following:

```
msg_tag motor;
#define MOTOR_STATE 1

when (io_changes(switch1) to 0)
{
    //send a request to the motor
```

```
            msg_out.tag = motor;
            msg_out.service = REQUEST;
            msg_out.code = MOTOR_STATE;
            msg_send();
        }
```

The request is packaged as shown in **Figure 11** on page 125. The application
program on the receiver device receives the request through a **when** clause (or
msg_receive( ) function) and must then formulate a response to this request, as
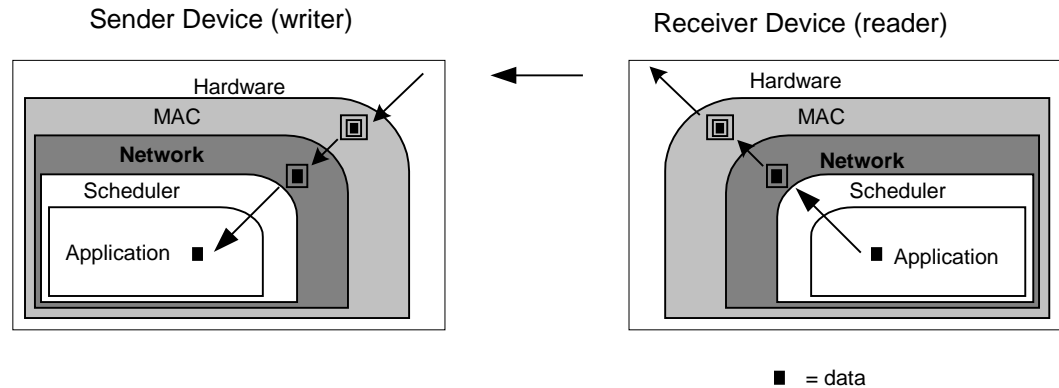shown in **Figure 13**.

Figure 13. Sending a Response

Constructing a Response

You can construct a response to a request message. As shown in **Figure 13**, the
response contains a data portion that is sent to the application processor of the
sender device. A response is different from an acknowledgment (**Figure 12** on
page 137), which does not contain a data portion and is sent only to the network
processor on the sender device.

The name of the outgoing response object is **resp_out**. The response inherits its
priority and authentication designation from the request to which it is replying.
Because the response is returned to the origin of the request, no message tag is
necessary. For the same reason, you cannot explicitly address a response.

The built-in outgoing response object is defined as shown below:

```
    struct {
        int code;                // message code
        int data[MAXDATA];       // message data
    } resp_out;
```

code   A numeric message code in the range 0 to 79. This field is required.
          See *Message Codes* on page 129 for a detailed description of numeric
          ranges used in the code field.

data   The data contained in the message. This field is optional.
          MAXDATA is a function of the **#pragma app_buf_in_size** directive
          (see Chapter 8, *Memory Management*, on page 185):

          MAXDATA = app_buf_in_size − 6

or
MAXDATA = app_buf_in_size – 17
(if explicit addressing is used)

Note: The Neuron firmware observes which locations in the data array have assignments and automatically sets the length of the outgoing message accordingly.

Sending a Response

You can send a response with the **resp_send( )** function. You must send responses from the same critical section that processed the incoming request. The response is constructed in the application input buffer in which the request arrived. Therefore, after you start response construction, you can no longer examine the incoming request. Also, no other intervening messages can be sent or received. This restriction only applies to the case in which an outgoing message uses an input application buffer.

The syntax for the **resp_send( )** function is the following:

> **void resp_send (void);**

This function sends a response using the **resp_out** object.

Note: While the response is constructed in the application input buffer by the application, the network processor uses a network output buffer to construct the response packet. So, the network output buffer must be sized to accommodate outgoing responses in addition to other outgoing messages.

Receiving a Response

A program usually receives a response through the predefined event **when(resp_arrives)**. The **resp_receive( )** function can also be used to receive a response.

The resp_arrives Event

The predefined event for receiving a response is **resp_arrives**.

Its syntax is the following:

> **resp_arrives** [**(***msg-tag-name***)**]

If a response arrives, this event evaluates to TRUE. The event can optionally be qualified by a message tag name; this qualification limits the event to a response message that corresponds to a previously sent request that used the named message tag. When there is no message tag name qualifying the event, the event evaluates to TRUE for each response message that arrives.

The resp_receive( ) Function

The **resp_receive( )** function has the following syntax:

> **boolean resp_receive(void);**

This function receives a response into the **resp_in** object. The function returns TRUE if a response is received, otherwise it returns FALSE. The response is automatically discarded at the end of the task that receives it.

Calling **resp_receive( )** has the side-effect of calling **post_events( )**, so a call to **resp_receive( )** defines a critical section boundary.

Format of a Response

The name of the incoming response object is **resp_in**.

The incoming response structure is predefined in the Neuron C Compiler as follows:

```
struct {
    int code;             // message code
    int len;              // length of message data
    int data[MAXDATA];    // message data
    resp_in_addr addr;    // explicit address - see the
                          // <msg_addr.h> include file
} resp_in;
```

code   A numeric message code in the range 0 to 79. See *Message Codes* on page 129.

len   The length of the message data.

data   The data contained in the message. This field is valid only if **len** is greater than 0. MAXDATA is a function of the **#pragma app_buf_in_size** (see Chapter 8, *Memory Management*, on page 185):

MAXDATA = app_buf_in_size – 6

or

MAXDATA = app_buf_in_size – 17

(if explicit addressing is used by any message or network variable in this program)

addr   An optional field in the incoming message that an application program can use to determine the source and destination of the message. You can find the definition of the type **resp_in_addr** in the **<msg_addr.h>** include file.

To use this field, you must include the **<addrdefs.h>** and **<msg_addr.h>** files.

Request/Response Examples

This example shows sending a request and asynchronously receiving the responses. The code for receiving this request and responding to it follows in the next example.

```
msg_tag  tag1;
#define DATA_REQUEST 0

when (io_changes(toggle))
{
    msg_out.tag = TAG1;
```

```
    msg_out.code = DATA_REQUEST;
    msg_out.service = REQUEST;
    msg_send();
}

when (resp_arrives(TAG1))
{
    if (resp_in.code == OK)
        process_response(resp_in.data[0]);
}
```

Here is the code for the responder to this request:

```
#define DATA_REQUEST 0
#define OK 1

when (msg_arrives(DATA_REQUEST))
{
    int x, y;
    x = msg_in.data[0];
    y = get_response(x);

    resp_out.code = OK;
        // msg_in no longer available

    resp_out.data[0] = y;
    resp_send();
}
```

The following example shows sending a request and receiving the responses directly:

```
int x;
msg_tag motor;
#define MOTOR_ON 0
#define DO_MOTOR_ON 3

when (command == DO_MOTOR_ON)
{
    // send a request
    msg_out.tag = motor; // construct the message
    msg_out.code = MOTOR_ON;
    msg_out.service = REQUEST;
    msg_send(); // send the message

    // wait for completion
    while (!msg_succeeds(motor)) {
        post_events();
        if (msg_fails(motor))
            node_reset();
        else if (resp_arrives(motor)) {
            x = x + resp_in.data[0];
            resp_free();            // optional
        }
    }
}
```

Comparison of resp_arrives and msg_succeeds

You can use both **resp_arrives** and the completion events (**msg_succeeds**, **msg_fails**, and **msg_completes**) for the same request transaction because these events give you different information. The following example illustrates this difference.

Suppose you send one request to six devices using multicast (group) addressing. Three of the responses are received and three are not received. In this case, the **resp_arrives** event is TRUE three times, once for each time a response arrives. The **msg_succeeds** event never becomes TRUE, because some of the responses did not arrive. The **msg_fails** event becomes TRUE when the allotted time for all responses to arrive is exceeded. In other words, for **msg_succeeds** to be TRUE, *all* of the responses must be received.

Response arrival events always occur *before* the message completion events (**msg_completes**, **msg_fails**, or **msg_succeeds**).

Idempotent Versus Non-Idempotent Requests

An *idempotent* transaction is one that can be safely repeated. For example, the command "turn on the light" can be sent repeatedly without changing the end effect (the light goes on).

A *non-idempotent* transaction cannot be safely repeated without changing the meaning. The command "turn up the volume by 10%" is an example of a non-idempotent message. Responding to it ten times is *not* the same thing as responding to it once.

LonTalk messages do not include an "idempotent" attribute. Instead, the receiving device infers the attribute through the lack or existence of application data in the response to the request.

If a response does not contain application data, the Neuron firmware assumes its request is non-idempotent and cannot be safely repeated to the application. In this case the firmware sends the original response to any repeated requests, and does not forward the repeated request to the application. This firmware feature simplifies application processing for responses without data because the application does not have to test for duplicate messages.

If a response does contain application data, the Neuron firmware assumes its request is idempotent and can be safely repeated. In this case the application sends any repeated requests to the application and the application must regenerate the response. This provides the opportunity for the application to update the response to a repeated request. If the application wants to treat these repeated request messages as non-idempotent, it can do so by buffering responses by receive transaction index and re-issuing those responses when duplicate requests arrive. An example is shown below.

Example:

```
#define OK 1
#define MAXRESP 10

struct RespBuffer {
    int code;
    unsigned int len;
```

```
        int data[MAXRESP];
    } resp_buffer[16];

when (msg_arrives) {
    struct RespBuffer *buf_p;

    if (msg_in.service == REQUEST) {
        buf_p = &resp_buffer[msg_in.rcvtx];
        if (!msg_in.duplicate) {
            int i;

            // Process initial request
            // . . .

            // Now save response
            buf_p->code = OK;
            buf_p->len = MAXRESP;
            for (i=0; i<MAXRESP; i++) {
                buf_p->data[i] = get_resp_data();
            }
        }

        // Generate the response.
        resp_out.code = buf_p->code;
        memcpy(resp_out.data, buf_p->data, buf_p->len);
        resp_send();
    }
}
```

The above example also shows that the **rcvtx** field of the **msg_in** object specifies the receive transaction index to which the request belongs.

Application Buffers

You can set the number of incoming and outgoing buffers available for use by a Neuron C application during compilation. The defaults for all models of the Neuron Chip and the Smart Transceivers, except the Neuron 3120 Chip and the Neuron 3120E1 Chip, are:

- Two priority application output buffers
- Two nonpriority application output buffers
- Two application input buffers

See Chapter 8, *Memory Management*, on page 185, for a discussion of buffer allocation. For most efficient response, set the number of application input buffers to equal the expected number of responses. If a disproportionately large number of responses (for example, more than 10) are expected for the same request, some responses might never be received if only a limited number of application input buffers are available.

Notes:

- Because of limited memory on the Neuron 3120 Chip and the Neuron 3120E1 Chip, if your program is linked for these chips, the linker adjusts the output buffer defaults to one priority and one nonpriority buffer. The number of input buffers still defaults to 2.

- The same pool of buffers is used for processing both incoming messages and responses. If you are processing events directly (that is, bypassing the services of the scheduler), be sure to check for messages as well as responses so that messages are processed and application buffers are freed up regularly.

Allocating Application Buffers

Normally, when an application builds a message, an application output buffer is allocated automatically by the Neuron firmware, and when the application leaves the critical section, any outstanding application output buffer is freed by the firmware automatically.

The following functions allow you to allocate and free application buffers explicitly:

boolean msg_alloc (void);

boolean msg_alloc_priority (void);

void msg_free (void);

A message travels along one of two paths: the priority path or the nonpriority path. As its name suggests, the priority path has precedence over the nonpriority path. Thus, if you allocate an application output buffer out of the priority buffer pool, the message is more likely to succeed on a congested network.

The **msg_alloc( )** and **msg_alloc_priority( )** functions return TRUE if a **msg_out** object can be allocated. Otherwise, these functions return FALSE. A program needs to call one of these functions if it does not want to wait for an application output buffer. If the function returns FALSE, the program can choose to do something else, then try again later.

The **msg_alloc_priority( )** function allocates a priority application output buffer. The **msg_alloc( )** function allocates a nonpriority application output buffer. If you are using the system default, up to two buffers of each type can be in use at the same time. You can allocate up to the maximum number of buffers set by the **#pragma app_buf_out_count** and **#pragma app_buf_out_priority_count** directives. See Chapter 8, *Memory Management*, on page 185, and also the *Neuron C Reference Guide* for more information on these directives.

The **msg_free( )** function frees the **msg_in** object. You do not normally need to free an application input buffer, because this is done for you when you exit a task. However, you might want to free an application input buffer explicitly if you are finished with it in a task, but you have more work to do before exiting the task.

Normally, you allocate an application output buffer by assigning a value to one of the fields of the **msg_out** object. In the event that an application buffer is not available, application processing is suspended (preemption mode) until one is available. If you want to avoid possibly suspending processing, use the **msg_alloc( )** function. If no application output buffer is available, a FALSE value is returned, and processing continues. This allows the application to do something else if there are no outgoing application buffers available, rather than stopping to wait for an application buffer.

An application input buffer is normally freed at the end of the critical section in which the **msg_receive( )** call is made. The application can choose to free the application buffer directly by calling **msg_free( )**. After this call, the **msg_in** object no longer contains the received message, but the network processor is able to use the freed application input buffer for another incoming message. The **msg_alloc( )** and **msg_free( )** functions are unlike standard memory allocation functions. An application output buffer allocated by the **msg_alloc( )** function is not freed by a matching call to the **msg_free( )** function. Instead, a **msg_send( )** or **msg_cancel( )** call automatically frees an output buffer allocated by **msg_alloc( )**, and a **msg_free( )** call automatically frees an input buffer allocated by **msg_receive( )**.

The analogous functions for allocating and freeing responses are:

boolean resp_alloc (void);

void resp_free (void);

The following example shows a task that creates two messages. Before creating and sending each message, the code checks buffer availability with **msg_alloc( )**.

```
msg_tag motor1;
msg_tag motor2;
#define MOTOR_ON 0

when (x == 2)
{
    if(msg_alloc() == FALSE)
        return;

    msg_out.tag = motor1;
    msg_out.code = MOTOR_ON;
    msg_send();

    if(msg_alloc() == FALSE)
        return;

    msg_out.tag = motor2;
    msg_out.code = MOTOR_ON;
    msg_send();
}
```

UDP Packets and Neuron C Applications

Introduction

Series 6000 devices support sending and receiving UDP packets. UPD packets are presented to a Neuron C application as normal application messages with a code set to 0x4f and the data portion of the message consisting of a 12 byte addressing header followed by the UDP payload.

UDP packets are connectionless, meaning the Neuron C application does not need to create a "socket" to send or receive UDP packets. Instead, the application

How Devices Communicate Using Application Messages

can simply send UDP messages using the proper message code and format, and receives messages with that same format.

Message Format

UDP packets appear to the Neuron application as unacknowledged Neuron application messages with the following format:

- Message code 0x4f

- a 4 byte source IP address

- a 2 byte source port number

- a 4 byte destination address

- a 2 byte destination port number

- a variable length UDP payload.

The following definitions may be used when accessing UDP packets from a Neuron C application:

```
 // This message code is used by the LonTalk application to send
 // and receive UDP messages.  They are presented as LonTalk
 // application messages using the UdpAppMsgHdr, followed by the
 // UDP payload.
#define UDP_APP_MSG_CODE            0x4f
     // The size of an IPV4 address, in bytes
#define UDP_APP_IP_ADDR_SIZE        4
typedef struct UdpAppMsgHdr
{
    unsigned int  sourceAddress[UDP_APP_IP_ADDR_SIZE];
    unsigned long sourcePort;     // The source UDP port
    unsigned int  destAddress[UDP_APP_IP_ADDR_SIZE];
    unsigned long destPort;       // The destination UDP port.
} UdpAppMsgHdr;
```

Sending a UDP packet

To send a UDP packet, the Neuron C application must send an unacknowledged broadcast packet with message code 0x4f and the message data consisting of the UdpAppMsgHdr followed by a UDP payload. The source IP address of the packet may be omitted. The following example function sends a UDP packet:

```
msg_tag bind_info(nonbind) explictMsgTag;

#include <stddef.h>
#include <addrdefs.h>
#include <msg_addr.h>

/* Send a UDP packet
   Parameters:
       sourcePort:  The source UDP port number
       destAddress: Pointer to the 4 byte designation IP
address
       destPort:    The destination UDP port number
```

```
      pPayload:      Pointer to the data to be sent
      payloadLen:    The length of the payload, in bytes
*/
void sendUdpPacket(unsigned long sourcePort,
                   unsigned int *destAddress, unsigned long
destPort,
                   void *pPayload, int payloadLen)
{
    msg_out.tag = explictMsgTag;
    msg_out.code = UDP_APP_MSG_CODE;
    msg_out.service = UNACKD;

    // Set the destination address to domain wide broadcast.
Note
    // that the Neuron system image firmware will typically
modify
    // the destination address (based on destAddress) when it
    // generates the compressed UDP packet to send on the
link.
    msg_out.dest_addr.bcast.type = BROADCAST;
    msg_out.dest_addr.bcast.domain = 0;
    msg_out.dest_addr.bcast.backlog = 0;
    msg_out.dest_addr.bcast.rpt_timer = 0;
    msg_out.dest_addr.bcast.retry = 0;
    msg_out.dest_addr.bcast.tx_timer = 0;
    msg_out.dest_addr.bcast.subnet = 0;

   // Fill in the UDP header
   memcpy(&msg_out.data[offsetof(UdpAppMsgHdr, sourcePort)],
          &sourcePort, sizeof(sourcePort));
   memcpy(&msg_out.data[offsetof(UdpAppMsgHdr, destAddress)],
          destAddress, UDP_APP_IP_ADDR_SIZE);
   memcpy(&msg_out.data[offsetof(UdpAppMsgHdr, destPort)],
          &destPort. sizeof(destPort));
   // Fill in the UDP payload
   memcpy(&msg_out.data[sizeof(UdpAppMsgHdr)], pPayload,
          payloadLen);
```

```
    msg_send();

}
```

Note that the message format sent on the link is different than the format shown when presented by the Neuron application. The format on the link uses the LonTalk/IP over FT compressed UDP packet format, which includes a 1 byte compression header followed by a compressed UDP packet header. Neuron system image firmware also updates the LonTalk destination address based on the destination IP address, typically allowing both source and destination IP addresses to be elided from the packet on the link.

Receiving a UDP packet

UDP packets are received like any other application message (typically in a when msg_arrives clause). There is no need to open a socket for a specific port. The destination port is included in the UdpAppMsgHdr header and may be used by the application to determine what, if any, processing should be performed.

The following function illustrates the application parsing a UDP packet that it receives:

```
/* Receive a UDP packet.
   Parameters:
      pPayload:           Pointer to payload buffer
      payloadBufferLen:   Size of payload buffer
      pUdpHeader:         Optional pointer to store the
incoming                                    UDP header
      Return:             The length of the UPD payload
 */
int receiveUdpPacket(void *pPayload, unsigned int
payloadBufferLen,
                     UdpAppMsgHdr *pUdpHeader)
{
   int payloadLen = 0;

    if (msg_in.code == UDP_APP_MSG_CODE &&
        msg_in.len >= sizeof(UdpAppMsgHdr) &&
        payloadBufferLen >= (msg_in.len -
sizeof(UdpAppMsgHdr)))
    {
        payloadLen = msg_in.len - (unsigned
short)sizeof(UdpAppMsgHdr);
        memcpy(pPayload, &msg_in.data[sizeof(UdpAppMsgHdr)],
            payloadLen);
        if (pUdpHeader != NULL)
        {
            memcpy(pUdpHeader, msg_in.data,
sizeof(UdpAppMsgHdr));
        }
    }
    return payloadLen;
}
```

The next example uses the `receiveUdpPacket` and `sendUdpPacket` functions to illustrate how an application can receiving a UDP packet on a specific destination port and echo the packet back to the sender on the source port.

```
far unsigned int udpPayload[255];
UdpAppMsgHdr udpHeader;

// If the sample application recieves a UDP packet on port
// UDP_TEST_ECHO_REQUEST_PORT will send the UDP packet back to
the
// sender
#define UDP_TEST_ECHO_REQUEST_PORT 10000

when (reset)
{
}

#pragma scheduler_reset
when (msg_arrives(UDP_APP_MSG_CODE))
{
    int payloadLen = receiveUdpPacket(udpPayload,
                                      (unsigned
int)sizeof(udpPayload),
                                      &udpHeader);

    if (payloadLen != 0 &&
        udpHeader.destPort == UDP_TEST_ECHO_REQUEST_PORT)
    {
        // Received a request to echo the packet back.
        sendUdpPacket(udpHeader.destPort,
                      udpHeader.sourceAddress,
udpHeader.sourcePort,
                      udpPayload, payloadLen);
    }
}

when (msg_arrives)
{
}
```

Reserved UDP Port Numbers

The Neuron system image reserves the following UDP ports which cannot be used for application UDP messaging:

68:	Reserved for DHCP client
161:	Reserved for SNMP client
2541	Reserved for LonTalk/IP packets

7

Additional Features

This chapter describes additional features in Neuron C. It describes the scheduler reset mechanism in more detail. In special cases requiring a scheduling algorithm different from that of the Neuron firmware scheduler, you might want a program to run in bypass mode and use the **post_events( )** function, also described here. Other topics discussed in this chapter include interrupts, sleep mode, error handling, and status reporting.

The Scheduler

Chapter 2, *Focusing on a Single Device*, on page 17, introduced the basic functioning of the Neuron firmware scheduler, shown in **Figure 14** on page 159. Priority **when** clauses are executed in the order specified every time the scheduler runs. If any priority **when** clause evaluates to TRUE, its task is run and the scheduler starts over. If none of the priority **when** clauses evaluates to TRUE, then a nonpriority **when** clause is evaluated, selected in a round-robin fashion. If the **when** clause is TRUE, its task is executed. If the nonpriority **when** clause is FALSE, its task is ignored. In either case, the scheduler returns to the top of the loop.

Interrupt tasks do not use the scheduler; see *Interrupts* on page 165 for more information.

Scheduler Reset Mechanism

The scheduler reset mechanism is normally disabled. When the reset mechanism is enabled, the round-robin part of the scheduler is reset to the first regular **when** clause whenever one of the following conditions is detected:

- A new network variable update is at the head of the queue.

- A new timer has expired.

- A new message is at the head of the queue.

Although these events can occur at any time, the scheduler recognizes them only when it is at the beginning of the scheduling loop (labeled "Top of Scheduling Loop" in **Figure 14**).

If you disable the reset mechanism, nonpriority **when** clauses are evaluated in the order in which they appear. When the last nonpriority **when** clause is reached, the scheduling loop returns to the first nonpriority **when** clause.

If ordering of **when** clauses is desired, you can turn on the reset mechanism with the following compiler directive:

#pragma scheduler_reset

Important: If the scheduler reset mechanism is enabled, then there is a risk that **when** clauses later in the program might never execute if the scheduler is reset too frequently. This can lead to application buffer starvation, because network variable and message processing tasks can only process buffers in the order in which they arrive. Thus, if the scheduler reset mechanism is enabled, you must ensure that the **when** clauses and tasks of your program are ordered such that the most frequently executed ones are last, or the rare ones are declared using the **priority** keyword.
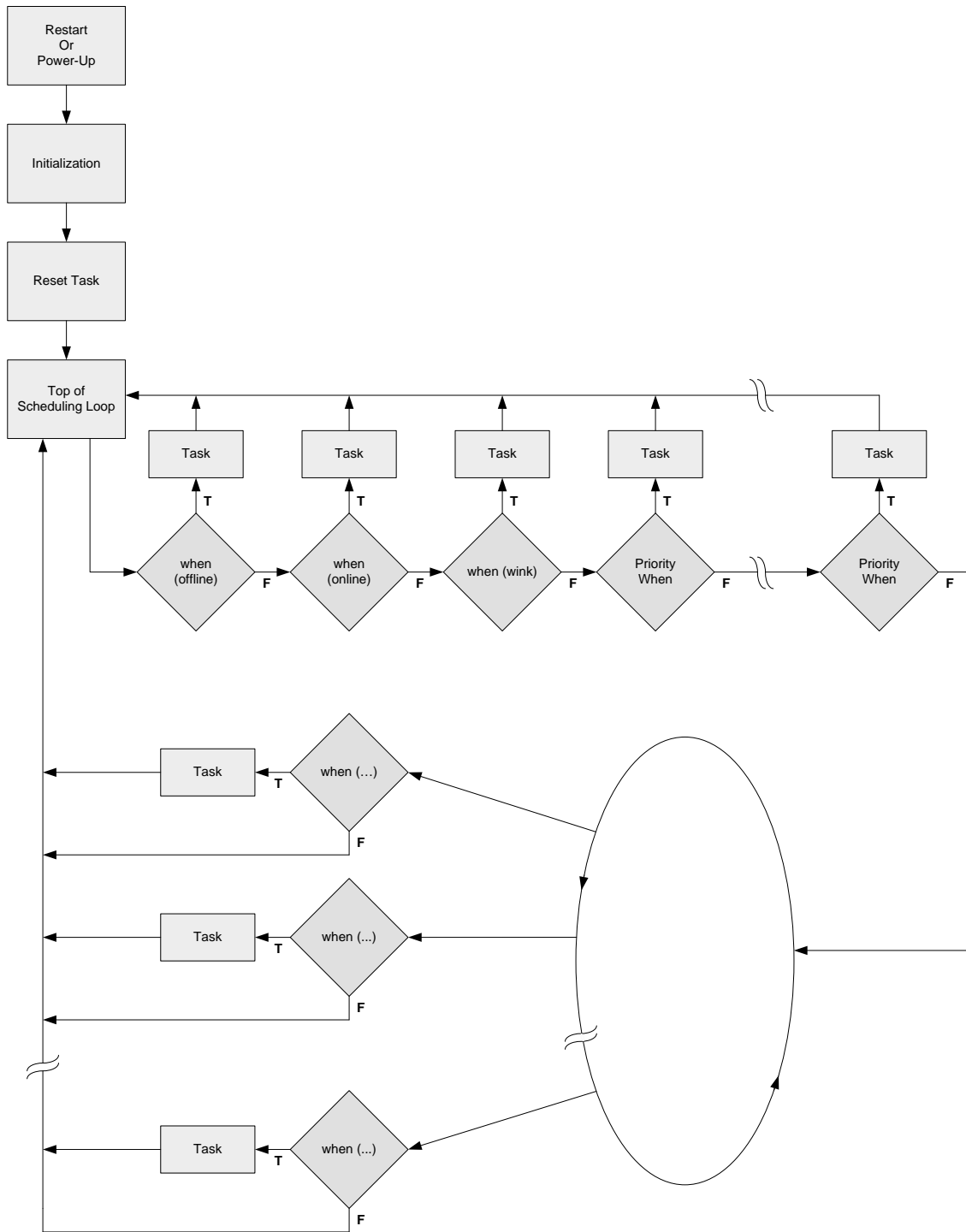
Figure 14. Neuron Firmware Scheduling of Nonpriority and Priority When Clauses

Note that application interrupts, supported by Series 5000 and 6000 chips, execute asynchronously, and are not governed by the scheduler. See *Interrupts* on page 165 for more information.

Incoming messages and network variable updates use application input buffers, which are processed by **when(msg_arrives)** and **when(nv_update_occurs)** tasks, respectively.

Outgoing messages, network variable updates, and network variable polls use application output buffers. If you check for completion events with any of the following tasks, then the application output buffers are automatically processed and freed by the scheduler in correspondence with these completion event tasks:

when (nv_update_completes)

when (nv_update_succeeds)

when (nv_update_fails)

when (msg_completes)

when (msg_succeeds)

when (msg_fails)

If there is no corresponding completion event task, then the output buffer is freed automatically by the scheduler when its corresponding event is discarded.

In either case, failure to reach the **when** clause that processes the application buffer at the head of the incoming or completion event queues (because of too frequent resets of the scheduler) lead to that queue's becoming blocked (or *stuck*) because the application buffer is never processed and freed.

Therefore, when using the scheduler reset mechanism, it is *important* to order the **when** clauses in a program such that events that occur frequently (such as I/O events that occur constantly, or short-interval timer events that expire continually) not lock out processing of messaging events.

Scheduler Example

Turning on the reset mechanism ensures that events are processed in the order intended. For example, you might want ensure that specific events are checked first, followed by a *catch-all* event, as illustrated in this code fragment:

```
#pragma scheduler_reset

network input SNVT_switch nviSwitch1;
network input SNVT_switch nviSwitch2;
network input SNVT_switch nviSwitch3;
network input SNVT_switch nviSwitch4;

when (nv_update_occurs(nviSwitch1))
{
    ...
}

when (nv_update_occurs(nviSwitch2))
{
    ...
}
when (nv_update_occurs)
{           // provides a generic check
    ...     // for all network variable
            // updates
}
```

Updates received for **nviSwitch1** cause both the first and third events to become TRUE. Similarly, updates for **nviSwitch2** cause the second and third events to

become TRUE. It is thus important that these **when** clauses be evaluated in their given order after a network variable update. Using **scheduler_reset**, the **nv_update_occurs** event for **nviSwitch1** is always checked first whenever a new network variable update is at the head of the queue.

