ECHELON

Neuron C Reference Guide

Provides reference info for writing programs using the Neuron C programming language.

078-0140-01G

Echelon Corporation
www.echelon.com

Welcome

This manual describes the Neuron® C Version 2.3 programming language. It is a companion piece to the *Neuron C Programmer's Guide*. It provides reference information for writing programs using Neuron C. Neuron C is a programming language based on ANSI C that is designed for applications that run on Neuron Chips and Smart Transceivers (Neuron-hosted devices). Neuron C includes network communication, I/O, and event-handling extensions to ANSI C, which make it a powerful tool for the development of LONWORKS® applications.

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.

This guide focuses on the Neuron C language used for Neuron-hosted application development, and only highlights differences for model file compilation where necessary.

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 Programmer's Guide* (078-0002-01I). This manual describes how to write 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 FX 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 lists the typographic conventions used in this manual for displaying Neuron C syntax:

Table 1. Typographic Conventions

Typeface or Symbol	Used for	Example
boldface type	keywords literal characters	**network** **{**
italic type	abstract elements	*identifier*
[square brackets]	optional fields	[*bind-info*]

Typeface or Symbol	Used for	Example
| 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

Neuron C Overview

Neuron C is a programming language based on ANSI C that is designed for Neuron Chips and Smart Transceivers. It includes network communication, input/output (I/O), and event- and interrupt-handling extensions to ANSI C, which make it a powerful tool for the development of LONWORKS applications.

Neuron C implements all 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 of 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 management tool such as the LonMaker Integration 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* and *configuration property types*. 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 (SFPTs).
- If you cannot find standard types or profiles that meet your requirements, Neuron C also provides full support for user network variable types (UNVTs), user configuration property types (UCPTs), and user functional profiles (UFPTs).

Neuron C applications run in the environment provided by the Neuron firmware. The Neuron firmware – also known as the *Neuron Chip Firmware* – implements the LonTalk® protocol and provides an *event-driven scheduling system*.

Neuron C also provides a lower-level *messaging service* integrated into the language in addition to the network variable model. 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, and when a timer expires, the Neuron firmware automatically manages the timer events and notifies the program of those events.

For Series 5000 and 6000 devices, Neuron C provides *interrupts* that allow applications to respond to asynchronous, time-sensitive, actions.

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

When using the Neuron C language to create model files for host-based device development, only the declarations for network variables, configuration properties, and functional blocks are relevant. Most other constructs, including executable code or I/O device declarations, are ignored (for example, when a model file shares source code with a Neuron-hosted Neuron C application). You can also use conditional compilation when sharing Neuron C source code between both application types.

The rest of this reference guide discusses these various aspects of Neuron C in much greater detail, accompanied by examples. See the *Neuron C Programmer's Guide* for additional information about how to use the Neuron C language.

1

Predefined Events

This chapter provides reference information on predefined events.

The predefined events described in this chapter apply only to application development for Neuron-hosted devices. However, the development tools for host-based device development support a similar API.

Introduction to Predefined Events

An *event* is a programmatic notification provided by the Neuron firmware that there has been an occurrence of something significant to the application program. For example, a network variable update has been received from the network, or an input pin has changed state.

Events are used in *when-clauses* to enable the execution of a *when-task*, using the following general syntax:

```
when( <event> ) {
   ...
}
```

Neuron C defines a number of predefined events for events that are managed by the Neuron firmware. Predefined events are represented by unique keywords, or by an event identifier with arguments, similar to a Neuron C function call, listed in **Table 2** below. Some predefined events, such as the I/O events, can be followed by a modifier that narrows the scope of the event. If the modifier is optional and is not supplied, any event of that type qualifies.

Most events can also be used as predicates in code; see *Bypass Mode* in Chapter 7 of the *Neuron C Programmer's Guide*.

Table 2. Events Listed by Functional Group

Functional Group	**Event**
System / Scheduler	**offline** **online** **reset** **timer_expires** **wink**
Input/Output	**io_changes** **io_in_ready** **io_out_ready** **io_update_occurs**
Sleep	**flush_completes**
Network Variables	**nv_update_completes** **nv_update_fails** **nv_update_occurs** **nv_update_succeeds**
Messages	**msg_arrives** **msg_completes** **msg_fails** **msg_succeeds** **resp_arrives**

Within a single program, the following predefined events, which reflect state transitions of the application processor, can appear in no more than one **when** clause:

offline
online
reset
timer_expires (unqualified)
wink

All other predefined events can be used in multiple **when** clauses. Predefined events (except for the **reset** event) can also be used in any Neuron C expression.

Event Directory

The following sections list Neuron C events alphabetically, providing relevant syntax information and a detailed description of each event.

flush_completes *Event*

The **flush_completes** event evaluates to TRUE, following a call to the **flush( )** function, when all outgoing transactions have been completed and no more incoming messages remain to be processed. For unacknowledged messages, "completed" means that the message has been transmitted by the media access control (MAC) layer. For acknowledged messages, "completed" means that the completion code has been processed. In addition, all network variable updates have completed.

See also the discussion of sleep mode in Chapter 7, *Additional Features*, of the *Neuron C Programmer's Guide*.

Syntax

flush_completes

Example

```
   ...
   flush();

when (flush_completes)
{
   sleep();
}
```

io_changes *Event*

The **io_changes** event evaluates to TRUE when the value read from the I/O object specified by *io-object-name* changes state. The state change can be one of the following three types:

- A change *to* a specified value
- A change *by* (at least) a specified amount (the absolute value)
- Any change (an unqualified change)

The reference value is the value read the last time the change event evaluated to TRUE. For the unqualified **io_changes** event, the event equates to TRUE when the current value is different from the reference value.

A task can access the input value for the I/O object through the **input_value** keyword. The **input_value** is always a **signed long**.

For the **bit**, **byte**, and **nibble** I/O objects, changes are not latched. The change must persist until the **io_changes** event is processed. The **leveldetect** input object can be used to latch changes that might not persist until the **io_changes** event can be processed.

Syntax

io_changes (*io-object-name***)** [**to** *expr* | **by** *expr*]

io-object-name	The I/O object name (see the *I/O Model Reference*). I/O objects of the following input object types can be used in an unqualified change event. The **by** and **to** options can also be used where noted. **bit** *(to)* **byte** *(by, to)* **dualslope** *(by)* **leveldetect** *(to)* **nibble** *(by, to)* **ontime** *(by)* **period** *(by, to)* **pulsecount** *(by)* **quadrature** *(by)*
to *expr*	The **to** option specifies the value of the I/O state necessary for the **io_changes** event to become TRUE. The compiler accepts an **unsigned long** value for *expr*, where *expr* is a Neuron C expression. However, each I/O object type has its own range of meaningful values.
by *expr*	The **by** option compares the current value with the reference value. The **io_changes** event becomes TRUE when the difference (absolute value) between the current value and the reference value is greater than or equal to *expr*.

The default initial reference value used for comparison purposes is zero. You can set the initial value by calling the **io_change_init( )** function. If an explicit reference value is passed to **io_change_init( )**, that value is used as the initial reference value: **io_change_init(***io-object-name*, *value***)**. If no explicit value is passed to **io_change_init( )**, the I/O object's current value is used as the initial value: **io_change_init(***io-object-name***)**.

Example 1

```
IO_0 input bit push_button;

when (io_changes(push_button) to 0)
{
   ...
```

```
}
```

Example 2

```
IO_7 input pulsecount total_ticks;

when (io_changes(total_ticks) by 100)
{
   ...
}
```

io_in_ready *Event*

The **io_in_ready** event evaluates to TRUE when a block of data is available to be read on some asynchronous I/O models. When data is available, the application then calls **io_in( )** to retrieve the data.

The **io_in_ready** event is used with the **parallel**, **sci**, and **spi** I/O models; see the *I/O Model Reference* for more information about these I/O models.

Syntax

io_in_ready (*io-object-name***)**

io-object-name The I/O object name (see the *I/O Model Reference*).

Example

```
when (io_in_ready(io_bus))
{
   io_in(io_bus, &data);
}
```

io_out_ready *Event*

The **io_out_ready** event evaluates to TRUE whenever the I/O interface is in a state where it can be written to, and the **io_out_request( )** function has been previously invoked.

The **io_out_ready** event is used with the **parallel**, **sci**, and **spi** I/O models; see the *I/O Model Reference* for more information about these I/O models.

Syntax

io_out_ready (*io-object-name***)**

io-object-name The I/O object name (see the *I/O Model Reference*).

Example

```
when (...)
{
   io_out_request(io_bus);
}
```

```
when (io_out_ready(io_bus))
{
   io_out(io_bus, &data);
}
```

io_update_occurs *Event*

The **io_update_occurs** event evaluates to TRUE when the input object specified by *io-object-name* has an updated value. The **io_update_occurs** event applies only to timer/counter input object types listed in **Table 3**.

Table 3. Timer/Counter Objects for the **io_update_occurs** Event

I/O Object	***io_update_occurs* evaluates to TRUE after:**
dualslope	The A/D conversion is complete
ontime	The edge is detected defining the end of a period
period	The edge is detected defining the end of a period
pulsecount	Every 0.8388608 seconds
quadrature	The encoder position changes

An input object may have an *updated* value that is actually the *same* as its previous value. To detect *changes* in value, use the **io_changes** event. A given I/O object cannot be included in when clauses with both **io_update_occurs** and **io_changes** events.

A task can access the updated value for the I/O object through the **input_value** keyword. The **input_value** type is always a **signed long**, but may be cast to another type as necessary.

Syntax

io_update_occurs (*io-object-name***)**

io-object-name The I/O object name (see the *I/O Model Reference*).

Example

```
#include <io_types.h>
ontime_t therm_value;   // 'ontime_t' defined in io_types.h
IO_7 input ontime io_thermistor;

when (io_update_occurs(io_thermistor))
{
   therm_value = (ontime_t)input_value;
}
```

msg_arrives *Event*

The **msg_arrives** event evaluates to TRUE once for each a message that arrives. This event can be qualified by a specific message code specified by the sender of the message. See Chapter 6, *How Devices Communicate Using Application Messages*, of the *Neuron C Programmer's Guide*, for a list of message code ranges and their associated meanings. You can reduce scheduling overhead by using an unqualified **msg_arrives** event followed by a **switch** statement on the code field of the **msg_in** object.

Syntax

msg_arrives [**(***message-code***)**]

message-code An optional integer message code. If this field is omitted, the event is TRUE for receipt of any message.

Example

```
when (msg_arrives(10))
{
   ...
}
```

msg_completes *Event*

The **msg_completes** event evaluates to TRUE when an outgoing message completes (that is, either succeeds or fails). This event can be qualified by a specific message tag.

Checking the completion event (**msg_completes**, **msg_fails**, **msg_succeeds**) is optional by message tag.

If a program checks for either the **msg_succeeds** or **msg_fails** event, it must check for *both* events. The alternative is to check only for **msg_completes**.

Syntax

msg_completes [**(***message-tag***)**]

message-tag An optional message tag. If this field is omitted, the event is TRUE for any message.

Example

```
msg_tag tag_out;

   ...
   msg_out.tag = tag_out;
   msg_send();
   ...

when (msg_completes(tag_out))
{
   ...
```

```
}
```

msg_fails *Event*

The **msg_fails** event evaluates to TRUE when a message fails to be acknowledged after all retries have been attempted. This event can be qualified by a specific message tag.

Checking the completion event (**msg_completes**, or **msg_fails** in combination with **msg_succeeds**) is optional by message tag. If a program checks for either the **msg_succeeds** or **msg_fails** event for a given message tag, it must check for *both* events for that tag. The alternative is to check only for **msg_completes**.

Syntax

msg_fails [**(***message-tag***)**]

message-tag	An optional message tag. If this field is omitted, the event is TRUE for any message.

Example

```
msg_tag tag_out;

   ...
   msg_out.tag = tag_out;
   msg_send();
   ...

when (msg_fails(tag_out))
{
   ...
}
```

msg_succeeds *Event*

The **msg_succeeds** event evaluates to TRUE when a message is successfully sent (see the *Neuron C Programmer's Guide* for the definition of success). This event can be qualified by a specific message tag.

Checking the completion event (**msg_completes**, or **msg_fails** in combination with **msg_succeeds**) is optional by message tag. If a program checks for either the **msg_succeeds** or **msg_fails** event for a given message tag, it must check for *both* events for that tag. The alternative is to check only for **msg_completes**.

Syntax

msg_succeeds [**(***message-tag***)**]

message-tag	An optional message tag. If this field is omitted, the event is TRUE for any message.

Example

```
msg_tag tag_out;
```

```
    ...
    msg_out.tag = tag_out;
    msg_send();
    ...

when (msg_succeeds(tag_out))
{
    ...
}
```

nv_update_completes *Event*

The **nv_update_completes** event evaluates to TRUE when an output network variable update completes (that is, either fails or succeeds) or a poll operation completes. Checking the completion event (**nv_update_completes**, or **nv_update_fails** in combination with **nv_update_succeeds**) is optional by network variable.

If an array name is used, then each element of the array is checked for completion. The event occurs once for each element that experiences a completion event. An individual element can be checked by using an array index. When **nv_update_completes** is TRUE for an event qualified by the name of an entire NV array, you can examine the **nv_array_index** built-in variable (type **short int**) to obtain the element's index to which the event applies.

If a network variable range is used, then the network variable at the beginning of the range must have a lower global index than the network variable at the end of the range. Each network variable in the range is checked for completion until the first such network variable with an event is found. The event occurs for each network variable in the range that experiences a completion event.

If a program checks for the **nv_update_succeeds** event, it must check for the **nv_update_fails** event as well. The alternative is to check *only* for **nv_update_completes**. A program is also permitted to check only for **nv_update_fails** as long as there is no use of **nv_update_completes** or **nv_update_succeeds** for *any* network variable.

Syntax

nv_update_completes [**(***network-var***)**]

nv_update_completes [**(***network-var1* **..** *network-var2***)**]

network-var — A network variable identifier, a network variable array identifier, or a network variable array element. A range can be specified with two network variable identifiers or network variable array elements separated with a range operator (two consecutive dots). The range is defined by the indices of the referenced network variables. If the parameter is omitted, the event is TRUE when any network variable update completes.

Example 1 – Event for a Single Network Variable

```
network output SVNT_abs_humid nvoHumidity;
   ...
   nvoHumidity = 32;    // This initiates an NV update
   ...

when (nv_update_completes(nvoHumidity))
{
   ...
}
```

Example 2 – Event for a Network Variable Array

```
network output SVNT_abs_humid nvoHumidity[4];

   ...
   nvoHumidity[1] = 32; // This initiates an NV update
   ...

when (nv_update_completes(nvoHumidity))
{
   ...
}
```

Example 3 – Event for a Range of Network Variables

```
network output SVNT_abs_humid nvoHumidity1, nvoHumidity2,
nvoHumidity3;

   ...
   nvoHumidity2 = 32;   // This initiates an NV update
   ...

when (nv_update_completes(nvoHumidity1 .. nvoHumidity3))
{
   ...
}
```

---

nv_update_fails *Event*

The **nv_update_fails** event evaluates to TRUE when an output network variable update or poll fails (see the *Neuron C Programmer's Guide* for the definition of success).

If an array name is used, then each element of the array is checked for failure. The event occurs once for each element that experiences a failure event. An individual element can be checked with use of an array index. When **nv_update_fails** is TRUE for an event qualified by the name of an entire NV

array, the **nv_array_index** built-in variable indicates the relative index of the element to which the event applies. The **nv_array_index** variable's type is a **short int**.

If a network variable range is used, then the network variable at the beginning of the range must have a lower global index than the network variable at the end of the range. Each network variable in the range is checked for failure until the first such network variable with an event is found. The event occurs for each network variable in the range that experiences a failure event.

Checking the completion event (**nv_update_completes**, or **nv_update_fails** in combination with **nv_update_succeeds**) is optional by network variable.

If a program checks for the **nv_update_succeeds** event, it must check for the **nv_update_fails** event as well. The alternative is to check only for **nv_update_completes**. A program is also permitted to check only for **nv_update_fails** as long as there is no use of **nv_update_completes** or **nv_update_succeeds** for *any* network variable.

Syntax

nv_update_fails [**(***network-var***)**]

nv_update_fails [**(***network-var1* **..** *network-var2***)**]

network-var — A network variable identifier, a network variable array identifier, or a network variable array element. A range can be specified with two network variable identifiers or network variable array elements separated with a range operator (two consecutive dots). The range is defined by the indices of the referenced network variables. If the parameter is omitted, the event is TRUE when any network variable update fails.

Example 1 – Event for a Single Network Variable

```
network output SVNT_abs_humid nvoHumidity;

   ...
   nvoHumidity = 32;
   ...

when (nv_update_fails(nvoHumidity))
{
   ...
}
```

Example 2 – Event for a Network Variable Array

```
network output SVNT_abs_humid nvoHumidity[4];

   ...
   nvoHumidity[1] = 32;
```

```
   ...

when (nv_update_fails(nvoHumidity))
{
   ...
}
```

Example 3 – Event for a Range of Network Variables

```
network output SVNT_abs_humid nvoHumidity1, nvoHumidity2,
nvoHumidity3;

   ...
   nvoHumidity2 = 32;
   ...

when (nv_update_fails(nvoHumidity1 .. nvoHumidity3))
{
   ...
}
```

nv_update_occurs *Event*

The **nv_update_occurs** event evaluates to TRUE when a value has been received for an input network variable.

If an array name is used, then each element of the array is checked to see if a value has been received. The event occurs once for each element that receives an update. An individual element can be checked with use of an array index. When **nv_update_occurs** is TRUE for an event qualified by the name of an entire NV array, the **nv_array_index** built-in variable (type **short int**) can be examined to obtain the element's index to which the event applies.

If a network variable range is used, then the network variable at the beginning of the range must have a lower global index than the network variable at the end of the range. Each network variable in the range is checked to see if a value has been received. The event occurs once for each network variable in the range that receives an update.

Syntax

nv_update_occurs [**(***network-var***)**]

nv_update_occurs [**(***network-var1* **..** *network-var2***)**]

network-var — A network variable identifier, a network variable array identifier, or a network variable array element. A range can be specified with two network variable identifiers or network variable array elements separated with a range operator (two consecutive dots). The range is defined by the indices of the referenced network variables. If the parameter is omitted, the event is TRUE for any network variable update.

Example 1 – Event for a Single Network Variable

```
network input SNVT_switch nviSwitch;

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

Example 2 – Event for a Network Variable Array

```
network input SNVT_switch nviSwitch[4];

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

Example 3 – Event for a Range of Network Variables

```
network input SNVT_switch nviSwitch1, nviSwitch2,
nviSwitch3;

when (nv_update_occurs(nviSwitch1 .. nviSwitch3))
{
   ...
}
```

---

nv_update_succeeds *Event*

The **nv_update_succeeds** event evaluates to TRUE once for each output network variable update that has been successfully sent and once for each poll that succeeds (see the *Neuron C Programmer's Guide* for the definition of success).

If an array name is used, then each element of the array is checked for success. The event occurs once for each element that experiences a success completion event. An individual element may be checked by using an array index. When **nv_update_succeeds** is TRUE for an event qualified by the name of an entire NV array, the **nv_array_index** built-in variable indicates the relative index of the element to which the event applies. The **nv_array_index** variable's type is a **short int**.

If a network variable range is used, then the network variable at the beginning of the range must have a lower global index than the network variable at the end of the range. Each network variable in the range is checked to see if a value has been received. The event occurs once for each network variable in the range that experiences a success completion event.

Checking the completion event (**nv_update_completes**, or **nv_update_fails** in combination with **nv_update_succeeds**) is optional by network variable.

If a program checks for the **nv_update_succeeds** event, it must check for the **nv_update_fails** event as well. The alternative is to check only for **nv_update_completes**. A program is also permitted to check only for **nv_update_fails** as long as there is no use of **nv_update_completes** or **nv_update_succeeds** for *any* network variable.

Syntax

nv_update_succeeds [**(***network-var***)**]

nv_update_succeeds [**(***network-var1* **..** *network-var2***)**]

network-var — A network variable identifier, a network variable array identifier, or a network variable array element. A range can be specified with two network variable identifiers or network variable array elements separated with a range operator (two consecutive dots). The range is defined by the indices of the referenced network variables. If the parameter is omitted, the event is TRUE when any network variable update succeeds.

Example 1 – Event for a Single Network Variable

```
network output SNVT_abs_humid nvoHumidity;

   ...
   nvoHumidity = 32;
   ...

when (nv_update_succeeds(nvoHumidity))
{
   ...
}
```

Example 2 – Event for a Network Variable Array

```
network output SNVT_abs_humid nvoHumidity[4];

   ...
   nvoHumidity[1] = 32;
   ...

when (nv_update_succeeds(nvoHumidity))
{
   ...
}
```

Example 3 – Event for a Range of Network Variables

```
network output SNVT_abs_humid nvoHumidity1, nvoHumidity2,
nvoHumidity3;

   ...
   nvoHumidity2 = 32;
   ...

when (nv_update_succeeds(nvoHumidity1 .. nvoHumidity3))
{
   ...
}
```

---

offline *Event*

The **offline** event evaluates to TRUE only if the device is online and an *Offline* network management message is received from a network tool, or when a program calls **go_offline( )**. The **offline** event is handled as the first priority **when** clause. It can be used in no more than one **when** clause in a program.

The offline state can be used in case of an emergency, for maintenance prior to modifying configuration properties, or in response to some other system-wide condition. After execution of this event and its task, the application program halts until the device is reset or brought back online. While it is offline, a device can respond to certain messages, but only *Reset* or *Online* messages from a network tool are processed by the application. For example, network variables on an offline device cannot be polled using a network variable poll request message but they can be polled using a *Network Variable Fetch* network management message.

If this event is checked for outside of a **when** clause, the programmer can confirm to the scheduler that the application program is ready to go offline by calling the **offline_confirm( )** function (see *Going Offline in Bypass Mode* in Chapter 7, *Additional Features*, of the *Neuron C Programmer's Guide*).

When an application goes offline, all outstanding transactions are terminated. To ensure that any outstanding transactions complete normally before the application goes offline, the application can call **flush_wait( )** in the **when(offline)** task.

Syntax

offline

Example

```
when (offline)
{
   flush_wait();
   // process shut-down command
}
```

```
when (online)
{
   // start-up again
}
```

online *Event*

The **online** event evaluates to TRUE only if the device is offline and an *Online* network management message is received from a network tool. The **online** event can be used in no more than one **when** clause in a program. The task associated with the **online** event in a **when** clause can be used to bring a device back into operation in a well-defined state.

Syntax

online

Example

```
when (offline)
{
   flush_wait();
   // process shut-down command
}

when (online)
{
   // resume operation
}
```

reset *Event*

The **reset** event evaluates to TRUE the first time this event is evaluated after a Neuron Chip or Smart Transceiver is reset. I/O object and global variable initializations are performed before processing any events. The **reset** event task is always the first when clause executed after reset of the Neuron Chip or Smart Transceiver. The **reset** event can be used in no more than one **when** clause in a program.

A typical application's reset task initializes peripheral I/O circuitry, prepares the APIs for utilities and libraries, and initializes application timers and the interrupt system. See also *Initial Value Updates for Input Network Variables* in Chapter 3 of the *Neuron C Programmer's Guide*.

The code in a reset task is limited in size. If you need more code than the compiler permits, move some or all of the code within the reset task to a function called from the reset task. The execution time for the code in a reset task must be less than 18 seconds to prevent installation errors due to time-outs in network tools. If your device requires more than 18 seconds for reset processing, use a separate and independent task to complete the reset processing. For example, you can set a global variable within a reset task that is tested within another **when** clause to create this independent task.

The **power_up( )** function can be called in a **reset** clause to determine whether the reset was due to power-up, or to some other cause such as a hardware reset, software reset, or watchdog timer reset.

Syntax

reset

Example

```
when (reset)
{
   // initialize peripheral devices, and so on
}
```

resp_arrives *Event*

The **resp_arrives** event evaluates to TRUE when a response arrives. This event can be qualified by a specific message tag.

Syntax

resp_arrives [**(***message-tag***)**]

message-tag An optional message tag. If this field is omitted, the event is TRUE for receipt of any response message.

Example

```
msg_tag tag_out;

   ...
   msg_out.tag = tag_out;
   msg_out.service = REQUEST;
   msg_send();
   ...

when (resp_arrives(tag_out))
{
   ...
}
```

timer_expires *Event*

The **timer_expires** event evaluates to TRUE when a previously declared timer object expires. If the *timer_name* option is not included, the event is an unqualified **timer_expires** event. Unlike all other predefined events, which are TRUE only once per occurrence, the unqualified **timer_expires** event remains TRUE as long as any timer object has expired. This event can be cleared only by checking for specific timer expiration events.

Syntax

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

timer-name — An optional timer object. If this field is omitted, the event is TRUE as long as any timer object has expired.

Example

```
mtimer countdown;

   ...
   countdown = 100;
   ...

when (timer_expires(countdown))
{
   ...
}
```

wink *Event*

The **wink** event evaluates to TRUE whenever a *Wink* network management message is received from a network tool. The device can be configured or unconfigured, but it must have a program running on it.

The **wink** event is unique in that it can evaluate to TRUE even though the device is unconfigured. This event facilitates installation by allowing an unconfigured device to perform an action in response to the network tool's wink request.

Each application implements an application-specific behavior in response to the *Wink* message, but all application-specific wink implementations share the same characteristic: the **wink** event should trigger a finite, harmless, visual or visual and audible physical response that allows for unambiguous identification of an individual physical device.

A typical implementation would flash one of its LEDs for 10 seconds, or show a similar response.