Updates to **nviSwitch3** or **nviSwitch4** trigger only the third event.

Bypass Mode

All scheduling of Neuron C programs, as described above, is event-driven and handled by the scheduler. Within a program, however, you can choose when to return control to the scheduler. The term *bypass mode* refers to a method of programming in which one **when** clause always evaluates to TRUE and never returns. In this case, a single task must handle all event processing.

You should use bypass mode rarely, and only in cases where you need a different scheduling algorithm than that provided by the Neuron firmware scheduler. While in bypass mode, your program is responsible for all event processing. You define critical sections through the **post_events( )** function (see the following section), and then check for predefined events in **if**, **while**, and **for** expressions.

If your application runs on a Series 5000 or 6000 device, consider using interrupts for specific kinds of events, so that you do not need to use bypass mode for the scheduler. See *Interrupts* on page 165 for more information.

The post_events( ) Function

Use the **post_events( )** function to define a boundary of a critical section at which network variable updates are sent and incoming network variable updates are processed.

Note: The **post_events( )** function is automatically called at the top of the scheduling loop.

When the **post_events( )** function is called, a number of things happen:

- Any outgoing network variable updates are sent. In the case of synchronous network variables, all the updates are sent. For nonsynchronous network variables, as many updates are sent as application output buffers are available. Any unsent updates are sent the next time **post_events( )** is called.

- Incoming network variable update events are received.

- New incoming messages are examined.

- Timers are examined to see if they have expired.

- The watchdog timer is reset (to keep it from timing out). See the following section about the watchdog timer.

You can use the **post_events( )** function to improve network performance by calling it immediately after modifying network variables. Call the **post_events( )** function to signal the network processor to start execution of the formatting of the outgoing packet before the **when** task completes, thus increasing parallel processing and utilizing the Neuron Chip and Smart Transceiver multi-processor architecture to its fullest.

Watchdog Timer

For Series 3100 devices, the watchdog timer times out within a range of 0.21 seconds to 0.42 seconds with a 40 MHz input clock. This value scales inversely with the input clock. The hardware timer has a period of 0.21 seconds, but a timeout occurs at the end of the current period only if the watchdog timer has not been retriggered since the beginning of the current period. Because the timer retrigger in software is asynchronous with the timeout period in hardware, from a software perspective the minimum time from retriggering to timeout is a single period, or 0.21 seconds, and the maximum time from retriggering to timeout is two periods, or 0.42 seconds.

For Series 5000 and 6000 devices, the watchdog timer period is fixed at 840 ms (1.19 Hz) for all system clock rates. The actual timeout range is between 0.8 s and 1.7 s.

The intention of the watchdog timer is to reset the device within a nominal value of one second should it experience a software failure, such as an unterminated loop or other fault, that prevents the software from retriggering the timer. Normally, the scheduler ensures that the watchdog timer is reset periodically, and the application program need not be concerned about the watchdog timer. If a program enters a very long task, however, the watchdog timer could expire, which causes a device reset.

To ensure that the watchdog timer does not expire, you can call the **watchdog_update( )** function periodically within long tasks (or when in bypass mode). The **post_events( )**, **msg_receive( )**, and **resp_receive( )** functions also update the watchdog timer, as well as use of the **io_out( )** function with the **pulsecount** output object, and the **io_in( )** function with the **magcard**, **magtrack1**, **neurowire slave,** and **wiegand** input objects.

Notes:

- Use the **watchdog_update( )** function with care, and, if possible, not within any loops. A software or hardware malfunction that prevents the loop from being terminated could cause a device not to respond; and, it would be unable to recover from this symptom by means of a watchdog timer reset, because the loop body would continuously re-trigger the watchdog timer.

- Firmware functions that write to EEPROM do not automatically update the watchdog timer.

An example of using the **watchdog_update( )** function is shown below:

```
when (TRUE)
{
   post_events();
   if (nv_update_occurs(nviA)){
      ...
   } else if (nv_update_occurs(nviB)){
      ... // long task
      watchdog_update();
      ... // more long task
   }
}
```

Additional Predefined Events

The following three predefined special events result from network management messages:

> **offline**
>
> **online**
>
> **wink**

The **offline** event occurs when an offline network management command is received from a network tool. This event is always handled as the first priority **when** clause. The **online** event occurs when an online message is received from a network tool. The **wink** event occurs when a **wink** command is received from a network tool.

The **offline** event can be used to place a device offline in case of an emergency, for maintenance, or in response to some other system-wide condition. While offline, a device responds only to network management messages until it is reset or brought back online. Reset can occur by physically resetting the device by activating the Neuron Chip or Smart Transceiver reset line, or through a *reset* network management message. After execution of the task associated with an offline **when** clause, the application program does not run until the device is either reset or brought back online.

A simple use of these two events is shown below:

```
when (offline)
{
   x();  // Clean up before going offline.
}         // Device goes offline here; application
          // program stops running.

when (online)
{
   y();  // Start up again
}
```

The application has no means to refuse a change into the offline or online states, respectively. The respective state becomes effective after the relevant task has been completed, allowing the application to prepare for that state by disabling peripheral hardware, stopping timers, and so on.

The device can change into the online state without the online **when** clause evaluating to TRUE: If the device is being taken offline into the soft-offline state, resetting the device loses, or discards, the soft-offline state and returns the device to normal, online, operation. The technique shown below can be used to handle this situation:

```
void HandleOnline (void)
{
   ...
}

when (reset)
{
   // regular reset code here:
   ...
```

```
        // handle case of device going online
        if (online) {
            HandleOnline();
        }
    }

    when (online)
    {
        HandleOnline();
    }
```

Going Offline in Bypass Mode

Use the **offline_confirm( )** function if the **offline** event is checked outside of a **when** clause, as in bypass mode. The **offline_confirm( )** function sets the state of the device to offline and returns immediately. Use this function to confirm that the device has finished its cleanup and is now going offline.

As shown below, in bypass mode, the program continues to run even though the device is offline. In bypass mode, it is up to you to determine which events are not processed when the device is offline.

Here is an example of using **offline_confirm( )** in bypass mode:

```
    when (TRUE)
    {
        while (TRUE) {
            post_events();
            if (online)
                continue;
            if (nv_update_occurs) {
                ...
            } else if (offline) {
                x();
                offline_confirm();
                // Wait for online
                while (!online) {
                    post_events();
                }
            } else {
                ...
            }
        }
    }
```

Wink Event

You can use the **wink** event to perform an action in response to a *wink* network management message from a network tool. A network tool can send a *wink* message to a device to help a network integrator physically identify a particular device. The **wink** event becomes TRUE any time a *wink* message is received by a device, whether it is configured or unconfigured.

For an unconfigured device, I/O and variable initialization occur before the **wink** event is evaluated. However, none of the initialization in the **when (reset)** task has occurred. In addition, the scheduler is not running on an unconfigured

device, so events can be processed only through direct event processing. Neither network variable updates, nor messages, are sent because the device is unconfigured. Timer objects can be set and read within the **wink** task (unless the device is unconfigured, in which case you can use the **delay( )** function). You also can explicitly check the **timer_expires( )** event as long as you first call **post_events( )**.

Interrupts

Series 5000 and 6000 chips provide a number of hardware interrupts that are available to a Neuron C application. At higher system clock rates, these interrupts run in a dedicated processor on the chip, the interrupt service routine (ISR) processor. At the lowest system clock rates (5 MHz and 10 MHz), these interrupts run in the same processor as the Neuron C application, the application (APP) processor. Running interrupts in their own ISR processor provides higher performance for both the interrupt routines and the Neuron C application.

In addition, Series 5000 and 6000 chips provide other hardware interrupts, such as those related to the SCI UART or SPI I/O models, that are automatically handled by the Neuron firmware and are not available to a Neuron C application.

This section describes the hardware interrupts that are available to a Neuron C application.

Interrupt Sources

The hardware interrupt support provides the following interrupt sources for application use:

- I/O interrupts

- Timer/counter interrupts

- Periodic system timer interrupts

The Neuron firmware, version 18 and later, combined with an application-specific interrupt dispatcher automatically generated by the compiler, identifies the interrupt source, manages the interrupt management flags, and calls the correct application-specific interrupt task for each interrupt request. Interrupts are dispatched in the order of source code declaration, so that interrupt tasks that require the lowest latency and smallest jitter should be declared first, and interrupt tasks that are least sensitive to latency and jitter should be declared last.

The Series 5000 and 6000 chips also supply one binary semaphore to support synchronized access to shared resources between the application and the ISR processor. A Neuron C application uses the **__lock** keyword to access this semaphore and synchronize access to shared resources. See the *Neuron C Reference Guide* for more information about this keyword, and see *Sharing Data with an Interrupt Task* on page 173 for more information about using the semaphore.

Network I/O is not allowed from within an interrupt task, but your interrupt task can access I/O devices and variables shared with the application.

In addition, you cannot use the NodeBuilder Debugger to set break points within an interrupt task. See *Debugging Interrupt Tasks* on page 176 for suggestions about how to debug an interrupt task.

I/O Interrupts

The Series 5000 and 6000 hardware supports two independent I/O interrupts, each derived from one of the chip's 12 I/O pins. Each of the two I/O interrupts can be configured to trigger with any one of the following I/O signal conditions (but not a combination of them):

- Rising edge of the I/O signal

- Falling edge of the I/O signal

- Both rising and falling edges of the I/O signal

- High level of the I/O signal

- Low level of the I/O signal

The interrupt source is independent of the I/O direction, that is, it works with both input and output signals. In addition, both interrupts are independent of each other. You can configure each I/O interrupt to trigger on separate I/O pins or on the same I/O pin. If both trigger on the same pin, they could also trigger on the same condition, although such a configuration is not likely to be useful to the application. For example, both could trigger on the falling edge of IO_3. A more useful example could have one I/O interrupt trigger with IO_3's rising edge and the other I/O interrupt trigger with IO_3's falling edge, but note that you can also declare a single I/O interrupt task that triggers with either edge of the I/O signal.

An interrupt that is configured to trigger on a high or low level of an I/O signal continues to trigger as long as the level condition is met. For example, you could trigger an interrupt to process a first-in first-out (FIFO) queue by specifying a level signal to represent a non-empty queue, and the interrupt triggers repeatedly until the level changes (when the queue is empty).

Timer/Counter Interrupts

As described in the *I/O Model Reference*, the Series 5000 and 6000 chip hardware supports two hardware timer/counters:

- Timer/counter 1 is connected to IO_0 (output) and IO_4..7 (input) through a multiplexer

- Timer/counter 2 is dedicated to IO_1 (output) and IO_4 (input)

Both timer/counters can generate one optional timer/counter interrupt each. The interrupt occurs when the counter overflows or underflows, or when the timer event latch triggers, depending on which I/O model the timer/counter is defined for.

For timer/counter I/O models that use multiplexed I/O pins, a timer/counter interrupt task runs when any of the pins within the defined I/O model meet the critera for an interrupt.

Periodic System Timer Interrupts

An application can also use the periodic system timer interrupt. The frequency for this interrupt can be configured within a wide range (2.4 kHz to 625 kHz).

The periodic system timer cannot be stopped (it is always running), and its frequency can be set only during compilation, but you can enable or disable the associated periodic system timer interrupt as needed; see *Controlling Interrupts* on page 171.

One application for periodic system timer interrupts could be to provide audio output for your application. Audio output generally requires a minimum of an 8 kHz signal, and a Series 5000 or 6000 chip can process interrupts at an 8 kHz rate while providing full network communication support for the application.

Defining an Interrupt Task

Within your Neuron C application, you can define one interrupt task for each interrupt source. The interrupt task definition specifies the interrupt source, interrupt type, and the interrupt condition. An interrupt task looks similar to the familiar Neuron C when task, but uses the **interrupt** keyword rather than the **when** keyword. However, interrupt tasks are not scheduled by the application scheduler.

The basic syntax for an interrupt task is:

> **interrupt (***pin***,** *condition***)**
> **interrupt (***object-name***)**
> **interrupt (repeating)**
> **interrupt (repeating**, *freq***)**

pin	specifies the I/O pin for an I/O interrupt
condition	specifies the interrupt trigger condition for an I/O interrupt
object-name	specifies the timer/counter I/O object name for a timer/counter interrupt
repeating	defines a periodic system timer interrupt
freq	specifies the frequency for a periodic system timer interrupt

The following sections describe how to define each type of interrupt task.

6000 HW interrupt tasks always compile into resident code. Unlike *when*-tasks, interrupt tasks can *only call resident utility functions*.

Defining an I/O Interrupt Task

An I/O interrupt references an I/O pin, **IO_0** .. **IO_11**. The I/O interrupt can use the same I/O pin keyword that you use to declare the I/O object, but you do not need to declare an I/O object for the pin used with an I/O interrupt. The I/O pin can be an input signal or an output signal.

You can define up to two I/O interrupts, which are dispatched in declaration order. If you define a third I/O interrupt task, the Neuron C compiler issues error **NCC#577**.

You declare the trigger condition for the I/O interrupt through a level indicator, '+' or '-', or by using the **clockedge** keyword, as shown in **Table 10** on page 168.

Table 10. I/O Interrupt Triggers

Trigger Type	Example Interrupt Specification
Positive level (default declaration)	IO_0
Positive level (explicit declaration)	IO_0, +
Negative level	IO_0, -
Falling edge	IO_0, clockedge(-)
Rising edge	IO_0, clockedge(+)
Both rising and falling edge	IO_0, clockedge(+-)

Example:

```
interrupt (IO_0, clockedge(+)) {     // rising edge of IO_0
   ...
}

interrupt (IO_5,-) {                 // neg. level @ IO_5
   ...
}

interrupt (IO_6) {                   // pos. level @ IO_6
   ...
}
```

Important: For a single pin, you cannot combine a dual-edge interrupt trigger ("**clockedge(+-)**") with a falling-edge trigger ("**clockedge(-)**"), because the hardware would interpret the combination as two dual-edge triggered interrupts. The Neuron C compiler issues error **NCC#586** when it encounters such a configuration.

Defining a Timer/Counter Interrupt Task

You define a timer/counter interrupt task by referencing a timer/counter I/O object. Up to two timer/counter interrupt tasks are supported.

Referring to a different I/O model yields compiler error **NCC#576**, and attempts to declare more than two timer/counter interrupts yield compiler error **NCC#576**.

When a timer/counter interrupt task completes, the system firmware clears the interrupt source, so a **when(io_update_occurs())** task cannot process the same timer/counter update that the interrupt task already processed. You can make calls to the **io_in()** function from within the interrupt task, if measurements are required, and you can share those measurements with the remaining application through global variables.

The following timer/counter I/O models that are supported as timer/counter interrupt sources:

- Dualslope Input
- Edgedivide Output
- Edgelog Input
- Frequency Output
- Infrared Input
- Infrared Pattern Output
- Oneshot Output
- Ontime Input

- Period Input
- Pulsecount Input
- Pulsecount Output
- Pulsewidth Output
- Stretched Triac
- Triac
- Triggeredcount Output

Note that the totalcount input and quadrature input I/O models are not supported by timer/counter interrupts, because the interrupt would not trigger until you called the **io_in()** function, which eliminates the need for an timer/counter interrupt for these two I/O models.

For timer/counter I/O models that use multiplexed I/O pins, you can declare an interrupt task for only one of the pins. Declaring an interrupt for more than one of the multiplexed pins yields an error (**NCC#584**). For example:

```
IO_4 input pulsecount ded pulsecount_A;
IO_5 input pulsecount mux pulsecount_B;
IO_6 input pulsecount mux pulsecount_C;
IO_7 input pulsecount mux pulsecount_D;

interrupt(pulsecount_B){
  // process the interrupt for the pulsecount
  // multiplexed I/O object
}

interrupt(pulsecount_C){
  // You cannot define a second interrupt task for the
  // pulsecount multiplexed I/O object
  // Compiler issues NCC#584:
  // "The hardware timer/counter unit is already used with
  // a different interrupt task"
}
```

Timer/counter interrupts for the following I/O models use overflow interrupt triggering:

- Dualslope input
- Frequency output
- Infrared pattern output
- Oneshot output
- Pulsecount output
- Pulsewidth output
- Triggered count output

Note: Triggering the interrupt on overflow for the dualslope input model means that the interrupt occurs at the end of the second integration period.

Timer/counter interrupts for the following I/O models use latch interrupt triggering:

- Edgelog input

- Infrared input

- Ontime input

- Period input

- Pulsecount input

- Stretched triac output

- Triac output

Note: Triggering the interrupt on the latch for the stretched triac output and triac output models means that the interrupt occurs at zero crossing. This trigger could be used for phase detection. If your application requires pulse monitoring, you could overload an I/O interrupt to monitor both the zero crossing and the trigger pulse.

Example:

```
IO_4 input edgelog mux clock(5) myEdgelog;
IO_5 input ontime myOnTime;
IO_1 output oneshot myOneShot;

interrupt(myEdgelog) {     // T/C 1 IRQ
    ...
}

interrupt(myOnTime) {      // T/C 2 IRQ
    ...
}

// The following yields a compilation error, even though
// the declaration of overlapping T/C models is correct
// (through use of the io_select function).

interrupt(myOneShot) {     // ERROR: Too many T/C interrupts
    ...
}
```

Defining a System Timer Interrupt Task

To define a periodic system timer interrupt, use the **repeating** keyword, followed by an optional frequency specification.

The default frequency is 8 kHz (a period of 125 µs). You can define other frequencies from 2,441.406 Hz to 625,000 Hz, in 256 steps. The interrupt frequency is a constant value in Hertz, specified as a double-quoted string. The number that you specify in that string follows the standard C format for floating point numbers. Additionally, you can append an optional multiplier: "Hz" (hertz), "kHz" (kilohertz), "MHz" (megahertz), or "GHz" (gigahertz). The multiplier is case-sensitive.

Examples:

```
interrupt(repeating) {     // periodic system timer, 8kHz
    ...
}

interrupt(repeating, ″8kHz″) {     // equivalent to default
    ...
}

interrupt(repeating, "3456.789Hz") {
    ...
}

interrupt(repeating, ″625E3″) {     // 625 kHz
    ...
}
```

You do not need to specify the frequency as one of the 256 allowed values within the range 2,441.406 Hz to 625,000 Hz. Instead, the Neuron C compiler accepts any correctly formatted value within the range, and uses the following formula to calculate an encoded rate value *Z*:

$$Z = \frac{1}{F * (1.6 * 10^{-6})} - 1$$

where *F* is the specified frequency value.

The compiler then rounds the calculated value *Z* to the nearest integer *Z'*. The *Z'* value defines the encoded step within the frequency range 2,441.406 Hz to 625,000 Hz, and has a value in the range [0 .. 255]. The compiler uses the *Z'* value to register the system timer interrupt with the system firmware:

$$n = \max[0, \min(255, Z')]$$

The compiler issues a warning message if the resulting frequency *f(n)* varies by more than 1% from the desired frequency F, according to the following formula:

$$f(x) = \frac{1}{(x + 1) * (1.6 * 10^{-6})}$$

The warning message (**NCC#578**) contains the exact error (in percent) and the resulting interrupt frequency. To eliminate the warning message, or to define the system timer frequency with high precision, use the formula for *Z* to specify a different frequency value.

Controlling Interrupts

By default, all application-defined interrupts are disabled, thus allowing the device to complete its initialization or reset processing before being interrupted. When the device is ready to receive interrupts, typically towards the end of the **when(reset)** task, the application calls the following function:

> **void interrupt_control(unsigned** *irqSelect***);**

The *irqSelect* argument specifies the type of interrupts to enable. You can use the following predefined symbols to specify the interrupt type:

```
#define INTERRUPT_IO                    0x03
#define INTERRUPT_TC                    0x0C
#define INTERRUPT_REPEATING            0x10
```

A value of zero (0x00) disables all interrupts. A value of -1 (0xFF) enables all interrupt tasks defined for the application.

You can call the **interrupt_control()** function at any time to enable or disable one or more of the three interrupt types.

The function prototype and predefined symbols are defined within the **<echelon.h>** header file. The Neuron C compiler automatically includes this file for every application; you do not need to include it explicitly.

Examples:

```
// enable I/O interrupts only
interrupt_control(INTERRUPT_IO);

// enable both I/O and timer/counter interrupts
interrupt_control(INTERRUPT_IO | INTERRUPT_TC);

// disable all interrupts
interrupt_control(0);
```

Recommendation: Disable all interrupts when the device goes to the offline state. Interrupt processing can sometimes interfere with device management tasks performed while a device is offline.

Example:

```
#include "status.h"

when(reset) {
    // query node status:
    status_struct status;
    retrieve_status(&status);

    // proceed with device initialization:
    ...

    // enable interrupt system if configured and online:
    if (status.status_node_state == 0x04) {
      interrupt_control(INTERRUPT_IO | INTERRUPT_REPEATING);
    }
}

when(offline) {
    interrupt_control(0);
}

when(online) {
    interrupt_control(INTERRUPT_IO | INTERRUPT_REPEATING);
}
```

If you call the **interrupt_control()** function for an interrupt type that does not have a corresponding interrupt task defined, the call does nothing. However, if you call the **interrupt_control()** function and there are no interrupt tasks defined, the Neuron linker issues an error message.

If you need to disable or enable all interrupts at the same time, you can also use the **io_idis()** function to disable interrupts and the **io_iena()** function to enable interrupts. These functions have the following declaration:

void io_idis(void);

void io_iena(void);

Sharing Data with an Interrupt Task

In general, an interrupt task can access any data that the main application can access. There are times when you might want an interrupt task to share data with the main application. Sharing data between asynchronous processes can lead to data integrity problems, so you should exercise care when sharing data.

Important: An interrupt task can read data that is declared with the **eeprom** keyword, but cannot write to it. The firmware logs system error 48 when it detects an attempt to write to **eeprom** data. Writing to an EEPROM or flash memory device is a time-consuming task that is not supported within an interrupt context. To ensure that **eeprom** data is written, the interrupt task should update an appropriate flag, and allow the main application to write the **eeprom** data.

When an interrupt task runs in the ISR processor, the system firmware provides a binary semaphore to synchronize access to shared resources between the application and the ISR processor. However, when an interrupt task runs in the application processor, there is no semaphore available, and you must manage resource sharing as you would for other function types (such as callback functions).

You use the Neuron C **__lock** keyword within the application or the interrupt task to access this semaphore and synchronize access to shared resources. The **__lock** keyword defines a block of code (within a pair of curly braces) that is controlled by the semaphor.

The **__lock** keyword is not supported for the lowest system clock rates (5 MHz and 10 MHz). However, you can use it within either Neuron C code or pure C code (for example, within a library).

Example:

```
interrupt (IO_0) {
    ...       // do something not guarded by the semaphore

    __lock { // acquire semaphore
       ...    // do something guarded by the semaphore
    }          // release semaphore

    ...       // more unguarded code
}
```

Because there is only one semaphore, you cannot nest locks. Nested locks yield a compiler error, **NCC#585**. Defining a lock with target hardware or firmware that does not support semaphores yields a linker error, **NLD#506**. If you define a lock for an interrupt task that runs within the application processor, the Neuron Exporter issue a warning message (**NEX#4014**) that the hardware semaphore and the **__lock{ }** statement are not operational for the specified

system clock setting. In addition, when the interrupt task runs, no semaphore is acquired, but a system error is logged.

You can define a lock both in Neuron C code and in pure C code (such as a function library).

While the compiler can detect and prevent direct nesting of **__lock{ }** statements, it cannot detect runtime-nesting of lock requests. Consider the following example.

Example:

```
unsigned long globalVariable;

void f() {
    __lock {
        globalVariable = ...;
    }
}

interrupt(IO_3, clockedge(-)) {
    __lock {
        f();
    }
}
```

Because the interrupt task acquires the semaphore and then calls function **f()**, the second lock request (the **__lock{ }** construct within the function **f()**) can never succeed. The chip resets after the watchdog timer expires, and a system error is logged.

While an interrupt task cannot perform network activity, it can read and evaluate values for network variables and configuration properties. You can guard those operations with a **__lock{ }** statement to synchronize between the interrupt task and the when tasks, but you cannot guard against asynchronous updates to these variables as they arrive over the network.

In an application that requires an interrupt task to read such data, it is recommended that the appropriate when task creates a locked copy of the relevant data, which can be safely accessed from the interrupt task.

Interrupt Latency

Interrupt *latency* is the amount of time that elapses between the generation of an interrupt by a device and the running of the associated interrupt task. Variations in this latency for servicing interrupts can lead to jitter in output signals generated by an interrupt task, or otherwise affect the algorithm performed by the interrupt task. This section describes the latency required by the system firmware, the Neuron C application, and the interrupt task.

When a Neuron C application contains one or more interrupt tasks, the Neuron C compiler generates an interrupt dispatcher that is added to the application. The system firmware calls this dispatcher when an interrupt is triggered.

When the system firmware receives an interrupt request from the hardware, the firmware prepares to run the interrupt task. This preparation adds the following latency to interrupt processing:

- 54 clock cycles if interrupt tasks run on the interrupt (ISR) processor

- 115 clock cycles if interrupt tasks run on the application (APP) processor

This preparation occurs once for each hardware interrupt request, regardless of the number of interrupt tasks defined within the application. Interrupts run on the ISR processor for system clock multiplier settings of 2 or higher (20 MHz or higher), or interrupts run on the application processor for system clock multiplier settings of ½ or 1 (5 or 10 MHz).

At the end of this preparation, the system firmware calls the interrupt dispatcher within the Neuron C application. The dispatcher attempts to dispatch each interrupt task, in source-code declaration order. For any given interrupt task, the dispatcher adds latency, as shown in **Table 11**.

Table 11. Dispatcher Latency

Number of Interrupt Tasks Defined within the Application	Clock Cycles Required for I/O or System Timer Interrupts	Clock Cycles Required for Timer/Counter Interrupts
1	7	10
More than 1	9	16

After the interrupt task completes, the system firmware performs a number of clean-up steps. For any given interrupt task, the clean-up steps add latency, as shown in **Table 12**. As the table shows, the clean-up steps apply only to timer/counter interrupt tasks.

Table 12. Clean-Up Latency

Number of Interrupt Tasks Defined within the Application	Clock Cycles Required for I/O or System Timer Interrupts	Clock Cycles Required for Timer/Counter Interrupts
1	0	63
More than 1	0	66

Figure 15 on page 176 summarizes how the system firmware and interrupt dispatcher add latency to interrupt processing. In the figure, the numbers are the clock cycles required for each step, as defined above. The numbers N_0, N_1, and N_2 represent the number of clock cycles required to run an interrupt task itself.
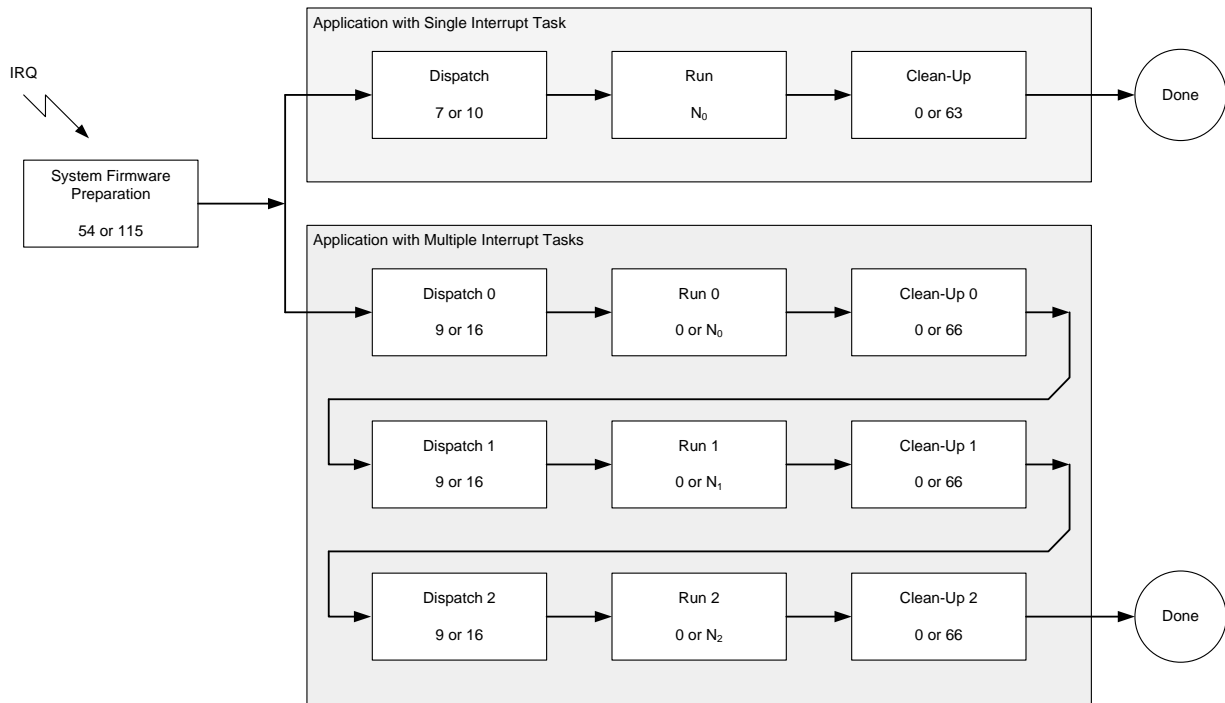
Figure 15. Interrupt Latency

As the figure shows, the first (in source-code declaration order) interrupt task always has the minimum latency and no jitter. If the application includes more than one interrupt task, a subsequent interrupt task experiences a number of clock cycles of latency, depending on the interrupt type and whether other interrupt tasks must run before it can be dispatched.

For a typical application, no more than two interrupts are pending at any one time, so that either no other interrupt task, or just one interrupt task, needs to run before the current interrupt task is dispatched.

Debugging Interrupt Tasks

You cannot use the NodeBuilder debugger to debug an interrupt task or any code called from an interrupt task. That is, you cannot set a breakpoint within an interrupt task or in a function that is called from an interrupt task. If you attempt to set such a breakpoint, the device logs a system error, resets, and comes up in the soft-offline state. If the reset clause re-enables interrupts before the device can go offline, the debugger might lose communications with the device, and thus need to set the device applicationless.

You can debug your code using application-specific debugging tools, such as dedicated global variables to track states or events, or application-specific output through a suitable I/O model. You could also run the code-under-test from a when task for testing and debugging purposes.

When sharing data between an interrupt task and the application, you cannot set a breakpoint on the __**lock** statement itself, but you can set breakpoints within or outside of the lock. However, if an interrupt trigger condition is met while the application is halted at a breakpoint within a lock, and the interrupt task tries to acquire a lock, the Neuron Chip or Smart Transceiver watchdog timer will time

out, and the device resets. To avoid the watchdog timer reset, use the **#pragma deadlock_is_infinite** compiler directive. Do not use this directive for release targets; it is intended only for debug targets. See the *Neuron C Reference Guide* for information about this directive.

Restrictions for Using Interrupts

Networking activity is prohibited within an interrupt task, and from any function called from the interrupt task. If the interrupt task calls a messaging or network variable API, the system firmware logs an error and ensures that no other action is taken. The messaging or network variable API call returns results that indicate an error condition, such as "no new message available" or "no buffers available."

Declaring one or more interrupt tasks with target hardware that does not support this hardware interrupts yields a linker error, **NLD#506**.

Declaring an interrupt task in pure-C mode yields a compiler error. That is, you cannot create a binary library of interrupt tasks, just as you cannot use when-tasks in a precompiled function library.

You cannot call transient utility functions from within an interrupt task.

Sleep Mode

You can use *sleep mode* to place a Series 3100 Neuron Chip or Smart Transceiver in a low-power state. To instruct a Series 3100 Neuron Chip or Smart Transceiver to enter sleep mode, perform the following steps:

1   Flush all pending network variable updates as well as all outstanding outgoing and incoming messages.

2   Put the Neuron Chip or Smart Transceiver into sleep mode after the flush completes. The Neuron Chip or Smart Transceiver always wakes up when the service pin is activated, or when there is activity on an I/O pin (the pin selected is configurable) or on the communications channel, or both.

Series 5000 and 6000 chips do not support sleep mode.

Example:

```
mtimer m_30;
network output SNVT_switch nvoValue;
static SNVT_switch temp;

when (timer_exp(m_30))
{
    nvoValue = temp;
    flush(TRUE);
}

when (flush_completes)
{
    sleep(COMM_IGNORE);
}
```

Flushing the Neuron Chip or Smart Transceiver

You can use the **flush( )** function to instruct the Neuron firmware to finish processing all outgoing and incoming messages. When the flush is complete, the **flush_completes** event becomes TRUE and the chip enters Quiet mode.

The flush( ) and flush_cancel( ) Functions

The **flush( )** function causes the Neuron firmware to monitor the status of all outgoing and incoming messages. Its syntax is as follows:

> **flush (boolean** *comm-ignore***);**

comm-ignore   Specify TRUE if the Neuron firmware is to ignore communications channel activity while it is flushing. Specify FALSE if the Neuron firmware is to accept incoming messages. This parameter should be the same as the **comm_ignore** parameter used with the **sleep( )** function that follows the flush.

While the flush is occurring, the program continues to run. The origination of new messages by the program while the flush is in progress delays the flush completion.

If the **comm_ignore** option is set to TRUE, new packets that arrive during the flush are discarded unless they are acknowledgments, responses, challenges, or replies.

You can cancel a flush operation that is in progress by calling the **flush_cancel( )** function.

flush_completes Event

The following predefined event becomes TRUE when the flush completes:

> **flush_completes**

This event becomes TRUE when all outgoing network buffers and application buffers are free, no more incoming messages are outstanding, and no network variable updates are outstanding.

Note: The **flush_wait( )** function should not be used in preparation for putting the device to sleep. The **flush_wait( )** function does not check for outstanding network variable updates or incoming messages.

Putting the Device to Sleep

You can use the **sleep( )** function to put a Neuron Chip or Smart Transceiver to sleep when the **flush_completes** event becomes TRUE. Its syntax is shown below:

> **sleep (***flags***)**
>
> **sleep (***flags***,** *io-object-name***)**
>
> **sleep (***flags***,** *io-pin***)**

<table>
<tr><td>flags</td><td>One or more of the following three flags, or 0 if no flag is to be specified. If two or more flags are used, they are OR'd together. You can specify any of the following flags:</td></tr>
</table>

COMM_IGNORE	Causes incoming messages to be ignored.
PULLUPS_ON	Enables all internal pullup resistors for Series 3100 devices (the default action is to disable the pullups, which lowers power consumption).
TIMERS_OFF	Turns off all timer objects (declared with **mtimer** and **stimer**) in the program.

<table>
<tr><td>io-object-name</td><td>An input object declared for any one of pins **IO_4** through **IO_7**. When any I/O transition occurs on the specified pin, the device wakes up. If this parameter is omitted, I/O is ignored while the device is in sleep mode. This I/O object can be defined for wakeup purposes only, or could be used for other I/O purposes as well.</td></tr>
<tr><td>io-pin</td><td>Specifies one of pins **IO_4** through **IO_7**. When any I/O transition occurs on the specified pin, the Neuron Chip or Smart Transceiver wakes up. If this parameter is omitted, I/O is ignored while the device is in sleep mode.</td></tr>
</table>

For example, to sleep and turn off timers and enable pullups, the call to **sleep( )** is as follows:

sleep(TIMERS_OFF | PULLUPS_ON);

Example:

```
IO_4 input bit wakeup_pin1;

when timer_expires(timer_2)
{
    sleep(COMM_IGNORE, wakeup_pin1);
    //or, sleep(COMM_IGNORE, IO_4);
}
```

You can force sleep mode even though the flush has not completed, as described in *Forced Sleep* on page 180.

When an event occurs that wakes the Neuron Chip or Smart Transceiver, the program resumes at the first statement after the sleep function call. If the **sleep( )** call is the last statement in a task, the program returns to the scheduler after it wakes up.

A device wakes up whenever a packet is received by the transceiver (unless you specified **COMM_IGNORE**). The device wakes up even if the packet is not addressed to the device. You are responsible for putting the device back to sleep if this occurs.

If a device sleeps for less than the receive timer duration and uses the **comm_ignore** option, it could receive duplicate messages or network variable update events. The default receive timer is set by a network tool during device installation. The receive timer has a hard-coded minimum value of 768 ms. This value can be increased by a network tool, depending on the network connections to the device.

Forced Sleep

You can force a device to sleep even though a flush operation is not complete. Under certain network conditions, such as extreme network congestion, the flush could take a long time to complete. To avoid consuming too much power, the application can stop waiting for the flush to complete and sleep anyway.

To force a device to sleep, call the **sleep( )** function without waiting for the **flush_completes** event. An example of forcing a device to sleep is shown below:

```
...
   flush(TRUE);             // start flush; ignore
                           // incoming packets
   flush_timeout = 300;    // start flush timeout
                           // timer (300 msec)
}

when (timer_expires(flush_timeout))
when (flush_completes)
{  // Ready to go to sleep since the flush
   // either completed or timed out
   flush_timeout = 0;     // First, turn off timer
                          // if not expired
   sleep(comm_ignore);
}
```

When you force sleep mode, the following occurs:

1   All pending network variable updates, outstanding application output buffers, and outstanding network output buffers are not sent and freed.

2   If you specify the **comm_ignore** option, any incoming network buffers are freed.

3   If any outstanding incoming application buffers remain, the device fails to sleep (regardless of whether the **COMM_IGNORE** option was specified). This feature prevents the device from receiving stale messages when it wakes up. In the example above, the application would have 300 milliseconds to process any incoming messages already in the queue. In addition, since the **comm_ignore** parameter was set to TRUE in the call to **flush( )**, no new incoming messages would arrive. Thus, it is likely that the device sleeps, assuming it processes, in the 300 msec prior to the timeout, any incoming messages that were outstanding prior to the call to **flush( )**.

Error Handling

You can take one or more of the following actions to recover from or report an application error:

- Reset the device
- Restart the application
- Take the application offline
- Disable a functional block
- Change functional block status

- Log an error

These actions can be combined. For example, you can log an error and then take the application offline. Alternatively, you can disable a functional block and change functional block status.

The Neuron firmware also logs system errors for errors detected by the firmware.

Resetting the Device

You can reset a device by calling the **node_reset( )** function. This function immediately resets all processors (application, network, media access, and for Series 5000 and 6000 devices, interrupt) on the Neuron Chip or Smart Transceiver. The Neuron reset pin is driven low; this signal can be used to reset external transceivers and logic. You typically take this action for catastrophic errors that require a hardware reset. A device can also be reset by expiry of the watchdog timer, or by a reset command received from the network.

When a device is reset, it executes the entire initialization sequence. The amount of time required for initialization is a function of the amount of off-chip memory as well as the size of the application program, but must be less than 18 seconds before the application is online. See the Smart Transceivers databooks for a detailed formula to calculate reset time.

When you reset a device, all state information that is not kept in EEPROM is lost. All pending incoming and outgoing messages and network variable updates are lost. The network processor could receive duplicate packets. In addition, any packets that have been acknowledged by the network processor but not processed by the application are lost.

Restarting the Application

You can reset the application processor, but not the network or media-access processors or (for Series 5000 and 6000 devices) the interrupt processor, using the **application_restart( )** function. You typically take this action for application errors that can be recovered by restarting the application, but that do not require an external hardware reset.

When you call this function, the Neuron firmware clears all timer objects and initializes I/O objects, non-configuration network variables, and static variables and then executes the reset clause. Some synchronization cleanup is done before restarting the application. Any outgoing messages in progress are terminated. Incoming messages are unaffected. Outstanding completion events and responses are discarded. An application restart does not lose network state information. Because only the application processor is reset, the network and MAC processors continue to process network traffic, and the interrupt processor continues to process interrupts.

Typical applications that use interrupt tasks share some amount of data between when tasks and interrupt tasks. During the application restart, some of the assumptions made for that data sharing might not be valid; you should consider disabling all interrupts (by calling the **interrupt_control( )** function) before calling **application_restart( )**. You can re-enable interrupts during the application's **when(reset)** task.

Taking an Application Offline

You can take a device offline using the **go_offline( )** function. You typically take this action if the error cannot be corrected by a device reset or application restart, and if the error is not localized to specific functional blocks on the device. The device can also be taken offline (and set back online again) through a command received over the network. Network tools frequently set devices offline during configuration.

The **go_offline( )** function terminates all outstanding transactions and stops all application processing. You can call the **flush_wait( )** function in the **when(offline)** task to ensure that any outstanding transactions complete normally.

The Neuron firmware continues to run when a device is offline so that a network integrator using a network tool can test the device status, take any required corrective actions, and then put the application back online. You can log an application error, as described in *Logging Application Errors* on page 183, to alert a network integrator as to the reason for going offline.

Disabling a Functional Block

You can disable an individual functional block for an error that cannot be corrected by a device reset or application restart, but is localized to a particular functional block or set of functional blocks on the device.

Functional block status is not built into the Neuron C language, but code to manage functional block status is automatically generated by the NodeBuilder Code Wizard. The code wizard creates an **fblockData[ ]** array that contains the functional block status for each functional block in an application. The members of this array are declared with the **SNVT_obj_status** type.

To disable a functional block, use the following code:

```
fblockData[fblockIndex].objectStatus.disabled = TRUE;
```

The *fblockIndex* parameter is the functional block index of the functional block to be disabled. A network tool can also disable or enable functional blocks from the network, generally while configuring a single functional block.

You must include code in your application to test the functional block status. The Code Wizard generates a **fblockNormalNotLockedOut( )** function that you can use to test functional block status. The syntax for this function is as follows:

boolean fblockNormalNotLockedOut(TFblockIndex *fblockIndex*);

The *fblockIndex* parameter is the functional block index of the functional block to be tested.

For example, the following call from the NodeBuilder example tests the functional block status for the functional block associated with a network variable input:

```
if
(fblockNormalNotLockedOut(fblock_index_map[nv_in_index]))
{
    . . .
}
```

See the *FT 5000 EVB Examples Guide, FT 6000 Examples Guide,* or the *IzoT NodeBuilder /PL Examples Guide* for more examples of using the **fblockNormalNotLockedOut( )** function.

You can change the functional block status, as described in *Changing Functional Block Status*, to alert a network integrator as to the reason for disabling a functional block.

Changing Functional Block Status

You can report a functional block error condition using the **nvoStatus** output of the Node Object functional block. Each functional block on a device has an independent status condition, so network tools use the **nviRequest** input to the Node Object functional block to request the status of an individual functional block, and this status is reported via the **nvoStatus** output.

Functional block status is not built into the Neuron C language, but as described in *Disabling a Functional Block* on page 182, code to manage functional block status is automatically generated by the NodeBuilder Code Wizard. You can update the functional block status by setting the appropriate fields within the **fblockData[ ]** array. See the definition of the **SNVT_obj_status** type at types.lonmark.org, or in the IzoT Resource Editor, for a description of the fields.

For example, the following statement changes the functional block status for the functional block identified by *fblockIndex* to report a mechanical fault:

```
fblockData[fblockIndex].objectStatus.mechanical_fault =
TRUE;
```

Logging Application Errors

You can report a device error condition using the **error_log( )** function, which is passed an error number between 1 and 127. This function writes the last number into a dedicated location in EEPROM. A network tool can use the *query status* network diagnostic command to read the last error. The syntax for the **error_log( )** function is as follows:

> **void error_log (unsigned int** *error_num***);**

The error number values between 1 and 127 are application-defined. You can assign numbers in this range to your device error conditions, and document these assignments as part of your device documentation.

System Errors

The Neuron firmware reports system errors using the same error log used to report application errors. System errors include programming errors and network errors and inconsistencies. As with application errors, network tools can retrieve the last value from the error log using the *query status* network diagnostic command.

System error numbers are in the range of 128 to 255; see the *Neuron Tools Errors Guide* for an annotated list of system error messages.

Access to Device Status and Statistics

From your application program, you can access the same diagnostic status information that is available to a network tool. The status information is stored in the status structure. To retrieve the status information, use the **retrieve_status( )** function. Its syntax is as follows:

> **void retrieve_status (status_struct ****status-p***);**

The fields of the status structure are described in detail in the *Neuron C Reference Guide.* Use the **clear_status( )** function to clear certain status structure fields (the statistics information, the reset cause register, and the error log).

8

Memory Management

This chapter describes system memory resources, such as
on-chip EEPROM, application buffers, and network buffers
that can be tailored to the needs of a specific application.
The following sections discuss how these resources can be
reallocated and when you might need to do so.

Memory Use

This section outlines the amount of memory used by certain elements in your program. For a description of the actual memory used by your program, see the link summary.

For Series 5000 and 6000 chips, the hardware implements all memory types in RAM, but software configures sections of this memory for volatile data storage (as if it was RAM), persistent data storage (as if it was EEPROM) and constant code and data (as if it was flash memory or EEPROM). The following discussion refers to sections of RAM, EEPROM and flash memory according to the behavior, especially with regards to data retention, not to the underlying physical implementation.

RAM Use

RAM is used as follows:

code	Size of code (for functions declared with **ram** keyword)
config_prop	0
cp_family	0
fblock	0
io_changes	3 bytes each (any type)
I/O object	0
msg_tag	0
mtimer	4 bytes each

The following sizes pertain to global and static data as declared in the program (except for **eeprom** and **config** variables). These amounts also apply to network variables.

char	1 byte
int	1 byte
enum	1 byte
long	2 bytes
structures	Sum of the size of the elements. Each 8 bits (or fraction of 8 bits) of consecutive bitfields uses up a byte. No bitfield can span a byte boundary. No padding is performed. The **float_type** and **s32_type** extended arithmetic structures each take 4 bytes.
unions	Size of the largest element

EEPROM Use

Approximately 65 bytes of EEPROM is used for constant system overhead, although this can vary depending on the firmware version. In addition, EEPROM or flash memory is used as follows:

- Each domain table entry requires 15 bytes for configurable information, to define the domain address, subnet number, device number, and authentication key. A system can have a maximum of two domain table entries and must have at least one domain table entry. The default is two domain table entries. See *Domain Table* on page 204.

- Each address table entry requires 5 bytes. A maximum of 15 addresses are allowed for the Series 3000 and 5000 chips. The Neuron C Compiler computes a default address table size based on inspection of the compiled application. At least 15 addresses are allocated by default, but larger numbers may be The Series 6000 chip supports 254 entries. The automatic default may be overridden using the *pragma addresses* directive; see *Address Table* on page 202.

- Each network variable declared (input or output) uses 3 bytes for its configuration information on targets limited to 15 addresses, and 4 bytes on targets supporting up to 254 addresses. In addition, it uses 3 bytes of read-only memory for its fixed information. If you use the SNVT self-identification (SI) feature, there is an additional 7-byte fixed overhead plus 2 additional bytes per network variable (minimum).

- Each network variable alias table entry uses 4 bytes on targets limited to 15 addresses, and 5 bytes on targets supporting up to 254 addresses. The Neuron C Compiler computes a default alias table size based on inspection of the compiled application. See *Alias Table* on page 203.

- Variables declared as **eeprom** and **config** in your program use an amount of EEPROM corresponding to its C data type. This includes network variables of the **config_prop** (or **cp**) class and modifiable configuration parameters declared using the **cp_family** keyword (the latter are stored together in a writable value file). See *Default Memory Usage* on page 193.

- The **when** clause table is placed in the code memory area (ROM, if available, or EEPROM). Each **when** clause uses a table entry from 3 to 6 bytes (most are 3 bytes). This code space is usually slightly smaller than the equivalent code generated by an **if** statement. Additional code space can result from **when** clauses containing user-defined events.

- The interrupt dispatcher and the **interrupt_control( )** function are placed in the code memory area (ROM, if available, or EEPROM or flash memory). The **interrupt_control( )** function takes a constant, small, amount of code space (12 bytes at the time of release, but the exact number is subject to change). The interrupt dispatcher is optimized for the each application, and consumes the smallest amount of memory possible. Its size varies with the number and type of interrupt tasks that your application defines.

- The read-only value file is normally placed in the CODE memory area (this can be ROM, if available, or EEPROM). The configuration value files use only the number of data bytes required by the data types of the configuration properties contained within. The **#pragma codegen put_read_only_cps_in_data_memory** directive instructs the linker to place the read-only value file in a modifiable memory area instead of the CODE memory area. See the *Compiler Directives* chapter in the *Neuron C Reference Guide* for more information about this directive. There is no additional overhead. The configuration template file is also placed in the

CODE memory area. The template file uses a number of bytes to describe each configuration property in the value files. This number varies based on the type and characteristics of the configuration property, but it is typically 12 or more bytes per configuration property instance. See the *Compiler Directives* chapter in the *Neuron C Reference Guide* for more information about the **#pragma codegen cp_family_space_optimization** compiler directive. Use of this directive can substantially reduce the size of the configuration property template file.

Using Neuron Chip Memory

The following section describes two different situations, using Neuron Chips or Smart Transceivers with off-chip memory, and using Neuron Chips or Smart Transceivers without off-chip memory.

Chips with Off-Chip Memory

On-chip memory for the Neuron 3150 Chip and FT 3150 Smart Transceiver consists of RAM and EEPROM. On-chip memory for the Neuron 5000 Processor and FT 5000 Smart Transceiver consists of RAM, but the memory map typically contains ROM, EEPROM, and RAM areas (optionally also flash memory). For a Series 5000 or 6000 chip, the system firmware automatically synchronizes non-RAM memory areas with the serial memory parts, as necessary.

For a device based on a Series 5000 chip, you specify the different offchip memory sections (RAM, EEPROM, etc) using the NodeBuilder Hardware Template Editor by their start and end addresses and additional attributes where necessary.

For a device based on a Series 6000 chip, you specify the exact type of serial flash memory used. The Neuron C toolchain automatically decides on the best split of this memory into volatile data ("RAM"), persistent data ("EEPROM") and application code and constant data ("flash"). The results of this automatic memory map tuning are reported in the Neuron Linker's memory map output.

Off-chip memory on these chips consists of one or more of ROM, RAM, EEPROM, NVRAM, or flash memory regions. You specify the starting page number for each region and the number of pages (a page is 256 bytes) when the device is defined. If ROM is used, its starting address must be 0000. If ROM is not used, then flash or NVRAM memory must take its place, starting at address 0000. The regions of memory must be in the order shown in **Figure 16**. They need not be contiguous, but they cannot overlap.

Memory mapped I/O devices can be connected to the Neuron 3150 Chip and FT 3150 Smart Transceiver. The devices should respond only to memory addresses that correspond to any of the shaded areas in **Figure 16**.
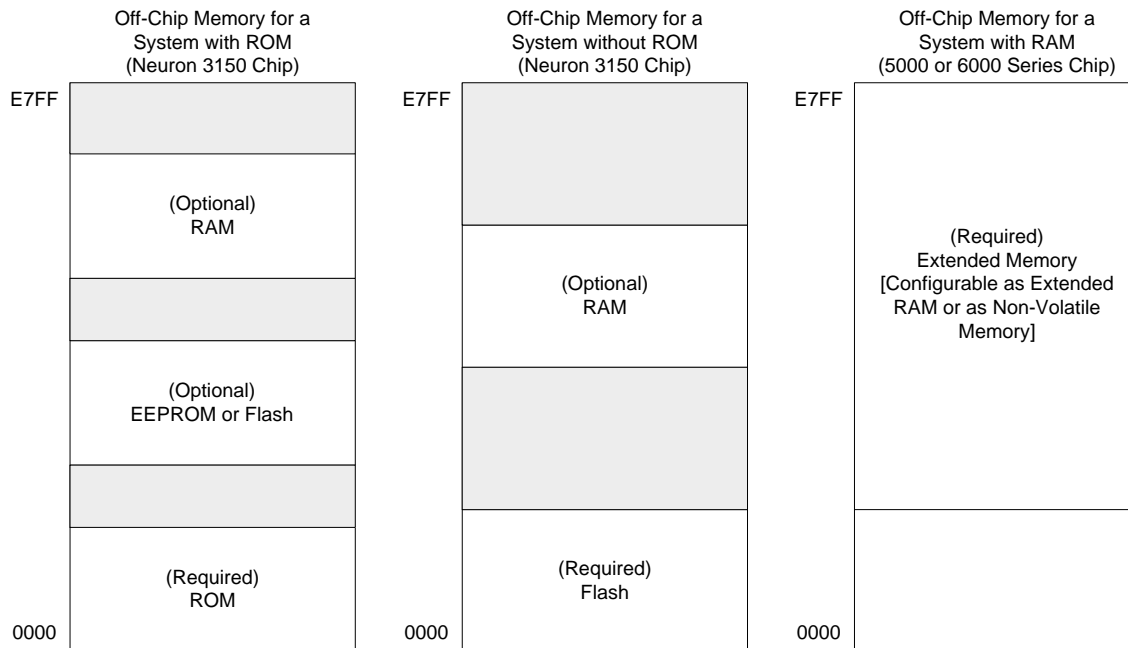
Figure 16. Off-Chip Memory for the Neuron 3150 Chip, the FT 3150 Smart Transceiver, the Neuron 5000 Processor, and the FT 5000 or 6000 Smart Transceiver

Chips without Off-Chip Memory

On-chip memory on the Neuron 3120 Chips and on the FT 3120 Smart Transceiver consists of ROM, RAM, and EEPROM. None of these devices supports off-chip memory. **Figure 17** summarizes the memory maps.
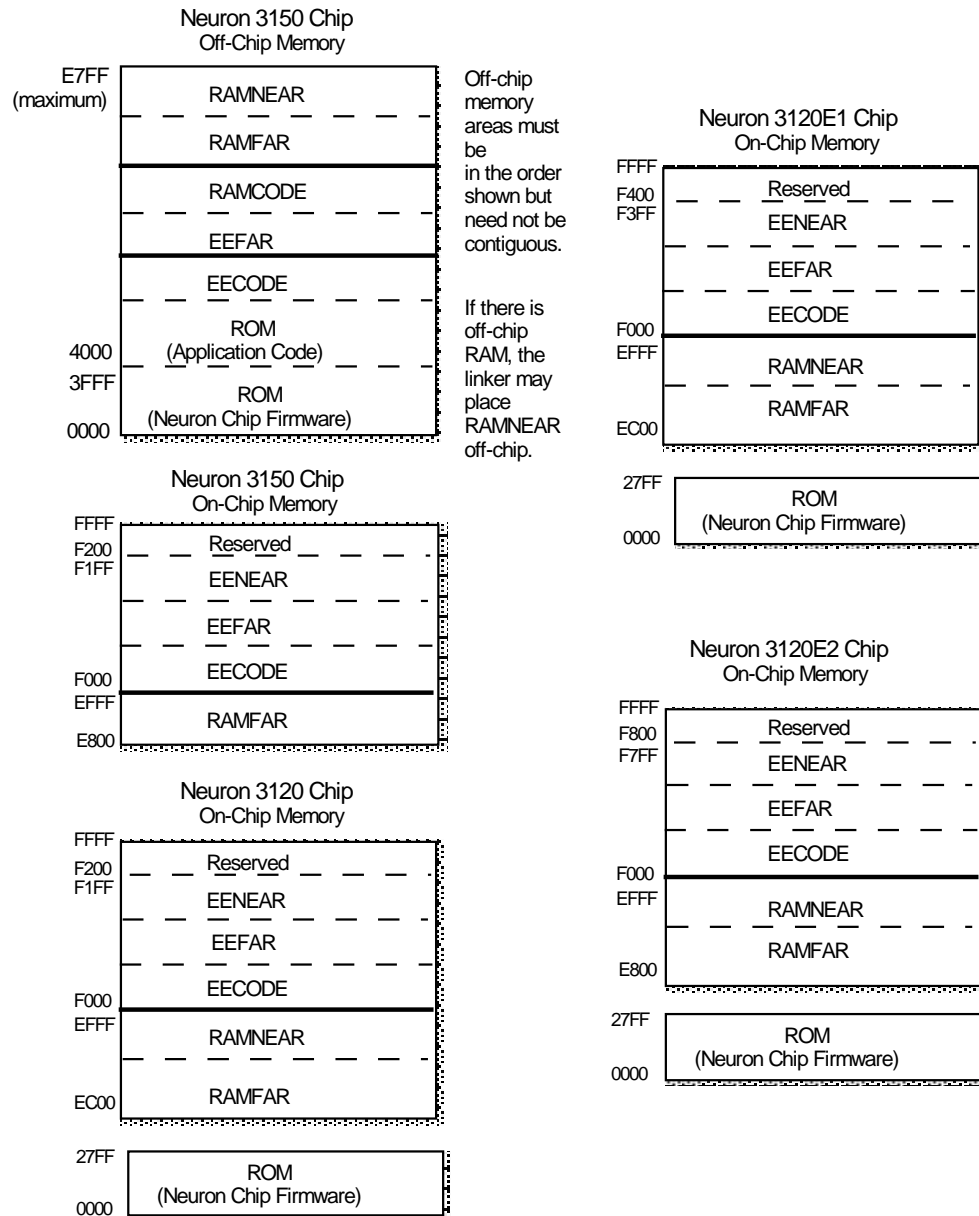
Figure 17. Memory Maps for the Various Chips, Showing Areas Defined by the Linker

Memory Regions

The definitions of the three memory regions are as follows:

- *ROM*: Non-volatile memory initialized before program execution on a device. ROM cannot be changed by the program. It is used for the Neuron firmware and can optionally (only on a Neuron 3150 Chip or FT 3150 Smart Transceiver) contain application code and constants. For a Series 5000 or 6000 chip, the ROM contains the system firmware image.