Note that the **wink** event and its task can execute when the device is in the unconfigured state. The application's implementation of the wink task should, therefore, not rely on application timers (**stimer**, **mtimer**), other when-tasks, or interrupts.

Syntax

wink

Example

```
when (wink)
{
   ...
   io_out(io_indicator_light, ON);
   delay(...);
```

```
    io_out(io_indicator_light, OFF);
    ...
}
```

2

Compiler Directives

This chapter provides reference information for compiler directives, also known as *pragmas*. The ANSI C language standard permits each compiler to implement a set of pragmas that control certain compiler features that are not part of the language syntax.

Many compiler directives apply to both Neuron-hosted and host-based application development, but some directives are not allowed in a model file.

Version 6 of the Neuron C compiler supports a fully-featured preprocessor, which is based on the MCPP open-source preprocessor.

Compiler Directives

ANSI C permits compiler extensions through the **#pragma** directive. These directives are implementation-specific. The ANSI standard states that a compiler can define any sort of language extensions through the use of these directives. Unknown directives can be ignored or discarded. The Neuron C Compiler issues warning messages for unrecognized directives.

In the Neuron C Compiler, pragmas can be used to set certain Neuron firmware system resources and device parameters such as buffer counts and sizes and receive transaction counts. See Chapter 8, *Memory Management,* of the *Neuron C Programmer's Guide* for a detailed description of the compiler directives for buffer allocation.

Other pragmas control code generation options, debugging options, error reporting options, and other miscellaneous features. Additional **#pragma** directives can be used to control other Neuron firmware-specific parameters. These directives can appear anywhere in the source file.

Pragma Directives

The following pragma directives are defined in Neuron C Version 2.3:

#pragma addresses *num*

Sets the number of address table entries to *num*. Valid values for *num* vary, depending on the chip you are using. For the Series 3000 and 5000 chips, values are 0 to 15. For the Series 6000 chip, valid values are 0 to 254.

The Neuron C Compiler version 6 or later automatically compites a recommended number of addresses based on inspection of your application's interface. You can inspect the resulting address table allocation through the Neuron Linker's *map* file.

You can use this pragma override the compiler's allocation algorithm, for example to trade EEPROM space for address table entries (see Chapter 8, *Memory Management,* of the *Neuron C Programmer's Guide*).

A minimum address table size of 15 records is recommended for all applications.

This directive is not supported in model files.

See *pragma num_addr_table_entries* for the legacy form of this directive.

#pragma aliases *num*

Controls the number of alias table entries allocated by the compiler. This number must be chosen during compilation; it cannot be altered at runtime. For Series 3100 devices with system firmware version 15 or earlier, valid values for *num* are 0 to 62. For Series 3100 devices with system firmware version 16 or later, and Series 5000 or 6000 devices, built with the NodeBuilder FX Development Tool, valid values for *num* are 0 to 127.

The Neuron C Compiler version 6 or later automatically computes a recommended number of aliases based on inspection of your application's interface. You can inspect the results of this in the Neuron Linker's *map* file, or you can use this directive to override the compiler's recommendations.

Earlier versions of the Neuron C Compiler *require* that you specify the number of aliases with this directive.

This directive is not supported in model files.

See *pragma num_alias_table_entries* for the legacy form of this directive.

#pragma all_bufs_offchip

This pragma is only used with the MIP/DPS. It causes the compiler to instruct the firmware and the linker to place all application and network buffers in off-chip RAM. This pragma is useful only on the Neuron 3150® Chip or 3150 Smart Transceiver, because these are the only parts that support off-chip memory. See the *Microprocessor Interface Program (MIP) User's Guide* for more information.

This directive is not supported in model files.

#pragma allow_duplicate_events

This directive causes the compiler to issue an **NCC#176** duplicate event message as a warning instead of as an error. The compiler normally treats a duplicate event as a programming error. However, there are rare situations where you want to test for a certain important event more than once within the scheduler loop by having multiple, duplicated **when** clauses at different points in the list of tasks run by the scheduler. This duplication can prevent such an event from having to wait too long to be serviced. For more information, see the discussion on *The Scheduler* in Chapter 7, *Additional Features*, in the *Neuron C Programmer's Guide.*

This directive is not supported in model files.

#pragma app_buf_in_count *count* [, *modifier*]

See *Allocating Buffers* in Chapter 8, *Memory Management,* of the *Neuron C Programmer's Guide* for detailed information on this pragma and its use.

This directive is not supported in model files.

#pragma app_buf_in_size *size* [*, modifier*]

See *Allocating Buffers* in Chapter 8, *Memory Management,* of the *Neuron C Programmer's Guide* for detailed information on this pragma and its use.

This directive is not supported in model files.

#pragma app_buf_out_count *count* [*, modifier*]

See *Allocating Buffers* in Chapter 8, *Memory Management,* of the *Neuron C Programmer's Guide* for detailed information on this pragma and its use.

This directive is not supported in model files.

#pragma app_buf_out_priority_count *count* [*, modifier*]

See *Allocating Buffers* in Chapter 8, *Memory Management,* of the *Neuron C Programmer's Guide* for detailed information on this pragma and its use.

This directive is not supported in model files.

#pragma app_buf_out_size *size* [*, modifier*]

See *Allocating Buffers* in Chapter 8, *Memory Management,* of the *Neuron C Programmer's Guide* for detailed information on this pragma and its use.

This directive is not supported in model files.

#pragma codegen *option*

This pragma allows control of certain features in the compiler's code generator. Application timing and code size could be affected by use of these directives. The valid *option*s that can be specified are:

cp_family_space_optimization
create_cp_value_files_uninit
expand_stmts_off
expand_stmts_on
no_cp_template_compression
no16bitstkfn
nofastcompare
noptropt
noshiftopt
nosiofar
optimization_off
optimization_on
put_cp_template_file_in_data_memory
put_cp_template_file_offchip
put_cp_value_files_offchip
put_read_only_cps_in_data_memory

pxopt
use_i2c_version_1

Some of these options are provided for compatibility with prior releases of the Neuron C Compiler. The **no16bitstkfn**, **nofastcompare**, **noptropt**, and **noshiftopt** options disable various optimizations in the compiler. The **nosiofar** option is provided for Neuron firmware versions that include the serial I/O functions in the near system-call area. In addition, the **cp_family_space_optimization**, **no_cp_template_compression**,

optimization_off, and **optimization_on** options perform code optimizations; use the **#pragma optimization** directive instead.

The **create_cp_value_files_uninit** option is used to prevent the compiler from generating configuration value files that contain initial values. Instead, the value files are generated with no initial value, such that the Neuron loader does not load anything into the block of memory; instead, the contents prior to load are unaltered. This can be helpful if an application image needs to be reloaded, but its configuration data is to remain unchanged.

The **expand_stmts_off** and **expand_stmts_on** options control statement expansion. Normally, statement expansion is off. To permit the network debug kernel to set a breakpoint at any statement whose code is stored in modifiable memory for a Series 3100 chip, the statement's code must be at least two bytes in length. Due to optimization, some statements can be accomplished in less than two bytes of generated Neuron machine code. Activating statement expansion tells the code generator to ensure that each statement contains at least two bytes of code by inserting a no-operation (NOP) instruction if necessary.

Applications targeted for a Series 5000 or 6000 chip do not need to enable statement expansion. These devices support a one-byte breakpoint instruction, which is automatically used when debugging. Thus, statement expansion and the related increase in application size are not required for these devices.

The automatic configuration property merging feature in NodeBuilder 3.1 (and later) might change the device interface for a device that was previously built with the NodeBuilder 3 tool. You can specify **#pragma codegen no_cp_template_compression** in your program to disable the automatic merging and compaction of the configuration property template file. Use of this directive could cause your program to consume more of the device's memory, and is intended only to provide compatibility with the NodeBuilder 3.0 Neuron C compiler. You cannot use both the **no_cp_template_compression** option and the **cp_family_space_optimization** option in the same application program. This feature is independent of the **#pragma optimization** directive.

The **noptropt** option can be desirable when debugging a program, because the debugger does not have knowledge of whether the compiler has eliminated redundant loads of a pointer between statement boundaries. If a breakpoint is set in such circumstances, modification of the pointer variable from the debugger would not modify the loaded pointer register which the compiler can then use in subsequent statements. Use of this pragma avoids this problem, but could also cause a substantial performance or size degradation in the generated code. This **codegen** option should not be used except while debugging.

The **put_cp_template_file_in_data_memory** option is used to direct the compiler to create the configuration template file in a device's data memory instead of code memory. The purpose of doing this would be to permit write access to the template file, or to permit more control over memory organization to accommodate special device memory requirements.

In certain situations when linking a program for a Neuron 3150 Chip or a 3150 Smart Transceiver, it might be necessary to force the configuration property template file into offchip memory rather than letting the linker

choose between offchip or onchip memory. Specify the **put_cp_template_file_offchip** option to force the template file into offchip memory.

In certain situations when linking a program for a Neuron 3150 Chip or a 3150 Smart Transceiver, it might be necessary to force the configuration property value files into offchip memory rather than letting the linker choose between offchip or onchip memory. Specify the **put_cp_value_files_offchip** option to force the value files into offchip memory.

The **put_read_only_cps_in_data_memory** option is used to direct the compiler to create the configuration read-only value file in a device's data memory instead of code memory. The purpose of doing this would be to permit write access to the read-only configuration properties (CPs), or to permit more control over memory organization to accommodate special device memory requirements.

The **pxopt** option is provided for a new form of pointer register optimization. When used in combination with optimization all it may be useful to compile twice, once with and once without **pxopt**. Then compare the memory footprint stated in the link map summary.

The **use_i2c_version_1** option is provided for compatibility with releases of the Neuron C Compiler prior to the introduction of Neuron C Version 2.1. The option disables use of a revised **i2c** I/O object in the compiler. Although unlikely, it is possible that a program using the **i2c** I/O object which compiled and linked with an older release of the Neuron C Compiler would not fit if compiled under the Neuron C Version 2.1 compiler or later, because the version 2 I/O object is a bit larger than the previous implementation, due to its greatly increased flexibility and support of additional I/O pins as compared to the version 1 implementation. See the description of the **i2c** I/O model in the *I/O Model Reference*.

Only the **cp_family_space_optimization** and **no_cp_template_compression** options are supported in model files.

#pragma deadlock_is_finite

This pragma allows the system semaphore to expire. The semaphore is used for sharing data between an interrupt task and the main application. This directive cancels the specification of the **#pragma deadlock_is_infinite** directive.

You can specify this directive as often as necessary to allow debugging of code within a lock construct (the Neuron C **__lock{ }** keyword). You cannot debug interrupt-related code when interrupt tasks run on the interrupt (ISR) processor.

This directive is not supported in model files.

#pragma deadlock_is_infinite

This pragma prevents the system semaphore from expiring. The semaphore is used for sharing data between an interrupt task and the main application. This directive should only be used for NodeBuilder debug targets; using it with release targets causes a compiler warning (**NCC#607**) and can allow deadlocks not to cause a watchdog timeout (an infinite deadlock).

You can specify this directive as often as necessary to allow debugging of code within a lock construct (the Neuron C **__lock{ }** keyword). You cannot debug interrupt-related code when interrupt tasks run on the interrupt (ISR) processor.

This directive is not supported in model files. **#pragma debug** *option*

This pragma allows selection of various network debugger features. A program using network debugger features can only be used with version 6 and later versions of the Neuron firmware.

The valid options are shown in the list below. This pragma can be used multiple times to combine options, but not all options can be combined.

network_kernel
no_event_notify
no_func_exec
no_node_recovery
no_reset_event
node_recovery_only

The debugger network kernel must be included to use the device with the network debugger supplied with the NodeBuilder Development Tool or the LCA Field Compiler API. The network kernel consists of several independent but interacting modules, all of which are included in the program image by default. To reduce the size of the network debug kernel included in a program, one or more of the following options can be specified in additional **#pragma debug** directives. See the *NodeBuilder User's Guide* and the NodeBuilder Online Help for more information.

Use of the **no_event_notify** option excludes the event notification module.

Use of the **no_func_exec** option excludes the remote function execution module.

Use of the **no_node_recovery** option turns off the device's reset recovery delay that the compiler automatically includes when the network debugging kernel is included.

Use of the **no_reset_event** option turns off the reset event notification feature. This feature is not necessary if the **no_event_notify** option is used to exclude all event notification, since the reset event notification is part of the event notification feature.

Use of the **node_recovery_only** option instructs the compiler to include the node recovery feature only, without the network debug kernel.

This directive is not supported in model files.

#pragma dhcp (enabled | disabled)

Enables or disables use of the dynamic host configuration protocol (DHCP), where supported. Only Series 6000 chips support LonTalk/IP which is required for DHCP.

Compilation targets with support for DHCP have DHCP enabled by default. You can use this directive to disable DHCP. When you use this directive to explicitly enable DHCP, a compile-time warning will be reported when the compilation target does not support DHCP.

#pragma disable_mult_module_init

Requests the compiler to 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 this directive, is slightly more efficient in memory usage, but might not permit a successful link for an application on a Neuron 3150 Chip or 3150 Smart Transceiver. This pragma should only be used when trying to fit a program into a Neuron 3120xx Chip or 3120 Smart Transceiver. See the discussion on *What to Try When a Program Does Not Fit on a Neuron Chip* in Chapter 8, *Memory Management,* of the *Neuron C Programmer's Guide*.

This directive cannot be used with the **debug** directive.

This directive is not supported in model files

. **#pragma disable_servpin_pullup**

Disables the internal pullup resistor on the service pin. This pullup resistor is normally enabled. The pragma takes effect during I/O initialization.

This directive has no effect when used for a Series 5000 or 6000 chip.

This directive is not supported in model files.

#pragma disable_snvt_si

Disables generation of the self-identification (SI) data. The SI data is generated by default, but can be disabled using this pragma to reclaim program memory when the feature is not needed. See *Standard Network Variable Types (SNVTs)* in Chapter 3, *How Devices Communicate Using Network Variables,* of the *Neuron C Programmer's Guide*.

The compiler does not issue error message **NCC#15** if you specify this directive multiple times in the application.

This directive is not supported in model files.

#pragma disable_warning *number*

Controls the compiler's printing of specific warning and hint messages Warning messages are less severe than errors, yet could indicate a problem in a program, or a place where code could be improved. To disable all warning messages, specify an asterisk (*****) for the *number*.

See the **enable_warning directive** to enable disabled warnings.

The **disable_warning** directive supercedes the **warnings_off** directive.

#pragma domains *num*

Sets the number of domain table entries to *num*. Valid values for *num* are 1 or 2. The default number of domain table entries is 2. You can use this pragma to trade EEPROM space for a domain table entry (see Chapter 8, *Memory Management,* of the *Neuron C Programmer's Guide*).

This directive is not supported in model files.

Two domains are recommended for all applications.

#pragma eeprom_locked

This pragma provides a mechanism whereby an application can lock its checksummed EEPROM. Checksummed EEPROM includes the application

and network images, but not application EEPROM variables. Setting the flag improves reliability because attempts to write EEPROM as a result of wild jumps fail. EEPROM variables are not protected. See the discussion of the **set_eeprom_lock( )** function in Chapter 3, *Functions*, on page 47, for more information.

There are drawbacks to using the EEPROM lock mechanism. A device with this pragma (or one using the **set_eeprom_lock( )** function) requires that the device be taken offline before checksummed EEPROM can be modified. So, if the device is configured by a network tool that does not take the device offline prior to changes, the tool does not change the configuration.

This directive is not supported in model files.

#pragma enable_io_pullups

Enables the internal pullup resistors on pins IO4 through IO7 for Series 3100 devices. The pragma takes effect during I/O initialization. These pullup resistors are normally disabled. Use of this pragma can eliminate external hardware components when pullup resistors are required. The PL 3120-E4 Smart Transceiver and the PL 3150 Smart Transceiver both have an extra I/O pin, IO11. On these models of Smart Transceiver, this directive also enables a pullup resistor for the IO11 pin.

This directive has no effect for a Series 5000 or 6000 chip because those chips do not have on-chip programmable pull-up resistors. The linker issues the **NLD#507** warning message if you include this directive for a Series 5000 or 6000 device.

This directive is not supported in model files.

#pragma enable_multiple_baud

Must be used in a program with multiple serial I/O devices that have differing bit rates. If needed, this pragma must appear prior to the use of any I/O function (for example, **io_in( )** and **io_out( )**).

This directive is not supported in model files.

#pragma enable_sd_nv_names

Causes the compiler to include the network variable names in the self-documentation (SD) information when self-identification (SI) data is generated. See *Standard Network Variable Types (SNVTs)* in Chapter 3, *How Devices Communicate Using Network Variables,* of the *Neuron C Programmer's Guide*.

#pragma enable_warning *number*

Controls the compiler's printing of specific warning and hint messages Warning messages are less severe than errors, yet could indicate a problem in a program, or a place where code could be improved. To enable all warning messages, specify an asterisk (*****) for the *number*.

See the **disable_warning** directive for the reverse operation.

The **enable_warning** directive supercedes the **warnings_on** directive.

#pragma enhanced_mode (enabled | disabled)

Enables or disables use of enhanced mode (Series 6000 chips only). Enhanced mode is a non-backward compatible extension of the LonWorks protocol that

improves reliability through the use of an expanded transaction ID space. Some devices, such as FT-6050 and Neuron 6050 devices can be configured to use either compatibility or enhanced mode. Other devices, such as Series 5000 devices, only support compatibility mode, while the FT-6010 only supports enhanced mode. Enhanced mode is always disabled by default, thereby enabling compatibility mode.

Enabling enhanced mode on targets without support for enhanced mode yields a compile time warning.

#pragma explicit_addressing_off

#pragma explicit_addressing_on

These pragmas are only used with the Microprocessor Interface Program (MIP) or for creating ShortStack Micro Server applications. See the *LonWorks Microprocessor Interface Program (MIP) User's Guide* or the *ShortStack User's Guide* for more information.

These directives are not supported in model files.

#pragma fyi_off

#pragma fyi_on

Controls the compiler's printing of informational messages. Informational messages are less severe than warnings, yet could indicate a problem in a program, or a place where code could be improved. Informational messages are off by default at the start of compilation. These pragmas can be intermixed multiple times throughout a program to turn informational message printing on and off as desired.

#pragma hidden

This pragma is for use only in the **<echelon.h>** standard include file.

.**#pragma idempotent_duplicate_off**

#pragma idempotent_duplicate_on

These pragmas control the idempotent request retry bit in the application buffer. This feature only applies to MIP or ShortStack Micro Server applications. One of these pragmas is required when compiling, if the **#pragma micro_interface** directive also is used. See the *LonWorks Microprocessor Interface Program (MIP) User's Guide* or the *ShortStack User's Guide* for more information.

These directives are not supported in model files.

#pragma ignore_notused *symbol*

Requests that compiler ignore the symbol-not-referenced flag for the named symbol. The compiler normally prints warning messages for any variables, functions, I/O objects, and so on, that are declared but never used in a program. This pragma may be used one or more times to suppress the warning on a symbol by symbol basis.

The pragma should appear after the variable declaration. A good coding convention is to place the pragma on the line immediately following the variable's declaration. For automatic scope variables, the pragma must appear no later than the line preceding the closing brace character ('}') that

terminates the scope containing the variable. There is no terminating brace for any variable declared at file scope.

#pragma include_assembly_file *filename*

This pragma can be used with the Neuron C Version 2 compiler to cause the compiler to open *filename* and copy its contents to the assembly output file. The compiler copies the contents such that the assembly code does not interfere with code being generated by the compiler. See the *Neuron Assembly Language Reference* for more information about including assembly language routines with Neuron C programs.

This directive is not supported in model files.

#pragma library "*library***"**

This pragma allows you to specify a library file (a file with a .lib extension) with which the application is to be linked. You can use this directive as an alternative to adding an explicit library reference to a NodeBuilder project. The directive is recommended for new development, because it promotes self-contained, modular development, and self-documenting code.

You can specify a library with either an absolute path or a relative path (relative to the location of the NodeBuilder device template file [*.nbdt file]), for example:

```
#pragma library "cenelec.lib"
#pragma library "..\myLibraries\myToolkit.lib"
```

You can also include macros in the path names. **Table 4** lists system-defined macros, but you can also define custom macros. A custom macro is expanded to the value of an operating system environment variable of the same name as the macro. For example, if you define an environment variable "MYPROJ" as "$LONWORKS$\myProjects", then the following directive defines the myToolkit.lib file within the "c:\LonWorks\myProjects" directory:

```
#pragma library "$MYPROJ$\myToolkit.lib"
```

If the environment variable "MYPROJ" is not defined, the linker replaces the unknown macro with an empty string; it does not issue an error message for the undefined macro.

The system macros listed in **Table 4** are always automatically pre-defined, and do not need defining in the system environment.

Table 4. System Macros for **#pragma library** Directive

Macro	Expansion
$LONWORKS$	The local LonWorks directory, generally **c:\LonWorks**. Note that there is no trailing backslash.
IMG	IMG enables a two-step search. First, IMG expands to the same as SYM (detailed below). When unsuccessful, IMG expands to $LONWORKS$\Images

Macro	Expansion
STD	$LONWORKS$\NeuronC\Libraries
SYM	The location of the selected firmware image and .SYM file.

For example, the NodeBuilder Code Wizard automatically adds the following directive to the base .nc file (the file that corresponds to the device template .nbdt file):

```
#ifdef  _NEURONC
#   ifndef _MODEL_FILE
#       ifndef USER_DEFINED_CODEWIZARD_LIB
#           pragma library "$IMG$\CodeWizard-3.lib"
#       endif // USER_DEFINED_CODEWIZARD_LIB
#    endif // _MODEL_FILE
#endif  // _NEURONC
```

You can nest macros up to five times, after which the recursion stops.

You can specify the **#pragma library** directive up to 19 times. That is, the compiler can process a total of 19 input libraries during one compilation, counting all included files and the original source file.

This directive is not supported in model files.

#pragma locate

This pragma is designed for use in exceptional cases only. It is used as a solution when rules and controls for placement of Neuron C functions or variables are insufficient. The pragma directive can be uses to assign a named segment type, or an origin or origin modifier within the segment, or both, to a named function or variable. The directive uses the syntax:

```
#pragma locate <name> { (seg|segment) <segment>} { (org|at)
<orgvalue>}
```

`<name>` refers to a variable or function. The variable or function must not yet be implemented at the location of the corresponding locate directive, but a function prototype may already be present, and a 'forward declaration' for a variable may be in place. The forward variable declaration is an extern declaration for a locally implemented variable.

The segment instruction begins with a keyword `seg` or `segment`. Both keywords have the same effect. The `<segname>` with follows is passed to the Neuron Assembler verbatim when code for the named item is generated. Where the compiler would normally choose the segment type according to built-in rules or controls, in this case it will generate a Neuron Assembly `SEG <segname>` instruction for this item.

This segment instruction now overrules all other preferences expressed in keywords, modifiers, directives or run-time options, without warning. It will not change the generated code other than the segment a certain item is allocated to. For example, a variable which normally would reside in the near RAM area can be forced into a far RAM segment with a segment instruction. The compiler still generates code assuming a near RAM location.

You must use the `far` modifier in the variable declaration to enforce the RAM placement.

The origin instruction begins with a keywork at `or org`. The orgvalue expression which follows can be onchip, offchip, a valid C integer constant, or any other arbitrary expression. The words `onchip` and `offchip` are recognized and automatically translated into the Neuron Assembler's `onchipmem` and `offchipmem` modifiers, respectively. Any valid C integer constant (using *0x* prefix for hexadecimal numbers) is automatically translated into the Neuron Assembler's number format. The result of the translations, or the verbatim `orgvalue` expression provided, is used as the modifier in the generated Neuron Assembly ORG instructions.

As is true of the segment instruction, the origin instruction overrules all other preferences expressed through keywords, modifiers, directives, or run-time options, without warning.

Neither segment nor origin instruction can be repeated within the same locate directive. The directive, as noted above, provides limited error checking and there is no warning if conflicting or incompatible preferences have been expressed elsewhere.

The following examples illustrate this pragma.

```
  //Fixed address for "myVar:"
#pragma locate myVar org 0xB123


  //Force the cp_modifiable_value_file onchip:
#pragma locate cp_modifable_value_file org onchip


  //Force function f into the EECODE segment:
#pragma locate f segment eecode
```

#pragma micro_interface

This pragma is only used with the Microprocessor Interface Program (MIP) or with ShortStack Micro Server applications. See the *LonWorks Microprocessor Interface Program (MIP) User's Guide* or the *ShortStack User's Guide* for more information.

#pragma names_compatible

This pragma is useful in Neuron C Version 2 (and later) to force the compiler to treat names starting with SCPT*, UNVT*, UCPT*, SFPT*, and UFPT* as normal variable names instead of as special symbols to be resolved through resource files. This list does *not* include names starting with SNVT*. Disabling the special behavior permits the compiler to accept programs written using Neuron C Version 1 that declare such names in the program.

This directive is not supported in model files.

#pragma net_buf_in_count *count* [*, modifier*]

See *Allocating Buffers* in Chapter 8, *Memory Management,* of the *Neuron C Programmer's Guide* for more detailed information on this pragma and its use.

This directive is not supported in model files.

#pragma net_buf_in_size *size* [*, modifier*]

See *Allocating Buffers* in Chapter 8, *Memory Management,* of the *Neuron C Programmer's Guide* for detailed information on this pragma and its use.

This directive is not supported in model files.

#pragma net_buf_out_count *count* [*, modifier*]

See *Allocating Buffers* in Chapter 8, *Memory Management,* of the *Neuron C Programmer's Guide* for detailed information on this pragma and its use.

This directive is not supported in model files.

#pragma net_buf_out_priority_count *count* [*, modifier*]

See *Allocating Buffers* in Chapter 8, *Memory Management,* of the *Neuron C Programmer's Guide* for detailed information on this pragma and its use.