  Off-chip ROM (only on a Neuron 3150 Chip or FT 3150 Smart Transceiver) may be implemented with any non-volatile memory

technology, including ROM, PROM, EPROM, EEPROM, flash memory, or non-volatile RAM. Off-chip ROM or any memory technology used in its place must have a write-time delay of 0 (zero) ms if the memory is used to include the Neuron firmware and start at address 0x0000. This requirement prevents EEPROM from being used for storage of the Neuron firmware. Off-chip EEPROM *can* be used for application code and data storage.

- *EEPROM*: Non-volatile memory that can be changed during program execution. Memory writes typically require 20 ms per byte for on-chip EEPROM. EEPROM can contain application code, constants, and EEPROM variables.

   Off-chip EEPROM can be implemented with EEPROM, flash memory, or non-volatile RAM. If you use flash memory you must configure the IzoT NodeBuilder hardware template editor to indicate flash memory. Memory writes to this area can cause the Neuron firmware to delay. This delay allows the memory to complete the write. When implemented with EEPROM, the delay for off-chip EEPROM writes is configurable from 0 to 255 milliseconds using the NodeBuilder Hardware Template Properties dialog. The delay for off-chip flash memory writes is fixed at 10 ms for each 64- or 128-byte sector.

   For Series 3100 devices, if flash is used for the EEPROM region, it can also take the place of the ROM region. In this case, you cannot write to the system area of the flash (see *Memory Areas*), but you can write to the user area. For more information, including the particular flash parts supported, see *Use of Flash Memory* on page 197.

   EEPROM is not zeroed when the Neuron Chip or Smart Transceiver is reset.

- *RAM*: Volatile memory that can be changed during program execution. RAM can contain application code, constants, or variables.

   The Neuron hardware does <u>not</u> implement wait states for slow devices. The memory must be readable and writable in one machine cycle at the selected system clock rate.

   The off-chip RAM region can be used for code. Any portion of the off-chip RAM used for code is retained over resets. The remainder of RAM, the area not used for code, is zeroed each time the chip is reset.

Memory Areas

The Neuron firmware and the Neuron linker divide the memory regions into memory areas as follows:

- The ROM region has a system area and a user area (Neuron 3150 Chip and FT 3150 Smart Transceiver only). The system area is 16 KB (or larger) on a Neuron 3150 Chip or Series 5000 chip. The user area is also named ROM. The Neuron C compiler and linker place executable code and constant data in the user area, unless flash memory is used. When flash memory is used for ROM, user code is placed in the EECODE area.

- The EEPROM region consists of the following three areas:

  EECODE
  EEFAR
  EENEAR

  If there is both on-chip and off-chip EEPROM, each region of EEPROM has its own section of EECODE and EEFAR. There is only one section of EENEAR, and it is always located on-chip. All of these are user areas.

  EECODE contains executable code and constant data. The **eeprom** keyword in Neuron C forces the compiler to place a specific object in this area.

  The EEFAR area contains variables declared with the **far** keyword combined with either the **config** or **eeprom** keywords. This area also contains configuration property network variables declared with the **config_prop** (or **cp**) keyword and the modifiable configuration property file for configuration properties declared with the **cp_family** keyword.

  You can use the **offchip** and **onchip** keywords in Neuron C to force the compiler and linker to place specific objects in the offchip and onchip EEFAR areas.

  The EENEAR area contains variables declared with either the **config** or **eeprom** keywords. It is the default, but is limited to a total size of 255 bytes.

- The RAM region consists of three areas:

  RAMCODE
  RAMFAR
  RAMNEAR

  RAMCODE can only be located off-chip (Neuron 3150 Chip or 3150 Smart Transceiver only). It contains executable code and constant data. By using the **ram** keyword in a declaration, you can explicitly place executable code and constant data in this area. This area can only be implemented in a RAM that is based on a non-volatile memory technology, such as battery-backed RAM.

  RAMFAR may be located both on-chip and off-chip. There can be one or two sections of RAMFAR in the on-chip RAM. If there is off-chip RAM, it can contain only one RAMFAR area. The RAMFAR area contains variables.

  You can use the **offchip** and **onchip** keywords in Neuron C to force the compiler and linker to place specific objects in the offchip and onchip RAMFAR areas, respectively.

  There can be only one RAMNEAR area. It can be located on-chip (all chips) or off-chip (Neuron 3150 Chip and FT 3150 Smart Transceiver only). The linker automatically determines the location of the RAMNEAR area. The RAMNEAR area is the default memory area for all

Neuron C variables. This area is limited to a total size of 256 bytes. However, the maximum allowable size may be smaller under certain circumstances depending on the amount of memory the user has allocated for buffers. See *Compiler Directives for Buffer Allocation* on page 207, and also see *Controlling Non-Default Memory Usage* on page 193.

Note that for Series 5000 chips, the Extended Memory region can be configured for either extended RAM (for three RAM regions) or for non-volatile memory (for the three EEPROM areas).

Default Memory Usage

If no special keywords are included in the declarations of variables or functions or other pieces of a Neuron C program, the pieces of the program are located in memory by the linker using the following rules.

All executable code objects (functions, **when** clauses, tasks) as well as string constants and data declared as **const** are placed in the ROM or EECODE areas. The linker places these objects wherever they fit. For the Neuron 3150 Chip or FT 3150 Smart Transceiver, the linker first tries to put an object in the user area of *off-chip* ROM. If the object does not fit in ROM, the linker attempts to put it in the *off-chip* EECODE area of memory. Finally, the linker attempts to put the object in the *on-chip* EECODE area of memory.

Data objects declared with any of the **config**, **config_prop**, **cp**, or **eeprom** keywords are normally placed in the on-chip EENEAR area of memory. The linker normally places all data objects without these keywords in the RAMNEAR area of memory.

Data objects declared with the **cp_family** keyword (configuration property families) create multiple items stored in memory. Each configuration property family member instance creates a descriptor entry in the template file, and a data value entry in either the writeable value file or the read-only value file. The configuration template file is placed in either the ROM or EECODE area. The writeable value file is placed in the EEFAR area of memory. The read-only value file is placed in either the ROM or EECODE area of memory.

The NodeBuilder CodeWizard declares network variables so that they are placed in RAMNEAR memory by default. However, because of the larger number of network variables that are available for devices built with the IzoT NodeBuilder Software your device could run out of available RAMNEAR memory. In this case, declare as many network variables as necessary to use RAMFAR memory.

You can modify the linker's placement of variables and functions using the special keywords described in the next section.

Controlling Non-Default Memory Usage

If you receive an error message at link time that part of your program does not fit into the available default memory, you can change the declarations of variables or functions using special Neuron C keywords and using certain compiler directives. These keywords and directives enable you to move the variables or functions to other locations in memory. The **eeprom**, **far**, **offchip**, **onchip**, **ram**, and **uninit** special keywords are described below.

Direct memory reads and writes (by the application program) to data in the EENEAR and RAMNEAR areas take advantage of special addressing modes in the Neuron Chip and Smart Transceiver that generate more efficient code (fewer bytes per instruction and fewer cycles per read or write). However, indirect memory access (by a pointer) is the same for near and far memory areas.

eeprom Keyword (for Functions and Data Declarations)

Functions and **const** data are located in ROM by default. When ROM is full, or when no ROM is available, the remaining functions and **const** data are placed in the EECODE areas, first in the offchip EECODE area, then in the onchip EECODE area. However, functions and **const** data can be explicitly redirected from the ROM area to the EECODE area of memory by including the **eeprom** keyword in the function definition or data declaration. For example:

```
eeprom int fn() { ... statements ... }
eeprom const type varname = {inits};
```

The **eeprom** keyword is useful for functions that might be occasionally but rarely changed after installation by a network tool. Likewise, a network tool would be able to modify a **const** data structure stored in EEPROM that might be used for calibration, or other configuration.

This keyword also allows the program to indicate variables whose values are preserved across power outages by locating the variables in EENEAR rather than in RAMNEAR. However, EEPROM memory has a limited capability to accept changes. See the EEPROM vendor's data sheets for a discussion of the limit to the number of writes that a particular EEPROM can support.

You can redirect variables from the RAMNEAR area to the EENEAR area of memory by including the **eeprom** keyword in the declaration, as described earlier. For example, the following declaration moves *varname* to the EENEAR area:

```
eeprom int varname;
```

You can direct network variables to the EENEAR area with either the **eeprom** or **config** keyword. You can also use the **far** keyword with network variables similar to the example above.

The EENEAR area is limited in size to a maximum of 255 bytes (but other factors can limit this area further). Any additional on-chip EEPROM, and all off-chip EEPROM are considered EEFAR areas. To move a particular variable to an EEFAR area, see the discussions of the **far**, **offchip**, and **onchip** keywords below.

Initializers for **eeprom** class variables take effect when the application image is loaded from an external system, such as the LonMaker Integration Tool or another network tool. Reloading a program has the effect of reinitializing all **eeprom** variables. Restarting a device or powering it up does *not* re-initialize the **eeprom** variables – they retain their existing values from before the restart or power outage. For an exception to these initialization rules, see the description of the **uninit** keyword, below.

Writing a value in on-chip EEPROM typically takes approximately 20 ms before the value takes effect (though this time can vary depending on the particular

chip). If the device loses power or is reset during this write time, the value might not have been written or, if written, the value might not be non-volatile.

far Keyword (for Data Declarations)

When data objects do not fit into the RAMNEAR area of memory, the following linker error messages appear:

```
Error:   No more memory in RAMNEAR area
Error:   Could not relocate segment in file '<program>.no'
```

You can direct the linker to put the objects into the RAMFAR area of memory by including the **far** keyword in the Neuron C data declaration. For example, the following declaration moves *varname* to the RAMFAR area:

```
far int varname;
```

To direct the linker to put the objects into the RAMFAR area of memory within the NodeBuilder CodeWizard:

1. Right-click the network variable and select **Properties** to open the NV Properties dialog.

2. Within the NV Properties dialog, click **Advanced** to open the Advanced NV Properties dialog.

3. Within the Advanced NV Properties dialog, select **far**. Click **OK** to close the dialog.

4. Within the NV Properties dialog, click **OK** to close the dialog.

Similarly, when **config** or **eeprom** objects do not fit into the EENEAR area of memory, the following messages appear:

```
Error:   No more memory in EENEAR area
Error:   Could not relocate segment in file '<program>.no'
```

You can direct the linker to put the objects into the EEFAR area of memory by also including the **far** keyword in the Neuron C data declaration. For example, the following declaration moves *varname* to the EEFAR area:

```
far eeprom int varname;
```

You could, for example, move a data table that is too large to fit into the EENEAR area to the EEFAR area of memory using this type of declaration.

As a general guideline, leave data that is *more* frequently used in the NEAR areas of memory if possible. Use of the NEAR areas generates relatively smaller instructions (which additionally execute in fewer cycles) than use of the FAR areas. Arrays that are referenced only with non-constant indices or pointers can be placed in FAR memory with no loss of efficiency.

offchip Keyword (for Functions and Data Declarations)

The Neuron linker typically places code, const data, and far variables in off-chip areas, if it can, and in on-chip areas when it must. However, the linker's default behavior is different when linking for flash memory. See *Use of Flash Memory* on page 197 for more information. To explicitly control the placement of these objects, any data or function declaration can include the **offchip** keyword.

If the appropriate off-chip memory area is available, the object is placed in the area. If the memory area is not available, the linker terminates with an error message to that effect. Examples of using the **offchip** keyword are shown below:

```
far offchip int a;             // offchip RAMFAR
far eeprom offchip int b;      // offchip EEFAR
const eeprom offchip int c = init;
    // offchip EECODE (no need for far kwd)
eeprom offchip void fn () {...}
    // offchip EECODE
```

onchip Keyword (for Functions and Data Declarations)

The Neuron linker typically places code, **const** data, and **far** variables in off-chip areas, if it can, and in on-chip areas when it must. To explicitly control the placement of these objects, any data or function declaration can include the **onchip** keyword. If the appropriate on-chip memory area is available, the object is placed in the area. If the memory area is not available, the linker terminates with an error message to that effect. See *Use of Flash Memory* on page 197 for more information. Examples of using the **onchip** keyword are shown below:

```
far onchip int a;              // onchip RAMFAR
far eeprom onchip int b;       // onchip EEFAR
const eeprom onchip int c = init;
    // onchip EECODE (no need for far kwd)
eeprom onchip void fn () {...}
    // onchip EECODE – would be in EEPROM
    // even without the eeprom keyword
```

The **onchip** keyword is useful for moving data to on-chip EEPROM when off-chip flash memory is used. The on-chip EEPROM supports more write cycles than off-chip flash memory. Frequently updated EEPROM variables should be located on-chip when off-chip flash memory is used. See the EEPROM vendor's data sheets for maximum write specifications.

ram Keyword (for Functions)

By default, functions and other executable code, as well as **const** data, are all placed in ROM, if available (on the Neuron 3150 Chip or FT 3150 Smart Transceiver), and then in off-chip or on-chip EECODE. You can redirect functions to the off-chip RAMCODE area of memory by including the **ram** keyword in the Neuron C function definition. The RAMCODE area is only available in off-chip RAM memory attached to a Neuron 3150 Chip or FT 3150 Smart Transceiver. The RAM must be non-volatile (for example, battery-backed), if the device is to be protected against a power cycle. See the *Neuron 3150 Chip External Memory Interface* engineering bulletin for special considerations on protecting code RAM against Neuron Chip resets.

The **ram** keyword can go anywhere before the function name. For example:

```
ram int fn() { ... statements ... }
```

The **ram** keyword is useful for functions that a network tool might change frequently after installation.

uninit Keyword (for Data Declarations)

You can combine the **uninit** keyword with **eeprom** variable declarations to declare data in EENEAR or EEFAR memory areas which is not affected by program load or chip reset. This type of declaration can provide two benefits. You might need a large area of allocated memory for database or calibration or other use, and might want the data to remain unaffected in these situations. Furthermore, **uninit** areas of EEPROM are not loaded, thus speeding up loading time. An example of using the **uninit** keyword to set aside 500 bytes of such memory is shown below:

```
uninit eeprom int datablock[500];
```

Compiler Directives

Configuration property value files and the configuration property template files, which hold values and self-documentation data for configuration properties declared with the **cp_family** keyword, can be allocated in on-chip EEPROM or off-chip EEPROM using the linker's default relocation algorithm. You can use the **#pragma codegen put_cp_template_file_offchip** and **#pragma codegen put_cp_value_files_offchip** compiler directives to force the template file or the value files into off-chip memory, if off-chip memory is available. If insufficient off-chip memory is available, forcing the files to off-chip memory cause the link to fail. See *Compiler Directives* in the *Neuron C Reference Guide* for more details about these directives.

When the Program Is Relinked

The compiler directs the application code to the proper areas of memory. The linker assigns data memory locations and resolves references to global symbolic addresses. These assignments to addresses occur in the order of declaration in the compilation. Therefore, to retain the same addresses from link to link, maintain the same order of declaration.

Use of Flash Memory

Use of flash memory is specified in the IzoT NodeBuilder hardware template editor. See the *IzoT NodeBuilder User's Guide* for more information on these features, and how to select use of flash memory.

Use of Flash Memory for Series 3100 Chips

For the Neuron 3150 Chip and the FT 3150 Smart Transceiver, Neuron firmware version 6 and later supports the use of flash memory. The firmware supports only the following flash parts:

- Atmel® AT29C256

- Atmel AT29C257 (32Kx8, 64 byte sector size)

- Atmel AT29C512 (64Kx8, 128 byte sector size)

- Atmel AT29C010 (128K x 8, 128 byte sector size)

Neuron 3150 Chips and FT 3150 Smart Transceivers that use these flash memory parts must run at 1.25 MHz clock speed, or faster.

You can use flash memory for just the EEPROM memory region, or for both the ROM and EEPROM memory regions. When you use flash memory for the EEPROM memory region, it can contain all memory areas normally associated with EEPROM, that is, EECODE and EEFAR. Flash memory can only be modified reliably a limited number of times (typically, 1000 to 10,000 times, but varies depending on the chip – consult the chip manufacturer's data books for specific limitations). Thus, even though the compiler, linker, and firmware support placing **eeprom** class variables in flash memory, you must take care that these variables are modified infrequently, within the specified lifetime of the flash memory parts.

When you also use flash memory for the Neuron firmware ROM, the ROMCODE area's size is fixed at the size of the system image (16 KB or more) and therefore cannot contain application code or data. The remainder of the flash memory address space is divided between EECODE and EEFAR. However, the flash memory should not contain both the system image and application read/write data, because when the flash is being written to, the system image cannot be read during the chip's programming interval (<10 msec). The Neuron firmware automatically locks itself out of the flash memory during the programming interval, but this lockout causes all system functions to be delayed, thus network messages can be lost during the lockout interval. If you must place data in flash memory, you can use the **offchip** keyword to direct the variables that are the least likely to be modified into flash memory.

Because flash memory is written on a sector basis (either 64, 128, or 256 bytes depending on the part), the number of writes possible for an **eeprom** variable located in flash is a function of the number of writes to the sector where it resides. As a comparison, consider a one-byte **eeprom** variable located in an off-chip EEPROM supporting 1,000 writes. This variable can be safely updated 1,000 times. If that same variable resided in an off-chip flash memory part with 64 byte sectors and a similar write limit, then the number of writes must be stated as a function of all the data in that sector. Specifically, up to 1,000 writes could be done for all the variables in the sector. Given 64 one-byte variables with equal frequency of modification, each could be written 1,000/64 or about 15 times.

Because modification of data in flash memory requires a flash programming cycle, the Neuron C linker uses an alternate linking algorithm, placing all data objects in on-chip memory if it can. You can control which objects are placed off-chip, if some must be, by use of the **offchip** keyword in the data object's declaration. If the linker places any data in flash memory, a warning is given to this effect.

Any direct write by the application to the flash memory causes a programming cycle, even if the flash memory is write protected. This condition is an error that occurs when the application bypasses the Neuron firmware for accessing the flash memory. The Neuron firmware uses the software write protection feature of the flash memory, so the invalid write does not change the contents of the flash memory, but a programming cycle is initiated. The flash memory provides invalid data during the programming cycle, causing a watchdog reset. It is possible for the reset to occur during a write cycle, which could, in the worst case, cause recurring resets. Therefore, having the system image in flash memory requires that there be a hardware mechanism to extend the device's reset state for at least the duration of the write cycle (typically 10 msec). The LVI circuitry

described in the Smart Transceivers databooks can accomplish this. See also the *Neuron 3150 Chip External Memory Interface* engineering bulletin for more information.

When writing to EEPROM, if a power cycle or other reset occurs, the data corruption is localized to the area being written. However, with flash memory, because an entire sector is always programmed at once, all data in the sector which was being written at the time of the failure is suspect. It is up to the application program to protect any critical non-checksummed read/write data through duplication, voting, journaling, or whatever technique is appropriate.

Because loading of flash memory occurs a sector at a time, it is important that the load image data be contiguous. Thus, **uninit eeprom** data and initialized **eeprom** data should not be interleaved. The linker processes declarations in the order they appear in the program, thus you can reduce loading time by grouping **uninit eeprom** declarations together.

Use of Flash Memory for Series 5000 Chips

For Series 5000 devices, the firmware supports the following flash memory parts:

- Atmel AT25F512B 512-Kilobit 2.7-volt Minimum SPI Serial Flash Memory

- Numonyx™ M25P05-A 512-Kbit, serial flash memory, 50 MHz SPI bus interface

- Silicon Storage Technology SST25VF512A 512 Kbit SPI Serial Flash

You can use flash memory for the Extended Non-Volatile memory region for EECODE data. You can also use an additional 16 KB of the flash memory to store an alternate Neuron firmware system image. Because of the large sector size of these flash memory parts, you cannot use flash memory for EENEAR or EEFAR data (**eeprom** class variables); the entire data region must be erased to modify data in this memory region. The use of flash memory for Series 5000 chips does not require additional RAM, but does require additional EEPROM for the storage of a device-specific flash memory driver.

If you use the **uninit eeprom far** keywords to create storage variables in flash memory, that application has access to memory that is initially erased to an "all ones" state after application load. This memory can only be modified by setting data bits to the zero state through memory writes.

When using a 32 KB sector flash device, the available space for the Extended Non-Volatile memory region is limited to 32 KB. For a smaller sector flash device (such as the SST25VF512A) the available space is 42 KB.

The flash memory that is used for the EECODE data is erased once during the application load process. If present, the alternate Neuron firmware image is not erased during this step. Flash memories typically have an endurance rating of 10 000 erase/program cycles – this number translates directly to the number of application load cycles.

The Series 5000 chip can operate at any system clock speed when using flash memory.

Use of Flash Memory for Series 6000 Chips

For Series 6000 devices, the firmware supports a number of 512 and 1024 KB serial flash memory parts (SPI interface). This is used for constant data and resident code, and portions of this memory are used for persistent data storage emulating EEPROM.

You must select the part matching your hardware from the *Non-volatile memory type* list on the *Memory* tab provided by the IzoT NodeBuilder Hardware Template editor.

Supported parts include:

Winbond W25X40CL and W25Q80BV 512 and 1024 KB parts, the

ON Semiconductor LE25U40CMC 512 KB part, the

Micron M25PX80 1024 KB part, the

Macronix MX25L8035E 1024 KB part, and the

Adesto AT25SF041 and AT25SF081 512 and 1024 KB serial flash memory parts.

(Note this list may change over time.)

The eeprom_memcpy( ) Function

You can write EEPROM memory, as well as flash memory, by direct assignment, including structure-to-structure assignment. In these cases, the compiler recognizes that the target variable is in EEPROM memory, and uses the appropriate firmware function to write the memory properly, with the correct delays, hardware interface sequences, and so on.

However, when writing to EEPROM through a pointer, the compiler cannot track what type of memory the pointer points to. Thus, addresses of EEPROM variables are automatically typed by the compiler as '**const ***', to prohibit use of the pointer for writing:

```
eeprom int x;

void y() {
    const int* pointer = &x;  // '&x' is 'const int *'
}
```

The compiler normally prevents removing the **const** attribute from any pointer so typed. This is prevented for both implicit and explicit cast operations. An implicit cast occurs when there is an assignment of a value to a variable of different type, or when there is an actual parameter passed to a function whose formal parameter is a different type, as illustrated in the following example:

```
eeprom int x;
int *p;
void f (int *p);

void y() {
    p = &x;          // implicit cast, compiler error
    f(&x);           // another erroneous implicit cast
    p = (int *)&x;   // explicit cast, also error
```

}

This behavior of the Neuron C compiler is stricter than the behavior specified by
ANSI C. However, if you specify the **#pragma relaxed_casting_on** directive,
the compiler only generates a warning message for each such implicit or explicit
cast. You can use the **#pragma warnings_off** or **#pragma disable_warning**
directive to further suppress the warning messages. You can use the
corresponding **warnings_on** (or **enable_warning**) and **relaxed_casting_off**
directives later in the program to restore the default behavior of the compiler for
the remainder of the program.

Use of this feature is dangerous, because you can circumvent the compiler's
checking and attempt a spurious write (that is, a write without knowledge of the
firmware) to EEPROM or flash memory. The **eeprom_memcpy( )** function is
provided to write by pointers which can (but are not required to) refer to
EEPROM or flash memory. The parameters of this function are the same as that
of **memcpy( )**, but this function supports the destination address being in
EEPROM or flash memory, whereas the normal **memcpy( )** function does not.
The **eeprom_memcpy( )** function limits the length parameter to 255 bytes.

Note: The use of this function in older versions of firmware with an excessive
length can cause the watchdog timer to time out, causing the device to reset. For
a Series 3100 device running at 10 MHz, the safe length is 32 bytes.

For Neuron 3120xx Chips and FT 3120 Smart Transceivers running version 7
firmware or later, Neuron 3150 Chips and FT 3150 Smart Transceivers running
version 12 firmware or later, and Series 5000 chips running version 18 firmware
or later, the firmware prevents a watchdog timer timeout during use of
eeprom_memcpy( ).

Reallocating On-Chip EEPROM

The Neuron C compiler generates four tables in EEPROM (on-chip for Series
3100 devices, off-chip for Series 5000 devices) that are used by the Neuron
firmware and network tools to define the network configuration for a device. Two
of these tables are the domain table and address table. By default, the domain
table is generated with two entries. This is the maximum and recommended
configuration. You can override this behavior and enforce a single domain record
with the *pragma domains* directive (also known as *pragma
num_domain_entries*).

The address table holds at least 15 address records by default, but the Neuron C
Compiler may suggest a larger address table based on inspection of your
application. You can override the automatic address table allocation using the
pragma addresses directive (also known as *pragma addr_table_entries*), enforcing
0 to 254 addresses where supported, or 0 to 15 addresses.

A third table, the alias table, is also automatically allocated by the Neuron C
Compiler based on inspection of your application's interface. The compiler
allocates 0 to 254 aliases (up to 15 on some legacy targets). You can override the
automatic alias table allocation using the *pragma aliases* directive (also known as
pragma num_alias_table_entries), enforcing 0 to 254 aliases on most systems.
Some older targets are limited to 15 aliases.

See the *Compiler Directives* chapter in the *Neuron C Reference Guide*, and the discussions below, for more information.

A fourth table, the network variable configuration table, is generated automatically with one entry for each network variable declared in the program. Each element of a network variable array counts separately, and the maximum size of the network variable configuration table is equal to the maximum number of network variables that the device can support. Each entry uses three bytes of EEPROM on legacy targets, four bytes on targets which support up to 254 address table records. You cannot change the size of this table, except by adding or deleting network variables.

If a program does not fit into the default memory areas, another alternative when using the Neuron 3150 Chip or the FT 3150 Smart Transceiver is to move parts of the program to other locations in memory. However, the domain table, address table, alias table, and network variable configuration table must be located in on-chip EEPROM. See *Chips with Off-Chip Memory* on page 188.

Address Table

The address table contains the list of network addresses to which the device sends network variable updates or polls, or sends implicitly addressed application messages. The address table can be configured through network management messages from a network tool.

Note: See the *ISO/IEC 14908-1 Control Network Protocol* standard for a description of the address table.

By default, the address table contains 15 entries for the Series 3000 and 5000 chips and 254 entries for the Series 6000 chip. Each address table entry uses five bytes of on-chip EEPROM. Use the following compiler directive to control the number of address table entries:

> **#pragma addresses <*num*>**
> (*nn* can be any value from 0 to 15 (or 254 for Series 6000 chips)

The original directive, **#pragma num_addr_table_entries *nn***, still works in the same way, but the simplified *pragma addresses* directive is recommended for new development. The maximum number of address table entries that a device could require is bounded by the maximum expected number of different destination entries required by connections (both network variables and message tags) for that device. A destination entry is required for an output network variable or message tag in a connection, if the output is not declared as a polled output, and also for the input if the input is polled or a member of a group connection. Two destination entries differ if they use a different service type, a different destination address, or different transport attributes (such as the repeat timer). Multiple network variables that use the same destination entry share a common address table entry. Fewer address table entries are consumed when address table entries can be shared by multiple connections. This capability can only be used if the network tool used to install the device generates shared entries (all LNS tools, including the LonMaker Integration Tool, provide this capability). Support for an extended address tables requires that the network variable and network variable aliases configuration structures (*nv_struct*, alias_struct) grow by one byte for each record to accommodate the extended address table index value.

As a general rule, the address table should be sized to at least 15 entries, if possible. However, because the address table is subject to frequent linear search operations, it is not recommended to allocate the maximum size of 254 entries (where supported), unless the application and the anticipated network variable connection complexity in a deployment requires it.

When compiling for Series 6000 Chips that support an extended address table, the compiler automatically defines the preprocessor symbol (_SUPPORT_EAT). This is used to control related definitions in some Neuron C header files (echelon.h, access.h) and can be used for other application-specific conditional compilation.

Alias Table

The alias table is generated based on a default size computed by the Neuron C Compiler. This may be overridden with a new **#pragma aliases *<num>*** compiler directive, shown below. (The original **#pragma num_alias_table_entries *nn*** directive still works in the same way, but *pragma aliases* is recommended for new development.) This compiler directive can be used to set the alias table size to any size between zero and 127 entries. Each alias entry uses 4 bytes of EEPROM (on-chip for Series 3100 devices or offchip for Series 5000 devices), and 5 bytes of EEPROM on systems which support more than 15 address table records (Series 6000 devices). An alias is an abstraction for a network variable that is managed by network tools and the Neuron firmware. Network tools use aliases to create connections that cannot be created solely with the address and network variable tables, providing network integrators with more flexibility in how devices are installed into networks. This feature requires Neuron firmware version 6 or later.

> **#pragma aliases *<num>***
> (see **Table 13** for the maximum value of *nn*)

How many network variable aliases a Neuron-hosted device can support depends on the device's memory map, the system firmware, and the development tool version, as shown in **Table 13**.

Table 13. Alias Limits

Neuron System Firmware Version	IzoT Nodebuilder Software Version	Maximum Number of Aliases
Version 15 or earlier	Any	62
Version 16 or later	3.1 or earlier	62
	FX or later	127

The maximum number of aliases for applications developed with the Mini EVK Evaluation Kit is 32. The limits for host-based applications depend on the development product used.

Note: See the *ISO/IEC 14908-1 Control Network Protocol* standard for a description of the alias table.

As a general rule, the alias table should be sized in proportion to the number of network variables that are likely to require an alias, but you should not allocate more alias table entries than the application could actually require. Although network variable aliases provide an important tool to enable versatile network variable connections, an excessively sized alias table can adversely affect overall system performance, especially if much of that alias table is unused.

The following rule-of-thumb allows calculation of a starting point for alias table size, *nn*:

$nn = 0$; for nv_count = 0

$nn = 10 + (\text{nv\_count} / 3)$; for nv_count > 0

Starting with this initial size, the alias table size can be refined based on your understanding of your application.

Domain Table

By default, the domain table is configured for two domains. Each domain uses 15 bytes of EEPROM (on-chip for Series 3100 devices or offchip for Series 5000 devices). The number of domain table entries is a function of the network where the device is installed, it is not a function of the application. You can reduce the size of the domain table using the following compiler directive:

#pragma domains 1

(The previous form of this directive, *pragma num_domain_entries*, still works in the same manner. However, *pragma domains* is recommended for new development.)

Notes:

- See the *ISO/IEC 14908-1 Control Network Protocol* standard for a description of the domain table.

- As a general rule, the domain table should be sized to the maximum of 2 entries, if possible. LONMARK International requires all interoperable LONWORKS devices to have two domain table entries. Reducing the size of the domain table to one entry prevents certification.

Allocating Buffers

You can use compiler directives to set certain Neuron firmware memory resources, such as buffer counts and sizes and receive transaction counts. These values can be set only during compilation. They cannot be configured at run-time. **Figure 18** on page 205 illustrates where application and network buffers are used. *Application buffers* are used between the application and network processors. *Network buffers* are used between the network and media access control (MAC) processors.
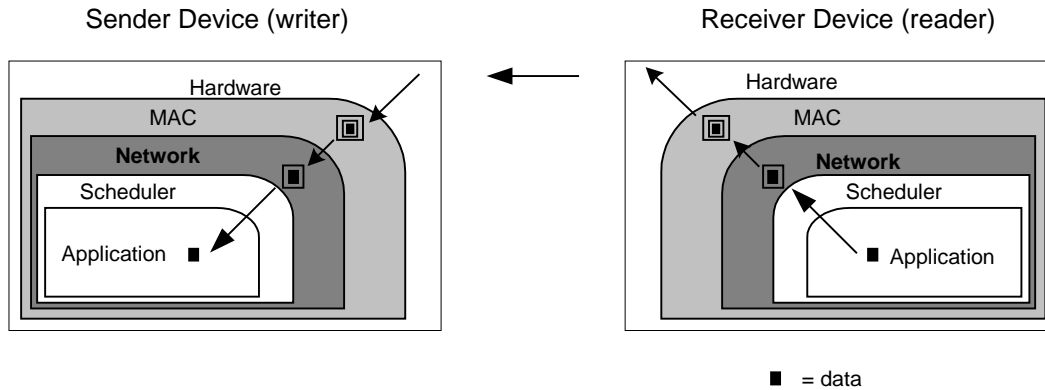
Figure 18. Application and Network Buffers

This section outlines a few guidelines for allocating buffers, depending on the needs of an individual application.

Buffer Size

If you use application messaging, you must set the appropriate buffer sizes to accommodate the largest message that the application or Neuron firmware could generate or receive for processing. In some cases, this could require an increase in buffer size. If you only use network variables, the compiler chooses buffer sizes based on the size of the largest network variable that you declare and the minimum sizes required by the Neuron firmware.

Figure 19 shows the basic components of an application buffer and a network buffer. An application buffer contains application message data and system overhead. A network buffer contains application message data, protocol layer 2 through layer 5 overhead, and system overhead.
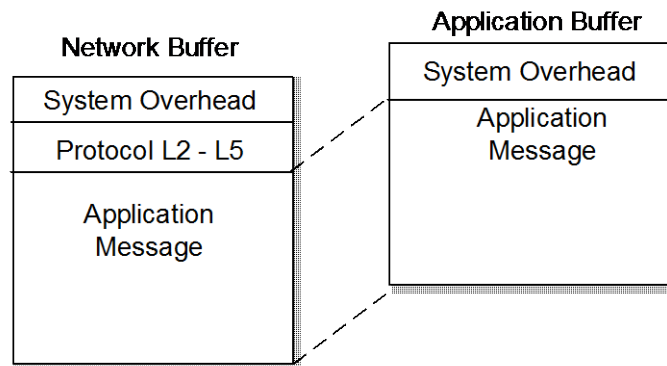
Figure 19. System and Protocol Overhead in Application and Network Buffers

Application Buffer Size

The size of an application buffer is equal to the following:

message_size + 5 bytes of system overhead

If explicit addressing is used, add an additional 11 bytes of system overhead.

For application messages, *message_size* equals 1 byte for the message code plus the number of bytes of data. For network variables, *message_size* equals 2 bytes plus the number of bytes in the network variable.

Table 14 on page 210 lists the valid sizes for application buffers. For example, if *message_size* is 40, then you need an application buffer of at least 45 bytes. However, the next largest valid size for an application buffer is 50 bytes.

Application buffers are also used to receive network management messages used by network tools for device configuration. A minimum input application buffer size of 22 bytes (34 bytes if explicit addressing is used) is required to be able to accommodate the largest possible network management messages.

The Neuron C Compiler selects a miniumum of 114 bytes for application buffers on systems with LonTalk/IP support. This buffer size supports most common frames, including a typical *ping* message and response.

Network Buffer Size

The size of a network buffer is less than or equal to the following:

message_size + 6 bytes of system overhead + 20 bytes of protocol overhead

Protocol overhead ranges from 5 to 20 bytes per message, and this formula uses the maximum. A 40-byte message would thus need a network buffer of at least 66 bytes (see **Table 14** on page 210).

Network buffers are also used to receive and respond to network management messages that are used by network tools for device configuration. A minimum input network buffer size of 42 bytes and output network buffer size of 50 bytes is required to be able to accommodate the largest possible network management messages.

The Neuron C Compiler selects a miniumum of 114 bytes for network buffers on systems with LonTalk/IP support. This buffer size supports most common frames, including a typical *ping* message and response.

Errors

If an input message fits into a network input buffer but does not fit into an application input buffer, the message is discarded. An **APP_BUF_TOO_SMALL** error code is logged by the Neuron firmware. If the message was sent with the acknowledged service, no acknowledgment is sent. If the message was a request, no response is sent.

If an output message fits into an application output buffer but does not fit into a network output buffer, a **NET_BUF_TOO_SMALL** error is logged and the device resets.

Buffer Counts

In most cases, the default number of output application buffers is sufficient. Increasing the number of application buffers on the output side decreases the likelihood of entering preemption mode if you are using synchronous network variable outputs (see *Preemption Mode* on page 61).

The number of input network buffers needed is a function of the types of service used and the types of connections between devices:

- If you are using authentication, you might need to increase the number of network buffers because authentication doubles the number of messages.

- If your device is installed with *unicast* connections (that is, one device sends a network variable or message to one other device), the default number of network buffers is probably sufficient.

- If your device is installed with *multicast* acknowledged or multicast request connections (that is, one device sends a message to a group of devices and expects a response from each), the number of network input buffers should be at least equal to the size of the largest group.

- If your application frequently has blocking sequences, that is, when tasks that do not complete for a relatively long time, the default application buffer count might be insufficient.

If, for example, a device sends a message with the acknowledged or request service to 63 different devices, the sender device could receive 63 almost simultaneous acknowledgments or responses. In general, large acknowledged connections should not be used because the same message delivery reliability can be achieved using repeated messaging, with far less network traffic. Network variable updates using the repeated delivery service do not generate any acknowledgements, and therefore do not require any input buffers.

The exact number of network input buffers required is a function of both bit rate and the system clock rate, so some experimentation might be necessary to determine the minimum number of buffers.

Compiler Directives for Buffer Allocation

The following sections describe the compiler directives used for setting the size and number of different types of buffers.

The compiler issues warnings when any of the buffer size compiler directives are used and the resulting settings are too small to accommodate all possible network management messages from being properly received or responded to.

Each of the buffer directives can include an optional modifier specification after the number of buffers. The modifier can be either of the following keywords:

- **minimum**

  The **minimum** keyword specifies that the number of buffers indicated for the pragma directive is the *minimum* number of buffers required for the application. The compiler defines at least as many buffers as specified in the directive, but could define additional buffers if the application requires them. This keyword encourages modular development, wherein different modules could have different buffer count or size requirements.

- **final**

  The **final** keyword specifies that the number of buffers indicated for the pragma directive is the *absolute* number of buffers to be used by the application. The compiler defines exactly as many buffers as specified in

the directive, and issues an error if additional buffers are required. This keyword promotes self-documenting source code.

Example: The following two directives specify the incoming network buffers. The first directive specifies that the compiler should define at least 66-byte buffers; the second directive specifies that the compiler should define exactly 114-byte buffers.

```
#pragma net_buf_in_size 66, minimum

#pragma net_buf_in_size 114, final
```

As the example shows, you can specify a **final** value that is larger than a previously specified **minimum** value. However, if you specify a **final** value that is smaller than a previously specified **minimum** value, the compiler issues an error message (**NCC#593**).

```
#pragma net_buf_in_size 114, minimum

#pragma net_buf_in_size 66, final
// ERROR – final smaller than minimum
```

Outgoing Application Buffers

These compiler directives set the size and number of nonpriority and priority buffers between the application and network processors for outgoing messages and network variables. See **Table 14** on page 210 for a list of default and allowable non-default values.

#pragma app_buf_out_size *n* [, *modifier*]

Sets the application buffer size (in bytes) for outgoing priority and nonpriority application messages and network variables.

#pragma app_buf_out_count *n* [, *modifier*]

Sets the number of application buffers available for outgoing nonpriority application messages and network variables.

#pragma app_buf_out_priority_count *n* [, *modifier*]

Sets the number of application buffers available for outgoing priority application messages and network variables.

Outgoing Network Buffers

These compiler directives set the size and number of nonpriority and priority buffers between the network and MAC processors for outgoing application messages and network variables. See **Table 14** on page 210 for a list of default and allowable non-default values.

#pragma net_buf_out_size *n* [, *modifier*]

Sets the network buffer size (in bytes) for outgoing priority and nonpriority application messages and network variables. A minimum size of 42 bytes is required to respond correctly to network management messages from network tools.

#pragma net_buf_out_count *n* [, *modifier*]

Sets the number of network buffers available for outgoing nonpriority messages and network variables.

#pragma net_buf_out_priority_count *n* [*, modifier*]

Sets the number of network buffers available for outgoing priority messages and network variables.

Incoming Network Buffers

These compiler directives set the size and number of buffers between the MAC and network processors for incoming explicit messages and network variables. See **Table 14** on page 210 for a list of default and allowable non-default values.

#pragma net_buf_in_size *n* [*, modifier*]

Sets the network buffer size (in bytes) for incoming application messages and network variables. A minimum size of 50 bytes is required to receive network management messages from network tools.

#pragma net_buf_in_count *n* [*, modifier*]

Sets the number of network buffers available for incoming application messages and network variables.

Incoming Application Buffers

These compiler directives set the size and number of buffers between the network and application processors for incoming application messages and network variables. See **Table 14** on page 210 for a list of default and allowable non-default values.

#pragma app_buf_in_size *n* [*, modifier*]

Sets the application buffer size (in bytes) for incoming application messages and network variables. A minimum size of 22 bytes (34 bytes if explicit addressing is used) is required to receive network management messages from network tools.

#pragma app_buf_in_count *n* [*, modifier*]

Sets the number of application buffers available for incoming application messages and network variables.

Number of Receive Transactions

The number of incoming transactions that can be handled concurrently by the network processor is determined by the receive transaction array. The following compiler directive sets the number of entries in the array. The size of a receive transaction entry is 13 bytes. See **Table 14** on page 210 for a list of default and allowable nondefault values.

#pragma receive_trans_count *n*

Sets the number of entries in the receive transaction array. The size of a receive transaction block is 13 bytes.

A receive transaction entry is required for any incoming message that uses either unacknowledged repeat, acknowledged, or request service. No receive

transaction entries are required for unacknowledged service. A receive transaction entry is required for each unique source address/destination address/priority attribute. Each receive transaction entry contains a current transaction number. A message is considered to be a duplicate if its source address, destination address, and priority attribute vector into an existing receive transaction and the message's transaction number matches the entry's transaction number.

Receive transaction entries are freed after the receive timer expires. The receive timer duration is determined by the destination device and varies as a function of the message addressing mode. For group addressed messages, the receive timer is determined by the address table. For Neuron ID addressed messages, the receive timer is fixed at eight seconds. For other addressing modes, the non-group receive timer in the configuration data structure is used.

Table 14. Values for Buffer Sizes and Counts

Pragma	Values Allowed	Default (See Notes)
app_buf_out_size	20, 21, 22, 24, 26, 30, 34, 42, 50, 66, 82, 114, 146, 210, or 255 bytes	A
app_buf_out_count	1, 2, 3, 5, 7, 11, 15, 23, 31, 47, 63, 95, 127, 191	E
app_buf_out_priority_count	0, 1, 2, 3, 5, 7, 11, 15, 23, 31, 47, 63, 95, 127, or 191	E
net_buf_out_size	*(20, 21, 22, 24, 26, 30, 34)*, 42, 50, 66, 82, 114, 146, 210, or 255 bytes (A minimum value of 42 bytes is recommended)	B
net_buf_out_count	1, 2, 3, 5, 7, 11, 15, 23, 31, 47, 63, 95, 127, 191	E
net_buf_out_priority_count	0, 1, 2, 3, 5, 7, 11, 15, 23, 31, 47, 63, 95, 127, or 191	E
net_buf_in_size	*(20, 21, 22, 24, 26, 30, 34, 42)*, 50, 66, 82, 114, 146, 210, or 255 bytes (A mimimum value of 50 bytes is recommended)	66
net_buf_in_count	1, 2, 3, 5, 7, 11, 15, 23, 31, 47, 63, 95, 127, 191	2

Pragma	Values Allowed	Default (See Notes)
app_buf_in_size	*(20, 21, 22, 24, 26, 30)*, 34, 42, 50, 66, 82, 114, 146, 210, or 255 bytes (A minimum value of 34 bytes is recommended if explicit addressing is used, otherwise a minimum value of 22 bytes is recommended)	C
app_buf_in_count	1, 2, 3, 5, 7, 11, 15, 23, 31, 47, 63, 95, 127, 191	2
receive_trans_count	1 .. 16	D

Notes:

A. **app_buf_out_size** default

If outgoing messages are sent with **msg_send( )**:

If explicit addressing is used:
A = 66

If explicit addressing is not used:
A = 50

If no outgoing explicit messages are sent (**msg_send( )** is not used), and:

If explicit addressing is used for network variables:
A = max(34, 19 + sizeof(largest output NV))

If explicit addressing is not used:
A = max(20, 8 + sizeof(largest output NV))

B. net_buf_out_size default

If outgoing explicit messages are sent with **msg_send( )** or **resp_send( )**:
B = 66

else:
B = max(42, 22 + sizeof(largest NV))

While the response is constructed in the application input buffer by the application, the network processor uses a network output buffer to construct the response packet. So, the network output buffer must be sized to accommodate outgoing responses in addition to other outgoing messages.

C. app_buf_in_size default

If any explicit message functions or events are used (incoming or outgoing):

If explicit addressing is used:
C = 66

If explicit addressing is not used:
C = 50

If no explicit message functions or events are used, and:

If explicit addressing is used for network variables:
$C = \max(34, 19 + \text{sizeof}(\text{largest NV}))$

The application input buffer must be sized to accommodate incoming responses in addition to other incoming messages.

D. receive_trans_count default

If explicit messages are received by the application program:
$D = \max(8, \min(16, \text{\# of non-config input NVs} + 2))$

If explicit messages are not received by the application program:
$D = \min(16, \text{\# of non-config input NVs} + 2)$

E. app_buf_out_count, **app_buf_out_priority_count**, **net_buf_out_count**, and **net_buf_out_priority_count** defaults

If the application is linked for a Neuron 3120 Chip or a Neuron 3120E1 Chip:

$E = 1$

If the application is linked for any other Neuron Chip or Smart Transceiver:

$E = 2$

When priority buffer counts are set to zero, all network variables are marked as non-priority nonconfig. If **app_buf_out_priority_count** or **net_buf_out_priority_count** is non-zero, then both must be non-zero, and two transmit transaction buffers are automatically allocated in RAM. If there are no priority output buffers, then only one transmit transaction buffer is allocated. The size of a transmit transaction buffer is 28 bytes (in versions 4, 6, and later, of the Neuron firmware), and 18 bytes (in earlier versions).

The Neuron C compiler determines that a program uses explicit addressing if it references any of the following:

msg_in.addr
resp_in.addr
msg_out.dest_addr
nv_in_addr

Usage Tip for Memory-Mapped I/O

You can attach memory-mapped I/O devices to a Neuron 3150 Chip or FT 3150 Smart Transceiver. You cannot use memory mapped I/O with Series 5000 or Series 6000 chips, but you can expand the I/O capabilities of all Neuron Chips and Smart Transceivers by accessing a peripheral I/O device using one of the supported I/O models, such as an SPI PIO device.

Memory-mapped devices should respond only to memory addresses that are outside the configured memory map areas for ROM, EEPROM, and RAM.

A convenient method of access to memory-mapped I/O from a Neuron C program is to declare a constant pointer to the block of control addresses for the device. In the following example, a hypothetical memory-mapped I/O device has two control registers and a 16-bit data register, at addresses *x*, *x+1*, *x+2*, and *x+3*, respectively. The device is connected to respond to the addresses of 0x8800 to 0x8803. The fragment of Neuron C code below accesses the device.

```
typedef struct {
```

```
        unsigned short int    controlReg1;
        unsigned short int    controlReg2;
        unsigned long int     dataReg;
}   *PMemMapDev;

const PMemMapDev pDevice = (PMemMapDev) 0x8800;

unsigned long read() {
    // Read from device ...

    pDevice->controlReg1 = … <control code 1>…;
    pDevice->controlReg2 = … <control code 2>…;
    unsigned long z = pDevice->dataReg;
}

void write(unsigned x, unsigned y, unsigned long z) {
    // Write to device ...

    pDevice->controlReg1 = x;
    pDevice->controlReg2 = y;
    pDevice->dataReg = z;
}
```

What to Try When a Program Does Not Fit on a Neuron Chip

The following discussion contains tips and techniques for reducing the EEPROM requirements of a program for purposes of getting it to fit or having it use less code space. Some of the techniques are tailored to a Neuron 3120 Chip or FT 3120 Smart Transceiver, but most are applicable to any Neuron C language program. Most of these manual optimization techniques improve both aspects of code size and code performance. The techniques below should be attempted roughly in the order presented.

While the memory map for a Series 3100 device is usually fixed and defined by hardware, some out-of-memory link-time errors for a Series 5000 chip can be addressed by adjusting the memory map according to the application's needs. Although you cannot define the memory map to exceed the FT 5000 Smart Transceiver's or Neuron 5000 Processor's inherent limits, you can increase or decrease the amount of RAM available (at the expense of EEPROM or flash memory available) in steps of 256 bytes.

For devices using a Series 6000 chip, the Neuron C tools automatically allocate the different memory areas as required by the hardware, the Neuron firmware and your application. However, many of the suggestions listed in this chapter apply, as they affect the amount of memory required for the various types of memory.

Many of the suggestions presented in the sections that follow lead not only to smaller code or data (to better fit the target device), but also lead to faster, more efficient, code. It is, therefore, recommended that you familiarize yourself with the all of the following recommendations, regardless of your preferred target Neuron Chip or Smart Transceiver model.

The link map contains information about a program's current memory usage. The summary information includes an estimate of the additional memory

required. The link summary is optionally output to the **BUILD.LOG** file, and is also included in the optionally produced link map file (if your Neuron C development tool supports link maps).

Reduce the Size of the Configuration Property Template File

Consider using the **#pragma codegen cp_family_space_optimization** directive. This results in an aggressive re-ordering of configuration property template and value file contents, aiming at reducing the total size of the template file. Depending on the specific application, this directive can have little effect or make a huge difference.

Note that optimizing the configuration property files for size might result in reduced performance when commissioning or configuring devices, especially when being used on or through low-bandwidth channels. See the *Compiler Directives* chapter in the *Neuron C Reference Guide* for more information and important considerations about these directives.

#pragma codegen cp_family_space_optimization is also enabled when you select the highest optimization level, using the **#pragma optimization all** directive.

Reduce the Number of Address Table Entries

A good rule of thumb is to assume that the minimum number of address table entries that a fully connected Neuron C application program can use is the sum of the number of non-polled output network variables, polled input network variables, and bindable message tags. A bindable message tag is one that does *not* include **bind_info (nonbind)** in its declaration. For example, an application with one message tag and two output network variables (one of which is an array of four elements), would need a maximum of six address table entries.

However, additional address table entries might be needed for input network variables which are in one or more groups, one entry being used for each group, or for any alias network variable that is associated with any of the input or output network variables on the device. Finally, each group connection to a device's **msg_in** tag uses an address table entry.

If your program does not explicitly receive messages (and therefore has no connections to the **msg_in** tag), and it has only a few network variables that are each connected only in a point-to-point manner (that is, no group connections), you could reduce the number of address table entries. Other situations could require further analysis to determine if the number of address table entries could be reduced.

The default number of address table entries is 15 for traditional systems, and a number between 15 and 254 for applications using a Series 6000 chip. The value can be controlled with the **#pragma addresses** directive (also known as num_addr_table_entries), detailed in the *Compiler Directives* chapter in the *Neuron C Reference Guide*. Reducing the number of address table entries saves 5 bytes of EEPROM per entry eliminated.

Remove Self-Identification Data if Not Needed

The Neuron C compiler places self-identification data in the device's program space. This data consumes EEPROM. If your program is not using SNVTs, you can consider removing the self-identification data. You can do this by specifying the following compiler directive:

#pragma disable_snvt_si

Remove Network Variable Names if Not Needed

The Neuron C compiler places information about the names of the network variables in the device's program space when the compiler directive **#pragma enable_sd_nv_names** appears in the program. This data consumes EEPROM.

You can remove the directive to regain one byte of EEPROM space for each character in a network variable's name, plus one byte for each network variable. When the device is installed, if there is no further information available about the network variable names, the network tool automatically assigns generic names such as "NVI1", "NVO7", and so on.

To assist the network integrator and allow the use of intuitive, self-explanatory names of the network variables as opposed to the generic, automatically generated ones, provide the external interface files along with your device. The network management software extracts the names for the network variables from the files (.XIF or .XFB extensions), without the names consuming code space in your device.

Declare Constant Data Properly

Use of the **const** or **eeprom** keyword in a declaration of constant data is very important, because, without either of these keywords, the compiler assumes the data is placed in RAM and thus the data needs to be initialized at runtime (each time the application program is reset). This process can be very expensive in terms of code space, it unnecessarily consumes RAM memory, and it also unnecessarily lengthens the time it takes the application to complete its reset processing.

Consider the following example. This example shows a *poorly declared* data table of four bytes in length. Unfortunately, because this declaration does not use the **const** keyword, the compiler places it in RAM, and it must therefore be initialized each time the application processor resets. The executable code fragment to initialize the array is an additional 9 bytes, and another four bytes are placed in code space containing the initial values for the table.

Furthermore, use of RAM for the data table means that there is a chance it could accidentally get modified by an unintentional programming error.

Example of poor declaration:

```
int lookup_table[4] = {1, 4, 7, 13};
```

The proper declaration of the data table only consumes four bytes of read-only memory (code space) for the data values themselves.

Example of proper declaration:

```
const int lookup_table[4] = {1, 4, 7, 13};
```

Use Efficient Constant Values

In the Neuron Chip and Smart Transceiver CPU architecture, constants in the range of 0 to 7 can be used more efficiently than larger 8-bit constants. Instructions that use these constant values are smaller and faster. Therefore, when choosing a sequence of constant values, normalize the sequence to begin with 0. An enumerated type (an **enum**) is, by default, normalized with zero in this manner.

Also, because the Neuron firmware initializes RAM to zero automatically when the application is reset, a constant sequence should be designed with zero as its initial value. The following section, *Take Advantage of Neuron Firmware Default Initialization Actions*, describes how you can use this fact to your advantage.

Another consideration is a comparison, especially when used in a loop control expression, such as in a **while** statement. The most efficient comparison of an expression with a constant is when the constant is zero. If you cannot arrange to have your loop test compare with zero, then try to arrange to have your loop test compare with one. Equality comparisons with one are not as efficient as comparisons with zero, but they are more efficient than comparisons with other constants.

Take Advantage of Neuron Firmware Default Initialization Actions

The Neuron firmware automatically sets all RAM variables to zero each time the chip resets, and also when the Neuron C **application_restart( )** function is called. After this action, the Neuron C application program is started. The first action of a Neuron C application program is to execute code to initialize any RAM variables to non-zero values. Then, if a task associated with the **when(reset)** clause exists, it is called.

Therefore, use of compile-time initializers to set RAM variables to zero is free. Eliminate any code in the **when(reset)** clause's task which is used to set RAM variables to zero, because it is unnecessary.

Also, compile-time initializers of I/O output objects are free. They are free regardless of initializer value. The use of compile-time initializers for I/O uses less code space than corresponding calls to **io_out( )** in the **when(reset)** clause's task.

Finally, during reset, the Neuron C application timers are all turned off automatically. Eliminate any code in the **when(reset)** clause's task that explicitly turns off application timers.

Use Neuron C Utility Functions Effectively

There are several Neuron C utility functions that can be used to reduce code requirements. For example, there are **min( )**, **max( )**, and **abs( )** functions, as well as other utility functions which can be used for common operations. Using these functions is generally more code-space efficient than coding the operations in-line using C operators.

The Neuron C utility functions include byte-manipulation functions, such as **high_byte( )**, **low_byte( )**, **make_long( )**, and **swap_bytes( )**. There are bit-manipulation functions such as **clr_bit( )**, **reverse( )**, **rotate_long_left( )**, **rotate_long_right( )**, **rotate_short_left( )**, **rotate_short_right( )**, **set_bit( )**, and **tst_bit( )**.

For extended precision arithmetic, Neuron C provides the **muldiv( )**, **muldivs( )**, **muldiv24( )**, and **muldiv24s( )** functions. These functions permit a multiply operation, followed by a divide operation, with the intermediate result and the operations using either 32-bit or 24-bit precision.

The Neuron C functions also include such utilities as the **timers_off( )** function. This function turns off all application timers with a single function call. This function call takes less space than the corresponding assignment of zero to a single timer, although it takes longer to execute. Thus, if your program contains a single application timer, and you turn it off by assigning zero to it, consider using this function instead in order to save code space.

Other miscellaneous functions include **bcd2bin( )** and **bin2bcd( )**, **delay( )** and **scaled_delay( )**, and **random( )**.

All of the Neuron C functions are described in detail, with examples, in the *Neuron C Reference Guide.*

Be Aware of Library Usage

Be aware of the system functions that are placed in application memory (see the table in the *Neuron C Reference Guide* for a complete list of the functions placed in memory for each chip and each version of firmware). If possible, avoid use of such things as **signed** bitfields in structures that cause use of library functions.

Use More Efficient Data Types

The Neuron C compiler generates more compact code when the data items and operations on them more closely match the underlying machine architecture and instruction set. If possible, change variables to be local rather than global, to be **short** rather than **long**, and to be **unsigned** rather than **signed**.

For example, consider the following function which finds an occurrence of **value** in the array **a** and returns the index where **value** was located:

```
<type> find(int a[], int value, <type> count) {
    <type>  i;
    for (i=0; i<count; ++i) {
        if (a[i] == value) break;
    }
    return i;
}
```

When this function is compiled, the following code sizes are obtained corresponding to the data types shown:

```
<type> is signed short:      25 bytes
<type> is unsigned short:    24 bytes
<type> is signed long:       34 bytes
<type> is unsigned long:     34 bytes
```

In addition to the code size numbers, all sequences above, except the one for **unsigned short**, make use of multiple calls to firmware helper functions. This implies that the runtime of the code sequence for **unsigned short** is even more efficient than it seems at first. Thus, the data type which permits the generation of the most efficient code is **unsigned short**. The Neuron Chip and Smart Transceiver instruction set is inherently most efficient when dealing with 8-bit unsigned integers.

Also, an awareness of the stack architecture employed in the CPUs of the Neuron Chip and of the Smart Transceiver can help in understanding how to write code that can be compiled efficiently (see the *Neuron Assembly Language Reference* for more information about the stack architecture). As a general guideline, you should keep the total size of active local variables plus parameters under eight bytes. This restriction permits all local variables and parameters to be accessed and stored using the smallest possible instructions. The following section, *Observe Declaration Order*, explains how the first local variable is accessed more efficiently, and how you can use this fact.