This directive is not supported in model files.

#pragma net_buf_out_size *size* [*, modifier*]

See *Allocating Buffers* in Chapter 8, *Memory Management,* of the *Neuron C Programmer's Guide* for detailed information on this pragma and its use.

#pragma netvar_processing_off

#pragma netvar_processing_on

This pragma is only used with the Microprocessor Interface Program (MIP). See the *LonWorks Microprocessor Interface Program (MIP) User's Guide* for more information.

These directives are not supported in model files.

#pragma no_hidden

This pragma is for use only in the **<echelon.h>** standard include file.

#pragma num_addr_table_entries *num*

Legacy form of *pragma addresses,* see there for details. The legacy form is supported and has the same effect than the shorter *pragma addresses* directive. The shorter form is recommended for all new development.

#pragma num_alias_table_entries *num*

Legacy form of *pragma aliases,* see there for details. The legacy form is supported and has the same effect than the shorter *pragma aliases* directive. The shorter form is recommended for all new development.

#pragma num_domain_entries *num*

Legacy form of *pragma domains,* see there for details. The legacy form is supported and has the same effect than the shorter *pragma domains* directive. The shorter form is recommended for all new development.

#pragma one_domain

Sets the number of domain table entries to 1. This pragma is provided for legacy application support and should no longer be used. New applications should use the **domains** pragma instead. The default number of domain table entries is 2.

This directive is not supported in model files.

#pragma optimization *level*

This pragma allows you to specify a code optimization level for optimal use of device memory. Supported levels are 0 (no optimization) to 5 (maximum optimization).

When you specify no optimization, the Neuron C compiler attempts to create fairly economical code. However, some improvements are generally possible, and sometimes necessary. In addition to specifying additional code optimization using this directive, consider manual optimization, as described in the *What to Try When a Program Does Not Fit on a Neuron Chip* section of Chapter 8 in the *Neuron C Programmer's Guide*.

With optimization enabled, the Neuron C compiler performs several types of code optimization. For example, it identifies common sub-expressions and moves them into sub-routines when economical, thus reducing the memory footprint of the generated code.

Table 5 lists the levels of optimization. Levels 2, 3, 4, and 5 are ignored if you disable optimation within the NodeBuilder FX Development Tool (select the **Disable optimizer** checkbox from the Compiler tab of the NodeBuilder Device Template Target Properties dialog). Levels 0 and 1 are ignored if you enable optimation within the NodeBuilder FX Development Tool (clear the **Disable optimizer** checkbox).

Table 5. Optimization Levels for the **#pragma optimization** Directive

Level	Optimization Performed	Notes
0	No optimization CP templates are not compressed	
1	No optimization	Default for debug targets
2	Minimal optimization CP templates are not compressed	
3	General optimization	Default for release targets
4	General optimization, plus optimization of space for **cp_family** definitions	
5	Maximum optimization	

You can use the following keywords instead of the numeric level indicators:

- *none* for level 0

- *debug* for level 1
- *standard* for level 3
- *all* for level 5

The keyword level indicators are generally preferred over their numeric counterparts because they are self-documenting.

As part of optimization levels 3 and 4, the Neuron C compiler can attempt to compact the configuration property template file by merging adjacent family members that are scalars into elements of an array. Any CP family members that are adjacent in the template file and value file, and that have identical properties, except for the item index to which they apply, are merged. Using optional *configuration property re-ordering and merging* can achieve additional compaction beyond what is normally provided by automatic merging of whatever CP family members happen to be adjacent in the files. With this feature enabled, the Neuron C compiler optimizes the layout of CP family members in the value and template files to make merging more likely.

Important: Configuration property re-ordering and merging can reduce the memory required for the template file, but could also result in slower access to the application's configuration properties by network tools. This could potentially cause a significant increase in the time required to commission your device, especially on low-bandwidth channel types such as power line channels. You should typically only use configuration property re-ordering and merging if you must conserve memory. If you use configuration property re-ordering and merging, be sure to test the effect on the time required to commission and configure your device.

The default for debug targets is no optimization because the NodeBuilder debugger allows you to place breakpoints in the source code, but after optimization, the compiler might have collapsed two or more statements together. In this case, the debugger might attempt to place a breakpoint in a statement that does not exist in the optimized code. Thus, debugging compiler-optimized code is not supported.

The **#pragma optimization** directive replaces the following directives:

#pragma codegen cp_family_space_optimization
#pragma codegen optimization_on
#pragma codegen optimization_off
#pragma codegen no_cp_template_compression

While all of these directives continue to work, the compiler issues the **NCC#589** warning message if you use these deprecated directives. If your application uses any of these directives with the **#pragma optimization** directive, the compiler issues the **NCC#588** warning message.

If you specify code optimization in NodeBuilder (or from the command line for the NCC tool), and you specify the **#pragma optimization** directive, the compiler issues the **NCC#590** warning message.

This directive is not supported in model files.

#pragma ram_test_off

For Series 3100 chips, disables the off-chip RAM buffer space test to speed up initialization. Normally the first thing the Neuron firmware does when it comes up after a reset or power-up is to verify basic functions such as CPUs,

RAM, and timer/counters. This can consume large amounts of time, particularly at slower clock speeds. By turning off RAM buffer testing, you can trade off some reset time for maintainability. All RAM static variables are nevertheless initialized to zero.

This directive has no effect for a Series 5000 or 6000 chip. System RAM (from 0xE800 to 0xEFFF) is always tested during reset, and the extended memory area (extended RAM or non-volatile memory, from 0x4000 to 0xE7FF) is not tested during reset.

This directive is not supported in model files.

#pragma read_write_protect

Allows a device's program to be read and write protected to prevent copying or alteration over the network. This feature provides protection of a manufacturer's confidential algorithms. A device cannot be reloaded after it is protected. The write protection feature is included to disallow Trojan horse intrusions. The protection must be specifically enabled in the Neuron C source program. After a device is loaded with an application containing this pragma, the application program can never be reloaded on a Neuron 3120xx Chip or 3120 Smart Transceiver. It is possible, however, to erase and reload a Neuron 3150 Chip or 3150 Smart Transceiver, with the use of the EEPROM blanking programs. Likewise, it is possible to erase and reload the external EEPROM of a Series 5000 or 6000 device. For more information on the use of the EEPROM blanking programs, see the Smart Transceivers data books.

This directive is not supported in model files.

#pragma receive_trans_count *num*

Sets the number of receive transaction blocks to *num*. Valid values for *num* are 1 to 16. See *Allocating Buffers* in Chapter 8, *Memory Management,* of the *Neuron C Programmer's Guide* for more detailed information on this pragma and its use.

This directive is not supported in model files.

#pragma relaxed_casting_off

#pragma relaxed_casting_on

These pragmas control whether the compiler treats a cast that removes the **const** attribute as an error or as a warning. The cast can either be explicit or implicit (as in an automatic conversion due to assignment or function parameter passing). Normally, the compiler considers any conversion that removes the **const** attribute to be an error. Turning on the relaxed casting feature causes the compiler to treat this condition as a warning instead. These pragmas can be intermixed throughout a program to enable and disable the relaxed casting mode as desired. See the example for *Explicit Propagation of Network Variables* in Chapter 3, *How Devices Communicate Using Network Variables,* of the *Neuron C Programmer's Guide*.

#pragma resident (enabled | disabled)

This directive switches the default compilation to resident mode (#*pragma resident enabled*) or transient mode (#*pragma resident disabled).* Without this directive (or the *--resident* compiler option), plain functions default to transient function when this feature is supported, and default to resident functions otherwise.

See *Transient and Resident Functions* in Chapter 3, *Functions*, for more details.

#pragma run_unconfigured

This pragma causes the application to run whenever it is online, even if the device is in the unconfigured state. Without this directive, the application runs only when the device is both online and configured. You can use this directive to have an application perform some form of local control prior to or independently of being installed in a network.

Applications that use this pragma and run on firmware versions prior to version 12 should not attempt to send messages when hard-offline. The hard-offline state can be detected by calling **retrieve_status( )** and checking the **status_node_state** field for the value **CNFG_OFFLINE**. The reason for this restriction is that the hard-offline state is used by network tools during configuration modification. Were one to send messages in this state, the message might be sent using invalid configuration and thus potentially go to the wrong location. Note that an application is typically taken soft-offline during modification so the device is only subject to these concerns if it is power-cycled while the modification is in progress. Applications that do not use this pragma do not ever run when hard-offline and thus are not vulnerable to this condition.

This directive is not supported in model files.

#pragma scheduler_reset

Causes the scheduler to be reset within the nonpriority **when** clause execution cycle, after each event is processed (see Chapter 7, *Additional Features,* of the *Neuron C Programmer's Guide* for more information on the Neuron scheduler).

This directive is not supported in model files.

#pragma set_guidelines_version *string*

The Neuron C version 2.1 (and later) compiler generates LONMARK information in the device's XIF file and in the device's SIDATA (stored in device program memory). By default, the compiler uses **"3.4"** as the string identifying the LONMARK guidelines version that the device conforms to. To override this default, specify the overriding value in a string constant following the pragma name, as shown. For example, a program could specify **#pragma set_guidelines_version "3.2"** to indicate that the device conforms to the 3.2 guidelines. This directive is useful for backward compatibility with older versions of the Neuron C compiler.

Note this directive can be used to state compatibility with a guidelines version that is not actually supported by the compiler. Future versions of the guidelines that require a different syntax for SI/SD data are likely to require an update to the compiler. This directive only has the effect described above, and does not change the syntax of SD strings generated.

The **set_guidelines_version** directive is typically used to specify a version string in the *major.minor* form (for example, "3.4"). The compiler issues a **NCC#604** warning message if the application-specific version string does not match that format, but permits the string.

Using this directive can prevent certification of the generated device.

#pragma set_id_string *"ssssssss"*

Provides a mechanism for setting the device's 8-byte program ID. This directive is provided for legacy application support and should no longer be used. The program ID should be set in the NodeBuilder device template instead, and should not be set to a text string except for network interface devices (for example, devices using the MIP). If this pragma is present, the value must be the same as the program ID set by the NodeBuilder tool.

This pragma initializes the 8-byte program ID located in the application image. The program ID is sent as part of the service pin message (transmitted when the service pin on a device is activated) and also in the response for the *Query ID* network management message. The program ID can be set to any C string constant, 8 characters or less.

This pragma can only be used to set a non-standard text program ID where the first byte must be less than 0x80. To set a standard program ID, use the **#pragma set_std_prog_id** directive, documented below. If this pragma is used, the **#pragma set_std_prog_id** directive cannot be used. Neither pragma is required or recommended.

#pragma set_netvar_count *nn*

This pragma is only used with the Microprocessor Interface Program (MIP) or ShortStack Micro Server applications. See the *LonWorks Microprocessor Interface Program (MIP) User's Guide* or the *ShortStack User's Guide* for more information.

This directive is not supported in model files.

#pragma set_node_sd_string *C-string-const*

Specifies and controls the generation of a comment string in the self-documentation (SD) string in a device's application image. Most devices have an SD string. The first part of this string documents the functional blocks on the device, and is automatically generated by the Neuron C Version 2 compiler. This first part can be followed by a comment string that documents the purpose of the device. This comment string defaults to a **NULL** string and can have a maximum of 1023 bytes (minus the length of the first part of the SD string generated by the Neuron C compiler), including the zero termination character. This pragma explicitly sets the comment string. Concatenated string constants are *not* allowed. This pragma can only appear once in the source program.

#pragma set_std_prog_id *hh*:*hh*:*hh*:*hh*:*hh*:*hh*:*hh*:*hh*

Provides a mechanism for setting the device's 8-byte program ID. This directive is provided for legacy application support and should not be used for new programs. The program ID should be set in the NodeBuilder device template instead. If this pragma is present, the value must agree with the program ID set by the NodeBuilder tool.

This pragma initializes the 8-byte program ID using the hexadecimal values given (each character other than the colons in the argument is a hexadecimal digit from **0** to **F**). The first byte can only have a value of **8** or **9**, with **8** reserved for devices certified by the LONMARK association. If this pragma is used, the **#pragma set_id_string** directive cannot be used. Neither pragma is required or recommended when using the NodeBuilder Development Tool.

For more information about standard program IDs, see the *LonMark Application Layer Interoperability Guidelines*.

#pragma skip_ram_test_except_on_power_up

Specify this directive to speed up reset processing by skipping the automatic testing of RAM by the Neuron firmware. For Series 3100 devices, RAM is still tested if the reset is a result of powering up the device. RAM is still always set to zero by each reset.

This directive is not supported in model files.

#pragma snvt_si_eecode

Causes the compiler to force the linker to locate the self-identification and self-documentation information in EECODE space. See *Memory Areas* in Chapter 8, *Memory Management,* of the *Neuron C Programmer's Guide* for a definition of the EECODE space. By default, the linker places the table in EEPROM or in ROM code space, as it determines. Placing this table in EEPROM ensures that it can be modified using *Memory Write* network management messages. A network tool can use this capability to modify self-documentation of a device during installation. This pragma is only useful on a Neuron 3150 Chip, 3150 Smart Transceiver, or a Series 5000 or 6000 device.

This directive is not supported in model files.

#pragma snvt_si_ramcode

Causes the compiler to force the linker to locate the self-identification and self-documentation information in RAMCODE space. See *Memory Areas* in Chapter 8, *Memory Management,* of the *Neuron C Programmer's Guide* for a definition of the RAMCODE space. By default, the linker places the table in EEPROM or in ROM code space, as it determines. Placing this table in RAM ensures that it can be modified using *Memory Write* network management messages.

Note: *RAMCODE space is always external memory, and is assumed to be non-volatile.* This pragma is only useful on a Neuron 3150 Chip, 3150 Smart Transceiver, or a Series 5000 or 6000 device.

This directive is not supported in model files.

#pragma specify_io_clock *string*

Specify this directive to inform the compiler of the value of the Neuron clock speed (external crystal frequency for Series 3100 devices, or the I/O clock frequency for Series 5000 and 6000 devices).

The directive is generally not required for Series 5000 or 6000 chips because the I/O clock frequency is fixed for those chips.

This directive is only useful in combination with the **sci** I/O model, and permits the compiler to calculate the register settings for the SCI I/O hardware in any Neuron Chip or Smart Transceiver equipped with SCI I/O hardware. The clock rate is specified with a string constant following the pragma name as shown. The only clock rates that may be used with SCI I/O hardware are "20 MHz", "10 MHz", "6.5536 MHz", "5 MHz", and "2.5 MHz". The strings must appear exactly as shown, including capitalization.

This directive is not supported in model files.

#pragma system_image_extensions nv_length_override

This directive enables the NV length override system image extension. This system image extension is used to implement changeable network variable types. You must provide an extension function named **get_nv_length_override()** as detailed below. Using this compiler directive together with a version of the Neuron firmware that does not support system extensions will cause a linker error (**NLD#477**).

You can continue to access the **nv_len** property as discussed in the *Neuron C Programmer's Guide*. However, the Neuron C Version 2.1 (and later) system image extension technique provides a more robust implementation and should therefore be used for all new designs. Writing to the nv_len property is not recommended.

Where **#pragma system_image_extensions nv_length_override** enables the nv_length_override system image extension, you must also provide the system extension. To do so, you must implement a function that meets the following prototype:

unsigned _RESIDENT get_nv_length_override(unsigned nvIndex);

The function returns the current length in bytes of the network variable with the given index, or the result of the *get_declared_nv_length()* API to signal that the length has not been changed. The get_declared_nv_length() API is supported with Neuron C version 2.3 (or later). You can use conditional compilation based on the _SUPPORT_LARGE_NV preprocessor symbol to manage source code backwards compatibility, as demonstrated in the following example:

```
unsigned _RESIDENT get_nv_length_override(unsigned nvIndex)
{
#if defined(_SUPPORT_LARGE_NV)
  unsigned uResult = get_declared_nv_length(nvIndex);
#else
  unsigned uResult = 0xFF;
#endif


  // TO DO: return the current length of the network variable
```

```
// with index "nvIndex."
// Example code follows:
//
// switch (nvIndex) {
//   case nviChangeableNv::global_index:
//     if (nviChangeableNv::cpNvType.type_category != NVT_CAT_INITIAL
//     && nviChangeableNv::cpNvType.type_category != NVT_CAT_NUL) {
//       uResult = nviChangeableNv::cpNvType.type_length;
//     }
//     break;
// } // switch

return uResult;
         }
```

You must maintain information about the current length (and type) for network variables with changeable types in some appropriate, persistent, variable. You can use the **sizeof( )** operator to obtain the initial size of the network variable. See the discussion on *Changeable Type Network Variables* in the chapter *How Devices Communicate Using Network Variables* of the *Neuron C Programmer's Guide* for more information.

This directive is not supported in model files.

#pragma transaction_by_address_off

#pragma transaction_by_address_on

These pragmas explicitly control which version of transaction ID allocation algorithm the Neuron firmware uses. Some versions of the Neuron firmware support a new version of transaction ID allocation that has superior duplicate rejection properties. For the Neuron 3150 Chip, 3150 Smart Transceiver, Neuron 3120E1 Chip, Neuron 3120E2 Chip, and 3120 Smart Transceiver, firmware version 6 (or later) supports either algorithm. For the Neuron 3120 Chip, firmware version 4 (or later) supports either algorithm. For a Series 5000 or 6000 device, firmware version 18 (or later) supports either algorithm. The newer version of transaction tracking (the *on* option) is used by default when available, unless the device is a LONWORKS network interface (for example, running the MIP), or the device's application program generates explicit destination addresses.

These directives are not supported in model files.

#pragma unknown_system_image_extension_isa_warning

This directive causes the [**NLD#477**] linker message, which normally reports an error for the use of **#pragma system_image_extensions nv_length_override** on a version of the Neuron firmware that does not support system image extension, to be changed to a warning. This change allows you to compile the same application code for different targets with respect to their system image support.

The Code Wizard in NodeBuilder 3.1 (and later) uses this directive to generate Neuron C source code that compiles, for example, for a LTM-10A target (debug platform), and a TP/FT-10F Flash Control Module (release

platform). See the discussion on Changeable Type Network Variables in the chapter *How Devices Communicate Using Network Variables* of the *Neuron C Programmer's Guide* for more information.

You normally do not need to specify this directive if your debug and release targets use the same system firmware version and hardware, or if you use conditional compilation to distinguish between the targets.

This directive is not supported in model files.

#pragma warnings_off

#pragma warnings_on

Controls the compiler's printing of warning messages. Warning messages generally indicate a problem in a program, or a place where code could be improved. Warning messages are on by default at the start of a compilation. These pragmas can be intermixed multiple times throughout a program to turn warning message printing on and off as desired.

These directives override the settings for the **#pragma enable_warning** *number* and **#pragma disable_warning** *number* directives.

The **warnings_off** and **warnings_on** directives are deprecated. Use the **enable_warning** and **disable_warning** directives instead.

Other Directives

The following additional directives are defined in Neuron C Version 2.3:

#error "*text***"**

This directive allows you to issue a custom error message. When this directive is processed, program compilation fails. This directive is useful for managing conditional compilation, for example:

```
#ifdef XXX
   ...
#else
   #ifdef YYY
      ...
   #else
      #error "You must define either XXX or YYY"
   #endif
#endif
```

This directive is supported in both the standard-compliant unquoted form or the backwards-compatible quoted one.

#warning "*text***"**

This directive allows you to issue a custom warning message. When this directive is processed, program compilation continues. Both the quoted and unquoted forms of the warning directive are supported. This directive is useful for managing conditional compilation, for example:

```
#ifdef XXX
   ...
#else
   #ifdef YYY
      ...
```

```
      #else
         #warning “You should define either XXX or YYY”
      #endif
   #endif
```

Or

```
   #ifdef XXX
      ...
   #else
      #ifdef YYY
         ...
      #else
         #warning You should define either XXX or YYY
      #endif
   #endif
```

When printing the diagnostic triggered by the warning directive, a quoted message will appear with quotes.

MCPP and Neuron C Compiler Versions 5 and 6

The current version of the Neuron C Compiler supports all standard preprocessor directives. If you want support for both Neuron C Compiler 6 and Neuron C Compiler 5, you should use conditional compilation, as demonstrated in the following example.

```
#ifdef _NCC_VERSION
   // this is a compiler version 6 or better. The value of
   // NCC_VERSION equates to the compiler’s major version
   // number.  The value can be evaluated in #if or #elif
   // statements, but these must be hidden from earlier versions
   // of the compiler:
# include “v6-definitions.h”
#else // _NCC_VERSION not defined:
// This is a compiler prior to version 6. #ifdef, #ifndef, #else
// and #endif are available, but not #if or #elif. These
// declarations could be made here or, for symmetry with the
// above, in a dedicated definition file:
#endif    // compiler version test
```

When source code relies on the availability of conditional compilation with use of #if or #elif directives, the following precautionary code snippet is recommended for inclusion in prime source code locations:

```
#ifndef _NCC_VERSION
# error “This source requires a Neuron C Compiler 6 or better”
#endif    // compiler version check
```

The *error* directive in the above example is presented in quoted form, but Version 6 of the Neuron C Compiler supports both quoted and unquoted, as shown below.

```
#ifndef _NCC_VERSION
# error “This source requires a Neuron C Compiler 6 or better”
# error This source requires a Neuron C Compiler 6 or better
#endif    // error directive example
```

The standard *line* directive is supported in the following form:

```
#line 123 “c:\mysource\xyz\main.nc”
  // line directive example
```

The *line* directive is used to synchronize compiler and debugger on the location of source code. While the directive is supported according to the language standard, it is not recommended for use in your code, as it may prevent debugging your application using NodeBuilder’s source code debugger.

3

Functions

This chapter provides reference information on the Neuron C built-in and library functions.

Built-in and library functions are used with executable code, and do not apply to host-based device development with model files.

Introduction

This chapter discusses some general attributes of Neuron C functions and lists available Neuron C functions, providing syntax information, descriptions, and examples of each function. Some functions are *built-in* functions. This means they are used as if they were function calls, but they are permanently part of the Neuron C language and are implemented by the compiler without necessarily mapping into an actual function call. Some built-in functions have special behaviors depending on their context and usage. The rest of the functions are library calls. Some library calls have function prototypes in one of the standard include files, as noted. The standard include files are:

<a2d.h>
<access.h> (this file includes **<addrdefs.h>**)
<addrdefs.h>
<byte.h>
<control.h>
<float.h>
<io_types.h>
<limits.h>
<mem.h>
<modnvlen.h>
<msg_addr.h>
<netdbg.h>
<netmgmt.h>
<nm_ckm.h>
<nm_err.h>
<nm_fm.h>
<nm_inst.h>
<nm_mod.h>
<nm_model.h>
<nm_nmo.h>
<nm_rqr.h>
<nm_sel.h>
<nm_ste.h>
<nm_sub.h>
<nm_wch.h>
<psg.h>
<psgreg.h>
<s32.h>
<status.h>
<stddef.h>
<stdlib.h>
<string.h>

Functions not defined in any of the above include files derive their prototypes from **<echelon.h>**, an include file that is automatically incorporated in each compilation. Except for **<echelon.h>**, you must incorporate the necessary include file (or files) to use a function. Although some of the following function descriptions list both an include file and a prototype, you should only specify the **#include** directive. The prototype is contained in the include file, and is shown here only for reference.

The functions listed in this chapter include floating-point and extended (32-bit) precision arithmetic support. A general discussion of the use of floating-point variables and floating-point arithmetic, and a discussion of the use of extended precision variables and extended precision arithmetic is included in the following list of functions.

Any existing application program developed for a Neuron 3120 Chip or 3120 Smart Transceiver using any system library functions might require more EEPROM memory on a Neuron 3120 Chip or 3120 Smart Transceiver than it would on a Neuron 3150 Chip, 3150 Smart Transceiverm or Series 5000 or 6000 chip. This is because more of the system functions are stored in the ROM firmware image on a Neuron 3150 Chip, a 3150 Smart Transceiver, or a Series 5000 or 6000 chip. Examination of the link map provides a measure of the EEPROM memory used by these functions. See *System Library on a Neuron 3120 Chip* in Chapter 8, *Memory Management,* of the *Neuron C Programmer's Guide* for more detailed information on how to create and examine a link map to obtain a measure of the Neuron 3120 Chip or 3120 Smart Transceiver EEPROM usage required for these functions. Also see the *NodeBuilder User's Guide* for additional information on the link map.

Note: "Neuron 3120 Chip" above refers to all the Neuron 3120 Chips, including 3120, 3120E1, 3120E2, 3120E3, 3120E4, 3120E5, and 3120A20 Chips, as well as the FT 3120 Smart Transceiver, the PL 3120 Smart Transceiver, and the PL 3170 Smart Transceiver.

Transient and Resident Functions

Compiling an application designed for use with a Series 6000 chip defaults to *transient* functions. Transient functions are loaded into the memory space on demand, and are transparently managed by the Neuron C Compiler and Neuron firmware. Neuron Chips and Smart Transceivers released prior to the Series 6000 chips do not support transient functions.

Transient functions support applications whose total code space exceed the available physical chip address space.

Resident functions are all those which are not transient. Resident functions reside within the chip's address space permanently, unrecoverably consuming address space but incurring no invocation overhead.

Note that *when*-tasks and *interrupt* tasks are always resident, but calling transient functions from *when*-tasks is possible. *Interrupt* tasks cannot call transient functions, however.

The compiler supports several tools to enforce resident functions:

The *__resident* keyword (note the leading double underscore) can be used to specify an individual function as resident. Function implementation and prototype must match; both must declare the function as resident or transient. Note that no keyword is supplied to enforce transient functions; transient functions are the default where this feature is supported by the target device hardware and firmware.