The Neuron C language permits aggregates, such as arrays, structures, and unions, to be declared on the local data stack. To the extent that such local variable aggregates are declared, the compiler uses larger and slower instructions to access these data items. Therefore, it is best to declare such variables as static items, rather than as local variables. If you are limited in data memory, and must declare these aggregates as local variables, then declare them after the non-aggregate local variables; this declaration order permits the compiler to use the shorter instructions for the non-aggregates.

Observe Declaration Order

The order of declaration of the automatic variables within a function can have an effect on code size. The compiler places the first variable declared on the top of the data stack, the second variable next, and so on. The Neuron C compiler generates more efficient code to access the topmost variable on the stack, especially when that variable is an eight bit scalar (**int**, **short**, or **char**). The least efficient accesses (loads and stores) are to variables deep in the stack. Thus, best results are generally obtained when the variable used most often is declared first.

For example, consider the following code fragment:

```
void arrayinit(int a[], int initval,
        unsigned count) {
    int j;
    unsigned i;
    j = initval;
    for (i=0; i<count; ++i) {
        a[i] = j;
    }
}
```

This function generates 23 bytes. However, if the variable **i** (which appears in more expressions than **j** and is thus used more often) were declared first, then the code generated would only be 21 bytes.

Use the Optional fastaccess Feature

Array accesses (both loads and stores) in Neuron C normally use the rules of ANSI Standard C. These rules permit the array index to be interpreted as a signed quantity, and furthermore permit the array index to exceed the bounds of the declared array. These characteristics of array indexing increase the code size for array references.

It is possible, given the Neuron machine instruction set, to generate better code for accessing small arrays if the following additional rules are observed:

1   The array index can be promoted to **unsigned** by the compiler if it is a **signed short.**

2   The program never attempts to access outside the bounds of the array, and never computes the address of an array element outside the bounds of the array. Computation of such an address is permitted in ANSI C for the purpose of terminating a loop using a pointer, but using this technique with **fastaccess** arrays yields undefined results.

To inform the compiler that these additional rules can be assumed for array access, include the **fastaccess** keyword in the definition of such an array. Also, the total size of the array must be no larger than 254 bytes. The **fastaccess** keyword can appear anywhere in the declaration and applies to all arrays in the declaration. For example, the following declares the arrays **a1** and **a2** to both be **fastaccess** style arrays:

        fastaccess int a1[4], a2[12];

You can combine the **fastaccess** keyword with other declaration syntax, including **network**, **far**, **eeprom**, and **const**. Fastaccess arrays can appear on the local procedure or function stack, as well as in global memory. The **fastaccess** feature does not apply to the indexing operator used with a pointer.

One potential drawback to using **fastaccess** arrays in global memory is that the linker locates these data items such that they do not span page boundaries (a memory page consists of 256 bytes). Thus, declaration of *many* global arrays as **fastaccess** could cause increased memory use due to possible fragmentation.

Eliminate Common Sub-Expressions

The Neuron C compiler automatically eliminates common sub-expressions when full optimization is enabled. Performing this optimization by hand can be straight-forward and, in some cases, yield more efficient code for all optimiziation levels. Consider the following Before-and-After example, which saves 4 bytes of code. The **temp** variable, in the After example, is declared such that it becomes the top variable on the stack.

Before (compiles to 28 bytes of code):

```
int a, b, c, d, e;
void f(void) {
    d = (a * 2) + (b * c * 4);
    e = a - (b * c * 4);
}
```

After (compiles to 24 bytes of code):

```
    int a, b, c, d, e;
    void f(void) {
        int temp;
        temp = b * c * 4;
        d = (a * 2) + temp;
        e = a - temp;
    }
```

Another form of common sub-expressions that might not be as obvious occurs with array indexing. Consider the following Before-and-After example that demonstrates the value in avoiding repeated indexing into an array element. Not only is there an obvious code savings by using a temporary pointer variable, there is a simplification of the code as well (the Before example contains three multiplies, one for each access to the array, whereas the After example only contains one multiply operation).

Before (compiles to 46 bytes of code):

```
    struct s {
        int x, y, z;
    } a[5];

    void f(int i) {
        a[i+2].x = 3;
        a[i+2].y = 5;
        a[i+2].z = 7;
    }
```

After (compiles to 33 bytes of code):

```
    struct s {
        int x, y, z;
    } a[5];

    void f(int i) {
        struct s *p;
        p = &(a[i+2]);
        p->x = 3;
        p->y = 5;
        p->z = 7;
    }
```

Use Function Calls Liberally

Because function calls are relatively cheap in terms of the code space and execution time overhead, replacing even a single line of complex code with an equivalent function can reduce code space if that line of code is used two or more times in a program. Some lines of code involving network variables might not look complex, but the underlying operations could be.

For example, consider the increment of an element in a structure which is part of an array of network variables; this operation generates a considerable amount of code. Replacing two such occurrences with a single function call saves code space at the expense of a minor performance penalty.

Also, consider passing expressions and values as function actual parameters, rather than using global variables. Accesses to parameters are generally more efficient than (or are no worse than) accesses to globals.

Use the Alternate Initialization Sequence

Use of the **#pragma disable_mult_module_init** directive saves 2 or 3 bytes of EEPROM code space. This directive specifies to the compiler that it should generate any required initialization code directly in the special init and event block, rather than as a separate procedure callable from the special init and event block.

The in-line method, which is selected as a result of use of this directive, is more efficient in memory usage (it typically saves 3 bytes if initialization code is present, and saves 2 bytes if no initialization code is present). However, the drawbacks of using the directive are the following: (1) the in-line initialization area is limited in length, and (2) there can be no linkage from the program's initialization code to application library or custom image initialization code (this is typically not a problem for any Neuron 3120 Chip or 3120 Smart Transceiver).

Use C Operators Effectively

The ANSI C language has a rich set of operators. Using them effectively can produce very efficient code.

For example, use of the C **? :** operator rather than use of an **if** - **else** statement for alternative assignments to the same left-hand-side can reduce code space, especially if the left-hand side expression is complex.

Also, the use of multiple **if - else** clauses can be slightly more efficient in code space than a **switch** clause. Consider the following Before-and-After example, which saves 2 bytes of code:

Before (compiles to 40 bytes of code):

```
void f (unsigned c) {
    switch (c) {
    case '1':
        f1();
    break;
    case '2':
        f2();
    break;
    case '3':
        f3();
    break;
    case '4':
        f4();
    break;
    default:
        f5();
    break;
    }
}
```

After (compiles to 38 bytes of code):

```
void f (unsigned c) {
    if (c == '1') {
        f1();
    } else if (c == '2') {
        f2();
```

```
        } else if (c == '3') {
            f3();
        } else if (c == '4') {
            f4();
        } else {
            f5();
        }
    }
```

Another C language operator that can improve code efficiency is the chained assignment. A chained assignment uses the fact that the value being assigned can continue to be used after the assignment. The chained assignment saves reloading or recomputing the value being assigned. Chained assignment is shown in the following Before-and-After example.

Before (compiles to 14 bytes of code):

```
mtimer t1;
unsigned long int l;

void f(void) {
    t1 = 5000;
    l = 5000;
}
```

After (compiles to 13 bytes of code):

```
mtimer t1;
unsigned long int l;

void f(void) {
    t1 = l = 5000;
}
```

Use of the logical operators **&&** and **||** for complex conditions typically perform faster than similar expressions that use the bit operators **&** and **|**. In addition, use of the logical operators can make the code smaller, especially when tests for equality or inequality with zero are part of the conditional expression. The following Before-and-After example demonstrates this efficiency:

Before (compiles to 15 bytes of code):

```
void f (int a, int b, int c) {
    if ((a < 0) | (b == 0) | (c > 0)) {
        // take some action
    }
}
```

After (compiles to 12 bytes of code):

```
void f (int a, int b, int c) {
    if ((a < 0) || (b == 0) || (c > 0)) {
        // take some action
    }
}
```

Use Neuron C Extensions Effectively

The Neuron C language contains features that exist primarily to help write efficient code.

For example, if a program has two input network variables, and has a single task executed when either variable is updated, it is more efficient to code it as shown in the After example, below. Likewise, use of a single **when** clause with the **nv_update_occurs** event referencing just an array name is more efficient than using multiple **when** clauses, one for each element of an array.

Before (compiles to 6 bytes of code):

```
when (nv_update_occurs(var1))
when (nv_update_occurs(var2))
{
    ...
}
```

After (compiles to 3 bytes of code):

```
// Use "unqualified" event to cover all variables
when (nv_update_occurs)
{
    ...
}
```

However, if you need to use specific **nv_update_occurs** events without the use of the unqualified event shown above, the following guidelines can be used.

Consider a program that declares two network variables **nviA** and **nviB**:

```
network input SNVT_switch nviA, nviB;
```

The following code fragments are all functionally equivalent, because they all respond to incoming network variable updates for either of these two network variables. The first of these implementations is the least efficient of the three, and the last one (equivalent to the Before example above) is the most efficient of the three:

Variant 1 (compiles to 15 bytes of code):

```
when (nv_update_occurs(nviA) || nv_update_occurs(nviB))
{
    ...
}
```

Variant 2 (compiles to 9 bytes of code):

```
when (nv_update_occurs(nviA..nviB))
{
    ...
}
```

Variant 3 (compiles to 6 bytes of code):

```
when (nv_update_occurs(nviA))
when (nv_update_occurs(nviB))
{
    ...
}
```

Use Variable Parameters

Function parameters may be used as local variables. This can reduce the overhead required to manage local *auto* variables. Consider this example:

```
unsigned xor_a(unsigned n, unsigned* data) {
  unsigned count = n;
  unsigned result = 0;

  while(count--) {
    result ^= *data++;
  }
  return result;
}
```

The argument *n* could be used instead of local variable *count*. This reduces the overhead for local *auto* variable management, reduces burden on the data stack, and leads to smaller stack frames. Smaller stack frames generally lead to more efficient code:

```
unsigned xor_b(unsigned count, unsigned* data) {
  unsigned result = 0;

  while(count--) {
    result ^= *data++;
  }
  return result;
}
```

Use Transient Code

On devices based on Series 6000 chips, functions default to *transient functions*. Transient functions are loaded on demand, and allow for the total application size to exceed the physical memory limits of the chip. However, when-tasks and interrupt-tasks always compile into resident code. Interrupt or when tasks which need to execute very frequently and with deterministic and minimum delays are best compiled into resident code, but many tasks execute infrequently or are not timing-critical.

Code which makes the body of such a task may best be moved from the resident task body into a transient function, as outlined in the following:

```
// Infrequent and not time-critical event:
when (<event>)
{
    <body>
}
```

This task could benefit from executing the body as a transient function:

```
// Infrequent and not time-critical event:
void mybody(…) {
    <body>
}

when (<event>)
{
    mybody(…);
```

```
        }
```

Using the Link Map

Compiling an application for various models of Neuron Chips and Smart Transceivers generally results in different application footprints, because a different set of system library functions might need to be linked into the application space (from a system library), or might already be part of the standard system image and does not need explicit linking into application space.

To understand how your device's memory is used, enable the link map feature (for Neuron C development tools that support link map generation, such as the IzoT NodeBuilder Software).

If linking and building the device's file set succeeds, the link map provides a detailed report of how your device's memory is used, and which code or data had to be linked in from other libraries.

After an unsuccessful build, the link map is incomplete and some of its information can be incorrect. However, the link map's memory usage statistics section provides a summary of consumed memory, and reports any shortfall.

Example: The following excerpt from a link map for an unsuccessful build shows that RAM allocation exceeds available resources by 2535 bytes.

```
RAM Usage:       (not necessarily in order of physical layout)
        System Data and Parameters          701 bytes
        Transaction Control Blocks           82 bytes
        Appl Timers and I/O Change Events     0 bytes
        Network and Application Buffers      424 bytes
        Application RAM Variables             0 bytes
        Library RAM Variables                 0 bytes
>>>>    Add'l Unsatisfied Requirements    30000 bytes
                                          -----
        Total RAM Requirement             31207 bytes
>>>>    Exceeds available RAM by           2535 bytes
            Note: May need more due to fragmentation.


>>>>    Link failed.  Program did not fit.
```

A

Neuron C Tools Stand-Alone Use

This appendix provides information on how to use the Neuron C tools as stand-alone programs from the command line.

A listing of the options supported by each tool can be obtained by typing the tool name at the command prompt. For example, typing **ncc** lists the Neuron C compiler command line options.

Stand-Alone Tools

The Neuron C tools listed in **Table 15** can be used *stand-alone*, meaning outside the integrated development environment, using the command prompt or command window only.

Table 15. Stand-Alone Tools

Description	Tool Name
Neuron Assembler	NAS
Neuron C Compiler	NCC
Neuron Exporter	NEX
Neuron Librarian	NLIB
Neuron Linker	NLD
Project Make	PMK

All of the NodeBuilder stand-alone tools share a common command-line technology, and thus have several aspects of use in common. These common aspects are described in the following section, *Common Stand-Alone Tool Use*. The sections following later in this appendix briefly introduce each of the tools listed above.

Note: Users of the IzoT Nodebuilder Software should not generally use the command line tools, with the exception of the Neuron Librarian and the Project Make Utility. The reason is that the build tools should be controlled by the Project Make Utility, **pmk.exe**. Not only does this utility manage the build process (it minimizes the number of build steps required), it also takes care of program ID management tasks and automatic boot ID processing. You must otherwise take care of these two important duties manually. See the *IzoT NodeBuilder User's Guide* for more details about the *Project Make Utility*.

Common Stand-Alone Tool Use

All stand-alone tools share a set of common syntax and common basic commands.

Common Syntax

If no command switches or arguments follow the name of the tool, the tool responds with usage hints.

Example:

C:\>**NAS**

Tool responds:

```
Neuron Assembler, version 5.00.21, build 0
Copyright (c) Echelon Corporation 1992-2009
```

```
Usage:  [optional command(s)] argument
```

... (Remaining output not listed here)

Most command switches come in two forms, a short form and a long form. The short form must be prefixed with a single slash '/' or dash '-' and consists of a single, case-sensitive, character that identifies the command.

Example of short form:

C:\>**NCC -DMYMACRO ...**

Short command switches can be separated from their respective values with a single space or an equal sign. Short command switches do not require a separator; the value can follow the command identifier immediately, as shown above.

The long form of the command must be prefixed with a double dash '- -', followed by the verbose, case-sensitive, name of the command.

Example of long form:

C:\>**NCC - -define=MYMACRO ...**

Long command switches do require a separator, which can consist of a single space, or an equal sign.

Multiple command switches can be separated by a single space.

Example:

C:\>**NCC - -define=MYMACRO1 - -define=MYMACRO2 ...**

Commands of a Boolean type need not be followed by a value, in which case **yes** is assumed. Possible values for Boolean commands are **yes**, **on**, **1**, **+**, **no**, **off**, **0**, **-** (a minus character).

Example:

C:\>**NCC - -kerneldbg=yes ...**

This is equivalent to the line shown below (because the Boolean type commands default to **yes**):

C:\>**NCC - -kerneldbg ...**

Commands can be read from the command line as shown in the examples above, and they can also be read from a command file (script), which contains empty lines, lines starting with a semicolon (comment lines), or lines containing one command switch on each line (with value as applicable).

For brevity, the short command syntax is most commonly used on the interactive command line, whereas the long command line syntax is preferred for command files due to its more self-explanatory nature.

Example command file:

```
; Example command file, containing
; some of the Exporter's commands
; Created Wednesday, November 21, 2008, 20:42:20

--bootflags=1024
--infolder=d:\lm\Source\EPR\23305\Development\IM
--outfolder=d:\lm\Source\EPR\23305\Development
```

```
--basename=23305
```

Most tools require additional arguments to be given; these arguments can appear at any location within the command line or in a separate line within a script.

Example of argument at end of command line:

C:\>**NCC - -define=MYMACRO mycode.nc**

Common Set of Basic Commands

In addition to the shared syntactical aspects introduced in the previous section, the stand-alone tools also share a common set of basic commands. Some of these common commands are listed below. To obtain a complete list of all available commands, you can type in the name of any of the stand-alone tools without specifying any command.

    -@ *file-pathname*   (Include a command file)

The **-@** (or: **- -file**) command specifies a command file (script). The commands are read from this script and used as if they were given at the command line and in the location of the @ command. Scripts themselves can refer to other scripts.

    - -defloc *dir*   (Location of an optional default command file)

The command line tools also search for a default script; a file that is read in addition to and after all other commands from the command line have been processed. These default script files need not be specified with the **-@** command, because they have a predefined name shown in **Table 16**. The command line tool assumes that the default script is located in the current working directory (and it is no error if there is none); the **- -defloc** command can be used to specify the location (*not* name) of the default script. The IzoT NodeBuilder Software uses the location of the NodeBuilder device template file (**.nbdt** extension) as the location of the default script.

Table 16. Default Script Names for the Stand-Alone Tools

Build Tool	Command Name	Default Script Name
Neuron C Compiler	NCC	LonNCC32.def
Neuron Assembler	NAS	LonNAS32.def
Neuron Linker	NLD	LonNLD32.def
Neuron Exporter	NEX	LonNEX32.def
Neuron Librarian	NLIB	LonLIB32.def

    - -mkscript *scriptfile*   (Generate command script in *scriptfile*)

The **- -mkscript** command produces a trace file that contains all commands the build tool received, no matter where these commands came from. When used within a default script, this feature can be used to capture the command sequence used by the project manager; a simple way to obtain machine-generated build scripts.

Important: Specify the *scriptfile* script file in such a way that it does not overwrite the default script file, or any other script file you want to preserve. The **- -mkscript** command allows for constant command flow tracking, and thus overwrites existing files without warning.

> **- -warning** *text*   (Display *text* as a warning)

This command is only useful in script files. It displays the message *text*, and indicates the message as a warning. The **- -mkscript** command automatically inserts a **- -warning** command into the generated script if the tool that executed the monitored command stream failed to complete without error.

When using the machine-generated script file, a warning states that the script was machine-generated, and based on a possible erroneous command stream.

Command Switches for Stand-alone Tools

The most useful and common command switches are documented in this section for each of the stand-alone tools.

Neuron C Compiler

The Neuron C compiler is named **ncc.exe**. You can run the stand-alone compiler from the command prompt to produce a Neuron assembly source file. The compiler command line contains the name of the executable file, then zero or more optional command switches, and finally the file name to compile.

> **Example:**

> C:\>**NCC mycode.nc**

The most interesting switches are the **-D** (**- -define**) and **-I** (**- -include**) switches. You can use the **-D** switch to define a symbol from the command line, which can then be tested from the program using the **#ifdef** and **#ifndef** directives.

You can use the **-I** (**- -include**) switch to specify a directory containing include files. You can specify additional include directories with additional **-I** switches. The search order corresponds to the order of the switches, if you specify more than one **-I** switch.

> **Example:**

> C:\>**ncc -DVERSION5 -I..\include -Id:\include mycode.nc**

When run for a filename with a .nc extension, the Neuron C compiler uses Neuron C rules for code generation. Libraries and custom system images cannot contain Neuron C code. To compile a pure C file, and use pure C rules for code generation, the filename must end with a .c extension as shown in the command line example below:

> **Example:**

> C:\>**ncc -I..\include mycode.c**

As a final, complete example, to compile **myfile.nc** with a **myinc.h** include file in a subdirectory named **myincs**, and to define the OPTION1 symbol for conditional compilation purposes, run the command shown below:

> **Example:**

C:\\>**ncc –Imyincs -DOPTION1 myfile.nc**

This command (assuming the compilation does not find errors) creates several output files, all sharing the basename that was used for the Neuron C source file. For the pure C example above, all generated files have a **myfile** base name, but different extensions.

For a pure C compilation, all generated files except the assembly source file, .ns extension, can be discarded. These files are required by other tools in case of a Neuron C compilation, and cannot be discarded in such case.

Neuron Assembler

The Neuron assembler is named **nas.exe**. The Neuron assembler is only provided for supporting the Neuron C compiler. It should not be used to generate Neuron Assembly Language applications.

You can run the Neuron assembler from the command prompt to produce a Neuron object file. The assembler command line contains the name of the executable file, then one or more optional switches, and finally the file name to assemble. The most useful assembler switch is the **-l** switch (the long form is **- - listing**), which tells the assembler to produce a listing.

Continuing the example from the compiler section above, the following command assembles the **myfile.ns** file to produce a **myfile.nl** listing file and a **myfile.no** object file. After the object file is produced, you can delete the **myfile.ns** intermediate assembly file to conserve disk space.

Example:

C:\\>**NAS -l myfile.ns**

Neuron Linker

The Neuron Linker is named **nld.exe**. You can run the linker from the command prompt to produce a Neuron executable file. The linker command line contains the name of the executable file, then one or more switches, and finally the object file name or names to link. Several switches must be used in combination to produce a correct link.

The **-a** (or **- -appimage**) switch should always be used when linking a Neuron C application program.

The **-t** (or **- -neurontype**) switch should be used to specify the name of the Neuron Chip or Smart Transceiver for which the application is being linked.

Example:

C:\\>**NLD -a -t3120E2 ....**

When linking for a Neuron 3150 Chip or 3150 Smart Transceiver or a Series 5000 chip, the external memory map must be specified using a set of switches. However, specifying the external memory map for a Series 6000 target yields a warning, and has no effect.

The switches specify the beginning or end of the external RAM, EEPROM, and ROM areas. Each of these switches is followed by a hex number corresponding to the first (last) page number of the area. A page is 256 bytes, thus the page number is the upper two hex digits of the four-digit byte address.

The **-r** and **-R** switches specify the first and last pages of external RAM, respectively. The **-e** and **-E** switches specify the first and last pages of external EEPROM, respectively. The **-Z** switch is used to specify the last page of ROM (there is no corresponding switch for the first page of ROM, because ROM must start at 0x0000). Do not use the switches for any area that has no external memory. Memory mapped I/O areas should not be specified, and should be outside the range of any external memory areas which are specified.

For example, a link that has external ROM from 0x0000 to 0x7FFF, no external EEPROM, external memory-mapped I/O devices from 0x8000 to 0x97FF, and external RAM from 0x9800 to 0x9FFF, uses the switches shown below:

> C:\>**nld –Z=7F –r=98 –R=9F ....**

The linker must input a symbol table corresponding to the system image for which the application is being linked. This is done using the **-p** switch, followed by a space, then the pathname of the image's symbol file.

For example, if linking for a 3150 custom device, the image name is SYS3150. Assume that the Neuron C software is installed in the C:\LonWorks directory. This main directory contains a number of subdirectories. The IMAGES subdirectory contains one or more further subdirectories named VER*nnn*, where *nnn* is a number from 2-255 (no leading zeros).

The standard image files are contained in the various VER*nnn* subdirectories. For example, if the software is installed in C:\LonWorks (the default), then the switch that links to firmware version 7 would be the following:

> C:\>**NLD -p C:\LonWorks\IMAGES\VER7\SYS3150.SYM ....**

The above switches are the minimum set of required switches for the Neuron Linker.

You can specify an output file different from the default with the **-o** switch. The default output file name is the name (without extension) of the first object file in the link command line.

You can use the **-A** switch to specify that EEPROM is to be implemented using flash memory as an alternative to EEPROM. The sector size (64 or 128) of the flash memory device must follow the **-A** switch.

You can include libraries in the link by specifying one or more instances of the **-l** switch, with a library name following. You can link to multiple libraries by specifying an **-l** switch for each library.

If you are using the IzoT NodeBuilder Software, refer to the *IzoT NodeBuilder User's Guide* for details about the use of libraries within a NodeBuilder project. The NodeBuilder tool does not support the construction of custom libraries except through the stand-alone tools as documented in Appendix B, but it does allow for their use.

See *Neuron Librarian* on page 235, as well as Appendix B, *Neuron C Function Libraries*, on page 237, for more about the construction of custom libraries.

Neuron Exporter

The Neuron exporter is named **nex.exe**. The exporter takes input from the compiler and the linker and produces the device file set. The device file set

contains the device interface files (.xif and .xfb extensions) as well as image files (.nri, .nfi, .nxe, .nei, .nme, .nmf, and .apb extensions, as needed).

The exporter command line contains a number of switches. Several switches must be used in combination to produce a correctly exported set of files.

Use the **-t** switch (**- -bootid**) to specify the boot ID. It is followed by a decimal number 0..65535, that denotes the boot ID in the exported image. You can use any value within this range, but each build should be built with a unique boot ID value. See the Smart Transceivers databooks for more about the Neuron Chip and Smart Transceiver reset procedure and the significance of the boot ID. Note that the **–t** switch is not required for targets using a member of the 3120 Neuron Chip family, but using this switch is highly recommended for targets using a member of the 3150 Neuron Chip family and for targets using Series 5000 chips.

Use the **-C** (**- -clock**) command to specify the Neuron clock rate as an encoded value, using "5" for 10 MHz for example. See the ClockSpeeds field in the **LonWorksUI.xml** file in the LonWorks Types directory (**C:\LonWorks\Types** by default) for a complete listing of clock speed values.

Use the **-c** (**- -xcvr**) command to specify the transceiver type used via the transceiver's standard ID. Use "7" for a TP/FT-10 free topology transceiver, for example. See the **std_id** field in the **StdXcvr.xml** file in the LonWorks Types directory (**C:\LonWorks\Types** by default) for a complete listing of standard transceiver IDs.

Use the **-I** (**- -infolder**) and **-O** (**- -outfolder**) commands to specify the location of the input files (generated by compiler and linker), and the output files (as generated by the exporter), respectively.

Use the **-b** (**- -basename**) command to specify the base name of the input and output files that are located in the input and output folders.

Use the **- -create***XXX* switch to enable the generation of a file type, where *XXX* is the file extension of the desired file type (in lower case). You can specify multiple **- -create***XXX* switches to generate multiple file types.

Here is an example that exports a device file set with base name of MyDevice for a 10 MHz based device that uses a TP/FT-10 free topology transceiver:

> C:\>**NEX -@commands.nex - -bootid=12345**

The commands.nex script file referenced by the command line contains the following set of commands:

```
; Sample Exporter command file
--clock=5
--xcvr=7
--infolder=MyProject\MyDevice\Development\IM
--outfolder=MyProject\MyDevice\Development
--basename=MyDevice
--createxif=yes
--createxfb=yes
--createnei=yes
--createnxe=yes
--createapb=yes
--createnri=yes
```

Note: When using the Neuron Exporter stand-alone tool, be sure to use two different folders for the **-I** (**--infolder**) and **-O** (**--outfolder**) commands, respectively.

Neuron Librarian

The Neuron librarian is named **nlib.exe**. You can use the librarian to create and manage libraries, or to add and remove individual object files to and from an existing library. A library consists of pure C functions; you cannot include Neuron C code in a library, with the exception of the pure C code subset of Neuron C. The librarian is the only tool discussed in this appendix that is not essential but purely optional; the librarian is not required to produce LONWORKS devices.

You can use the libraries that are created or modified by the librarian within the NodeBuilder project managers, or with the stand-alone **nld.exe** Neuron linker. For using libraries with the stand-alone linker, see the **-l** and **-L** commands in the linker's command set. For using libraries within the project manager, refer to the *IzoT NodeBuilder User's Guide*.

You can run the librarian from the command prompt by specifying the name of the command, an optional set of switches, the library name, and an optional list of object file names.

To create a new library, enter the following command line:

> C:\>**nlib -c -a** *library-name object-file* [*object-file* ...]

To add modules to an existing library, enter the following command line:

> C:\>**nlib -a** *library-name object-file* [*object-file* ...]

To replace (or update) existing modules in an existing library, enter the following command line:

> C:\>**nlib -u** *library-name object-file* [*object-file* ...]

To report on the contents of an existing library, enter the following command. The report is output on the console, but can be redirected to a file.

> C:\>**nlib -r** *library-name*

To create a summary report, enter the following command line:

> C:\>**nlib -s** *library-name*

For example, the command shown below uses the long form switch and adds the zorro.no and garcia.no object files to the mylib.lib library:

> C:\>**nlib - -add mylib.lib zorro.no garcia.no**

You can use script files to specify inputs to the librarian. For example, to combine the ten object files named "f1.no" through "f10.no" into a mylib.lib library, enter the following command line and command file:

> C:\>**nlib -c2 -a mylib.lib @mylib.lst**

The contents of the mylib.lst file are as follows:

> f1.no
> f2.no
> f3.no

f4.no
f5.no
f6.no
f7.no
f8.no
f9.no
f10.no

The librarian command line can contain more than one script file, if desired. Alternately, the command line in the preceding example could just have contained the object file names, or it could have contained a mixture of the two.

You can add a Neuron object file (with a .no extension) from a pure C compilation, to a custom library as shown in the above example. See Appendix B, *Neuron C Function Libraries*, on page 237, for more information on using libraries. See *Neuron C Compiler* on page 231 for instructions on compiling a pure C file. See *Neuron Assembler* on page 232 for instructions on assembling the compiled output to a Neuron object file.

B

Neuron C Function Libraries

This appendix discusses how to construct and use your own function and data libraries for use with the Neuron C tools. You can use the IzoT NodeBuilder Software to construct and use function libraries.