Example:

```
extern unsigned __resident xor_n(unsigned n, unsigned* data);

unsigned __resident xor_n(unsigned n, unsigned* data) {
   unsigned result = 0;
```

```
   while (n--) result ^= *data++;
   return result;
}
```

The standard Neuron C include files also define a *_RESIDENT* preprocessor symbol. This can be used to create code which is backwards-compatible with earlier versions of the Neuron C Compiler. Earlier versions of the compiler do not define the necessary preprequisites, leading to the definition of *_RESIDENT* as an empty string. The current version of the Neuron C Compiler supplies the required prerequisites and the *_RESIDENT* symbol expands to the *__resident* keyword.

Example:

```
extern unsigned _RESIDENT xor_n(unsigned n, unsigned* data);

unsigned _RESIDENT xor_n(unsigned n, unsigned* data) {
   unsigned result = 0;
   while (n--) result ^= *data++;
   return result;
}
```

Most function prototypes included with standard Neuron C include files use this technique, as most Neuron C runtime utility functions and system API are declared resident.

Instead of defining individual functions as resident explicitly, you can also use the pragma resident directive to enable or disable the default mode, which applies to plain functions.

In the next example, function a will be transient with compilation targets which support transient functions, and function b will always be resident. Function c has the same residence as function a:

```
void a(void) {
  …
}

#pragma resident enabled
void b(void) {
  …
}
#pragma resident disabled

void c(void) {
  …
}
```

The compiler also supports a new command line option, --resident, for use with the command console or automated build scripts. This option has the same effect as a global *#pragma resident enabled* directive and sets the default of plain function definitions (such as function ‘a’ in the previous example) to *resident*.

See appendix A, *Neuron C Tools Stand-Alone Use* in the *Neuron C Programmer’s Guide* for more about using the Neuron C tools from the console or build scripts.

Overview of Neuron C Functions

You can call the functions listed in the following sections from a Neuron C application program. These functions are built into the Neuron C Compiler, or are part of the Neuron firmware, or are linked into the application image from a system library. The availability of these functions varies by model of Neuron

Chip or Smart Transceiver, as well as by firmware version. This detailed information is available at www.echelon.com/downloads.

At the time of this release, all Neuron C runtime utilities and system API are resident. The *__resident* or *_RESIDENT* modifier is implied in all the prototypes which follow in this guide.

Execution Control

Table 6 lists the execution control functions.

Table 6. Execution Control Functions

Function	Description
delay( )	Delay processing for a time independent of input clock rate
flush( )	Flush all outgoing messages and network variable updates
flush_cancel( )	Cancel a flush in process
flush_wait( )	Wait for outgoing messages and updates to be sent before going off-line
get_tick_count( )	Read hardware timer
go_offline( )	Cease execution of the application program
interrupt_control( )	Enable or disable interrupts
msec_delay( )	Delay processing for a specified number of milliseconds
post_events( )	Define a critical section boundary for network variable and message processing
power_up( )	Determine whether last processor reset was due to power up
preemption_mode( )	Determine whether the application processor scheduler is currently running in preemption mode.
propagate( )	Force propagation of an output network variable
scaled_delay( )	Delay processing for a time that depends on the input clock rate

Function	Description
sleep( )	Enter low-power mode by disabling system clock
timers_off( )	Turn off all software timers
watchdog_update( )	Re-trigger the watchdog timer to prevent device reset

Network Configuration

Table 7 lists the network configuration functions.

Table 7. Network Configuration Functions

Function	Description
access_address( )	Read device's address table
access_alias( )	Read device's alias table
access_domain( )	Read device's domain table
access_nv( )	Read device's network variable configuration table
addr_table_index( )	Determine address table index of message tag
application_restart( )	Begin application program over again
get_current_nv_length( )	Read a network variable's current length
go_unconfigured( )	Reset this device to an uninstalled state
node_reset( )	Activate the reset pin, and reset all CPUs
nv_table_index( )	Determine global index of a network variable
offline_confirm( )	Inform network tool that this device is going offline
update_address( )	Write device's address table
update_alias( )	Write device's alias table

Function	Description
update_clone_domain( )	Write device's domain table with clone entry
update_config_data( )	Write device's configuration data structure
update_domain( )	Write device's domain table with normal entry
update_nv( )	Write device's network variable configuration table

Integer Math

Table 8 lists the integer mathematics functions.

Table 8. Integer Math Functions

Function	Description
abs( )	Arithmetic absolute value
bcd2bin( )	Convert binary coded decimal data to binary
bin2bcd( )	Convert binary data to binary coded decimal
high_byte( )	Extract the high byte of a 16-bit number
low_byte( )	Extract the low byte of a 16-bit number
make_long( )	Create a 16-bit number from two 8-bit numbers
max( )	Arithmetic maximum of two values
min( )	Arithmetic minimum of two values
muldiv( )	Unsigned multiply/divide with 32-bit intermediate result
muldiv24( )	Unsigned multiply/divide with 24-bit intermediate result
muldiv24s( )	Signed multiply/divide with 24-bit intermediate result

Function	Description
muldivs( )	Signed multiply/divide with 32-bit intermediate result
random( )	Generate 8-bit random number
reverse( )	Reverse the order of bits in an eight-bit number
rotate_long_left( )	Rotate left a 16-bit number
rotate_long_right( )	Rotate right a 16-bit number
rotate_short_left( )	Rotate left an 8-bit number
rotate_short_right( )	Rotate right an 8-bit number
s32_abs( )	Take the absolute value of a signed 32-bit number
s32_add( )	Add two signed 32-bit numbers
s32_cmp( )	Compare two 32-bit signed numbers
s32_dec( )	Decrement a 32-bit signed number
s32_div( )	Divide two signed 32-bit numbers
s32_div2( )	Divide a 32-bit signed number by 2
s32_eq( )	Return TRUE if first argument equals second argument
s32_from_ascii( )	Convert an ASCII string into a 32-bit signed number
s32_from_slong( )	Convert a signed long number into a 32-bit signed number
s32_from_ulong( )	Convert an unsigned long number into a 32-bit signed number
s32_ge( )	Return TRUE if first argument is greater than or equal to second argument
s32_gt( )	Return TRUE if first argument is greater than second argument
s32_inc( )	Increment a 32-bit signed number

Function	Description
s32_le( )	Return TRUE if first argument is less than or equal to second argument
s32_lt( )	Return TRUE if first argument is less than second argument
s32_max( )	Take the maximum of two signed 32-bit numbers
s32_min( )	Take the minimum of two signed 32-bit numbers
s32_mul( )	Multiply two signed 32-bit numbers
s32_mul2( )	Multiply a 32-bit signed number by 2
s32_ne( )	Return TRUE if first argument does not equal second argument
s32_neg( )	Return the negative of a signed 32-bit number
s32_rand( )	Return a random 32-bit signed number
s32_rem( )	Return the remainder of a division of two signed 32-bit numbers
s32_sign( )	Return the sign of a 32-bit signed number
s32_sub( )	Subtract two signed 32-bit numbers
s32_to_ascii( )	Convert a 32-bit signed number into an ASCII string
s32_to_slong( )	Convert a 32-bit signed number into signed long

Floating-Point Math

Table 9 lists the floating-point mathematics functions.

Table 9. Floating-Point Functions

Function	Description
fl_abs( )	Take the absolute value of a floating-point number
fl_add( )	Add two floating-point numbers

Function	Description
fl_ceil( )	Return the ceiling of a floating-point number
fl_cmp( )	Compare two floating-point numbers
fl_div( )	Divide two floating-point numbers
fl_div2( )	Divide a floating-point number by two
fl_eq( )	Return TRUE if first argument equals second argument
fl_floor( )	Return the floor of a floating-point number
fl_from_ascii( )	Convert an ASCII string to floating-point
fl_from_s32( )	Convert a signed 32-bit number to a floating-point number
fl_from_slong( )	Convert a signed long number into a floating-point number
fl_from_ulong( )	Convert an unsigned long number to a floating-point number
fl_ge( )	Return TRUE if first argument is greater than or equal to second argument
fl_gt( )	Return TRUE if first argument is greater than second argument
fl_le( )	Return TRUE if first argument is less than or equal to second argument
fl_lt( )	Return TRUE if first argument is less than second argument
fl_max( )	Find the maximum of two floating-point numbers
fl_min( )	Find the minimum of two floating-point numbers
fl_mul( )	Multiply two floating-point numbers
fl_mul2( )	Multiply a floating-point number by two

Function	Description
fl_ne( )	Return TRUE if first argument is not equal to second argument
fl_neg( )	Return the negative of a floating-point number
fl_rand( )	Return a random floating-point number
fl_round( )	Round a floating-point number to the nearest whole number
fl_sign( )	Return the sign of a floating-point number
fl_sqrt( )	Return the square root of a floating-point number
fl_sub( )	Subtract two floating-point numbers
fl_to_ascii( )	Convert a floating-point number to an ASCII string
fl_to_ascii_fmt( )	Convert a floating-point number to a formatted ASCII string
fl_to_s32( )	Convert a floating-point number to signed 32-bit
fl_to_slong( )	Convert a floating-point number to signed long
fl_to_ulong( )	Convert a floating-point number to unsigned long
fl_trunc( )	Return the whole number part of a floating-point number

Strings

Table 10 lists the string functions.

Table 10. String Functions

Function	Description
strcat( )	Append a copy of a string at the end of another
strchr( )	Scan a string for a specific character

Function	Description
strcmp( )	Compare two strings
strcpy( )	Copy one string into another
strlen( )	Return the length of a string
strncat( )	Append a copy of a string at the end of another
strncmp( )	Compare two strings
strncpy( )	Copy one string into another
strrchr( )	Scan a string in reverse for a specific character

Utilities

Table 11 on page 58 lists the utlility functions.

Table 11. Utility Functions

Function	Description
ansi_memcpy( )	Copy a block of memory with ANSI return value
ansi_memset( )	Set a block of memory to a specified value with ANSI return value
clear_status( )	Clear error statistics accumulators and error log
clr_bit( )	Clear a bit in a bit array
crc8( )	Calculate an 8-bit CRC over an array
crc16( )	Calculate a 16-bit CRC over an array
crc16_ccitt( )	Calculate a 16-bit CCITT CRC over an array
eeprom_memcpy( )	Copy a block of memory to EEPROM destination
error_log( )	Record software-detected error
fblock_director( )	Call the director associated with an **fblock**

Function	Description
get_fblock_count( )	Return the number of **fblock** declarations in the program
get_nv_count( )	Return the number of network variable declarations in the program
memccpy( )	Copy a block of memory
memchr( )	Search a block of memory
memcmp( )	Compare a block of memory
memcpy( )	Copy a block of memory: • from **msg_in.data** and **resp_in.data** • to **resp_out.data** • length greater than or equal to 256 bytes • others
memset( )	Set a block of memory to a specified value: • length greater than or equal to 256 bytes • others
retrieve_status( )	Read statistics from protocol processor
service_pin_msg_send( )	Send a service pin message
service_pin_state( )	Read the service pin state
set_bit( )	Set a bit in a bit array
set_eeprom_lock( )	Set the state of the checksummed EEPROM's lock
tst_bit( )	Return TRUE if bit tested was set

Input/Output

Table 12 lists the I/O functions.

Table 12. Input/Output Functions

Function	Description
io_change_init( )	Initialize reference value for **io_changes** event
io_edgelog_preload( )	Define maximum value for **edgelog** period measurements
io_edgelog_single_preload( )	Define maximum value for **edgelog single_tc** period measurements
io_idis( )	Disable the I/O interrupt used in the hardware support for the **sci** and **spi** I/O objects
io_iena( )	Enable the I/O interrupt used in the hardware support for the **sci** and **spi** I/O objects
io_in( )	Input data from I/O object: • Dualslope input • Edgelog input • Infrared input • Magcard input • Neurowire I/O slave mode • Neurowire I/O with invert option • Serial input • Touch I/O • Wiegand input • others
io_in_ready( )	Event function which evaluates to TRUE when a block of data is available from the parallel I/O object
io_in_request( )	Start dualslope A/D conversion

Function	Description
io_out( )	Output data to I/O object: • Bitshift output • Neurowire I/O slave mode • Neurowire I/O with invert option • Serial output • Touch I/O • others
io_out_ready( )	Event function which evaluates to TRUE when a block of data is available from the parallel I/O object
io_out_request( )	Request ready indication from parallel I/O object
io_preserve_input( )	Preserve first timer/counter value after reset or **io_select( )**
io_select( )	Set timer/counter multiplexer
io_set_baud( )	Set the serial bit rate for an **sci** I/O object
io_set_clock( )	Set timer/counter clock rate
io_set_direction( )	Change direction of I/O pins
sci_abort( )	Abort pending **sci** transfer
sci_get_error( )	Read most recent **sci** error code
spi_abort( )	Abort pending **spi** transfer
spi_get_error( )	Read most recent **spi** error code

Signed 32-Bit Integer Support Functions

The Neuron C compiler does not directly support the use of the C arithmetic and comparison operators with 32-bit integers. However, there is a complete library of functions for signed 32-bit integer math. These functions are listed in *Integer Math* on page 53. For example, in standard ANSI C, to evaluate **X = A + B * C** in long (32-bit) arithmetic, the '**+**' and '*****' infix operators can be used as follows:

```
long X, A, B, C;
X = A + B * C;
```

With Neuron C, this can be expressed as follows:

```
s32_type X, A, B, C;
s32_mul(&B, &C, &X);
s32_add(&X, &A, &X);
```

The signed 32-bit integer format can represent numbers in the range of ±2,147,483,647 with an absolute resolution of ±1.

An **s32_type** structure data type for signed 32-bit integers is defined by means of a **typedef** in the **<s32.h>** file. It defines a structure containing an array of four bytes that represents a signed 32-bit integer in Neuron C format. This is represented as a two's complement number stored with the most significant byte first. The type declaration is shown here for reference:

```
typedef struct {
   int   bytes[4];
} s32_type;
```

All of the constants and functions in the **<s32.h>** file are defined using the Neuron C signed 32-bit data type, which is a structure. Neuron C does not permit structures to be passed as parameters or returned as values from functions. When these objects are passed as parameters to C functions, they are passed as addresses (using the '**&**' operator) rather than as values. However, Neuron C does support structure assignment, so signed 32-bit integers can be assigned to each other with the '**=**' assignment operator.

No errors are detected by the 32-bit functions. Overflows follow the rules of the C programming language for integers, namely, they are ignored. Only the least significant 32 bits of the results are returned.

Initializers can be defined using structure initialization syntax. For example:

```
s32_type some_number = {0, 0, 0, 4};
// initialized to 4 on reset

s32_type another_number = {-1, -1, -1, -16};
// initialized to -16
```

A number of constants are defined for use by the application if desired. **s32_zero**, **s32_one**, **s32_minus_one** represent the numbers 0, 1, and -1.

If other constants are desired, they can be converted at runtime from ASCII strings using the function **s32_from_ascii( )**.

Example:

```
s32_type one_million;

when(reset) {
    s32_from_ascii("1000000", one_million);
}
```

Because this function is fairly time consuming, it could be advantageous to pre-compute constants with the NXT Neuron C Extended Arithmetic Translator utility. This program accepts an input file with declarations using standard integer initializers, and creates an output file with Neuron C initializers. See *Using the NXT Neuron C Extended Arithmetic Translator* on page 79.

For example, if the input file contains the following statement:

```
const s32_type one_million = 1000000;
```

then the output file contains the following:

```
const s32_type one_million = {0x00,0x0f,0x42,0x40}
/* 1000000 */;
```

Users of the NodeBuilder tool can use Code Wizard to create initializer data for **s32_type** network variables and configuration properties. The NodeBuilder Neuron C debugger can display signed 32-bit integers through the **s32_type** shown above.

The Neuron C debugger can display signed 32-bit integers as raw data at a specific address. To examine the value of one or more contiguous signed 32-bit integer variables, enter the address of the first variable into the raw data evaluation window, select **Raw Data at Address**, **Data Size** as **quad**, **Count** as the number of variables that you want to display, and **Format** as **Dec**. The data is displayed as unsigned, even if it is negative. To view the data as signed, click on the value field, and the Modify Variable window shows the data in both formats. You can also modify signed 32-bit integer variables by clicking on the value field, and entering new data in the usual format for integers.

The signed 32-bit integer arguments are all passed to the support functions as addresses of structures. The calling function or task is responsible for declaring storage for the arguments themselves. Argument lists are ordered so that input arguments precede output arguments. In all cases, any of the signed 32-bit integer input arguments can be reused as output arguments to facilitate operations in place.

Binary Arithmetic Operators

Table 13 on page 63 lists the binary arithmetic operator functions.

Table 13. Binary Arithmetic Operators

Short Name	Function
add	`void s32_add(const s32_type *arg1, const s32_type *arg2, s32_type *arg3);` Adds two signed 32-bit integers. **(arg3 = arg1 + arg2)**.
sub	`void s32_sub(const s32_type *arg1, const s32_type *arg2, s32_type *arg3);` Subtracts two signed 32-bit integers. **(arg3 = arg1 - arg2)**.
mul	`void s32_mul(const s32_type *arg1, const s32_type *arg2, s32_type *arg3);` Multiplies two signed 32-bit integers. **(arg3 = arg1 *arg2)**.
div	`void s32_div(const s32_type *arg1, const s32_type *arg2, s32_type *arg3);` Divides two signed 32-bit integers. **(arg3 = arg1 / arg2)**.

Short Name	Function
rem	`void s32_rem(const s32_type *arg1, const s32_type *arg2, s32_type *arg3);` Returns the remainder of the division of two signed 32-bit integers **(arg3 = arg1 % arg2)**. The sign of arg3 is always the same as the sign of arg1.
max	`void s32_max(const s32_type *arg1, const s32_type *arg2, s32_type *arg3);` Returns the maximum of two signed 32-bit integers. **(arg3 = max(arg1, arg2))**.
min	`void s32_min(const s32_type *arg1, const s32_type *arg2, s32_type *arg3);` Returns the minimum of two signed 32-bit integers. **(arg3 = min(arg1, arg2))**.

Unary Arithmetic Operators

Table 14 lists the unary arithmetic operator functions.

Table 14. Unary Arithmetic Operators

Short Name	Function
abs	`void s32_abs(const s32_type *arg1, s32_type *arg2);` Returns the absolute value of a signed 32-bit integer. **(arg2 = abs(arg1))**.
neg	`void s32_neg(const s32_type *arg1, s32_type *arg2);` Returns the negative of a signed 32-bit integer. **(arg2 = - arg1)**.

Comparison Operators

Table 15 lists the comparison operator functions.

Table 15. Comparison Operators

Short Name	Function
eq	`boolean s32_eq(const s32_type *arg1, const s32_type *arg2);` Returns TRUE if the first argument is equal to the second argument, otherwise FALSE. **(arg1 == arg2)**.

Short Name	Function
ne	`boolean s32_ne(const s32_type *arg1, const s32_type *arg2);` Returns TRUE if the first argument is not equal to the second argument, otherwise FALSE. **(arg1 != arg2)**.
gt	`boolean s32_gt(const s32_type *arg1, const s32_type *arg2);` Returns TRUE if the first argument is greater than the second argument, otherwise FALSE. **(arg1 > arg2)**.
lt	`boolean s32_lt(const s32_type *arg1, const s32_type *arg2);` Returns TRUE if the first argument is less than the second argument, otherwise FALSE. **(arg1 < arg2)**.
ge	`boolean s32_ge(const s32_type *arg1, const s32_type *arg2);` Returns TRUE if the first argument is greater than or equal to the second argument, otherwise FALSE. **(arg1 >= arg2)**.
le	`boolean s32_le(const s32_type *arg1, const s32_type *arg2);` Returns TRUE if the first argument is less than or equal to the second argument, otherwise FALSE. **(arg1 <= arg2)**.
cmp	`int s32_cmp(const s32_type *arg1, const s32_type *arg2);` Returns +1 if the first argument is greater than the second argument, -1 if it is less, and 0 if it is equal.

Miscellaneous Signed 32-bit Functions

Table 16 lists miscellaneous signed 32-bit functions.

Table 16. Signed 32-Bit Functions

Short Name	Function
sign	`int s32_sign(const s32_type *arg);` Sign function, returns +1 if the argument is positive, 0 if the argument is zero, and -1 if the argument is negative.
inc	`void s32_inc(s32_type *arg);` Increments a signed 32-bit integer.

Short Name	Function
dec	`void s32_dec(s32_type *arg);` Decrements a signed 32-bit integer.
mul2	`void s32_mul2(s32_type *arg);` Multiplies a signed 32-bit integer by two.
div2	`void s32_div2( s32_type *arg );` Divides a signed 32-bit integer by two.
rand	`void s32_rand(s32_type *arg);` Returns a random integer uniformly distributed in the range [-2,147,483,648 to +2,147,483,647].

Integer Conversions

Table 17 lists the integer conversion functions.

Table 17. Integer Conversions

Short Name	Function
to slong	`signed long s32_to_slong(const s32_type *arg);` Converts a signed 32-bit integer to a Neuron C signed long integer (range -32,768 to +32,767). Overflow is ignored.
to ulong	`unsigned long s32_to_ulong(const s32_type *arg);` Converts a signed 32-bit integer to a Neuron C unsigned long integer (range 0 to 65,535). Overflow is ignored.
from slong	`void s32_from_slong(signed long arg1, s32_type *arg2);` Converts a Neuron C signed long integer (range -32,768 to +32,767) to a signed 32-bit integer.
from ulong	`void s32_from_ulong(unsigned long arg1, s32_type *arg2);` Converts a Neuron C unsigned long integer (range 0 to +65,535) to a signed 32-bit integer.

Conversion of Signed 32-bit to ASCII String

Table 18 lists the conversion function for a signed 32-bit number to an ASCII string.

Table 18. Conversions of Signed 32-Bit Numbers to ASCII Strings

Short Name	Function
`to ascii`	`void s32_to_ascii(const s32_type *arg1, char *arg2);` Converts a signed 32-bit integer ***arg1** to an ASCII string followed by a terminating null character. The *arg2 output buffer should be at least 12 bytes long. The general output format is **[-]xxxxxxxxxx**, with one to ten digits.

Conversion of ASCII String to Signed 32-bit

Table 19 lists the conversion function for an ASCII string to a signed 32-bit number.

Table 19. Conversions of ASCII Strings to Signed 32-Bit Numbers

Short Name	Function
`from ascii`	`void s32_from_ascii(const char *arg1, s32_type *arg2);` Converts an ASCII string arg1 to a signed 32-bit integer in ***arg2**. The conversion stops at the first invalid character in the input buffer – there is no error notification. The acceptable format is **[-]xxxxxxxxxx**. The number of digits should not exceed ten. Embedded spaces within the string are not allowed.

Signed 32-Bit Performance

Table 20 on page 67 lists times in milliseconds for the various 32-bit functions. They were measured using a Series 3100 Neuron Chip with a 10 MHz input clock. These values scale with a faster or slower clock. The measurements are maximums and averages over random data uniformly distributed in the range [-2,147,483,648 to +2,147,483,647].

Table 20. Signed 32-Bit Function Performance

Function	Maximum	Average
Add/subtract	0.10	0.08
Multiply	2.07	1.34
Divide	3.17	2.76
Remainder	3.15	2.75
Maximum/minimum	0.33	0.26
Absolute value	0.25	0.12

Function	Maximum	Average
Negation	0.20	0.20
Arithmetic Comparison	0.33	0.26
Conversion to ASCII	26.95	16.31
Conversion from ASCII	7.55	4.28
Conversion to 16-bit integer	0.12	0.10
Conversion from 16-bit integer	0.10	0.10
Random number generation	0.12	0.11
Sign of number	0.15	0.11
Increment	0.07	0.04
Decrement	0.10	0.04
Multiply by two	0.10	0.10
Divide by two	0.30	0.16

Floating-Point Support Functions

The Neuron C compiler does not directly support the use of the ANSI C arithmetic and comparison operators with floating-point values. However, there is a complete library of functions for floating-point math. These functions are listed in *Floating-Point Math* on page 55. For example, in standard ANSI C, to evaluate X = A + B * C in floating-point, the '**+**' and '*****' infix operators can be used as follows:

```
float X, A, B, C;
X = A + B * C;
```

With Neuron C, this can be expressed as follows:

```
float_type X, A, B, C;
fl_mul(&B, &C, &X);
fl_add(&X, &A, &X);
```

The floating-point format can represent numbers in the range of approximately $-1*10^{1038}$ to $+1*10^{1038}$, with a relative resolution of approximately $\pm1*10^{-7}$.

A **float_type** structure data type is defined by means of a **typedef** in the **<float.h>** file. It defines a structure that represents a floating-point number in IEEE 754 single precision format. This has one sign bit, eight exponent bits and 23 mantissa bits, and is stored in big-endian order. Processors that store data in little-endian order represent IEEE 754 numbers in the reverse byte order. The **float_type** type is identical to the type used to represent floating-point network variables. The type declaration is shown here for reference.

```
typedef struct {
    unsigned int sign         : 1;
            // 0 = positive, 1 = negative
    unsigned int MS_exponent  : 7;
    unsigned int LS_exponent  : 1;
    unsigned int MS_mantissa  : 7;
    unsigned long LS_mantissa;
} float_type;
```

See the IEEE 754 standard documentation for more details.

All the constants and functions in the **<float.h>** file are defined using the Neuron C **float_type** floating-point format, which is a structure. Neuron C does not permit structures to be passed as parameters or returned as values from functions. When these objects are passed as parameters to C functions, they are passed as addresses (using the '**&**' operator) rather than as values. However, Neuron C does support structure assignment, so floating-point objects can be assigned to each other with the '**=**' assignment operator.