Definitions

Application program   A Neuron C source program that has been compiled, assembled, and linked with a system firmware image. The application program is not a stand-alone executable. It contains external references to the system image, and must be loaded into the memory of a device that contains the corresponding system image.

Library   A file produced by the **nlib.exe** Neuron librarian, containing one or more pure C object files produced by the Neuron assembler. The Neuron Linker can extract these object files from the library and combine them with a Neuron C application program.

Pure C   The Neuron C language is a set of extensions added to a subset of the ANSI Standard C language. The term *pure C* in this document refers to use of the subset language without the Neuron C extensions.

Stand-alone tool   The term *stand-alone* means that a tool is available from the Windows command prompt, so that it can be used outside of the project manager. All of the tools described in this appendix can be run in a stand-alone manner.

NodeBuilder Support for Libraries

You can use the NodeBuilder Neuron C compiler, assembler, and librarian to construct Neuron libraries from pure C source files as described in Appendix A, *Neuron C Tools Stand-Alone Use*, on page 227. You can then use libraries within a NodeBuilder device template to construct application programs using the NodeBuilder project manager.

The NodeBuilder project manager does not detect when a library needs to be rebuilt due to changes in the source files that make up the library. When you construct libraries, you should use standard software engineering techniques to manage your software baseline and ensure that current versions of your software are being used. Use of a third-party **make** program can help in managing software dependencies.

The NodeBuilder tool contains full support for linking an application program with one or more libraries containing Neuron object code and data written in pure C. The NodeBuilder tool does full dependency checking on the use of libraries. When a library is modified, the NodeBuilder tool automatically detects this condition and re-links the device's application program when you call the build function.

To make a library known to a NodeBuilder project, add the library to the Libraries folder in the Project pane. Refer to the *IzoT NodeBuilder User's Guide* for details about the use of libraries within a NodeBuilder project and device template.

Any Neuron C program that references one or more symbols from the library automatically includes the appropriate modules from the library at link time.

The linker only looks at libraries when the object files being linked and combined with the system image file have not already defined all symbols needed by the application program. Each library is examined, in the order in which it is specified in the linker command line. The NodeBuilder Project Make Utility places the user-defined libraries in order ahead of the standard libraries, thereby allowing symbols from the standard libraries to be superseded by symbols defined and exported in the user-defined library. Use this feature with caution because standard symbols might be accidentally overwritten. When a symbol is found in a library, the corresponding object file is extracted from the library, and all objects contained in that object file are added to the link.

An object file in a library can, in turn, introduce other symbols that are undefined. Such symbols cause the linker to search all the libraries again for other object files that can resolve these previously undefined symbol references, and these object files are also be included in the link. The process continues until all symbols included in the link are resolved, or until the list of libraries is exhausted.

Tradeoffs, Advantages, and Disadvantages

The use of libraries provides you with certain advantages and disadvantages as described in the following sections.

Advantages of a Library

Use of a library to contain utility routines and constant data tables can provide the following advantages:

1   Use of a library can speed up compilation, because utility routines are not recompiled each time.

2   A library can provide modularity, encapsulation, and reuse — software engineering techniques which can be used to increase quality and decrease development costs.

3   The library can contain several related constant tables and procedures. When organized properly, only the pieces used by a given application are linked into the application. Unused pieces do not consume any code space in the device's application.

4   A library can contain data declarations for objects in any part of a Neuron Chip's or Smart Transceiver's memory space, including near RAM and EEPROM areas. Libraries also can contain initialized RAM variables. The initialization rules are identical to Neuron C application programs.

Disadvantages of a Library

Use of a library has the following disadvantages:

1   The NodeBuilder tool offers no way to debug the contents of a library. However, the contents of data objects in a library can be examined from the Neuron C debugger, provided that the data is declared as **extern** in the application program. Procedures should be fully debugged prior to placing them in a library.

2 The NodeBuilder project manager cannot be used to manage the dependencies of a library or its component files. Keeping the library up-to-date is left entirely up to the user.

3 A library can only contain pure C functions and data objects. It cannot contain or reference Neuron C extensions, such as network variables, I/O objects, timers, or **when** statements. The functions documented in the *Neuron C Reference Guide* can all be used from a library module as pure C functions *except* the functions that pertain to network variables, messages, or input/output. Similar use restrictions apply to the Neuron C built-in variables. See *Performing Neuron C Functions* on page 250 for techniques that you can use to workaround this disadvantage.

4 Any objects from a library that are linked into an application become part of the application. When the application changes, the library objects are linked into it again, and their locations could change as a result. In particular, this means that the objects from a library must be re-loaded into the device's memory each time the application is re-linked. This re-load can be done over the network or by programming some non-volatile memory device, as appropriate. This disadvantage is presented as a contrast to Neuron C Custom system images; see Appendix C, *Neuron C Custom System Images*, on page 243, for more information on this topic.

Library Construction Using the Librarian

You can construct libraries using the stand-alone versions of the Neuron C tools, as documented in Appendix A, *Neuron C Tools Stand-Alone Use*, on page 227. Use the stand-alone compiler and assembler to compile and assemble the pure C source files that make up the library. This compilation and assembly process produces a Neuron object file (.no extension) corresponding to each C source file. Then, use the **nlib.exe** Neuron librarian to combine the Neuron object files into a library that can be used by the Neuron linker.

Following are additional guidelines for constructing libraries:

1 If any symbol in a library module (corresponding to a pure C file) is referenced, all of that module is included in a link. Therefore, you should separate unrelated functions to minimize program space use when an application program uses components of the library.

For example, if you were building a string library containing the **strcpy( )**, **strcat( )**, and **strlen( )** functions, you could minimize code space by placing each function in a separate file, because any given application program might only want a subset of the functions to be linked in. If all three functions were placed in a single file, all of them would be linked in any time any one of them were used.

2 Use the **static** keyword to declare functions or data items within a library module that are not to be exposed to the application program or other library modules. This effectively hides that symbol.

3 Construct include files containing **extern** function prototypes and **extern** data declarations for the users of the library to include in their programs. You must use the **extern** keyword for each declaration to allow the compiler to establish the correct calling sequence and use the appropriate assembler commands to permit linking to the data object or function contained in the library.

Use these include files in the library source files that define these functions and data objects as well, so that the Neuron C compiler can ensure that your **extern** declarations and prototypes match your actual declarations and function definitions. You can follow a particular **extern** declaration with the actual declaration, as long as the declarations match.

This technique can help prevent calling a function with incorrect parameters (due to an incorrect **extern** prototype, for example), which can result in an overwritten data stack and thus could result in a device that repeatedly experiences a watchdog timer reset, overwritten variables, or similar software failures.

4 The NodeBuilder Project Make Utility advises the linker of all libraries listed in the **STDLIBS.LST** file, but precedes this list by the list of user-defined libraries that are specified through the NodeBuilder device template. Therefore, to reduce the chances of a symbol conflict between two or more libraries, use a naming convention to establish unique names for library objects. Use the **nlib -r** command option (as described in Appendix A, *Neuron C Tools Stand-Alone Use*, on page 227) to generate a report listing the symbols that are defined in each existing library.

Performing Neuron C Functions from Libraries

The pure C code that is placed in a library cannot contain references to network variables, messages, I/O, timers, or other Neuron C objects. However, the library can be designed mainly for the purpose of performing Neuron C related tasks, such as standard I/O device management, or message construction, or timer manipulation.

You can access Neuron C objects from a library function by making it the responsibility of the application program to actually perform the Neuron C operation in an application function. The library can then call the application function in the Neuron C application program, and effectively perform Neuron C operations.

For example, consider a library that contains routines for management of a standard LCD display device. This library would contain various routines for formatting information and for managing the display in response to various commands from the application program. It is desirable to have the library code automatically perform the I/O operations to update the device. However, due to the pure C restriction the required Neuron C code cannot be implemented as part of the library.

For the I/O operations on the display, assume that the display has a Neurowire device interface. The library could be accompanied with an include file for the benefit of whatever Neuron C application program uses it. The include file can contain the Neurowire I/O declarations and the function definitions necessary to support the display's I/O. Then, the library can access the functions as necessary without further intervention from the application program, and without the application program's being responsible for passing some special parameters each time it wants to interact with the library display management software.

As this discussion shows, it is possible to create utility functions that depend on Neuron C features. By dividing the Neuron C code from the pure C code, and

then placing the Neuron C code in the include file for the utility, the utility can effectively function as if it were an encapsulated utility.

C

Neuron C Custom System Images

This appendix discusses how to build and use your own custom system images for use with the Neuron C tools. You can use the IzoT NodeBuilder Software to construct and use custom system images.

Custom system images are not supported for Series 5000 devices.

Definitions

Application program   A Neuron C source program that has been compiled, assembled, and linked with a system image. The application program is not a stand-alone executable. It contains external references to the system image, and must be loaded into the memory of a device that contains the corresponding firmware image.

Custom system image   A set of files produced by combining a standard system image (or another custom system image) with one or more pure C object files produced by the Neuron assembler. The extra code and data objects in the custom system image are additional objects that can be referenced by any application program that is linked with the custom system image. You can use the Neuron linker to construct a custom system image.

Library   A file containing one or more pure C object files produced by the Neuron librarian. The Neuron linker can extract these object files from the library and combine them with a Neuron C application program.

Pure C   The Neuron C language is a set of extensions added to a subset of the ANSI Standard C language. The term *pure C* in this document refers to the subset language without the Neuron C extensions.

Stand-alone tool   The term *stand-alone* means that a tool is available from the Windows command prompt, so that it can be used outside of the project manager. All of the tools described in this appendix can be run in a stand-alone manner as explained in Appendix A, *Neuron C Tools Stand-Alone Use*, on page 227.

Standard system image

A standard system image is a set of Neuron firmware files included with the NodeBuilder software. The currently defined standard system images are listed below:

B3120A20	- Firmware incorporated into a Neuron 3120 Chip.
BIN3120	- Firmware incorporated into a Neuron 3120 Chip.
B3120E1	- Firmware incorporated into a Neuron 3120E1 Chip.
B3120E2	- Firmware incorporated into a Neuron 3120E2 Chip.
B3120E3	- Firmware incorporated into a Neuron 3120E3 Chip.
B3120E4	- Firmware incorporated into a Neuron 3120E4 Chip or a FT 3120 Smart Transceiver.
B3120E4A	- Firmware incorporated into a Neuron 3120E4 Chip or a FT 3120 Smart Transceiver.
B3120E4X	- Firmware incorporated into a PL 3120 Smart Transceiver.
B3120E5	- Firmware incorporated into a Neuron 3120E5 Chip.
B3120A20	- Firmware incorporated into a Neuron 3120A20 Chip.
B3170E4X	- Firmware incorporated into a PL 3170 Smart Transceiver.

Neuron C Custom System Images

BFT5000	- Firmware incorporated into a Neuron 5000 Processor or an FT 5000 Smart Transceiver.
SYS3150	- For all devices using a Neuron 3150 Chip or FT 3150 Smart Transceiver.
SYS3150A	- For all devices using a PL 3150 Smart Transceiver.
SYS3150C	- For all devices using an FT 3150 Smart Transceiver.
SYS3150D	- For all devices using a PL 3150 Smart Transceiver.

System image   A system image is a set of files that contain the Neuron firmware needed to operate a Neuron Chip or Smart Transceiver and implement the LonTalk protocol. The software in the image is a pre-linked executable. It must not contain any external references.

NodeBuilder Use of Custom System Images

For Series 3100 devices, you can use the NodeBuilder Neuron C compiler, assembler, and librarian to construct a custom system image as described in *Constructing a Custom System Image* on page 247. You can then use this custom system image within a NodeBuilder device template to construct application programs using the NodeBuilder project manager.

Important: Custom system images are not supported for Series 5000 devices.

The NodeBuilder project manager does not detect when a custom system image needs to be rebuilt due to changes in the source files that make up the custom system image. When you construct custom system images, you should use standard software engineering techniques to manage your software baseline and insure that current versions of your software are being used. Use of a third-party **make** program can help in managing software dependencies.

The NodeBuilder project manager can use a custom system image in much the same way as it handles the standard system images. The NodeBuilder project manager does full dependency checking on the use of custom system images. When you modify a custom system image, the NodeBuilder project manager automatically detects this condition and re-links the program for the device when you invoke the build function.

See the *IzoT NodeBuilder User's Guide* and the NodeBuilder Online Help for details about choosing the firmware image to be used for a device target, and for details about using custom firmware images.

Following are guidelines for using custom system images with the NodeBuilder tool:

1   Re-program any custom device PROMs using a custom system image when you generate a new version of the custom system image. If you do not re-program the PROMs, inconsistencies between the PROM version of the system image and the NodeBuilder version can cause the device's program to fail in odd ways—often with a watchdog timer timeout or other diagnostic failure.

2   You must keep the set of files that comprise a system image together. For example, keep all the system image files together when you copy them to the appropriate **Ver***nnn* directory. *When updating system image files, ensure that all files in the set are updated together.* Failure to do so can result in errors similar to those mentioned above. The set of files

consists of the following:

image.nx	Loadable image in Intel hex format
image.nxb	Binary version of *image*.nx
image.sym	Symbol table file

Tradeoffs, Advantages, and Disadvantages

The use of custom system images provides you with certain advantages in exchange for some other disadvantages. Sometimes, it is more appropriate to use a library, and at other times it is more appropriate to use a custom system image. These issues are discussed in this section, and also in Appendix B, *Neuron C Function Libraries*, on page 237.

Advantages of a Custom System Image

Use of a custom system image to contain utility routines and constant data tables can provide the following advantages:

1   Use of a custom system image can speed up compilation, because utility routines are not recompiled each time.

2   A custom system image can provide encapsulation and reuse—software engineering techniques which can be used to increase quality and decrease development time.

3   The custom system image can be constructed once and can be used across many application programs. This can be particularly helpful in the scenario where the custom system image is programmed into a ROM or flash memory for a Neuron 3150 Chip or a 3150 Smart Transceiver, and one of several different application programs which use the custom system image are then directed into EEPROM or flash memory by network load at a later date. This scenario can be used by Neuron C field compiler users who want to deliver devices with a custom system image that includes support for the application I/O hardware. This support can be provided as standard functions that can be called by end-user application code.

Disadvantages of a Custom System Image

Use of a custom system image has the following disadvantages:

1   The NodeBuilder tool offers no way to debug the contents of a system image. However, the contents of data objects in a system image can be examined from the Neuron C debugger, provided that the data is declared as **extern** in the application program. Procedures should be fully debugged prior to placing them in a custom system image. Debugger function is not affected for application programs that *use* custom system images.

2   The NodeBuilder project manager cannot be used to manage the dependencies of a custom system image on its component files. Keeping the custom system image up-to-date is left entirely up to the user.

3   A custom system image can only contain pure C functions and data objects. It cannot contain Neuron C extensions, such as network variables, I/O objects, timers, **when** statements, and so on. The functions documented in the *Neuron C Reference Guide* can all be used as pure C functions *except* the functions pertaining to network variables, messages, or input/output. Similar use restrictions apply to the Neuron C built-in variables.

4   A custom system image cannot be used for a Neuron 3120xx Chip or 3120 Smart Transceiver.

5   A custom system image can contain many utility routines and data tables, not all of which are useful for a given application program, thus this could waste memory space.

6   A custom system image cannot contain references to any unresolved symbols. All procedures and data objects in the custom system image must refer to other procedures and objects in that same image, or to procedures and objects that make up the underlying base image. The base image is typically one of the standard system images, although it could be another custom system image that is based on a standard system image. Custom system images can be built in layers upon other custom system images.

7   A custom system image can make only limited use of a Neuron Chip's RAM memory, and it cannot use EEPROM memory. Functions and constant data objects can only be placed in ROM or flash memory. Only a limited amount of **far** RAM is available - a total of 64 bytes (for all layers of a custom system image). No **near** RAM or EEPROM, nor any **far** EEPROM can be declared or used directly. A custom system image *can* contain initialized RAM variables. The initialization rules are identical to those of Neuron C application programs.

8   The process of constructing a system image is complex. There are several details to keep straight, including keeping all the files up to date, and keeping the image in the devices in sync with the image version files. Because the NodeBuilder project manager does not assist in this, the probability of making a mistake in the process is increased, and a mistake in the custom system image process is not always self-evident.

Constructing a Custom System Image

You can construct a custom system image using the stand-alone versions of the Neuron C compiler, Neuron assembler, and Neuron linker tools, as documented in Appendix A, *Neuron C Tools Stand-Alone Use*, on page 227. Use the stand-alone compiler and assembler to compile and assemble the pure C source files that make up the custom portion of the system image. This compilation and assembly process produces a Neuron Object file corresponding to each pure C source file. Use the **nld.exe** Neuron linker to combine the Neuron object files into a custom system image that can be used as described above.

You can use a standard system image as the base for a new custom system image, or you can use another custom system image that was constructed previously.

To construct a custom system image, perform the following steps.

1   Run the stand-alone Neuron linker as described in Appendix A. In addition to the linker switches described in Appendix A, several other switches, specific to custom system image creation, are required as follows.

- Specify the **-c** switch. This must be the first switch that you specify to the linker. There are no arguments associated with the **-c** switch.

- Specify the base system image with the **-b** switch. The **-b** switch must be followed by the filename or pathname of the base image (without a filename extension). Start with the Sys3150 system image or a custom system image based on the Sys3150 system image. For example, to use the standard version 13 system image for Neuron 3150 Chip-based custom devices, the switch would be as follows:

  -b \LonWorks\Images\Ver13\SYS3150

- A custom system image can only use ROM and a limited amount of far onchip RAM. Thus, the only memory map switch that is needed for custom system image construction is the **-Z** switch. When the linker is constructing a custom system image, the **-Z** switch specifies the end of the reserved ROM for the custom system image. The base image already reserves some amount of ROM from address 0x0000 to some value x (for a standard system image, x is 0x3FFF).

  The **-Z** switch followed by some value yp reserves the ROM area from $x+1$ to y for the custom system image being created. Thus, yp is the page number corresponding to the address y. For example, to reserve from 0x4000 to 0x4FFF for the custom system image being constructed, the value for the **-Z** switch would be "4F." The reserved area must at least be large enough to contain the object files being included. The minimum value for the **-Z** switch is "40," which would be just one page (256 bytes) additional ROM space beyond the standard system image. This custom system image would end at 0x40FF.

  Any additional unused space in the new reserved area is kept for later use by future versions of this particular custom system image. The new reserved amount of ROM is automatically made known to the linker when it uses the custom system image in linking a Neuron C application program at some later time, and the application program is then permitted to use only any remaining unreserved ROM.

- Specify the **-V** switch to assign a version number to the custom system image, within the range 128..255. Follow the **-V** switch with the desired version number, in decimal. Move custom system image files to the **VER***nnn* subdirectory in the **IMAGES** directory that corresponds to the version number chosen. If the appropriate **VER***nnn* directory does not exist, create one.

The following example creates a custom system image from the SYS3150 standard system image and the object files named in the Objs.lst script

file. The output files are directed (with the "-o" switch) to files named "myimage.*."

nld –c –b c:\LonWorks\Images\Ver13\sys3150 –t 3150 –Z 4F –V 128 –o myimage -@objs.lst

After using the linker as described above, the following custom system image files are created:

- The **myimage.sym** file contains a list of the image's exported symbols for use when linking an application program to the image.

- The **myimage.nx** file is an Intel hex-format file containing the binary linked system image.

- The **myimage.map** file (if requested) is a link map and report for the image.

2   Move the system image files into the appropriate **Ver*nnn*** directory corresponding to the version number you supplied in step 1.

Following are additional guidelines for constructing custom system images:

1   You can use libraries when constructing custom system images, but only to the extent that they are referenced. The complete library contents are not automatically transferred into the custom system image. Any global variables defined in the library must be declared as **far** (RAM) variables in order to be used in constructing a custom system image.

2   You cannot program PROMs directly with a custom system image. To program a PROM, you must build a ROM image file that is based on the custom system image using the NodeBuilder tool. You can use an empty application to build the image if you just want to load the custom system image into PROM.

3   Use the **static** keyword to declare functions or data items that are not to be exposed to the application program. This declaration effectively hides that symbol.

4   Construct include files containing **extern** function prototypes and **extern** data declarations for the users of the custom system image to include in their programs. You must use the **extern** keyword for each declaration to allow the compiler to establish the correct calling sequence and use the appropriate assembler commands to permit linking to the data object or function contained in the custom system image.

Then, include these include files in the custom system image source files that define these functions and data objects so that the Neuron C compiler can ensure that your **extern** declarations and prototypes match your actual declarations and function definitions. You can have a specific **extern** declaration precede the actual declaration, as long as the declarations match. This technique can help prevent calling a function with incorrect parameters (due to an incorrect **extern** prototype, for example), which could result in an overwritten data stack and thus a device which repeatedly experiences a watchdog timer reset, overwritten variables, or similar software failures.

5   You can use a maximum of 64 bytes of RAM within a custom system image. If one custom system image is used as a base for another, this

limit is cumulative, in other words, the two (or more) images cannot use more than 64 bytes in total. Any unused RAM from this 64-byte area is made available to the application program when it is linked. Custom system images can only use **far** RAM variables. Include the **far** keyword in any **extern** declarations of these variables (see point # 4, above).

6	You cannot access EEPROM variables from a custom system image.

Providing a Large RAM Space

The total amount of RAM space available to a custom system image at link time is 64 bytes. However, if a custom system image needs a larger amount of RAM for certain functions, a large RAM block can be declared by an application program that uses the custom system image, and the application program can make this known to the custom system image during reset. This functionality can be placed inside a reset routine provided in an include file associated with the custom image. An application using the custom image would then be expected to include the include file, and call the reset routine from the **when(reset)** task.

A pointer to this RAM block can be passed as a parameter to the appropriate custom system image functions each time they are called. For a more efficient implementation, the custom system image can use 2 bytes of its RAM space to declare a global pointer to such a block of memory, and initialize the pointer to NULL. When the application program resets, it has the responsibility of correctly setting this global pointer variable to point to the block of memory, or at least the responsibility of calling an initialization function (providing this pointer).

Performing Neuron C Functions

The pure C code that is placed in a custom system image cannot contain references to network variables, messages, I/O objects, timers, or other Neuron C objects. However, the custom system image can be designed mainly for the purpose of performing Neuron C related tasks, such as standard I/O device management, message construction, or timer manipulation.

You can access Neuron C objects from a custom system image function by making it the responsibility of the application program to actually perform the Neuron C operation in a function. The custom system image can declare a RAM variable that the application sets to a pointer to the application function. The custom system image can then call the function in the Neuron C application program, and effectively perform Neuron C operations.

For example, consider a custom system image that contains routines for management of a standard LCD display device. This custom system image would contain various routines for formatting information and for managing the display in response to various commands from the application program. It is desirable to have the custom system image code automatically perform the I/O operations to update the device. It might also be necessary for such a custom system image to have access to a large buffer in RAM. However, due to the pure C restriction and the custom system image RAM memory restriction, neither of these requirements can be implemented solely within the custom system image. The RAM buffer can be provided by the application as discussed above.

As for the I/O operations on the display, assume that the display has a Neurowire device interface. The custom system image could be accompanied with an include file for the benefit of whatever Neuron C application program uses it. The include file can contain the Neurowire I/O declarations and the function definitions necessary to support the display's I/O. It can even contain a routine to be called from the application's **when(reset)** task that initializes the appropriate function pointers and memory buffer pointers. Then, the custom system image can access the functions and memory buffers as necessary without further intervention from the application program, and without the application program's being responsible for passing some special parameters each time it wants to interact with the custom system image display management software.

This indirection through function pointers is necessary for a custom system image, because a custom system image cannot contain any unresolved external symbols when the linker creates it. However, a library of display management software could just access such I/O functions as discussed above directly by name, without having to resort to function pointers.

D

Neuron C Language Implementation Characteristics

This appendix discusses how the Neuron C language implements certain language characteristics that are defiend as "language-specific" in the ISO and ANSI standards for the C language.

Neuron C Language Implementation Characteristics

The International Organization for Standardization (ISO) and International Electrotechnical Commission (IEC) International Standard ISO/IEC 9899 for the C programming language states (in Appendix J, Section 3) that each C language implementation "is required to document its choice of behavior in each of the areas listed in this subclause. The following [aspects of the language] are implementation-defined."

The standard defines the term "implementation-defined behavior" as "unspecified behavior where each implementation documents how the choice is made." Thus, all these items are language definition issues not specified in the ISO/IEC standard, but instead left up to the individual implementer. They are also potential portability issues.

Each heading below references the subclause in Appendix J of the ISO/IEC C language standard, and the appropriate section of that appendix. In addition, some of the sections below refer to clauses of Appendix F of the older ANSI C standard. Each answer applies to the latest implementation, as of the date of printing, of the Neuron C Version 2 compiler supplied by Echelon Corporation.

Translation (J.3.1)

Q: *How is a diagnostic identified? (Sec. 3.10, Sec. 5.1.1.3)*

A: Each Neuron C diagnostic consists of at least two lines output to the standard output file. One of these keywords introduces the diagnostic: FYI (For Your Information), Warning, Error, or FATAL. The remainder of the first line consists of the full path name of the source or include file to which the diagnostic applies, followed by a line number, and a column number in parentheses.

The second (and possibly subsequent) lines contain the diagnostic. Each of the diagnostic message lines is indented one tab stop.

FYI and warning diagnostics *do not* prevent the compiler from successfully completing translation. All *warning* diagnostics should be examined and corrected, however, as they are likely to indicate programming problems or poor programming practice.

Error diagnostics *do* prevent the compiler from successfully completing translation. They may also result in masking of other errors; thus the compiler may not be able to locate all errors in a single compilation pass.

FATAL diagnostics prevent the compiler from performing any further translation. These diagnostics result from resource problems (out of memory, disk full, and so on) or from internal checking on the compiler itself. Any diagnostic of the form ***TRAP *n*** *, where *n* is a decimal number, should be reported to Echelon Customer Support.

Environment (J.3.2)

Q: What are the semantics of the arguments to **main***? (Sec. 5.1.2.2.1)*

A: Neuron C places no special meaning on the procedure **main**. The name **main** can be used as any other legal identifier.

Q: What constitutes an interactive device? (Sec. 5.1.2.3)

A: Neuron C defines no interactive devices.

Identifiers (J.3.3)

Q: What is the number of significant initial characters (beyond 31) in an identifier without external linkage? (Sec. 5.2.4.1, Sec. 6.4.2)

A: An identifier without external linkage can extend to 256 characters. All characters are significant.

Identifiers (F.3.3 of ANSI C Standard)

Q: What is the number of significant initial characters (beyond 6) in an identifier with external linkage? (Sec. 3.1.2)

A: There are two forms of external linkage in Neuron C: *traditional external* and *network external*. Traditional external consists of the **extern**, **static**, and file scope variables and procedure names. These names are used by the Neuron C linker when linking the program to construct a load image. Names declared with the **extern** or **static** storage classes, or declared at file scope, cannot exceed 63 characters. In some cases, the compiler may append characters to the name to make it unique, and in these cases, the external identifier may be further restricted in length, but in no case is the name required to be shorter than 50 characters. The compiler produces a warning diagnostic when such names have excessive length, and it also truncates these names to the maximum allowable length. Therefore, it is best to restrict traditional external names to 50 characters or less.

The second form of external linkage, network external, consists of the names used by the network and by a network tool. These names include names of network variables, names of message tags, and names of **typedef**s used in defining network variables of nonstandard types. The compiler produces an error diagnostic for each network external name that exceeds 16 characters. Functional block names are considered network external names when there is no **external_name** or **external_resource_name** option in the **fblock** declaration. If the option is supplied, an internal functional block name can be up to 64 characters.

Q: Are case distinctions significant in an identifier with external linkage? (Sec. 3.1.2)

A: Yes, case is significant in an identifier with external linkage, for both forms of external linkage described above.

Characters (J.3.4)

Q: *What are the members of the source and execution character sets beyond what the standard explicitly defines? (Sec. 5.2.1)*

A: The Neuron C character set uses the basic ASCII character encoding for its source and execution character sets. The Neuron C source character set is the character set as explicitly defined in the standard. The ASCII carriage return character (hex 0D) and the ASCII backspace character (hex 08) are both accepted as white space. The end-of-line character is the ASCII new-line (hex 0A). Additionally, the Neuron C compiler accepts the remaining basic ASCII printable characters @ (at-sign) and ` (accent-grave) in character constants and string literals.

The Neuron C compiler interprets the ASCII EOT character (hex 04) as an end-of-file marker. Likewise, the character Ctrl-Z (hex 1A), which is the MS-DOS end-of-text-file character, is an end-of-file marker. However, neither of these characters is *required* by the Neuron C compiler.

The execution character set is intended to be basic ASCII (character values 0 .. 127). However, a program written in Neuron C is free to use any interpretation of character values outside the range 0 .. 127.