An **fl_error** global variable stores the last error detected by the floating-point functions. If error detection is desired for a calculation, application programs should set the **fl_error** variable to **FL_OK** before beginning a series of floating-point operations, and check the value of the variable at the end.

The errors detected are as follows:

FL_UNDERFLOW	A non-zero number could not be represented because it was too small for the floating-point representation. Zero was returned instead.
FL_INVALID_ARG	A floating-point number could not be converted to integer because it was out of range; or, an attempt was made to evaluate the square root of a negative number.
FL_OVERFLOW	A number could not be represented because it was too large for the floating-point representation.
FL_DIVIDE_BY_ZERO	An attempt was made to divide by zero. This does **not** cause the Neuron firmware DIVIDE_BY_ZERO error to be logged.

A number of **#define** literals are defined for use by the application to initialize floating-point structures. **FL_ZERO, FL_HALF, FL_ONE, FL_MINUS_ONE** and **FL_TEN** can be used to initialize floating-point variables to 0.0, 0.5, 1.0, -1.0, and 10.0 respectively.

Example:

```
float_type some_number = FL_ONE;
// initialized to 1.0 at reset
```

Five floating-point constants are pre-defined: **fl_zero, fl_half, fl_one, fl_minus_one**, and **fl_ten** represent 0.0, 0.5, 1.0, -1.0, and 10.0 respectively.

Example:

```
fl_mul(&some_number, &fl_ten, &some_number);
```

```
// multiply some number by 10.0
```

If other constants are desired, they can be converted at runtime from ASCII strings using the **fl_from_ascii( )** function.

Example:

```
float_type ninety_nine; // constant 99.0
when(reset) {
    fl_from_ascii("99", &ninety_nine);
         // initialize constant
}
```

Because this function is fairly time consuming, it could be advantageous to pre-compute constants with the NXT Neuron C Extended Arithmetic Translator. This program accepts an input file with declarations using standard floating-point initializers, and creates an output file with Neuron C initializers. It recognizes any **SNVT_*xxx*_f** data type, as well as the **float_type** type. See *Using the NXT Neuron C Extended Arithmetic Translator* on page 79.

For example, if the input file contains the following statements:

```
network input float_type var1 = 1.23E4;
const float_type var2 = -1.24E5;
SNVT_temp_f var3 = 12.34;
```

then the output file contains the following:

```
network input float_type var1 = {0,0x46,0,0x40,0x3000}
/* 1.23E4 */;
const float_type var2 = {1,0x47,1,0x72,0x3000}
/* -1.24E5 */;
SNVT_temp_f var3 = {0,0x41,0,0x45,0x70a4}
/* 12.34 */;
```

Users of the NodeBuilder tool can also use Code Wizard to create initializer data for **float_type** objects.

Variables of a floating-point network variable type are compatible with the Neuron C **float_type** format. The ANSI C language requires an explicit type cast to convert from one type to another. Structure types cannot be cast, but pointers to structures can. The following example shows how a local variable of type **float_type** can be used to update an output network variable of type **SNVT_angle_f**.

Example:

```
float_type local_angle;  // internal variable
network output SNVT_angle_f nvoAngle;  // network variable

void f(void) {
   nvoAngle = *(SNVT_angle_f *) &local_angle;
}
```

The following example shows how an input **SNVT_length_f** network variable can be used as an input parameter to one of the functions in this library.

Example:

```
network input SNVT_length_f nvoLength;
// network variable

when(nv_update_occurs(nvoLength)) {
    if(fl_eq((const float_type *)&nvoLength, &fl_zero))
      // compare length to zero
   . . .
}
```

The IEEE 754 format defines certain special numbers such as Infinity, Not-a-Number (NaN), and Denormalized Numbers. This library does not produce the correct results when operating on these special numbers. Also, the treatment of roundoff, overflow, underflow, and other error conditions does not conform to the standard.

To assign the IEEE value of NaN to a floating-point object, you can use the hex value 0x7FC00000 as shown in the example below:

Example:

```
float_type fl = {0,0x7F,1,0x40,0};  // NaN
```

The NodeBuilder debugger can display floating-point objects according to their underlying **float_type** structure.

The debugger can display floating-point objects as raw data at a specific address. To examine the value of one or more contiguous floating-point variables, enter the address of the first variable into the raw data evaluation window, select **Raw Data at Address**, **Data Size** as **quad**, **Count** as the number of variables that you want to display, and **Format** as **Float**. You can also modify floating-point variables by clicking on the value field, and entering new data in the usual format for floating-point numbers.

The floating-point function arguments are all passed by pointer reference. The calling function or task is responsible for declaring storage for the arguments themselves. Argument lists are ordered so that input arguments precede output arguments. In all cases, floating-point output arguments can match any of the input arguments to facilitate operations in place.

Binary Arithmetic Operators

Table 21 on page 72 lists the binary arithmetic operator functions.

Table 21. Binary Arithmetic Operators

Short Name	Function
add	`void fl_add(const float_type *arg1, const float_type *arg2, float_type *arg3);` Adds two floating-point numbers: (**arg3 = arg1 + arg2**).
sub	`void fl_sub(const float_type *arg1, const float_type *arg2, float_type *arg3);` Subtracts two floating-point numbers: (**arg3 = arg1 - arg2**).
mul	`void fl_mul(const float_type *arg1, const float_type *arg2, float_type *arg3);` Multiplies two floating-point numbers: (**arg3 = arg1 *arg2**).
div	`void fl_div(const float_type *arg1, const float_type *arg2, float_type *arg3);` Divides two floating-point numbers: (**arg3 = arg1 / arg2**).
max	`void fl_max(const float_type *arg1, const float_type *arg2, float_type *arg3);` Finds the max of two floating-point numbers: (**arg3 = max(arg1, arg2)**).
min	`void fl_min(const float_type *arg1, const float_type *arg2, float_type *arg3);` Finds the minimum of two floating-point numbers: (**arg3 = min(arg1, arg2)**).

Unary Arithmetic Operators

Table 22 lists the unary arithmetic operator functions.

Table 22. Unary Arithmetic Operators

Short Name	Function
abs	`void fl_abs(const float_type *arg1, float_type *arg2);` Returns the absolute value of a floating-point number: (**arg2 = abs(arg1)**).
neg	`void fl_neg(const float_type *arg1, float_type *arg2);` Returns the negative of a floating-point number: (**arg2 = - arg1**).
sqrt	`void fl_sqrt(const float_type *arg1, float_type *arg2);` Returns the square root of a floating-point number: (**arg2 = √ arg1**).

Short Name	Function
trunc	`void fl_trunc(const float_type *arg1, float_type *arg2);` Returns the whole number part of a floating-point number: (**arg2 = trunc(arg1)**). Truncation is towards zero. For example, **trunc(-3.45) = -3.0.**
floor	`void fl_floor(const float_type *arg1, float_type *arg2);` Returns the largest whole number less than or equal to a given floating-point number: (**arg2 = floor(arg1)**). Truncation is towards minus infinity. For example, **floor(-3.45) = -4.0.**
ceil	`void fl_ceil(const float_type *arg1, float_type *arg2);` Returns the smallest whole number greater than or equal to a given floating-point number: (**arg2 = ceil(arg1)**). Truncation is towards plus infinity. For example, **ceil(-3.45) = -3.0.**
round	`void fl_round(const float_type *arg1, float_type *arg2);` Returns the nearest whole number to a given floating-point number: (**arg2 = round(arg1)**). For example, **round(-3.45) = -3.0.**
mul2	`void fl_mul2(const float_type *arg1, float_type *arg2);` Multiplies a floating-point number by two: (**arg2 = arg1 * 2.0**).
div2	`void fl_div2(const float_type *arg1, float_type *arg2);` Divides a floating-point number by two: **arg2 = arg1 / 2.0**).

Comparison Operators

Table 23 lists the comparison operator functions.

Table 23. Comparison Operators

Short Name	Function

eq	`boolean fl_eq(const float_type *arg1,`
	`  const float_type *arg2);`
	Returns **TRUE** if the first argument is equal to the second argument, otherwise **FALSE**: (**arg1 == arg2**).

ne	`boolean fl_ne(const float_type *arg1,`
	`  const float_type *arg2);`
	Returns **TRUE** if the first argument is not equal to the second argument, otherwise **FALSE**: (**arg1 != arg2**).

Short Name	Function
gt	`boolean fl_gt(const float_type *arg1, const float_type *arg2);` Returns **TRUE** if the first argument is greater than the second argument, otherwise **FALSE**: (**arg1 > arg2**).
lt	`boolean fl_lt(const float_type *arg1, const float_type *arg2);` Returns **TRUE** if the first argument is less than the second argument, otherwise **FALSE**: (**arg1 < arg2**).
ge	`boolean fl_ge(const float_type *arg1, const float_type *arg2);` Returns **TRUE** if the first argument is greater than or equal to the second argument, otherwise **FALSE**: (**arg1 >= arg2**).
le	`boolean fl_le(const float_type *arg1, const float_type *arg2);` Returns **TRUE** if the first argument is less than or equal to the second argument, otherwise **FALSE**: (**arg1 <= arg2**).
cmp	`int fl_cmp(const float_type *arg1, const float_type *arg2);` Returns +1 if the first argument is greater than the second argument, -1 if it is less, or 0 if it is equal.

Miscellaneous Floating-Point Functions

Table 24 lists miscellaneous floating-point functions.

Table 24. Floating-Point Functions

Short Name	Function
sign	`int fl_sign(const float_type *arg);` Sign function, returns +1 if the argument is positive, 0 if the argument is zero, or -1 if the argument is negative.
rand	`void fl_rand(float_type *arg);` Returns a random number uniformly distributed in the range [0.0, 1.0) – that is, including the number 0.0, but not including the number 1.0.

Floating-Point to/from Integer Conversions

Table 25 on page 75 lists the conversion functions for floating-point and integer numbers.

Table 25. Floating-Point Conversion Functions

Short Name	Function
to slong	`signed long fl_to_slong(const float_type *arg);` Converts a floating-point number to a Neuron C signed long integer (range -32,768 to +32,767). Truncation is towards zero. For example, **fl_to_slong(-4.56) = -4**. If the closest integer is desired, call **fl_round( )** before calling **fl_to_slong( )**.
to ulong	`unsigned long fl_to_ulong(const float_type *arg);` Converts a floating-point number to a Neuron C unsigned long integer (range 0 to 65,535). Truncation is towards zero. For example, **fl_to_ulong(4.56) = 4**. If the closest integer is desired, call **fl_round( )** before calling **fl_to_ulong( )**.
to s32	`void fl_to_s32(const float_type *arg1, void *arg2);` Converts a floating-point number to a signed 32-bit integer (range ±2,147,483,647). The second argument is the address of a four-byte array, compatible with the signed 32-bit integer type **s32_type**. Truncation is towards zero. For example, **fl_to_s32(-4.56) = -4**. If the closest integer is desired, call **fl_round( )** before calling **fl_to_s32( )**.
from slong	`void fl_from_slong(signed long arg1, float_type *arg2);` Converts a Neuron C signed long integer (range -32,768 to +32,767) to a floating-point number.

from ulong	`void fl_from_ulong(unsigned long arg1,`
	`  float_type *arg2);`
	Converts a Neuron C unsigned long integer (range 0 to +65,535) to a floating-point number.

from s32	`void fl_from_s32(const void *arg1, float_type *arg2);` Converts a signed 32-bit number (range ±2,147,483,647) to a floating-point number. The first argument is the address of a four-byte array.

Conversion of Floating-Point to ASCII String

Table 26 on page 76 lists the conversion functions for floating-point numbers to ASCII strings.

Table 26. Conversions of Floating-Point Numbers to ASCII Strings

<table>
<tr><th>Short Name</th><th>Function</th></tr>
<tr><td>to ascii</td><td>
<pre>void fl_to_ascii(const float_type *arg1, char *arg2,
            int decimals, unsigned buf_size);</pre>
Converts a floating-point number *arg1 to an ASCII string followed by a terminating NUL character. The decimals value is the required number of decimal places after the point. The buf_size value is the length of the output buffer pointed to by arg2, including the terminating null. If possible, the number is converted using non-scientific notation, for example [-]xxx.xxxxx. If the result would not fit in the buffer provided, the number is converted using scientific notation, for example [-]x.xxxxxxE[-]nn. This function uses repeated multiplication and division, and can be time-consuming, depending on the input data. If decimals is 0, the buffer includes a trailing decimal point. If decimals is -1, there is no trailing decimal point. The number is rounded to the specified precision.
Example: Converting the number -12.34567, with a buf_size of 10.
<table>
<tr><th>decimals</th><th>output string</th></tr>
<tr><td>5</td><td>-12.34567</td></tr>
<tr><td>4</td><td>-12.3457</td></tr>
<tr><td>3</td><td>-12.346</td></tr>
<tr><td>2</td><td>-12.35</td></tr>
<tr><td>1</td><td>-12.3</td></tr>
<tr><td>0</td><td>-12.</td></tr>
<tr><td>-1</td><td>-12</td></tr>
</table>
</td></tr>
<tr><td>to ascii fmt</td><td>
<pre>void fl_to_ascii_fmt(const float_type *arg1, char *arg2,
            int decimals, unsigned buf_size,
            format_type format);</pre>
Converts the *arg1 floating-point number to an ASCII string followed by a terminating null. This function operates in the same way as fl_to_ascii(), except that the caller specifies the output format. The format parameter can be set to FMT_DEFAULT, FMT_FIXED or FMT_SCIENTIFIC to specify the default conversion (same as fl_to_ascii()), non-scientific notation or scientific notation respectively.
</td></tr>
</table>

Conversion of ASCII String to Floating-Point

Table 27 lists the conversion functions for an ASCII srtring to a floating-point number.

Table 27. Conversions of ASCII Strings to Floating-Point Numbers

Short Name	Function
`from ascii`	void fl_from_ascii(const char *arg1, float_type *arg2); Converts an ASCII string to a floating-point number. The conversion stops at the first invalid character in the input buffer—there is no error notification. The acceptable format is the following: **[+/-][xx][.][xxxxx][E/e[+/-]nnn]** **Examples:** 0, 1, .1, 1.2, 1E3, 1E-3, -1E1 There should be no more than nine significant digits in the mantissa portion of the number, or else the results are unpredictable. A significant digit is a digit following any leading zeroes. Embedded spaces within the number are also not allowed. This routine uses repeated multiplication and division, and can be time-consuming, depending on the input data. **Examples:** 0.00123456789E4 // is acceptable 123.4567890 // is not acceptable 123 E4 // is not acceptable The value 123.4567890 is not acceptable because it has 10 significant digits, and the value 123 E4 is not acceptable because it has an embedded space.

Floating-Point Performance

Table 28 lists times in milliseconds for the various functions in the floating-point library. They were measured using a Series 3100 Neuron Chip with a 10 MHz input clock. These values scale with faster or slower input clocks. The measurements are maximums and averages over random data logarithmically distributed in the range 0.001 to 1,000,000.

Table 28. Floating-Point Function Performance

Function	Maximum	Average
Add	0.56	0.36
Subtract	0.71	0.5

Function	Maximum	Average
Multiply	1.61	1.33
Divide	2.43	2.08
Square Root	10.31	8.89
Multiply/Divide by two	0.15	0.13
Maximum	0.61	0.53
Minimum	0.66	0.60
Integer Floor	0.25	0.21
Integer Ceiling	0.92	0.63
Integer Rounding	1.17	1.01
Integer Truncation	0.23	0.17
Negation	0.10	0.08
Absolute Value	0.10	0.08
Arithmetic Comparison	0.18	0.09
Conversion to ASCII	22.37	12.49
Conversion from ASCII	27.54	22.34
Conversion to 16-bit integer	2.84	1.03
Conversion from 16-bit integer	2.58	0.75
Conversion to 32-bit integer	5.60	2.71
Conversion from 32-bit integer	0.99	0.72
Random number generation	2.43	0.43
Sign of number	0.02	0.02

Using the NXT Neuron C Extended Arithmetic Translator

You can use the NXT Neuron C Extended Arithmetic Translator to create initializers for signed 32-bit integers and floating-point variables in a Neuron C program. To use the NXT translator, open a Windows command prompt and enter the following command:

nxt *input-file output-file*

(where *input-file* contains Neuron C variable definitions)

The source file can contain only one variable per line. Initializers of **float_type**, and **SNVT_<xxx>_f** variables are converted appropriately.

The output file is generated with properly converted initializers. Unaffected lines are output unchanged. The output file can be included in a Neuron C application with the **#include** directive. The output file is overwritten if it exists and was generated originally by this program.

In some cases, such as for **struct**s and **typedef**s, the translator cannot identify signed 32-bit or floating-point initializers. These can be identified by adding 's' or 'S' (for signed 32-bit integers), or 'f' or 'F' (for floating-point values) to the end of the constant.

As an example, if the input file contains the following statements:

```
s32_type var1 = 12345678;
struct_type var2 = {0x5, "my_string", 3333333S};
float_type var1 = 3.66;
struct_type var2 = {5.66f, 0x5, "my_string"};
```

then the output file will contain the following:

```
s32_type var1 = {0x00,0xbc,0x61,0x4e} /* 12345678 */;
struct_type var2 = {0x5,
   "my_string", {0x00,0x32,0xdc,0xd5} /* 3333333 */};

float_type var1 = {0,0x40,0,0x6a,0x3d71} /* 3.66 */;
struct_type var2 = {{0,0x40,1,0x35,0x1eb8} /* 5.66 */, 0x5,
   "my_string"};
```

Note: Users of the NodeBuilder Development Tool can also use Code Wizard to generate initializer data for **s32_type** and **float_type** network variables or configuration properties.

Function Directory

The following sections list the Neuron C functions alphabetically, providing relevant syntax information and a detailed description of each function.

abs( ) *Built-in Function*

The **abs( )** built-in function returns the absolute value of *a*. The argument *a* can be of **short** or **long** type. The return type is **unsigned short** if *a* is **short**, or **unsigned long** if *a* is **long**.

Syntax

type **abs (***a***);**

Example

```
int i;
long l;

void f(void)
{
   i = abs(-3);
   l = abs(-300);
}
```

access_address( ) *Function*

The **access_address( )** function returns a **const** pointer to the address structure that corresponds to the index parameter. This pointer can be stored, used to perform a structure copy, or used in other ways common to C pointers, except that the pointer cannot be used for writes.

See the ISO/IEC 14908 (ANSI/EIA/CEA-709.1) *Control Network Specification* for a description of the data structure.

Syntax

#include <access.h>
const address_struct *access_address (unsigned *index***);**

Example

```
#include <access.h>
address_struct addr_copy;

void f(void)
{
   addr_copy = *(access_address(2));
}
```

access_alias( ) *Function*

The **access_alias( )** function returns a **const** pointer to the alias structure that corresponds to the index parameter. This pointer can be stored, used to perform a structure copy, or used in other ways common to C pointers, except that the pointer cannot be used for writes.

The Neuron 3120 Chip with version 4 firmware does not support aliasing.

See the ISO/IEC 14908 (ANSI/EIA/CEA-709.1) *Control Network Specification* for a description of the data structure.

Series 6000 chips and version 21 Neuron firmware introduce support for an extended address table, which requires a extended alias configuration structure to accommodate the potentially larger address table index associated with the

alias. The standard Neuron C *access.h* include file defines both the traditional alias configuration structure and the extended form (alias_struct, alias_struct_ex). An *ALIAS_STRUCT_TYPE* preprocessor definition is supplied which equates to the correct type of the alias configuration structure for the current compilation target.

Syntax

#include <access.h>
const ALIAS_STRUCT_TYPE *access_alias (unsigned *index***);**

Example

```
#include <access.h>
ALIAS_STRUCT_TYPE alias_copy;
void f(void)
{
   alias_copy = *(access_alias(2));
}
```

access_domain( ) *Function*

The **access_domain( )** function returns a **const** pointer to the domain structure that corresponds to the index parameter. This pointer can be stored, used to perform a structure copy, or used in other ways common to C pointers, except that the pointer cannot be used for writes.

See the ISO/IEC 14908 (ANSI/EIA/CEA-709.1) *Control Network Specification* for a description of the data structure.

Syntax

#include <access.h>
const domain_struct *access_domain (unsigned *index***);**

Example

```
#include <access.h>
domain_struct domain_copy;

void f(void)
{
   domain_copy = *(access_domain(0));
}
```

access_nv( ) *Function*

The **access_nv( )** function returns a **const** pointer to the network variable configuration structure that corresponds to the index parameter. This pointer can be stored, used to perform a structure copy, or used in other ways common to Neuron C pointers, except that the pointer cannot be used for writes.

See the ISO/IEC 14908 (ANSI/EIA/CEA-709.1) *Control Network Specification* for a description of the data structure.

Series 6000 chips and version 21 Neuron firmware introduce support for an extended address table, which requires a extended network variable configuration structure to accommodate the potentially larger address table index associated with the network variable. The standard Neuron C *access.h* include file defines both the traditional network variable configuration structure and the extended form (nv_struct, nv_struct_ex). An *NV_STRUCT_TYPE* preprocessor definition is supplied which equates to the correct type of the network variable configuration structure for the current compilation target.

Syntax

#include <access.h>
const NV_STRUCT_TYPE *access_nv (unsigned *index***);**

Example

```
#include <access.h>
network output SNVT_amp nvoAmpere;
NV_STRUCT_TYPE nv_copy;

void f(void)
{
   nv_copy = *(access_nv(nv_table_index(nvoAmpere));
}
```

addr_table_index( ) *Built-in Function*

The **addr_table_index( )** built-in function is used to determine the address table index of a message tag as allocated by the Neuron C compiler. The returned value is in the range of 0 to 14.

The Neuron C compiler does not allow this function to be used for a non-bindable message tag (that is, a message tag declared with the **bind_info(nonbind)** option).

Syntax

unsigned int addr_table_index (*message-tag***);**

Example

```
unsigned mt_index;
msg_tag my_mt;

void f(void)
{
   mt_index = addr_table_index(my_mt);
}
```

ansi_memcpy( ) *Function*

The **ansi_memcpy( )** function copies a block of *len* bytes from *src* to *dest*. It returns the first argument, which is a pointer to the *dest* memory area. This function cannot be used to copy overlapping areas of memory, or to write into EEPROM or flash memory.

The **ansi_memcpy( )** function as implemented here conforms to the ANSI definition for **memcpy( )**, as it returns a pointer to the destination array. See **memcpy( )** for a non-conforming implementation (does not have a return value), which is a more efficient implementation if the return value is not needed. See also **ansi_memset( )**, **eeprom_memcpy( )**, **memccpy( )**, **memchr( )**, **memcmp( )**, **memcpy( )**, and **memset( )**.

Syntax

#include <mem.h>
void *ansi_memcpy (void *_dest_, void *_src_, unsigned long _len_);

Example

```
#include <mem.h>

unsigned buf[40];
unsigned *p;

void f(void)
{
   p = ansi_memcpy(buf, "Hello World", 11);
}
```

ansi_memset( ) *Function*

The **ansi_memset( )** function sets the first *len* bytes of the block pointed to by *p* to the character *c*. It also returns the value *p*. This function cannot be used to write into EEPROM or flash memory.

The **ansi_memset( )** function as implemented here conforms to the ANSI definition for **memset( )**, as it returns the pointer *p*. See **memset( )** for a non-conforming implementation (does not have a return value), which is a more efficient implementation if the return value is not needed. See also **ansi_memcpy( )**, **eeprom_memcpy( )**, **memccpy( )**, **memchr( )**, **memcmp( )**, and **memcpy( )**.

Syntax

#include <mem.h>
void *ansi_memset (void *_p_, int _c_, unsigned long _len_);

Example

```
#include <mem.h>

unsigned target[20];
```

```
unsigned *p;

void f(void)
{
   p = ansi_memset(target, 0, 20);
}
```

application_restart( ) *Function*

The **application_restart( )** function restarts the application program running on the application processor only. The network, MAC, and interrupt processors are unaffected. When an application is restarted, the **when(reset)** event becomes TRUE.

Recommendation: For applications that include interrupt tasks, call **interrupt_control(0)** to suspend the interrupt processing prior to restarting the application.

Syntax

#include <control.h>
void application_restart (void);

Example

```
#define MAX_ERRS 50 int error_count;
...
when (error_count > MAX_ERRS)
{
   application_restart();
}
```

bcd2bin( ) *Built-in Function*

The **bcd2bin( )** built-in function converts a binary-coded decimal structure to a binary number. The structure definition is built into the compiler. The most significant digit is *d1*. Note that *d1* should always be 0.

Syntax

unsigned long bcd2bin (struct bcd *a);

```
struct bcd {
   unsigned d1:4,
            d2:4,
            d3:4,
            d4:4,
            d5:4,
            d6:4;
};
```

Example

```
void f(void)
```

```
{
   struct bcd digits;
   unsigned long value;

   memset(&digits, 0, 3);
   digits.d3=1;
   digits.d4=2;
   digits.d5=3;
   digits.d6=4;
   value = bcd2bin(&digits);
      //value now contains 1234
}
```

bin2bcd( ) *Built-in Function*

The **bin2bcd( )** built-in function converts a binary number to a binary-coded decimal structure.

Syntax

void bin2bcd (unsigned long *value***, struct bcd ****p***);**

For a definition of **struct bcd**, see *bcd2bin*, above.

Example

```
void f(void)
{
   struct bcd digits;
   unsigned long value;
   ...
   value = 1234;
   bin2bcd(value, &digits);
   // digits.d1 now contains 0
   // digits.d2 now contains 0
   // digits.d3 now contains 1
   // digits.d4 now contains 2
   // digits.d5 now contains 3
   // digits.d6 now contains 4
}
```

clear_status( ) *Function*

The **clear_status( )** function clears a subset of the information in the status structure (see the **retrieve_status( )** function on page 138). The information cleared is the statistics information, the reset cause register, and the error log.

Syntax

#include <status.h>
void clear_status (void);

Example

```
when (timer_expires(statistics_reporting_timer))
{
   retrieve_status(status_ptr);   // get current statistics
   report_statistics(status_ptr); // check it all out
   clear_status();
}
```

clr_bit( ) *Function*

The **clr_bit( )** function clears a bit in a bit array pointed to by *array*. Bits are numbered from left to right in each byte, so that the first bit in the array is the most significant bit of the first byte in the array. Like all arrays in C, this first element corresponds to index 0 (*bitnum* 0). When managing a number of bits that are all similar, a bit array can be more code-efficient than a series of bitfields because the array can be accessed using an array index rather than separate lines of code for each bitfield. See also the **set_bit( )** function and the **tst_bit( )** function.

Syntax

#include <byte.h>
void clr_bit (void ****array*, unsigned** *bitnum***);**

Example

```
#include <byte.h>

unsigned short a[4];

void f(void)
{
   memset(a, 0xFF, 4);  // Sets all bits
   clr_bit(a, 4);       // Clears a[0] to 0xF7 (5th bit)
}
```

crc8( ) *Function*

The **crc8( )** function iteratively calculates an 8-bit cyclic redundancy check (CRC) over an array of data using the following polynomial:

$$x^8 + x^5 + x^4 + 1$$

This function is useful in conjunction with the support for the Touch I/O model, but can also be used whenever a CRC is needed.

Syntax

#include <stdlib.h>
unsigned crc8 (unsigned *crc***, unsigned** *new-data***);**

Example

```
#include <stdlib.h>

unsigned data[SIZE];

void f(void)
{
   unsigned i; // Or 'unsigned long' depending on SIZE
   unsigned crc = 0;
   for (i = 0;  i < SIZE;  ++i) {
      // Combine partial CRC with next data byte
      crc = crc8(crc, data[i]);
   }
}
```

---

crc16( ) *Function*

The **crc16( )** function iteratively calculates a 16-bit cyclic redundancy check (CRC) over an array of data bytes using the following polynomial:

$$x^{16} + x^{15} + x^2 + 1$$

This function is useful in conjunction with the support for the Touch I/O model, but can also be used whenever a CRC is needed.

Syntax

#include <stdlib.h>
unsigned long crc16 (unsigned long *crc*, **unsigned** *new_data*);

Example

```
#include <stdlib.h>

unsigned data[SIZE];

void f(void)
{
   unsigned i; // Or 'unsigned long' depending on SIZE
   long crc = 0;
   for (i = 0;  i < SIZE;  ++i) {
      // Combine partial CRC with next data value
      crc = crc16(crc, data[i]);
   }
}
```

crc16_ccitt( ) *Function*

The **crc16_ccitt( )** function iteratively calculates a 16-bit Comité Consultatif International Téléphonique et Télégraphique[1] (CCITT) cyclic redundancy check (CRC) over an array of data bytes using the following polynomial:

$$x^{16} + x^{12} + x^5 + 1$$

Apart from using a different polynomial, this function differs from the **crc16( )** function in that this function operates on a data array, which is generally faster.

This function is useful in conjunction with the support for the Touch I/O model, but can also be used whenever a CRC is needed.

Syntax

#include <stdlib.h>

extern system far unsigned long crc16_ccitt (unsigned long *crc_in***,**
const unsigned ****sp*, unsigned** *len***);**

crc-in	Specifies the input seed for the CRC calculation.
sp	Specifies a pointer to the buffer that contains the data to be checked.
len	Specifies the length of the data buffer, 1 to 255 bytes. A value of 0 represents 256 bytes.

Example

```
#include <stdlib.h>

unsigned data[SIZE];

void f(void)
{
   long seed = 0x04C1;
   long crc crc16_ccitt(seed, *data, SIZE);
}
```

delay( ) *Function*

The **delay( )** function allows an application to suspend processing for a given time. This function provides more precise timing than can be achieved with application timers.

Table 29 lists the formulas for determining the duration of the delay.

[1] International Telegraph and Telephone Consultative Committee.

Table 29. Delay Values for Various System Clock Rates

Series 3100 Input Clock	Series 5000 and 6000 System Clock	Delay (in microseconds)
—	80 MHz	0.0375*(max(1,min(65535,count*16))*42+221)
—	40 MHz	0.075*(max(1,min(65535,count*8))*42+204)
—	20 MHz	0.15*(max(1,min(65535,count*4))*42+187)
40 MHz	—	0.15*(max(1,min(65535,count*4))*42+176)
—	10 MHz	0.3*(max(1,min(65535,count*2))*42+170)
20 MHz	—	0.3*(max(1,min(65535,count*2))*42+159)
—	5 MHz	0.6*(max(1,count)*42+139)
10 MHz	—	0.6*(max(1,count)*42+128)
6.5536 MHz	—	0.9155*((max(1,floor(count/2))*42+450)
5 MHz	—	1.2*((max(1,floor(count/2))*42)+155)
2.5 MHz	—	2.4*((max(1,floor(count/4))*42)+172)
1.25 MHz	—	4.8*((max(1,floor(count/8))*42)+189)
625 kHz	—	9.6*((max(1,floor(count/16))*42)+206)

For example, for a Series 3100 device with a 10 MHz input clock, the formula above yields durations in the range of 88.8 microseconds to 840 milliseconds by increments of 25.2 microseconds. Using a count greater than 33,333 (for a Series 3100 device at 10 MHz) could cause the watchdog timer to time out. See also the **scaled_delay( )** function, which generates a delay that scales with the input clock.

Note: Because of the multiplier used by **delay( )**, and because the watchdog timer timeout scales with the input clock (for Series 3100 devices), there is the potential for a watchdog timeout at 20 MHz and 40 MHz operation. The maximum inputs to **delay( )** for Series 3100 devices are 16666 at 20 MHz and 8333 at 40 MHz. Timing intervals greater than the watchdog interval must be done through software timers or through a user routine that calls **delay( )** and **watchdog_update( )** in a loop. Also see **msec_delay( )**.

Syntax

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

count	A value between 1 and 33333. The formula for determining the duration of the delay is based on the count parameter and the input clock (see above).

Example

```
IO_4 input bit io_push_button;
boolean debounced_button_state;

when(io_changes(io_push_button))
{
   delay(400); //delay approx. 10ms at any clock rate
   debounced_button_state=(boolean)io_in(io_push_button);
}
```

eeprom_memcpy( ) *Function*

The **eeprom_memcpy( )** function copies a block of *len* bytes from *src* to *dest*. It does not return any value. This function supports destination addresses that reside in EEPROM or flash memory, where the normal **memcpy( )** function does not. This function supports a maximum length of 255 bytes.

See also **ansi_memcpy( )**, **ansi_memset( )**, **memccpy( )**, **memchr( )**, **memcmp( )**, **memcpy( )**, and **memset( )**.

Syntax

void eeprom_memcpy (void ****dest*, void ****src***, unsigned short** *len***);**

Example

```
#pragma relaxed_casting_on
eeprom far unsigned int widget[100];
far unsigned int ram_buf[100];

void f(void)
{
   eeprom_memcpy(widget, ram_buf, 100);
}
```

Because the compiler regards a pointer to a location in EEPROM or FLASH as a pointer to constant data, **#pragma relaxed_casting_on** must be used to allow for the **const** attribute to be removed from the first argument, using an implicit or explicit cast operation. A compiler warning still occurs as a result of the **const** attribute being removed by cast operation. See the discussion of the **eeprom_memcpy( )** function in the *Memory Management* chapter of the *Neuron C Programmer's Guide*.

error_log( ) *Function*

The **error_log( )** function writes the error number into a dedicated location in EEPROM. Network tools can use the *Query Status* network diagnostic command

to read the last error. The NodeBuilder Neuron C debuggers maintain a log of the last 25 error messages.

The *Neuron Tools Errors Guide* lists the error numbers that are used by the Neuron Chip firmware. These are in the range 128 ... 255. The application can use error numbers 1 ... 127.

Syntax

#include <control.h>
void error_log (unsigned int *error_num***);**

error_num	A decimal number between 1 and 127 representing an application-defined error.

Example

```
#define MY_ERROR_CODE 1
...
when (nv_update_fails)
{
   error_log(MY_ERROR_CODE);
}
```

fblock_director( ) *Built-in Function*

The **fblock_director( )** built-in function calls the director function associated with the functional block whose global index is *index*. If the *index* is out of range, or the functional block does not have a director function, the **fblock_director( )** built-in function does nothing except return. Otherwise, it calls the director function associated with the functional block specified, passes the *cmd* parameter on to that director function, and returns when the called director function completes.

Syntax

void fblock_director (unsigned int *index***, int** *cmd***);**

index	A decimal number between 0 and 254 representing a functional block global index.
cmd	A decimal number between –128 and 127, interpreted as an application-specific command.

Example

```
void f(void)
{
   fblock_director(myFB::global_index, 3);
}
```

Floating-Point Support *Functions*

void fl_abs (const float_type ****arg1*, float_type ****arg2***);**

void fl_add (const float_type ****arg1*, const float_type ****arg2***, float_type ****arg3***);**

void fl_ceil (const float_type ****arg1*, float_type ****arg2***);**

int fl_cmp (const float_type ****arg1*, const float_type ****arg2***);**

void fl_div (const float_type ****arg1*, const float_type ****arg2***, float_type ****arg3***);**

void fl_div2 (const float_type ****arg1*, float_type ****arg2***);**

void fl_eq (const float_type ****arg1*, const float_type ****arg2***);**

void fl_floor (const float_type ****arg1*, float_type ****arg2***);**

void fl_from_ascii (const char ****arg1*, float_type ****arg2***);**

void fl_from_s32 (const void ****arg1*, float_type ****arg2***);**

void fl_from_slong (signed long *arg1***, float_type ****arg2***);**

void fl_from_ulong (unsigned long *arg1***, float_type ****arg2***);**

void fl_ge (const float_type ****arg1*, const float_type ****arg2***);**

void fl_gt (const float_type ****arg1*, const float_type ****arg2***);**

void fl_le (const float_type ****arg1*, const float_type ****arg2***);**

void fl_lt (const float_type ****arg1*, const float_type ****arg2***);**

void fl_max (const float_type ****arg1*, const float_type ****arg2***, float_type ****arg3***);**

void fl_min (const float_type ****arg1*, const float_type ****arg2***, float_type ****arg3***);**

void fl_mul (const float_type ****arg1*, const float_type ****arg2***, float_type ****arg3***);**

void fl_mul2 (const float_type ****arg1*, float_type ****arg2***);**

void fl_ne (const float_type ****arg1*, const float_type ****arg2***);**

void fl_neg (const float_type ****arg1*, float_type ****arg2***);**

void fl_rand (float_type ****arg1*);**

void fl_round (const float_type ****arg1*, float_type ****arg2***);**

int fl_sign (const float_type ****arg1*);**

void fl_sqrt (const float_type ****arg1*, float_type ****arg2***);**

void fl_sub (const float_type ****arg1*, const float_type ****arg2***, float_type ****arg3***);**

void fl_to_ascii (const float_type ****arg1*, char ****arg2***, int** *decimals*,
unsigned *buf-size*);

void fl_to_ascii_fmt (const float_type ****arg1*, char ****arg2***, int** *decimals*,
unsigned *buf-size*, **format_type** *format*);

void fl_to_s32 (const float_type ****arg1*, void ****arg2***);**

signed long fl_to_slong (const float_type ****arg2*);**

unsigned long fl_to_ulong (const float_type ****arg2*);**

void fl_trunc (const float_type ****arg1*, float_type ****arg2***);**

These functions are described in *Floating-Point Support Functions* on page 68.

flush( ) *Function*

The **flush( )** function causes the Neuron firmware to monitor the status of all outgoing and incoming messages.

The **flush_completes** event becomes TRUE when all outgoing transactions have been completed and no more incoming messages are outstanding. For unacknowledged messages, "completed" means that the message has been fully transmitted by the MAC layer. For acknowledged messages, "completed" means that the completion code has been processed. In addition, all network variable updates must be propagated before the flush can be considered complete.

Syntax

#include <control.h>
void flush (boolean *comm_ignore***);**

comm_ignore	Specify TRUE if the Neuron firmware should ignore any further incoming messages. Specify FALSE if the Neuron firmware should continue to accept incoming messages.

Example

```
boolean nothing_to_do;
...
when (nothing_to_do)
{
   // Getting ready to sleep
   ...
   flush(TRUE);
}

when (flush_completes)
{
   // Go to sleep
   nothing_to_do = FALSE;
   sleep();
}
```

flush_cancel( ) *Function*

The **flush_cancel( )** function cancels a flush in progress.

Syntax

#include <control.h>
void flush_cancel (void);

Example

```
boolean nothing_to_do;
...
when (nv_update_occurs)
{
   if (nothing_to_do) {
   // was getting ready to sleep but received an input NV
      nothing_to_do = FALSE;
      flush_cancel();
   }
}
```

flush_wait( ) *Function*

The **flush_wait( )** function causes an application program to enter preemption mode, during which all outstanding network variable and message transactions are completed. When a program switches from asynchronous to direct event processing, **flush_wait( )** is used to ensure that all pending asynchronous transactions are completed before direct event processing begins.

During preemption mode, only pending completion events (for example, **msg_completes** or **nv_update_fails**) and pending response events (for example, **resp_arrives** or **nv_update_occurs**) are processed. When this processing is complete, **flush_wait( )** returns. The application program can now process network variables and messages directly and need not concern itself with outstanding completion events and responses from earlier transactions.

Syntax

#include <control.h>
void flush_wait (void);

Example

```
msg_tag TAG1;
network output SNVT_volt nvoVoltage;

when (...)
{
   msg_out.tag = TAG1;
   msg_out.code = 3;
   msg_send();
   flush_wait();

   nvoVoltage = 3;
   while (TRUE) {
      post_events();
      if (nv_update_completes(nvoVoltage)) break;
   }
}

when (msg_completes(TAG1))
{
   ...
```

```
}
```

get_current_nv_length( ) Function

The **get_current_nv_length( )** function returns the currently defined length of a network variable in bytes, given the global index, *netvar-index*, of that network variable. This is useful when working with changeable-type network variables.

In Neuron C Version 2.3, the get_current_nv_length() API is functionally equivalent to reading the *nv_len* property. Earlier versions of Neuron C limited the *nv_len* property to changeable-type network variables.

Syntax

unsigned int get_current_nv_length (unsigned int *netvar-index***);**

netvar-index	The global index for the network variable whose current length is required. The global index can be obtained through the **global_index** property, or through the **nv_table_index( )** built-in function.

Example

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

void f(void)
{
    unsigned currentLength =
        get_current_nv_length(nvoVolt::global_index);
}
```

Recommendation: Use the **sizeof( )** operator to obtain the length of a network variable that is not a changeable type, or to obtain the length of the initial type of a changeable type network variable.

With Neuron C version 2.3 or better, you can use the *nv_len* property or the get_current_nv_length() API to obtain the current length of all network variables, regardless of their changeable-type attribute.

get_declared_nv_length( ) Function

This API is added with version 2.3 of the Neuron C language.

The **get_declared_nv_length( )** function returns the initial length of a network variable in bytes, given its global index, *netvar-index*. The function returns 255 (0xFF) to indicate that the declared (initial) length of the given network variable should be obtained from the *nv_fixed* system data structure instead.

This API is used in the implementation of changeable-type network variables on applications which implement where larger than 31 byte network variables.

Syntax

unsigned int get_declared_nv_length (unsigned int *netvar-index***);**

netvar-index	The global index for the network variable whose current length is required. The global index can be obtained through the **global_index** property, or through the **nv_table_index( )** built-in function.

Example

```
#pragma system_image_extensions nv_length_override

unsigned _RESIDENT get_nv_length_override(unsigned nvIndex)

{

#if defined(_SUPPORT_LARGE_NV)
  unsigned uResult = get_declared_nv_length(nvIndex);
#else
  unsigned uResult = 0xFF;
#endif


  // TO DO: add code to return the current length of the network variable

  // with index "nvIndex."

  // Example code follows:

  //

  // switch (nvIndex) {

  //   case nviChangeableNv::global_index:

  //     if (nviChangeableNv::cpNvType.type_category != NVT_CAT_INITIAL

  //     && nviChangeableNv::cpNvType.type_category != NVT_CAT_NUL) {

  //       uResult = nviChangeableNv::cpNvType.type_length;

  //     }

  //     break;

  // } // switch


  return uResult;

}
```

Recommendation: Use the **sizeof( )** operator to obtain the length of a network variable that is not a changeable type, or to obtain the length of the initial type of a changeable type network variable.

get_fblock_count( ) *Built-in Function*

The **get_fblock_count( )** built-in function is a compiler special function that returns the number of functional block (**fblock**) declarations in the program. For an array of functional blocks, each element counts as a separate **fblock** declaration.

Syntax

unsigned int get_fblock_count (void);

Example

```
unsigned numFBs;

void f(void)
{
   numFBs = get_fblock_count();
}
```

get_nv_count( ) *Built-in Function*

The **get_nv_count( )** built-in function is a special compiler function that returns the number of network variable declarations in the program. For each network variable array, each element counts as a separate network variable.

Syntax

unsigned int get_nv_count (void);

Example

```
network input SNVT_time_stamp nviTimeStamp[4];
unsigned numNVs;

void f(void)
{
   numNVs = get_nv_count();  // Returns '4' in this case
}
```

get_tick_count( ) *Function*

The **get_tick_count( )** function returns the current system time. The tick interval, in microseconds, is defined by the literal **TICK_INTERVAL**. This function is useful for measuring durations of less than 50 ms for a Series 3100 device at 40 MHz. For Series 3100 devices, the tick interval scales with the input clock. For Series 5000 and 6000 devices, the tick interval is fixed at (0.5 * **TICK_INTERVAL).**

Syntax

unsigned int get_tick_count (void);

Example

```
void f(void)
{
   unsigned int start, delta;
```

```
    start = get_tick_count();
    ...
    delta = get_tick_count() - start;
}
```

go_offline( ) *Function*

The **go_offline( )** function takes an application offline. This function call has the same effect on the device as receiving an *Offline* network management message. The offline request takes effect as soon as the task that called **go_offline( )** exits. When that task exits, the **when(offline)** task is executed and the application stops.

When an *Online* network management message is received, the **when(online)** task is executed and the application resumes execution.

When an application goes offline, all outstanding transactions are terminated. To ensure that any outstanding transactions complete normally, the application can call **flush_wait( )** in the **when(offline)** task.

Syntax

#include <control.h>
void go_offline (void);

Example

```
boolean maintenanceMode;

...

when (maintenanceMode)
{
   go_offline();
}

when (offline)
{
   // process shut-down command
   flush_wait();
   io_out(maintenanceLed, 0);
}

when (online) {
   ...
   // re-start suspended operation, I/O devices,
   // interrupts, etc
   io_out(maintenanceLed, 1);
}
```

go_unconfigured( ) *Function*

The **go_unconfigured( )** function puts the device into an unconfigured state. It also overwrites all the domain information, which clears authentication keys as well.

The **go_unconfigured( )** function can be used after detecting and logging a serious, unrecoverable error. Some security devices also call this function when they detect a attempt to tamper with the device, and thus render the device inoperational and erase the secret authentication keys.

Syntax

#include <control.h>
void go_unconfigured (void);

Example

```
void f() {
   ...
   if (unrecoverable) {
      error_log(MY_UNRECOVERABLE_ERROR);
      go_unconfigured();
   }
}
```

high_byte( ) *Built-in Function*

The **high_byte( )** built-in function extracts the upper single-byte value from the *a* double-byte operand. This function operates without regard to signedness. See also **low_byte( )**, **make_long( )**, and **swap_bytes( )**.

Syntax

unsigned short high_byte (unsigned long *a***);**

Example

```
short b;
long a;

void f(void)
{
   a = 258;    // Hex value 0x0102
   b = high_byte(a);    // b now contains the value 0x01
}
```

interrupt_control( ) *Built-in Function*

The **interrupt_control( )** built-in function enables or disables interrupts. You can call the **interrupt_control( )** function at any time to enable or disable one or more of the three interrupt types: I/O interrupts, timer/counter interrupts, or periodic system timer interrupts. This function applies only to the hardware interrupts provided by Series 5000 and 6000 devices, and is only available to an application that defines at least one interrupt task.

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.

The Neuron C Compiler automatically generates the interrupt_control() function when your application defines at least one interrupt task. Note that the compiler does not generate this API unnecessarily; if you use conditional compilation to control inclusion or exclusion of interrupt tasks in your application, you should use the same conditional compilation conditions to control calls to the *interrupt_control()* API.

Syntax

void interrupt_control(unsigned *irqSelect***);**

irqSelect 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 disables all interrupts. A value of -1 (0xFF) enables all interrupt tasks defined within the application. You can enable and disable interrupts at any time, as required for your application, but you cannot enable an interrupt type for which your application does not define an interrupt task.

All interrupts are disabled after device reset. An application that uses interrupts must enable these interrupts when it is ready, typically towards the end of the **reset** task. Enable interrupts only when the device is in the online state.

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.

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);
}
```

io_change_init( ) *Built-in Function*

The **io_change_init( )** built-in function initializes the I/O object for the **io_changes** event. If this function is not used, the I/O object's initial reference value defaults to 0.

Syntax

void io_change_init (*input-io-object-name* **[,** *init-value***]);**

input-io-object-name	Specifies the I/O object name, which corresponds to *io-object-name* in the I/O declaration.
init-value	Sets the initial reference value used by the **io_changes** event. If this parameter is omitted, the object's current value is used as the initial reference value. This parameter can be **short** or **long** as needed.

Example

```
IO_4 input ontime signal;

when (reset)
{
   // Set comparison value for 'signal'
   // to its current value
   io_change_init(signal);
}
...
when (io_changes(signal) by 10)
{
    ...
}
```

io_edgelog_preload( ) *Built-in Function*

The **io_edgelog_preload( )** built-in function is optionally used with the edgelog I/O model. The *value* parameter defines the maximum value, in units of the clock period, for each period measurement, and can be any value from 1 to 65535. If the period exceeds the maximum value, the **io_in( )** call is terminated.

The default maximum value is 65535, which provides the maximum timeout condition. By setting a smaller maximum value with this function, a Neuron C program can *shorten* the length of the timeout condition. This function need only be called once, but can be called multiple times to change the maximum value. The function can be called from a **when(reset)** task to automatically reduce the maximum count after every start-up.

If a preload value is specified, it must be added to the value returned by **io_in( )**. The resulting addition could cause an overflow, but this is normal.

Syntax

void io_edgelog_preload (unsigned long *value***);**

value A value between 1 and 65535 defining the maximum value for each period measurement.

Example

```
IO_4 input edgelog elog;

when (reset)
{
   io_edgelog_preload(0x4000);      // One fourth timeout
                                    // value: 16384
}
```

io_edgelog_single_preload( ) *Built-in Function*

The **io_edgelog_single_preload( )** built-in function is optionally used with the edgelog I/O model when declared with the **single_tc** option keyword. The *value* parameter defines the maximum value, in units of the clock period, for each period measurement, and can be any value from 1 to 65535. If the period exceeds the maximum value, the **io_in( )** call is terminated.

The default maximum value is 65535, which provides the maximum timeout condition. By setting a smaller maximum value with this function, a Neuron C program can *shorten* the length of the timeout condition. This function need only be called once, but can be called multiple times to change the maximum value. The function can be called from a **when(reset)** task to automatically reduce the maximum count after every start-up.

If a preload value is specified, it must be added to the value returned by **io_in( )**. The resulting addition could cause an overflow, but this is normal.

Syntax

void io_edgelog_single_preload (unsigned long *value***);**

value A value between 1 and 65535 defining the maximum value for each period measurement.

Example

```
IO_4 input edgelog single_tc elog;

when (reset)
{
   io_edgelog_single_preload(0x4000);
   // One fourth timeout value: 16384
}
```

io_idis( ) *Function*

For Series 3100 devices, the **io_idis( )** function disables the I/O interrupt used in the hardware support for the **sci** and **spi** I/O models. You can turn off interrupts when going offline or to assure that other time-critical application functions are not disturbed by SCI or SPI interrupts.

For Series 5000 and 6000 devices, the **io_idis( )** function disables all application interrupts. This function does not affect the I/O interrupt used in the hardware support for the **sci** and **spi** I/O models.

Syntax

void io_idis (void);

Example

```
when (...)
{
   io_idis();
}
```

io_iena( ) *Function*

For Series 3100 devices, the **io_iena( )** function enables the I/O interrupt used in the hardware support for the **sci** and **spi** I/O models. You can turn off interrupts when going offline or to assure that other time-critical application functions are not disturbed by SCI or SPI interrupts.

For Series and 6000 devices, the **io_iena( )** function enables all application interrupts. This function does not affect the I/O interrupt used in the hardware support for the **sci** and **spi** I/O models.

Syntax

void io_iena (void);

Example

```
when (...)
{
   io_iena();
}
```

io_in( ) *Built-in Function*

The **io_in( )** built-in function reads data from an input object.

The **<io_types.h>** include file contains optional type definitions for each of the I/O object types. The type names are the I/O object type name followed by "**_t**". For example **bit_t** is the type name for a **bit** I/O object.

The data type of the *return-value* is listed below for each object type.