Q: *What is the mapping of the members of the source character set (in character constants and string literals) to members of the execution character set? (Sec. 6.4.4.4, Sec. 5.1.1.2)*

A: The mapping from the source character set to the execution character set is the identity relationship.

Q: *What is the value of an integer character constant that contains a character or an escape sequence not represented in the basic execution character set or the extended character set for a wide character constant? (Sec. 6.4.4.4)*

A: An integer character constant can only contain characters in the basic execution character set. With escape sequences, character constants can be constructed ranging from 0 to 255, or if **signed** chars are used, ranging from -128 (**\x80**) through 127 (**\x7F**).

Q: *What is the value of an integer character constant that contains more than one multibyte character? What is the current locale used to convert multibyte characters into corresponding wide characters (codes) for a wide character constant? (Sec. 6.4.4.4)*

A: The Neuron C compiler does not implement multibyte characters.

Q: *Does a "plain"* **char** *have the same range of values as* **signed char** *or* **unsigned char***? (Sec. 6.2.5, Sec. 6.3.1.1)*

A: A "plain" **char** is identical to an **unsigned char**.

Characters (F.3.4 of ANSI C Standard)

Q: *What are the shift states used for the encoding of multibyte characters? (Sec. 2.2.1.2)*

A: Neuron C does not support multibyte characters. Character constants containing more than one character are errors.

Q: *What are the number of bits in a character in the execution character set? What is the size of a wide character—that is, the type of* **wchar_t**? *(Sec. 2.2.4.2.1)*

A: The execution character set uses an 8-bit representation. The Neuron C compiler does not support wide characters, but the type of a wide character, **wchar_t**, is defined to be **unsigned long**. (Note that Neuron C defines **unsigned long** as 16 bits.)

Integers (J.3.5)

Q: *What is the result of converting an integer to a shorter signed integer? What is the result of converting an unsigned integer to a signed integer of equal length, if the signed integer cannot represent the unsigned integer's value? (Sec. 6.3.1.3)*

A: Conversion from **long** to **short** may result in data loss, depending on the value being converted, since this conversion is performed by discarding the most significant byte of the **long** integer. If, for example, a **long** integer containing the value 513 (hex 0201) was converted to a **signed short**, discarding the most significant byte of the **long** integer would result in the value 1.

Conversion from an **unsigned** integer to a **signed** integer of equal length may result in a negative number. For example, an **unsigned short** integer may have the value 255 (hex FF), but when converted to a **signed short** integer, it is then interpreted using two's complement, and the value becomes -1.

The Neuron C compiler produces diagnostic messages when data might be lost as the result of an *implicit* conversion operation. *Explicit* conversion, such as through a cast operation, does not produce a diagnostic message. As an example, in the code fragment below, the assignment to **x** results in a diagnostic *warning* message, but the assignment to **y** does not.

```
int x, y;
x = 285;           // Data is lost, x is assigned 29.
                   // Warning is produced.

y = (int)285;      // Data is lost, y is assigned 29.
                   // No warning is produced.
```

Q: *What are the results of bitwise operations on signed integers? (Sec. 6.5)*

A: Bitwise operations on signed integers are performed as if the values of the operands were **unsigned**. The result is interpreted as **signed**. Thus, the result of **(-2)|1** is **-1**.

Integers (F.3.5 of ANSI C Standard)

Q: *What are the representations and sets of values of the various types of integers? What is the order of bits in a multi-unit integer representation? What is the method of encoding an unsigned integer? What is the method of encoding a signed integer? (Sec. 3.1.2.5)*

A: An **int** implies a **short int** by default, which is 8 bits in Neuron C. The 8-bit byte is the fundamental unit of storage on the Neuron Chip. A **long int** is a 16-bit, or 2-byte, integer representation. The **<limits.h>** include file contains definitions of the various integer-type ranges. These values are:

```
  -128 ..    127    signed short
     0 ..    255    unsigned short
-32768 ..  32767    signed long
     0 ..  65535    unsigned long
```

All unsigned integer values use binary representations. Signed integers use two's complement binary representations. The **long int**, a multi-unit representation, is stored such that the most significant byte is at the lowest address.

Q: What is the sign of the remainder on integer division? (Sec. 3.3.5)

A: The sign of the remainder of an integer division (that is, **op1 % op2**) is always the same as the sign of **op1**.

Q: What is the result of a right shift of a negative-valued signed integral type? (Sec. 3.3.7)

A: When a negative-valued signed integral type is right-shifted, binary ones are shifted in from the left. Thus, for **int x** and **long int x**, **(x>>1)** is always equal to **(x/2)**.

Floating Point (J.3.6)

Neuron C does not support floating-point operations with C syntax or operators. A floating-point library is included with Neuron C that implements a limited set of floating point operations as function calls. The floating-point library operates on data that conforms to IEEE 754.

Arrays and Pointers (J.3.7)

Q: What is the result of casting a pointer to an integer, or vice versa? What is the result of casting a pointer to one type to a pointer to another type? (Sec. 6.3.2.3)

A: The binary representations of pointers and **unsigned long** integers are the same. Thus, the result of casting a pointer to an integer of a certain type is the same as casting an **unsigned long** to an integer of the same type. Integers cast to pointer undergo the same conversions as integers cast to **unsigned long**.

All pointer representations are interchangeable. Thus, no conversion results from casting a pointer to one type to a pointer to another type, and the use of such a pointer produces the expected results.

*Q: What is the type of integer required to hold the difference between two pointers to elements of the same array, **ptrdiff_t**? (Secs. 6.5.6)*

A: The result of subtraction between two pointers is a [**signed**] **long**.

Arrays and Pointers (F.3.7 of ANSI C Standard)

*Q: What is the type of integer required to hold the maximum size of an array— that is, the type of the **sizeof** operator, **size_t**? (Secs. 3.3.3.4, 4.1.1)*

A: The maximum size of an array (32,767 elements) requires an **unsigned long**.

Hints (J.3.8)

Q: *What is the extent to which objects are actually placed in registers by use of the* **register** *storage class specifier? (Sec. 6.7.1)*

A: The Neuron Chip uses a stack-based architecture. Because this architecture has no general-purpose registers, the compiler ignores the **register** storage class. The compiler also produces a warning diagnostic whenever the **register** class is used.

Structures, Unions, Enumerations, and Bit-Fields (J.3.9)

Q: *Is a "plain"* **int** *bit-field treated as a* **signed int** *bit-field or as an* **unsigned int** *bit-field? (Sec. 6.7.2, Sec. 6.7.2.1)*

A: A "plain" **int** bit-field is treated as a **signed int** bit-field. Use of **unsigned** bit-fields is recommended, unless a sign is needed, since **unsigned** bit-fields are more efficient in runtime and code space.

Q: *What is the order of allocation of bit-fields within a unit? (Sec. 6.7.2.1)*

A: Bit-fields are allocated from high-order bit (most-significant bit) to low-order bit (least-significant bit) within a byte.

Q: *Can a bit-field straddle a storage-unit boundary? (Sec. 6.7.2.1)*

A: No. A bit-field cannot straddle a byte boundary. Therefore, the largest bit-field is 8 bits.

Q: *What is the integer type chosen to represent the values of an enumeration type? (Sec. 6.7.2.2)*

A: The integer type **int** is used to represent the values of an enumeration type. Thus, the valid range of enumerator values is -128 ... 127.

Structures, Unions, Enumerations, and Bit-Fields (F.3.9 of ANSI C Standard)

Q: *What are the consequences of accessing a member of a union object with a member of a different type? (Sec. 3.3.2.3)*

A: Union members of different types overlay each other at the same offsets within a union. Thus, the consequences of accessing a pointer as a **long** or as an **unsigned long**, or vice versa, are the same as casting the member. Likewise, the consequences of accessing an **int**, or **char**, or **short**, as another typed member from the same list is the same as casting the member. Accessing a **long** data type or pointer data type as a **short** results in the value of the most significant byte. Accessing a **short** data type as a **long** results in reading or changing an unused byte (the least significant byte of the **long**), and the most significant byte of the **long** mapping to the **short**.

Q: *What is the padding and alignment of members of structures? (Sec. 3.5.2.1)*

A: Because the architecture of the Neuron Chip is **byte** aligned, no padding is needed or performed between members of structures in Neuron C.

Qualifiers (J.3.10)

*Q: What constitutes an access to an object that has **volatile**-qualified type? (Sec. 6.7.3)*

A: Neuron C does not support **volatile**-qualified type. The compiler also produces a warning diagnostic whenever the **volatile** qualifier is used.

Declarators (F.3.11 of ANSI C Standard)

Q: What is the maximum number of declarators that can modify an arithmetic, structure, or union type? (Sec. 3.5.4)

A: There is no limit to the maximum number of declarators that modify any type. The limit is determined at run-time by the amount of heap memory and stack space available to the compiler.

Statements (F.3.12 of ANSI C Standard)

*Q: What is the maximum number of **case** values in a **switch** statement? (Sec. 3.6.4.2)*

A: The Neuron C **switch** statement only accepts an **int** expression for the switch value. Because no two **case** labels in a **switch** statement can have the same value, there are only 256 choices permitted. Neuron C accepts all 256 different **case** values for a single **switch** statement.

Preprocessing Directives (J.3.11)

Q: Does the value of a single-character constant in a constant expression that controls conditional inclusion match the value of the same character constant in the execution character set? Can such a character constant have a negative value? (Sec. 6.10.1)

A: Yes, and yes.

Q: What is the method for locating includable source files? (Sec. 6.10.2)

A: The normal include directive should use the quoted form. To access the *system* include files, the directive should use the bracketed form.

Example:

```
#include <stddef.h>
```

System include files are defined as the include files installed by the IzoT NodeBuilder Software. They are located in the LonWorks Neuron C\Include directory (**c:\LonWorks\NeuronC\Include** by default). With the exception of the LonWorks directory, the location of the system include files cannot be changed.

Use the following quoted form for all user include files:

```
#include "[drive:][pathname\]filename.ext"
```

The quoted form causes the compiler to use the filename as it stands if it is absolute or drive-relative. Otherwise, the compiler first searches the working directory (using a relative pathname if one is supplied). Next, it searches each of

the directories specified in **Include Directories** of the NodeBuilder Device Templates Properties dialog and Project Properties dialog.

When working from within the NodeBuilder Project Manager, or from the command line but through the NodeBuilder Project Make Utility, the *current working directory* is the folder that contains the main Neuron C source file.

The bracketed form shown below:

```
#include <filename.ext>
```

searches the **include** subdirectory within the *standard files directory* for system include files (such as **<limits.h>** and **<stddef.h>**). (The search for include files specified by the bracketed form is unaffected by the directories specified in **Include Directories** of the NodeBuilder's properties dialogs.)

Q: What is the support of quoted names for includable source files? (Sec. 6.10..2)

A: The quoted names in the **#include** directive can be any valid filename under the Windows operating system, with absolute, drive-relative, or relative pathname, if any. Pathnames can be relative to the current working directory, or relative to any of the directories on the include file search path.

When working from within the NodeBuilder Project Manager, or from the command line but through the NodeBuilder Project Make Utility, the *current working directory* is the folder that contains the main Neuron C source file.

*Q: What is the mapping of source file character sequences in the **#include** directive? (Sec. 6.10.2)*

A: The source file character sequences can be upper or lower case. Any valid filename character can be used. Case is not significant.

*Q: What is the behavior of each recognized **#pragma** directive? (Sec. 6.10.6)*

A: The **#pragma** directives are documented in the *Compiler Directives* chapter of the *Neuron C Reference Guide*.

Q: What are the definitions for __DATE__ and __TIME__ when respectively, the date and time of the translation are not available? (Sec. 6.10.8)

A: Neuron C does not support the __**DATE**__ and __**TIME**__ macros.

Library Functions (J.3.12)

*Q: What is the **NULL** pointer constant to which the macro **NULL** expands? (Sec. 7.17)*

A: The **NULL** pointer constant is defined to be **0** in the **<stddef.h>** file.

Neuron C is, generally, a "freestanding implementation." This means that Neuron C does not include a full Standard C library as part of the implementation. However, some Standard C functions are available. There are some string functions, and memory functions, such as **strcpy( )** and **memcmp( )**. Consult the *Functions* chapter of the *Neuron C Reference Guide* for more information on the functions supported.

Index

conversion
   cast, 242
   integer, 241
   pointer, 242
cp keyword. *See* config_prop keyword
cp_family keyword, 6, 85, 86, 175, 179, 180, 184
cp_info keyword, 87
CPT, definition, 3
create a new library, 219
critical sections, 138, 143, 144
   boundary, 48, 127, 135, 139, 149
   definition of, 47
ctrl-Z character, 240
custom functional profiles, 9
custom system images
   advantages of, 230
   construction of, 231
   definition, 228
   disadvantages of, 230
   providing a large RAM space, 234

D

data keyword, 121, 128
declaration, I/O object, 29
declarations, 7
   order of, 204
declarators
   limits on, 244
delay( ) function, 40
dest_addr keyword, 122
device
   bringing online, 151
   commissioning, 21
   context for properties. *See* context
      expression, for device
   forced sleep, 168
   initialization
      and the wink event, 152
   interface, 3, 9, 10, 102, 105
   reset, 7, 134, 151, 169
      causes of, 150
      effect of, 21
      time required, 169
direct event processing, 19, 127, 135, 153
director keyword, 104, 106, 112
disable_mult_module_init pragma, 207
disable_snvt_si pragma, 11, 48, 201
disadvantages of a library, 223
distributed systems, 9
domain table, 188, 190
   memory use, 174
duplicate keyword, 128

E

EECODE memory area, 179
EEFAR memory area, 179

EENEAR memory area, 179
EEPROM, 178
   on-chip, address table, 189
   on-chip, alias table, 189
   on-chip, domain table, 190
   on-chip, reallocating, 188
   pointers to, 187
   use of, 174
   variables
      pointers to, 13
   write timer, 37, 41
eeprom keyword, 6, 7, 179, 180, 181
eeprom_memcpy( ) function, 13, 187
efficiency of code, 204
enable_sd_nv_names pragma, 10, 48, 201
end-of-file marker, 240
end-of-line character, 240
enum variable type, 202
   predefined, 5
enumeration type, 243
EOT character, 240
error diagnostic from compiler, 238
error handling, 168
error status, access, 172
error_log( ) function, 75, 171
event-driven scheduling, 4
events, 18, 20
   blocking queue, 21, 22, 127, 129, 148
   expression, 22
   latency, 37
   posting of, 37
   predefined, 18
   processing of, 20
      completion events, 20
      network events, 20
      queue, 20
      responses, 20
      when clause, 21
   scheduler, 16
   unqualified, 26, 131
   unsolicited, 21
   user-defined, 18, 22
expired timers. *See* timers
explicit addresses, 197
   for network variable updates, 131
explicit addressing, 192, 198
explicit messages, 119, 191
   events, 197
   functions, 197
   receiving
      implementation caveat, 127
exporter
   command line switches, 218
exporter command line switches, 217
extended arithmetic, 5
extern keyword, 6, 233, 239
external_name keyword, 13, 104, 106, 239
external_resource_name keyword, 104, 106, 239

F

far keyword, 7, 181, 182
fastaccess keyword, 205
fatal error diagnostic from compiler, 238
fblock keyword, 104, 239
fblock_director( ) function, 114
fblock_index_map variable, 114
file transfer protocol, 118
firmware, 16
   error handling, 169
   helper functions, 204
   I/O objects, 11
   initialization actions, 202
   initialization time, 21
   offline processing, 170
   preemption mode, 134
   scheduler, 146
   version, 217
fixed timers, 36
flash memory, 176, 178, 179, 187
   effects of writing, 185
   sectors, 185
   Series 3100, 185
   Series 5000, 186
   use of, 174, 178
floating-point, 5, 242
   syntax and operators, 13
flush
   pending updates, 165
flush( ) function, 166, 168
flush_cancel( ) function, 166
flush_completes event, 166, 168
flush_wait( ) function, 134, 136, 166, 170
foreign-frame messages, 124
forward declarations, 24
function calls, 206
function prototypes, 14, 24
functional blocks, 10, 102
   accessing members, 110
   accessing properties, 110
   director function, 106, 113
      examples, 114
   implementation-specific members, 10, 103, 105
   limitations on name length, 13
   member list, 104
   members, 10
   members, definition of, 3
functional profile templates. *See* functional profiles
functional profiles, 3, 10, 102, 105
   custom, 9
   standard, 9
   using inheritance, 109
FYI diagnostic from compiler, 238

G

gateway, 120, 124
get_nv_length_override( ) function, 74, 75
get_tick_count( ) function, 39
global data, 6
global keyword, 90, 96, 107, 108, 115
global_index keyword, 112
go_offline( ) function, 75
going offline in bypass mode, 152
group, 193

H

host-based compilation, 12

I

I/O devices, 11
I/O object
   declaring, 29
   summary, 28
   syntax, 29
I/O objects, 11, 13
   initialization of, 7
idempotent transaction, 141
identifiers, 239
implementation_specific keyword, 104, 105
implements keyword, 104
include directive, 245
include files, 244
incoming message queue
   blocked. *See* events, blocking queue
initial value updates, 62
initialization
   Neuron Chip, 169
   time required, 169
input clock frequency, 36
input_value variable
   examples, 34
int, 241, 243
integer character constant, 240
integer constants, 4
   types for various values, 241
integer conversion
   unsigned to signed, 241
integer division
   sign of result, 242
interoperability, 4, 9, 44, 84
   proprietary interface, 118
   requirements for certification, 10, 190
interoperable devices, 2
interrupt
   controlling, 159
   debugging, 164
   I/O, 154, 155
   latency, 162
   lock keyword, 161
   overview, 24

periodic, 154, 158
restrictions, 165
semaphore, 161
sharing data, 161
sources, 153
task, 155
timer/counter, 154, 156
timing, 162
interrupt service routine. *See* ISR
io_changes event, 19
memory use, 174
io_update_occurs event
examples, 31, 33
is_bound( ) function, 50
ISR
defining, 155

L

len keyword, 128
librarian, 220, 222
command line switches, 219
libraries
advantages of, 223
disadvantages of, 223
including in link, 217
report of library contents, 219
library, 222
definition, 228
functions, 13
limits.h, 241
link map, 209
linker, 222, 223
command line switches, 216
linking a program, 184
lock keyword, 161
logging system errors, 171
long int, 13, 241
long to short integer conversion, 241
LONMARK Interoperability Association
website, 10
LonTalk protocol, 119
LONWORKS messages, 9
lowering power consumption, 167

M

magcard I/O object, 150
magtrack1 I/O object, 150
main( ), 14, 239
max( ) function, 202
max_rate_est option, 122
Media Access Control (MAC) layer, 119
memcpy( ) function, 124, 188
memory
page, definition of, 205
usage
default, 180
non-default, 181

use by program elements, 174
wait states, 178
memory-mapped I/O
usage tip, 198
message codes, 120, 123
application-specific, 123
ranges, 123
message data
block transfer, 124
message tags, 13, 123, 200
and explicit addressing, 130
connecting, 130
declaration, 122
default msg_in tag, 130
limitations on name length, 13
non-bindable, 130
syntax, 122
messages, 9
cancelling, 126
code, 118
completion status, 131
data field, 118
events, 153
explicit, 11
explicit addressing of, 130, 197
foreign-frame, 120, 124
implicit, 119
incoming, 122
format of, 127
list of steps, 120
priority, 143
processing completion events, 133
protocol overhead, 192
receiving, 126
sending, 125
unwanted, 128
messaging service, 4
millisecond timers, 25
min( ) function, 202
Miranda prototype rule, 25
model file, 12
monitoring device, 65
msec_delay( ) function, 40
msg_alloc( ) function, 135, 143
msg_alloc_priority( ) function, 143
msg_arrives event, 20, 126, 127
msg_cancel( ) function, 125, 144
msg_completes event, 20, 123, 131, 148
msg_fails event, 21, 131, 141, 148
msg_free( ) function, 143
msg_in message tag, 122
msg_in object, 127, 142, 143
addr field, 128, 198
fields invalidated, 128
msg_out object, 120, 130, 144
defined, 121
dest_addr field, 198
tag field, 123
msg_receive( ) function, 126, 127, 137, 144, 150

msg_send( ) function, 120, 125, 130, 144, 197
msg_succeeds event, 21, 131, 141, 148
    comparison with resp_arrives event, 141
msg_tag keyword, 122, 130
msg_tag_index variable, 131
mtimer, 37
    accuracy of, 37
    and clock speed, 37
    keyword, 25
multibyte characters, 240
multicast connections
    and buffer use, 193
multi-character displays, 35
multi-processor architecture, 149

N

net_buf_in_count pragma, 195, 196
net_buf_in_size pragma, 195, 196
net_buf_out_count pragma, 194, 196, 198
net_buf_out_priority_count pragma, 194, 196, 198
net_buf_out_size pragma, 194, 196, 197
network buffers, 190, *See* buffers
network congestion, effects of, 168
network tool, 151
    initialization of configuration properties, 87
network variables, 44
    advantages of, 8
    alias, 189
    arrays of, 47, 48
    changeable type, 99
    changeable-type, 68
    communication model, 11
    configuration properties, 84
    configuration table, 188
    connecting, 45, 49
    declaring, 46, 47
      as polled, 58
    definition, 3
    events, 51
    example of declarations, 48
    explicitly addressed, 120
    how they work, 119
    initial value updates, 62
    initialization of, 7
    limitations on name length, 13
    maximum number of, 47
    memory use, 175
    outgoing updates, 149
    pointers to, 13
    polling of, 56, 65, 136, 148
      examples, 57, 59
    priority
      examples, 49
    processing completion events, 55
    propagation of, 9, 53
    sharing a common address table entry, 189
    size, 48

structures, 13
    synchronous, 53, 54, 134, 135, 149
      effects on performance, 54
      updating, 54
    synchronous vs. nonsynchronous, 54
    syntax of, 47
    updates, 47, 53, 153
      scheduling of, 47
    using explicit addressing to send an update, 131
network, storage class, 6
Neuron C
    and ANSI C, differences, 13
    character set, 240
    compared to ANSI C, 13
    declarations, 7
    definition of, 2
    portability issues, 238
    storage classes. *See* storage classes
    thread of execution, 14
Neuron C compiler
    command line switches, 215
Neuron C tools, 211
Neuron Chip
    definition of memory page, 205
    flushing messages, 166
    initialization, 169
    sleep mode, 166
    wake up, 165
Neuron-hosted compilation, 12
neurowire I/O object, 37, 150
    examples, 35
node_reset( ) function, 169
NodeBuilder project manager, 222, 229
nonauth keyword, 67
nonauthenticated keyword, 67
nonbind keyword, 122, 130
non-bindable message tags, 130
nonconfig keyword used with authentication, 67
non-idempotent transaction, 141
NULL, definition of, 245
num_addr_table_entries pragma, 189, 200
num_alias_table_entries pragma, 189
num_domain_entries pragma, 190
nv_in_addr variable, 65, 198
nv_len property, 74
nv_properties keyword, 93
nv_update_completes event, 20, 51, 53, 148
    examples, 53
nv_update_fails event, 20, 51, 52, 148
    examples, 52
nv_update_occurs event, 20, 51, 57
    examples, 51
nv_update_succeeds event, 20, 50, 51, 52, 148
    examples, 52
NVT
    definition, 3

O

object files, 222, 223
offchip keyword, 7, 179, 181, 183, 185
off-chip memory
    use of, 176
offline event, 18, 20, 127, 151, 152
offline_confirm( ) function, 152
onchip keyword, 7, 179, 181, 183
online event, 18, 20, 127, 151
optimization
    common sub-expressions, 205
outgoing network variable updates, 149

P

padding of structures, 243
parallel processing, 149
partial completion event testing. *See*
    completion events
pending updates
    flushing, 165
pointers, 13, 242
    subtraction of, 242
poll( ) function, 57
polled applications, 11
polled keyword, 57, 58
polled network variables. *See* network
    variables, polling of
polling
    definition of, 56
post_events( ) function, 47, 127, 139, 149, 150
power consumption
    limiting, 168
    lowering, 167
power failure
    effects on flash memory, 186
pragmas, 245
predefined events, 18
preempt_safe keyword, 17, 55, 134
preemption mode, 17, 54, 55, 134, 135, 144, 192
preemption_mode( ) function, 135
preprocessor directives, 244
priority keyword, 17, 146
priority when clauses, 23
    starving execution of nonpriority, 24
priority_on keyword, 121
processor execution
    lockout when writing flash or EEPROM
        memory, 185
propagate( ) function, 60
propagation
    of network variables. *See* network variables,
        propagation of
property lists, 88
    for functional blocks, 107
    for network variables, 90
protocol
    overhead, 192

ptrdiff_t, 242
pullup resistors, internal, 167
pulsecount I/O object, 36, 150
pure C, 222, 225, 228, 234
    definition, 228

Q

quadrature I/O object
    examples, 31, 33

R

RAM, 178
    custom image needs, 234
    use, 174
ram keyword, 7, 179
    for functions, 183
RAMCODE memory area, 179
RAMFAR memory area, 179
RAMNEAR memory area, 179
range_mod_string keyword, 90, 107
rate_est option, 122
raw data value, 73
rcvtx keyword, 128
reader devices, 66
    behavior of, 46
receive transactions
    number of, 195
    requirements, 195
    size, 195
receive_trans_count pragma, 195, 197, 198
receiving a message, 126
register keyword, 13, 243
registers, 243
relaxed_casting_on pragma, 13, 61, 111, 187
relinking a program, 184
remainder operation
    sign of result, 242
repeating keyword, 25
repeating timers, 39
request message. *See* request/response
    message service
REQUEST service type, 122, 128
request/response message service, 137
    examples, 140
    for messages, 136
    using, 136
    with explicit messages, 118
reserved words, 14
reset cause register, 172
reset event, 21, 202
reset pin, 169
reset time, 21
resetting a device. *See* device, reset
resource files, 3, 9, 10, 84, 87, 103
    Resource Editor, 9, 10, 84, 103
resp_alloc( ) function, 144
resp_arrives event, 21, 138, 141

domain. *See* domain table
    network variable configuration, 188
tag keyword, 121
tasks, 16, 18, 47
    order of execution, 23
    priority, 23
    returning from, 18
template file. *See* configuration properties,
    template file
timeout
    waiting on buffer, 134
timer
    millisecond, 25, 37
    preemption mode timeout, 36
    pulsecount input, 36
    second, 25, 37
    triac pulse, 37
    write, EEPROM. *See* EEPROM, write timer
timer objects. *See* timers
timer_expires event, 26, 153
    examples, 17
timers, 4, 13, 16
    accuracy of, 36, 37
    checking for specific, 27
    duration, formula for, 38
    examples, 26
    expiration of, 149
    expired, 25
    fixed duration, 36
    in the wink task, 153
    initialization of, 7
    maximum number of, 25
    measuring very short durations of time, 39
    memory use, 174
    repeating, 25, 39
    starting over, 25
    time remaining, 25
    turning off, 25
    turning off before sleep, 167
    unqualified event expressions, 26
    use in the debugger, 26
timers_off( ) function, 203
transaction, idempotent, 141
transmit transaction buffers, 198
TRAP *n* diagnostic from compiler, 238
triac I/O object, 37
    examples, 33
type qualifiers, 6
typedef keyword, 86

U

UCPT

    definition, 3
    use of, 86
UFPT, 103
UFPT, definition, 3
UNACKD service type, 122, 128
UNACKD_RPT service type, 122, 128
unacknowledged service, 49
unicast connections
    and buffer use, 193
uninit keyword, 7, 182, 184
unions, 13, 243
unqualified events. *See* events, unqualified
unsigned long, 241
unsigned to signed integer conversion, 241
UNVT
    definition, 3
user-defined events, 18, 22

V

value files. *See* configuration properties, value
    files
variables
    declaration order, 204
    initialization, 7
volatile keyword, 13, 244

W

wait states
    Neuron Chip, 178
wake up Neuron Chip, 165
warning diagnostic from compiler, 238
warnings_off pragma, 187
watchdog timer, 37, 41, 134, 149, 150, 186, 188
watchdog_update( ) function, 150
    examples, 150
wchar_t, 241
when clauses, 16
    default, 127
    memory use, 175
    priority, 23, 146
    scheduling, 22
when statement, 14
wiegand I/O object, 150
wink command, 151
wink event, 20, 127, 151, 152
working directory, 245
writer device, 66
    behavior of, 46

Appendix: MCPP License Terms

The Neuron C Compiler uses a pre-processor based on the open-source
implementation of *MCPP* by Kiyoshi Matsui. Following are the license terms:

```
/*-
 * Copyright (c) 1998, 2002-2008 Kiyoshi Matsui <kmatsui@t3.rim.or.jp>
 * All rights reserved.
 *
 * Some parts of this code are derived from the public domain software
 * DECUS cpp (1984,1985) written by Martin Minow.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 * 1. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 *
 * THIS SOFTWARE IS PROVIDED BY THE AUTHOR ``AS IS'' AND
 * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR BE LIABLE
 * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
 * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
 * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
 * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
 * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
 * SUCH DAMAGE.
 */
```