Object Type	***Returned Data Type***
bit input	**unsigned short**
bitshift input	**unsigned long**
byte input	**unsigned short**

dualslope input	**unsigned long**
edgelog input	**unsigned short**
i2c	**unsigned short**
infrared input	**unsigned short**
leveldetect input	**unsigned short**
magcard input	**signed short**
magcard_generic input	**unsigned long**
magtrack1 input	**unsigned short**
muxbus input	**unsigned short**
neurowire master	**void**
neurowire slave	**unsigned short**
nibble input	**unsigned short**
ontime input	**unsigned long**
parallel	**void**
period input	**unsigned long**
pulsecount input	**unsigned long**
quadrature input	**signed long**
serial input	**unsigned short**
spi	**unsigned short**
totalcount input	**unsigned long**
touch	**void**
wiegand input	**unsigned short**

Syntax

return-value **io_in (***input-io-object-name* **[,** *args***]);**

return-value	The value returned by the function. See below for details.
input-io-object-name	The I/O object name, which corresponds to *io-object-name* in the I/O declaration.
args	Arguments, which depend on the I/O object type, as described below. Some of these arguments can also appear in the I/O object declaration. If specified in both places, the value of the function argument overrides the declared value for that call only. If the value is not specified in either the function argument or the declaration, the default value is used.

General

For all input objects except those listed below, the syntax is:

io_in (*input-obj***);**

The type of the *return-value* of the **io_in( )** call is listed in the table above.

bitshift

For **bitshift** input objects, the syntax is:

io_in (*bitshift-input-obj* [**,** *numbits*]**);**

numbits	The number of bits to be shifted in, from 1 to 127. Only the last 16 bits shifted in are returned. The unused bits are 0 if fewer than 16 bits are shifted in.

edgelog

For **edgelog** input objects, the syntax is:

io_in (*edgelog-input-obj***,** *buf***,** *count***);**

buf	A pointer to a buffer of **unsigned long** values.
count	The maximum number of values to be read.

The **io_in( )** call has an **unsigned short** *return-value* that is the actual number of edges logged.

ic2

For **i2c** I/O objects, the syntax is:

io_in (*i2c-io-obj***,** *buf***,** *addr***,** *count***);**

io_in (*i2c-io-obj***,** *buf***,** *addr***,** *count*, *stop***);**

buf	A (**void ***) pointer to a buffer.
addr	An **unsigned** short int I2C device address.
count	The number of bytes to be transferred.
stop	A Boolean value to specify whether the stop condition is asserted after the transfer.

The **io_in( )** call has a **boolean** *return-value* that indicates whether the transfer succeeded (TRUE) or failed (FALSE).

infrared

For **infrared** input objects, the syntax is:

io_in (*infrared-obj***,** *buf***,** *ct***,** *v1***,** *v2***);**

buf	A pointer to a buffer of unsigned short values.
ct	The maximum number of bits to be read.
v1	The maximum period value (an **unsigned long**). See the I/O model description in the *I/O Model Reference*.
v2	The threshold value (an **unsigned long**). See the infrared I/O model description in the *I/O Model Reference*.

The **io_in( )** call has an **unsigned short** *return-value* that is the actual number of bits read.

magcard

For **magcard** input objects, the syntax is:

io_in (*magcard-input-obj***,** *buf***);**

buf	A pointer to a 20 byte buffer of **unsigned short** bytes, which can contain up to 40 hex digits, packed 2 per byte.

The **io_in( )** call has a **signed short** *return-value* that is the actual number of hex digits read. A value of -1 is returned in case of error.

magcard_bitstream

For **magcard_bitstream** input objects, the syntax is:

io_in (*magcard-bitstream-input-obj, buf, count***);**

buf	A pointer to a buffer of **unsigned short** bytes, sufficient to hold the number of bits (packed 8-per-byte) to be read.
count	The number of data bits to be read.

The **io_in()** call has an **unsigned long** *return-value* that is the actual number of data bits read. This value is either identical to the count argument, or a smaller number that indicates a timeout event occurred during the read.

magtrack1

For **magtrack1** input objects, the syntax is:

io_in (*magtrack1-input-obj***,** *buf***);**

buf	A pointer to a 78 byte buffer of **unsigned short** bytes, which each contain a 6-bit character with parity stripped.

The **io_in( )** call has an **unsigned short** *return-value* that is the actual number of characters read.

muxbus

For **muxbus** I/O objects, the syntax is:

io_in (*muxbus-io-obj* [**,** *addr*]**);**

addr	An optional address to read. Omission of the address causes the firmware to reread the last address read or written (muxbus is a bi-directional I/O device).

neurowire

For **neurowire** I/O objects, the syntax is:

io_in (*neurowire-io-obj***,** *buf***,** *count***);**

buf	A (**void ***) pointer to a buffer.
count	The number of bits to be read.

The **io_in( )** call has an **unsigned short** *return-value* signifying the number of bits actually transferred for a **neurowire slave** object. For other neurowire I/O object types, the *return-value* is **void**. See the *Driving a Seven Segment Display with the Neuron Chip* engineering bulletin (part number 005-0014-01) for more information.

parallel

For **parallel** I/O objects, the syntax is:

io_in (*parallel-io-obj***,** *buf***);**

buf	A pointer to the **parallel_io_interface** structure.

serial

For **serial** input objects, the syntax is:

io_in (*serial-input-obj***,** *buf***,** *count***);**

buf	A (**void ***) pointer to a buffer.
count	The number of bytes to be read (from 1 to 255).

spi

For **spi** I/O objects, the syntax is:

io_in (*spi-io-obj, buf, len***);**

buf	A pointer to a buffer of data bytes for the bidirectional data transfer.
len	An **unsigned short** number of bytes to transfer.

The **io_in()** function has an **unsigned short** *return-value* that indicates the number of bytes transferred on the previous transfer. Calling **io_in()** for a **spi** object is the same as calling **io_out()**. In either case, the data in the buffer is output and simultaneously replaced by new input data.

touch

For **touch** I/O objects, the syntax is:

io_in (*touch-io-obj***,** *buf***,** *count***);**

buf	A (**void ***) pointer to a buffer.
count	The number of bytes to be transferred.

wiegand

For **wiegand** input objects, the syntax is:

io_in (*wiegand-obj***,** *buf***,** *count***);**

buf	An (**unsigned ***) pointer to a buffer.
count	The number of bits to be read (from 1 to 255).

Example

```
IO_0 input bit d0;
boolean value;
...
void f(void)
{
   value = io_in(d0);
}
```

---

io_in_request( ) *Built-in Function*

The **io_in_request( )** built-in function is used with a **dualslope** I/O object to start the **dualslope** A/D conversion process.

Syntax

void io_in_request (*input-io-object-name***,** *control-value***);**

input-io-object-name	Specifies the I/O object name, which corresponds to *io-object-name* in the I/O declaration. This built-in function is used only for **dualslope** and **sci** I/O models.
control-value	An **unsigned long** value used to control the length of the first integration period. See the descriptions of the **dualslope** and **sci** I/O objects in the *I/O Model Reference* for more information.

Example 1

```
IO_4 input dualslope ds;
stimer repeating t;

when (online)
{
   t = 5;      // Do a conversion every 5 sec
}

when (timer_expires(t))
{
   io_in_request(ds, 40000);
}
```

The **io_in_request( )** is used with a **sci** I/O object to start the serial transfer.

Example 2

```
#pragma specify_io_clock "10 MHz"
IO_8 sci baud(SCI_2400) iosci;
unsigned short buf[20];

when (...)
{
   io_in_request(iosci, buf, 20);
}
```

The **io_out( )** built-in function writes data to an I/O object.

The **<io_types.h>** include file contains optional type definitions for each of the I/O object types. The type names are the I/O object type name followed by "**_t**". For example **bit_t** is the type name for a **bit** I/O object. The data type of *output-value* is listed below for each object type.

Object Type	*Output Value Type*
bit output	**unsigned short**
bitshift output	**unsigned long** (also, see below)
byte output	**unsigned short**
edgedivide output	**unsigned long**
frequency output	**unsigned long**
i2c	(see below)
infrared_pattern output	(see below)
muxbus output	**unsigned short**
neurowire master	(see below)
neurowire master	**void** (also, see below)
neurowire slave	(see below)
neurowire slave	**unsigned short** (also, see below)
nibble output	**unsigned short**
oneshot output	**unsigned long**
parallel	(see below)
pulsecount output	**unsigned long**
pulsewidth output	**unsigned short**
sci	Not applicable
serial output	(see below)
spi	**unsigned short**
stretchedtriac	**unsigned short**
touch	(see below)
triac output	**unsigned long**
triggeredcount output	**unsigned long**

Syntax

return-value **io_out (***output-io-object-name***,** *output-value* [**,** *args*]**);**

return-value The value returned by the function. **void** for all models except **i2c** (for which *return-value* is a Boolean value).

output-io-object-name	Specifies the I/O object name, which corresponds to *io-object-name* in the I/O declaration.
output-value	Specifies the value to be written to the I/O object.
args	Arguments, which depend on the object type, as described below. Some of these arguments can also appear in the object declaration. If specified in both places, the value of the function argument overrides the declared value for that call only. If the value is not specified in either the function argument or the declaration, the default value is used.

General

For all output objects except those listed below, the syntax is:

io_out (*output-obj***,** *output-value***);**

The type of the *output-value* of the **io_out( )** call is listed in the table above.

bitshift

For **bitshift** output objects, the syntax is:

io_out (*bitshift-output-obj* **,** *output-value* [**,** *numbits*]**);**

numbits	The number of bits to be shifted out, from 1 to 127. After 16 bits, zeros are shifted out.

i2c

For **i2c** I/O objects, the syntax is:

io_out (*i2c-io-obj***,** *buf***,** *addr***,** *count***);**

io_out (*i2c-io-obj***,** *buf***,** *addr***,** *count*, *stop***);**

buf	A (**void ***) pointer to a buffer.
addr	An unsigned int I2C device address.
count	The number of bits to be written (from 1 to 255).
stop	A Boolean value to specify whether the stop condition is asserted after the transfer.

infrared_pattern

For **infrared_pattern** output objects, the syntax is:

io_out (*infrared-pattern-obj, freqOut, timing-table, count***);**

freqOut	An unsigned long value that selects the output-frequency.
timing-table	An array of unsigned long timing values.
count	An unsigned short value specifying the number of entries in the timing table. The number of values in the table, and therefore the *count* value, must be an odd number. See the detailed description of the **infrared_pattern** I/O

model in the *I/O Model Reference* for a detailed explanation of this restriction.

muxbus

For **muxbus** I/O objects, the syntax is:

io_out (*muxbus-io-obj***,** [*addr*,] *data***);**

addr	An optional address to write (from 0 to 255). Omission of the address causes the firmware to rewrite the last address read or written (**muxbus** is a bi-directional I/O device).
data	A single byte of data to write.

neurowire

For **neurowire** I/O objects, the syntax is:

io_out (*neurowire-io-obj***,** *buf***,** *count***);**

buf	A (**void ***) pointer to a buffer.
count	The number of bits to be written (from 1 to 255).

Calling **io_out( )** for a **neurowire** output object is the same as calling **io_in( )**. In either case, data is shifted into the buffer from pin IO_10.

parallel

For **parallel** I/O objects, the syntax is:

io_out (*parallel-io-obj***,** *buf***);**

buf	A pointer to the **parallel_io_interface** structure.

serial

For **serial** output objects, the syntax is:

io_out (*serial-output-obj***,** *buf***,** *count***);**

buf	A (**void ***) pointer to a buffer.
count	The number of bytes to be written (from 1 to 255).

spi

For **spi** I/O objects, the syntax is:

io_out (*spi-io-obj, buf, len***);**

buf	A pointer to a buffer of data bytes for the bidirectional data transfer.
len	An unsigned short number of bytes to transfer.

Calling **io_out()** for a **spi** object is the same as calling **io_in()**. In either case, the data in the buffer is output and simultaneously replaced by new input data.

touch

For **touch** I/O objects, the syntax is:

io_out (*touch-io-obj***,** *buf***,** *count***);**

buf	A (**void ***) pointer to a buffer.
count	The number of bits to be written (from 1 to 255).

Example

```
boolean value;
IO_0 output bit d0;

void f(void)
{
   io_out(d0, value);
}
```

---

io_out_request( ) *Built-in Function*

The **io_out_request( )** built-in function is used with the **parallel** I/O object and the **sci** I/O object.

The **io_out_request( )** sets up the system for an **io_out( )** on the specified parallel I/O object. When the system is ready, the **io_out_ready** event becomes TRUE and the **io_out( )** function can be used to write data to the parallel port. See Chapter 2, *Focusing on a Single Device,* of the *Neuron C Programmer's Guide* for more information.

Syntax

void io_out_request (*io-object-name***);**

io-object-name	Specifies the I/O object name, which corresponds to *io-object-name* in the I/O object's declaration.

Example 1

```
when (...)
{
   io_out_request(io_bus);
}
```

The **io_out_request( )** is used with a **sci** I/O object to start the serial transfer.

Example 2

```
IO_8 sci baud(SCI_2400) iosci;
unsigned short buf[20];

when (...)
{
   io_out_request(iosci, buf, 20);
}
```

io_preserve_input( ) *Built-in Function*

The **io_preserve_input( )** built-in function is used with an input timer/counter I/O object. If this function is not called, the Neuron firmware discards the first reading on a timer/counter object after a reset (or after a device on the multiplexed timer/counter is selected using the **io_select( )** function because the data might be suspect due to a partial update). Calling the **io_preserve_input( )** function prior to the first reading, either by an **io_in( )** or implicit input, overrides the discard logic.

The **io_preserve_input( )** call can be placed in a **when (reset)** clause to preserve the first input value after reset. The call can be used immediately after an **io_select( )** call to preserve the first value after select.

Syntax

void io_preserve_input (*input-io-object-name***);**

input-io-object-name Specifies the I/O object name that corresponds to *io-object-name* in the I/O declaration. This built-in function is only applicable to input timer/counter I/O objects.

Example

```
IO_5 input ontime ot1;
IO_6 input ontime ot2;
unsigned long variable1;

when (io_update_occurs(ot1))
{
   variable1 = input_value;
   io_select(ot2);
   io_preserve_input(ot2);
}
```

io_select( ) *Built-in Function*

The **io_select( )** built-in function selects which of the multiplexed pins is the owner of the timer/counter circuit, and optionally specifies a clock for the I/O object. Input to one of the timer/counter circuits can be multiplexed among pins 4 to 7. The other timer/counter input is dedicated to pin 4.

When **io_select( )** is used, the I/O object automatically discards the first value obtained.

Syntax

void io_select (*input-io-object-name* [**,** *clock-value*]**);**

input-io-object-name The I/O object name that corresponds to *io-object-name* in the I/O declaration. This built-in function is used only for the following timer/counter input objects:

infrared
ontime
period
pulsecount
totalcount

clock-value — Specifies an optional clock value, in the range 0 to 7, or a variable name for the clock. This value permanently overrides a clock value specified in the object's declaration. The clock value option can only be specified for the **infrared**, **ontime**, and **period** objects.

Example

```
IO_5 input ontime pcount1;
IO_6 input ontime pcount2;
unsigned long variable1;

when (io_update_occurs(pcount_1))
{
   variable1 = input_value;
   // select next I/O object
   io_select(pcount_2);
}
```

io_set_baud( ) — *Built-in Function*

The **io_set_baud( )** built-in function allows an application to optionally change the baud rate for an SCI device. The SCI device optionally has an initial bit rate setting from its declaration.

See also Chapter 2, *Compiler Directives*, on page 21, for information about the **specify_io_clock** compiler directive.

Syntax

void io_set_baud (*io-object-name***,** *baud-rate***);**

io-object-name — The I/O object name that corresponds to *io-object-name* in the I/O declaration. This built-in function is used only for **sci** I/O objects.

baud-rate — The serial bit rate through use of the enumeration values found in the **<io_types.h>** include file. These enumeration values are SCI_300, SCI_600, SCI_1200, SCI_2400, SCI_4800, SCI_9600, SCI_19200, SCI_38400, SCI_57600, and SCI_115200. The enumeration values select serial bit rates of 300, 600, 1200, 2400, 4800, 9600, 19200, 38400, 57600, and 115200, respectively.

Example

```
IO_8 sci baud(SCI_2400) iosci;

when (...)
```

```
{
   io_set_baud(iosci, SCI_38400);
   // Optional baud change
}
```

io_set_clock( ) *Built-in Function*

The **io_set_clock( )** built-in function allows an application to specify an alternate clock value for any input or output timer/counter object that permits a clock argument in its declaration syntax. The objects are listed below:

dualslope
edgelog
frequency
infrared
oneshot
ontime
period
pulsecount
pulsewidth
stretchedtriac
triac

For multiplexed inputs, use the **io_select( )** function to specify an alternate clock.

When **io_set_clock( )** is used, the I/O object automatically discards the first value obtained.

You can call this function at any time. However, if your application specifies an alternate clock value, it must call this function within the reset task and after each call to the **io_select()** function.

Syntax

void io_set_clock (*io-object-name***,** *clock-value***);**

io-object-name	The I/O object name that corresponds to *io-object-name* in the I/O declaration. This built-in function is used only for timer/counter I/O objects.
clock-value	Required clock selector value in the range of 0 to 7 (for Series 3100 devices) or 0 to 15 (for Series 5000 and 6000 devices), or a variable name for the clock. This value overrides a clock value specified in the object's declaration. You can use a **TCCLK_*** macro value from **<echelon.h>** to specify the clock selector value. For Series 3100 devices, you can use the macros whose values are in the range 0..7; for Series 5000 and 6000 devices, you can use any of the macros (values 0..15) for increased clock resolution. However, for Series 5000 and 6000 devices, you cannot specify a value that defines a clock rate that is higher

than one-half of the device's system clock. For example, if your system clock rate is 20 MHz, you can specify any **TCCLK_*** macro that defines a 10 MHz or lower clock rate (that is, you cannot specify **TCCLK_40MHz** or **TCCLK_20MHz** – no error is issued, but the effective value used in this case is **TCCLK_10MHz**).

Example

```
IO_1 output pulsecount clock(3) pcout;

when(...)
{
   io_set_clock(pcout, 5);
   // equivalent to io_set_clock(pcout, TCCLK_156k2Hz);
   ...
}
```

io_set_direction( ) *Built-in Function*

The **io_set_direction( )** built-in function allows the application to change the direction of any **bit**, **nibble**, or **byte** type I/O pin at runtime. The *dir* parameter is optional. If not provided, **io_set_direction( )** sets the direction based on the direction specified in the declaration of *io-object-name*.

A program can define multiple types of I/O objects for a single pin. When directions conflict and a timer/counter object is defined, the direction of the timer/counter object is used, regardless of the order of definition. However, if the program uses the **io_set_direction( )** function for such an object, the direction is changed as specified.

To change the direction of overlaid I/O objects, at least one of the objects must be one of the allowed types for **io_set_direction( )** and that I/O object must be used to change directions, even if the subsequent I/O object used is a different one.

For example, if you overlaid a **bit** input with a **oneshot** output, you only can use the **bit** I/O object with **io_set_direction( )** to change the direction from input to output, thus enabling the **oneshot** output.

Any **io_changes** events requested for input objects can trigger when the object is redirected as an output, because the Neuron firmware returns the last value output on an output object as the input value. Thus, the user might want to qualify **io_changes** events with flags maintained by the program to indicate the current direction of the device.

Syntax

typedef enum {IO_DIR_IN=0, IO_DIR_OUT=1} io_direction;
void io_set_direction (*io-object-name***,** [**io_direction** *dir*]**);**

io-object-name	The I/O object name that corresponds to *io-object-name* in the I/O declaration. This built-in function is used only for direct I/O objects such as **bit**, **nibble**, and **byte**.
dir	An optional direction, using a value from the **io_direction** enum shown above. If omitted, uses the

declared direction of the I/O device to set the pin direction.

Example

```
IO_0 output bit b0;
IO_0 input byte byte0;
int read_byte;

void f(void)
{
   io_set_direction(b0, IO_DIR_OUT);
   io_out(b0, 0);
   io_set_direction(byte0);   // Defaults to IO_DIR_IN
   read_byte = io_in(byte0);
}
```

io_set_terminal_count( ) *Built-in Function*

The **io_set_terminal_count( )** built-in function allows the application to change the terminal count for the stretched triac I/O object at runtime. This function allows a device to:

- Have a single application for both 50 Hz and 60 Hz power domains
- Operate at a non-standard power line frequency
- Provide higher-than-typical tolerances to changes in frequency

The application can determine the current values for frequency at runtime, and use this function to adjust the triac on-time as needed.

See the *I/O Model Reference* for more information about the stretched triac model.

Syntax

void io_set_terminal_count (*io-object-name***,** *terminal_count***);**

io-object-name	The I/O object name that corresponds to *io-object-name* in the I/O declaration. This built-in function is used only for the **stretchedtriac** direct I/O object.
terminal_count	A value from 0 to 193 for triac devices running with 50 Hz AC power, or a value from 0 to 160 for triac devices running with 60 Hz AC power. This value represents the amount of stretching for the triac trigger pulse within the half-cycle. Specify this value to calibrate the triac device.

Example

```
IO_0 output stretchedtriac sync (IO_5) frequency(60)
ioTriac;

when (...) {
  io_out(ioTriac, 160);  // full on
}
```

```
when (...) {
  io_out(ioTriac, 80);  // half on
}

when (...) {
  io_out(ioTriac, 0);   // full off
}
```

is_bound( ) *Built-in Function*

The **is_bound( )** built-in function indicates whether the specified network variable or message tag is connected (bound). The function returns TRUE if the network variable or message tag is connected, otherwise it returns FALSE.

Typically, the function is used to override error detection algorithms. Many actuators, for example, are configured to receive updates to input network variables periodically for new values, or for re-sent values (heartbeats). Failing such updates for a period of time might constitude an error for such an application, but the device cannot expect such an update if its network variables (or message tags) are not yet bound.

The **is_bound()** function can be used to detect this case, and prevent entering the error condition.

For network variables, **is_bound( )** returns TRUE if the network variable selector value is less than 0x3000. For message tags, **is_bound( )** returns TRUE if the message tag has a valid address in the address table.

Syntax

boolean is_bound (*net-object-name***);**

net-object-name Either a network variable name or a message tag.

Example

```
mtimer update_monitor;

network input cp SCPTmaxSentTime cpMaxSendTime;
network input SNVT_color nviColor nv_properties {
   cpMaxSendTime
};

...

when(timer_expires(update_monitor))
{
   if (is_bound(nviColor) && nviColor::cpMaxSendTime)
   {
      heartbeat_failure();
      update_monitor = nviColor::cpMaxSendTime * 100ul;
   }
}
```

low_byte( ) *Built-in Function*

The **low_byte( )** built-in function extracts the lower single-byte value from the double-byte operand *a*. This function operates without regard to signedness. See also **high_byte( )**, **make_long( )**, and **swap_bytes( )**.

Syntax

unsigned short low_byte (unsigned long *a***);**

Example

```
short b;
long a;

void f(void)
{
   a = 258;    // Hex value 0x0102
   b = low_byte(a);     // b now contains the value 0x02
}
```

make_long( ) *Built-in Function*

The **make_long( )** built-in function combines the *low-byte* and *high-byte* single-byte values to make a double-byte value. This function operates without regard to signedness of the operands. See also **high_byte( )**, **low_byte( )**, and **swap_bytes( )**.

Syntax

unsigned long make_long (unsigned short *low-byte***,**
unsigned short *high-byte***);**

Example

```
short a, b;
long l;

void f(void)
{
   a = 16;        // Hex value 0x10
   b = -2;        // Hex value 0xFE
   l = make_long(a, b); // l now contains 0xFE10
   l = make_long(b, a); // l now contains 0x10FE
}
```

max( ) *Built-in Function*

The **max( )** built-in function compares *a* and *b* and returns the larger value. The result *type* is determined by the types of *a* and *b*, as shown in **Table 30** on page 120.

Table 30. Result Types for the **max()** Function

Larger Type	Smaller Type	Result
unsigned long	(any)	**unsigned long**
signed long	**signed long** **unsigned short** **signed short**	**signed long**
unsigned short	**unsigned short** **signed short**	**unsigned short**
signed short	**signed short**	**signed short**

If the result type is **unsigned**, the comparison is **unsigned**, else the comparison is **signed**. Arguments can be cast, which affects the result type. When argument types do not match, the smaller type argument is promoted to the larger type prior to the operation.

Syntax

type **max (***a***,** *b***);**

Example

```
int a, b, c;
long x, y, z;

void f(void)
{
   a = max(b, c);
   x = max(y, z);
}
```

Note: The description of the **max()** function result types and type promotion of arguments also applies equally to the **min()** function.

memccpy( ) *Function*

The **memccpy( )** function copies *len* bytes from the memory area pointed to by *src* to the memory area pointed to by *dest*, up to and including the first occurrence of character *c*, if it exists. The function returns a pointer to the byte in *dest* immediately following *c*, if *c* was copied, else **memccpy( )** returns NULL. This function cannot be used to write to EEPROM or flash memory. See also **ansi_memcpy( )**, **ansi_memset( )**, **eeprom_memcpy( )**, **memchr( )**, **memcmp( )**, **memcpy( )**, and **memset( )**.

Syntax

#include <mem.h>
void* memccpy (void *****dest*, const void *****src***, int** *c***, unsigned long** *len***);**

Example

```
#include <mem.h>

unsigned array1[40];

void f(void)
{
   // Copy up to 40 bytes to array1,
   // but stop if an ASCII "W" value is copied.
   unsigned *p;
   p = memccpy(array1, "Hello World", 'W', sizeof(array1));
}
```

When the function returns, *array1* contains "Hello W", but no terminating '\0' character, and *p* points to the 8th byte in the *array1* array (following the 'W').

memchr( ) *Function*

The **memchr( )** function searches the first *len* bytes of the memory area pointed to by *buf* for the first occurrence of character *c*, if it exists. The function returns a pointer to the byte in *buf* containing *c*, else **memchr( )** returns NULL. See also **ansi_memcpy( )**, **ansi_memset( )**, **eeprom_memcpy( )**, **memccpy( )**, **memcmp( )**, **memcpy( )**, and **memset( )**.

Syntax

#include <mem.h>
void *memchr (const void *_buf_, int _c_, unsigned long _len_);

Example

```
#include <mem.h>

unsigned array[40];

void f(void)
{
   unsigned *p;

   // Find the first 0xFF byte, if it exists
   p = memchr(array, 0xFF, sizeof(array));
}
```

memcmp( ) *Function*

The **memcmp( )** function compares the first *len* bytes of the memory area pointed to by *buf1* to the memory area pointed to by *buf2*. The function returns 0 if the memory areas match exactly. Otherwise, on the first non-matching byte, the byte from each buffer is compared using an unsigned comparison. If the byte from *buf1* is larger, then a positive number is returned, else a negative number is returned. See also **ansi_memcpy( )**, **ansi_memset( )**, **eeprom_memcpy( )**, **memccpy( )**, **memchr( )**, **memcpy( )**, and **memset( )**.

Syntax

#include <mem.h>
int memcmp (void *_buf1_, const void *_buf2_, unsigned long _len_);

Example

```
#include <mem.h>

unsigned array1[40], array2[40];

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

memcpy( ) *Built-in Function*

The **memcpy( )** built-in function copies a block of *len* bytes from *src* to *dest*. It does not return any value. This function cannot be used to copy overlapping areas of memory, or to write into EEPROM or flash memory. The **memcpy( )** function can also be used to copy to and from the data fields of the **msg_in**, **resp_in**, **msg_out**, and **resp_out** objects.

The **memcpy( )** function as implemented here does not conform to the ANSI C definition, because it does not return a pointer to the destination array. See **ansi_memcpy( )** for a conforming implementation. See also **ansi_memset( )**, **eeprom_memcpy( )**, **memccpy( )**, **memchr( )**, **memcmp( )**, and **memset( )**.

Syntax

void memcpy (void *_dest_, void *_src_, unsigned long _len_);

Example

```
void f(void)
{
   memcpy(msg_out.data, "Hello World", 11);
}
```

memset( ) *Built-in Function*

The **memset( )** built-in function sets the first *len* bytes of the block pointed to by *p* to the character *c*. It does not return any value. This function cannot be used to write into EEPROM or flash memory.

The **memset( )** function as implemented here does not conform to the ANSI C definition, because it does not return a pointer to the array. See **ansi_memset( )** for a conforming implementation. See also **ansi_memcpy( )**, **eeprom_memcpy( )**, **memccpy( )**, **memchr( )**, **memcmp( )**, and **memcpy( )**.

Syntax

void memset (void *_p_, int _c_, unsigned long _len_);

Example

```
unsigned target[20];

void f(void)
{
   memset(target, 0, sizeof(target));
}
```

min( ) *Built-in Function*

The **min( )** built-in function compares *a* and *b* and returns the smaller value. The result *type* is determined by the types of *a* and *b*, as described for **max( )** on page 119.

Syntax

type **min (***a*, *b***);**

Example

```
int a, b, c;
long x, y, z;

void f(void)
{
   a = min(b, c);
   x = min(y, z);
}
```

msec_delay( ) *Function*

The **msec_delay( )** function allows an application to suspend processing for a time interval specified by *milliseconds*. The maximum delay is 255 ms. This function provides more precise timing than can be achieved with application timers, and provides an easier way to specify millisecond delays than the **delay( )** or **scaled_delay( )** functions. See **delay( )** and **scaled_delay( )** for functions that can delay the application program for a longer duration.

Syntax

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

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

Example

```
IO_4 input bit io_push_button;
boolean debounced_button_state;
```

```
when (io_changes(io_push_button))
{
   msec_delay(10); // Delay 10ms at any clock rate
   debounced_button_state = (boolean)io_in(io_push_button);
}
```

msg_alloc( ) *Built-in Function*

The **msg_alloc( )** built-in function allocates a nonpriority buffer for an outgoing message. The function returns TRUE if a **msg_out** object can be allocated. The function returns FALSE if a **msg_out** object cannot be allocated. When this function returns FALSE, a program can continue with other processing, if necessary, rather than waiting for a free message buffer.

See Chapter 6, *How Devices Communicate Using Application Messages,* in the *Neuron C Programmer's Guide* for more information about application messages.

Syntax

boolean msg_alloc (void);

Example

```
void f(void)
{
   if (msg_alloc()) {
      // OK.  Build and send message
      ...
   }
}
```

msg_alloc_priority( ) *Built-in Function*

The **msg_alloc_priority( )** built-in function allocates a priority buffer for an outgoing message. The function returns TRUE if a priority **msg_out** object can be allocated. The function returns FALSE if a priority **msg_out** object cannot be allocated. When this function returns FALSE, a program can continue with other processing, if desired, rather than waiting for a free priority buffer.

See Chapter 6, *How Devices Communicate Using Application Messages,* in the *Neuron C Programmer's Guide* for more information about application messages.

Syntax

boolean msg_alloc_priority (void);

Example

```
void f(void)
{
   if (msg_alloc_priority()) {
      // OK.  Build and send message
      ...
```

```
   }
}
```

msg_cancel( ) *Built-in Function*

The **msg_cancel( )** built-in function cancels the message currently being built and frees the associated buffer, allowing another message to be constructed.

If a message is constructed but not sent before the critical section (for example, a task) is exited, the message is automatically cancelled. This function is used to cancel both priority and nonpriority messages.

See Chapter 6, *How Devices Communicate Using Application Messages,* in the *Neuron C Programmer's Guide* for more information about application messages.

Syntax

void msg_cancel (void);

Example

```
void f(void)
{
   if (msg_alloc()) {
   ...
      if (offline()) {
      // Requested to go offline
         msg_cancel();
      } else {
         msg_send();
      }
   }
}
```

msg_free( ) *Built-in Function*

The **msg_free( )** built-in function frees the **msg_in** object for an incoming message.

See Chapter 6, *How Devices Communicate Using Application Messages,* in the *Neuron C Programmer's Guide* for more information about application messages.

Syntax

void msg_free (void);

Example

```
void f(void)
{
...
   if (msg_receive()) {
      // Process message
         ...
      msg_free();
```

```
   }
...
}
```

msg_realloc( ) *Function*

The **msg_realloc ( )** function allows using an input buffer for an outgoing message. This ensures that propagation of the outgoing message will not enter preemption mode. However, the maximum size of the outgoing message is constrained by the applicaiton input buffer size, not the application output buffer size. Access to the msg_in built-in object is prohibited after successfully calling the msg_realloc() function until msg_cancel() or msg_send() has been called. The msg_realloc() function succeeds when at least one input buffer is available.

See Chapter 6, *How Devices Communicate Using Application Messages,* in the *Neuron C Programmer's Guide* for more information about application messages.

Syntax

boolean msg_realloc (void);

msg_receive( ) *Built-in Function*

The **msg_receive( )** built-in 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 pending at the head of the message queue, this function does not wait for one. A program might need to use this function if it receives more than one message in a single task, as in bypass mode. If there already is a received message, the earlier one is discarded (that is, its buffer space is freed).

Note: Because this function defines a critical section boundary, it should never be used in a **when** clause expression (that is, it *can* be used within a *task*, but *not* within the **when** *clause* itself). Using it in a **when** clause expression could result in events being processed incorrectly.

The **msg_receive( )** function receives all messages in raw form, so that the **online**, **offline**, and **wink** special events cannot be used. If the program handles any of these events, it should use the **msg_arrives** event, rather than the **msg_receive( )** function.

See Chapter 6, *How Devices Communicate Using Application Messages,* in the *Neuron C Programmer's Guide* for more information about application messages.

Syntax

boolean msg_receive (void);

Example

```
void f(void)
{
...
   if (msg_receive()){
      // Process message
         ...
```

```
      msg_free();
   }
...
}
```

msg_send( ) *Built-in Function*

The **msg_send( )** built-in function sends a message using the **msg_out** object.

See Chapter 6, *How Devices Communicate Using Application Messages,* in the *Neuron C Programmer's Guide* for more information about application messages.

Syntax

void msg_send (void);

Example

```
msg_tag motor;
# define MOTOR_ON 0
# define ON_FULL 1

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();
}
```

muldiv( ) *Function*

The **muldiv( )** function permits the computation of (*A***B*)/*C* where *A*, *B*, and *C* are all 16-bit values, but the intermediate product of (*A***B*) is a 32-bit value. Thus, the accuracy of the result is improved. There are two versions of this function: **muldiv( )** and **muldivs( )**. The **muldiv( )** function uses **unsigned** arithmetic, while the **muldivs( )** function (see below) uses **signed** arithmetic.

See also **muldiv24( )** and **muldiv24s( )** for functions which use 24-bit intermediate accuracy for faster performance.

Syntax

#include <stdlib.h>
unsigned long muldiv (unsigned long *A***, unsigned long** *B***,**
unsigned long *C***);**

Example

```
#include <stdlib.h>
unsigned long a, b, c, d;
...
```

```
void f(void)
{
   d = muldiv(a, b, c);   // d = (a*b)/c
}
```

muldiv24( ) *Function*

The **muldiv24( )** function permits the computation of (*A***B*)/*C* where *A* is a 16-bit value, and *B* and *C* are both 8-bit values, but the intermediate product of (*A***B*) is a 24-bit value. Thus, the performance of the function is improved while maintaining the accuracy of the result. There are two versions of this function: **muldiv24( )** and **muldiv24s( )**. The **muldiv24( )** function uses **unsigned** arithmetic, while the **muldiv24s( )** function (see below) uses **signed** arithmetic.

See also **muldiv( )** and **muldivs( )** for functions which use 32-bit intermediate accuracy for greater accuracy at the expense of slower performance. You can always use **muldiv24()** without loss of precision, compared to **muldiv()**, if neither A nor B ever exceeds 256.

Syntax

#include <stdlib.h>
unsigned long muldiv24 (unsigned long *A***, unsigned int** *B***,**
unsigned int *C***);**

Example

```
#include <stdlib.h>
unsigned long a, d;
unsigned int  b, c;
...

void f(void)
{
   d = muldiv24(a, b, c);   // d = (a*b)/c
}
```

muldiv24s( ) *Function*

The **muldiv24s( )** function permits the computation of (*A***B*)/*C* where *A* is a 16-bit value, and *B* and *C* are both 8-bit values, but the intermediate product of (*A***B*) is a 24-bit value. Thus, the performance of the function is improved while maintaining the accuracy of the result. There are two versions of this function: **muldiv24s( )** and **muldiv24( )**. The **muldiv24s( )** function uses **signed** arithmetic, while the **muldiv24( )** function (see above) uses **unsigned** arithmetic.

See also **muldiv( )** and **muldivs( )** for functions which use 32-bit intermediate accuracy for greater accuracy at the expense of slower performance. You can always use **muldiv24s()** without loss of precision, compared to **muldivs()**, if either A or B always is in the -128..+127 value interval.

Syntax

#include <stdlib.h>
signed long muldiv24s (signed long *A***, signed int** *B***, signed int** *C***);**

Example

```
#include <stdlib.h>
signed long a, d;
signed int  b, c;
...

void f(void)
{
   d = muldiv24s(a, b, c);   // d = (a*b)/c
}
```

muldivs( ) *Function*

The **muldivs( )** function permits the computation of (*A***B*)/*C* where *A*, *B*, and *C* are all 16-bit values, but the intermediate product of (*A***B*) is a 32-bit value. Thus, the accuracy of the result is improved. There are two versions of this function: **muldivs( )** and **muldiv( )**. The **muldivs( )** function uses **signed** arithmetic, while the **muldiv( )** function (see above) uses **unsigned** arithmetic.

See also **muldiv24( )** and **muldiv24s( )** for functions which use 24-bit intermediate accuracy for faster performance.

Syntax

#include <stdlib.h>
signed long muldivs (signed long *A***, signed long** *B***, signed long** *C***);**

Example

```
#include <stdlib.h>
signed long a, b, c, d;
...

void f(void)
{
   d = muldiv(a, b, c);   // d = (a*b)/c
}
```

node_reset( ) *Function*

The **node_reset( )** function resets the Neuron Chip or Smart Transceiver hardware. When **node_reset( )** is called, all the device's volatile state information is lost. Variables declared with the **eeprom** or **config** class and the device's network image (which is stored in EEPROM) are preserved across resets and loss of power. The **when(reset)** event evaluates to TRUE after this function is called.

Syntax

#include <control.h>
void node_reset (void);

Example

```
#define MAX_ERRORS1 50
#define MAX_ERRORS2 55
int error_count;
...

when(error_count > MAX_ERRORS2)
{
    node_reset();
}

when(error_count > MAX_ERRORS1)
{
    application_restart();
}
```

nv_table_index( ) *Built-in Function*

The **nv_table_index( )** built-in function is used to determine the index of a network variable as allocated by the Neuron C compiler. The returned value is limited by the application's number of static network variables, and is never more than 61 for devices that are limited to 62 static network variables, or 253 for devices that are limited to 254 static network variables.

The **global_index** property, introduced in Neuron C Version 2, is equivalent to the **nv_table_index( )** built-in function. The *global_index* property is recommended for new development.

Syntax

int nv_table_index (*netvar-name***);**

netvar-name	A network variable name, possibly including an index expression.

Example

```
unsigned nv_index;
network output SNVT_lux nvoLux;

void f(void)
{
   nv_index = nv_table_index(nvoLux);
   // Equivalent statement, recommended:
   nv_index = nvoLux::global_index;
}
```

offline_confirm( ) *Function*

The **offline_confirm( )** function allows a device to confirm to a network tool that the device has finished its clean-up and is now going offline. This function is normally only used in bypass mode (that is, when the **offline** event is checked outside of a **when** clause). If the program is not in bypass mode, use **when (offline)** rather than **offline_confirm( )**.

In bypass mode, when the Neuron firmware goes offline using **offline_confirm( )**, the program continues to run. It is up to the programmer to determine which events are processed when the Neuron firmware is offline.

Syntax

#include <control.h>
void offline_confirm (void);

Example

```
void f(void)
{
   ...
   if (offline){
      // Perform offline cleanup
      ...
      offline_confirm();
   }
}
```

poll( ) *Built-in Function*

The **poll( )** built-in function allows a device to request the latest value for one or more of its input network variables. Any input network variable can be polled at any time. If an array name without an index is used, then each element of the array is polled. An individual element can be polled by using an array index.

When writing a Neuron hosted application in Neuron C, the input network variable does not need to be declared as **polled**. However, you must declare input network variables that will use the **poll( )** function, or an equivalent API, as **polled** when writing model files for host-based device development.

The new, polled value can be obtained through use of the **nv_update_occurs** event.

If multiple devices have output network variables connected to the input network variables being polled, multiple updates are sent in response to the poll. The polling device cannot assume that all updates are received and processed independently. This means it is possible for multiple updates to occur before the polling device can process the incoming values. To ensure that all values sent are independently processed, the polling device should declare the input network variable as a synchronous input.

An input network variable that is polled with the **poll( )** function consumes an address table entry when it is bound to any output network variables.

The device interface file must identify all polled network variables. This identification occurs automatically, however, a device's program ID must be updated if **poll( )** calls are added or deleted from an application.

See also the *Initial Value Updates for Input Network Variables* section in Chapter 3 of the *Neuron C Programmer's Guide* for additional guidance about how to use the **poll( )** function.

Syntax

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

network-var	A network variable identifier, array name, or array element. If the parameter is omitted, all input network variables for the device are polled.

Example

```
network input SNVT_privacyzone nviZone;

   ...
   poll(nviZone);
   ...

when (nv_update_occurs(nviZone))
{
   // New value of nviZone arrived
}
```

post_events( ) *Function*

The **post_events( )** function defines a boundary of a critical section at which network variable updates and messages are sent and incoming network variable update and message events are posted.

The **post_events( )** function is called implicitly by the scheduler at the end of every task body. If the application program calls **post_events( )** explicitly, the application should be prepared to handle the special events **online**, **offline**, and **wink** before checking for any **msg_arrives** event.

The **post_events( )** function can also be used to improve network performance. See *The post_events*( ) *Function* in Chapter 7, *Additional Features,* of the *Neuron C Programmer's Guide* for a more detailed discussion of this feature.

Syntax

#include <control.h>
void post_events (void);

Example

```
boolean still_processing;
...
void f(void)
{
```

```
   while (still_processing) {
      post_events();
      ...
   }
}
```

power_up( ) *Function*

The **power_up( )** function returns TRUE if the last reset resulted from a power-up. Any time an application starts up (whether from a reset or from a power-up), the **when(reset)** task runs, and you can use this function to determine whether the start-up resulted from a power-up.

Syntax

#include <status.h>
boolean power_up (void);

Example

```
when (reset)
{
   if (power_up()) {
      initialize_hardware();
   } else {
      // hardware already initialized
      ...
   }
}
```

preemption_mode( ) *Function*

The **preemption_mode( )** function returns a TRUE if the application is currently running in preemption mode, or FALSE if the application is not in preemption mode. Preemption mode is discussed in Chapter 3, *How Devices Communicate Using Network Variables,* of the *Neuron C Programmer's Guide.*

Syntax

#include <status.h>
boolean preemption_mode (void);

Example

```
void f(void)
{
   if (preemption_mode()) {
      // Take some appropriate action
      ...
   }
}
```

The **propagate( )** built-in function allows a device's application program to request that the latest value for one or more of its output network variables be sent out over the network. Any bound (that is, connected) output network variable can be propagated at any time. Propagating an unbound output network variable has no effect on the network, allowing the application to run the exact same code, regardless of whether the network variable is unbound or is bound to many network variables.

If an array name is used, then each element of the array is propagated. An individual element can be propagated by using an array index.

Input network variables cannot be propagated, and calls to **propagate( )** for input network variables have no effect.

This function allows variables to be sent out even if they are declared **const**, and are thus in read-only memory (normally a network variable's value is sent over the network only when is the application writes a new value to the network variable). Also, it permits updating a network variable through a pointer, and then causing the variable to be propagated separately.

Polled output network variables can be propagated with the **propagate( )** function. However, if an output network variable is declared as **polled**, but is also affected by the **propagate( )** function, the polled attribute does not appear in the device interface (XIF) file. Thus, network tools can handle the network address assignment for the variable properly. If any member of an array is propagated, the **polled** attribute is blocked for all elements of the array. If a **propagate( )** call appears without arguments, all output variables' **polled** attributes are blocked.

Syntax

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

network-var	A network variable identifier, array name, or array element. If the parameter is omitted, all output network variables for the device are propagated.

Example 1

```
// The pragma permits network variable addresses
// to be passed to functions with non-const pointers,
// with only a warning.

network output UNVT_whatever nvoWhatever;

void f(const UNVT_whatever* p);

when (...)
{
   f(&nvoWhatever);  // Process by address in function f
   propagate(nvoWhatever);    // Cause NV to be sent out
}
```

Example 2

```
network output const eeprom SNVT_address nvoAddress;

// Propagate nvoAddress on request
when (...)
{
   propagate(nvoAddress);
}
```

random( ) *Function*

The **random( )** function returns a random number in the range 0 ... 255. The random number is seeded using the unique 48-bit Neuron ID. The **random( )** function is computed from the data on all three CPU buses. If, after each reset, the **random( )** function is called at exactly the same time, the returned random number is the same. However, if your device does anything different, based on I/O processing or messages received, or based on data changes, and so on, the random number sequence is different.

Syntax

unsigned int random (void);

Example

```
void f(void)
{
   unsigned value = random();
}
```

resp_alloc( ) *Built-in Function*

The **resp_alloc( )** built-in function allocates an object for an outgoing response. The function returns TRUE if a **resp_out** object can be allocated. The function returns FALSE if a **resp_out** object cannot be allocated. When this function returns FALSE, a program can continue with other processing, if necessary, rather than waiting for a free message buffer.

See Chapter 6, *How Devices Communicate Using Application Messages,* in the *Neuron C Programmer's Guide* for more information about application messages.

Syntax

boolean resp_alloc (void);

Example

```
when (...)
{
   if (resp_alloc()) {
      // OK.  Build and send message
      ...
```

```
   }
}
```

resp_cancel( ) Built-in Function

The **resp_cancel( )** built-in function cancels the response being built and frees the associated **resp_out** object, allowing another response to be constructed.

If a response is constructed but not sent before the critical section (for example, a task) is exited, the response is automatically cancelled. See Chapter 6, *How Devices Communicate Using Application Messages,* of the *Neuron C Programmer's Guide* for more information.

Syntax

void resp_cancel (void);

Example

```
void f(void)
{
   if (resp_alloc()) {
   ...
      if (offline()) {
         // Requested to go offline
         resp_cancel();
      } else {
         resp_send();
      }
   }
}
```

resp_free( ) Built-in Function

The **resp_free( )** built-in function frees the **resp_in** object for a response. See Chapter 6, *How Devices Communicate Using Application Messages,* of the *Neuron C Programmer's Guide*.

Syntax

void resp_free (void);

Example

```
void f(void)
{
...
   if (resp_receive()) {
      // Process message
      ...
      resp_free();
   }
...
}
```

resp_receive( ) *Built-in Function*

The **resp_receive( )** built-in function receives a response into the **resp_in** object. The function returns TRUE if a new response is received, otherwise it returns FALSE. If no response is received, this function does not wait for one. A program might need to use this function if it receives more than one response in a single task, as in bypass mode. If there already is a received response when the **resp_receive( )** function is called, the earlier one is discarded (that is, its buffer space is freed).

Note: Because this function defines a critical section boundary, it should never be used in a **when** clause expression (that is, it *can* be used within a task, but *not* within the **when** *clause* itself). Using it in a **when** clause expression could result in events being processed incorrectly.

See Chapter 6, *How Devices Communicate Using Application Messages,* of the *Neuron C Programmer's Guide* for more information.

Syntax

boolean resp_receive (void);

Example

```
void f(void)
{
...
   if (resp_receive()) {
      // Process message
         ...
      resp_free();
   }
...
}
```

resp_send( ) *Built-in Function*

The **resp_send( )** built-in function sends a response using the **resp_out** object. See Chapter 6, *How Devices Communicate Using Application Messages,* of the *Neuron C Programmer's Guide* for more information.

Syntax

void resp_send (void);

Example

```
# define DATA_REQUEST 0
# define OK 1

when (msg_arrives(DATA_REQUEST)))
{
   unsigned 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();
   }
```

retrieve_status( ) *Function*

The **retrieve_status( )** function returns diagnostic status information to the Neuron C application. This information is also available to a network tool over the network, through the *Query Status* network diagnostics message. The **status_struct** structure, defined in **<status.h>**, is shown below.

Syntax

#include <status.h>
void retrieve_status (status_struct *_p_);

```
typedef struct status_struct {
   unsigned long  status_xmit_errors;
   unsigned long  status_transaction_timeouts;
   unsigned long  status_rcv_transaction_full;
   unsigned long  status_lost_msgs;
   unsigned long  status_missed_msgs;
   unsigned    status_reset_cause;
   unsigned    status_node_state;
   unsigned    status_version_number
   unsigned    status_error_log;
   unsigned    status_model_number;
} status_struct;
```

status_xmit_errors A count of the transmission errors that have been detected on the network. A transmission error is detected through a CRC error during packet reception. This error could result from a collision, noisy medium, or excess signal attenuation.

status_transaction_timeouts
A count of the timeouts that have occurred in attempting to carry out acknowledged or request/response transactions initiated by the device.

status_rcv_transaction_full
The number of times an incoming repeated, acknowledged, or request message was lost because there was no more room in the receive transaction database. The size of this database can be set through a pragma during compilation (**#pragma receive_trans_count**).

status_lost_msgs The number of messages that were addressed to the device and received in a network buffer that were discarded because there was no application buffer available for the message. The number of application buffers can be set through a pragma at compile time (**#pragma app_buf_in_count**).

status_missed_msgs The number of messages that were on the network but could not be received because there was no network buffer available for the message. The number of network buffers can be set through a pragma during compilation (**#pragma net_buf_in_count**).

status_reset_cause Identifies the source of the most recent reset. The values for this byte are (*x* = don't care):

Power-up reset	0b*xxxxxxx*1
External reset	0b*xxxxxx*10
Watchdog timer reset	0b*xxxx*1100
Software-initiated reset	0b*xxx*10100

status_node_state The state of the device. The states are:

Unconfigured	0x02
Unconfigured/no application	0x03
Configured/online	0x04
Configured/hard-offline	0x06
Configured/soft-offline	0x0C
Configured/bypass-mode	0x8C

status_version_number
The version number, which reflects the Neuron firmware version.

status_error_log The most recent error logged by the Neuron firmware or application. A value of 0 indicates no error. An error in the range of 1 to 127 is an application error and is unique to the application. An error in the range of 128 to 255 is a system error (system errors are documented in the *Neuron Tools Errors Guide*). The system errors are also available in the **<nm_err.h>** include file.

status_model_number
The model number of the Neuron Chip or Smart Transceiver. The value for this byte is one of the following:

0x00	for all Neuron 3150 Chips, and for an FT 3150 Smart Transceiver
0x01	for a PL 3150 Smart Transceiver
0x08	for Neuron 3120 Chip
0x09	for Neuron 3120E1 Chip
0x0A	for Neuron 3120E2 Chip
0x0B	for Neuron 3120E3 Chip
0x0C	for Neuron 3120A20 Chip
0x0D	for Neuron 3120E5 Chip
0x0E	for Neuron 3120E4 Chip or an FT 3120 Smart Transceiver
0x0F	for a PL 3120 Smart Transceiver
0x11	for a PL 3170 Smart Transceiver
0x20	for an FT 5000 Smart Transceiver
0x21	for a Neuron 5000 Processor
0x24	for an FT 6050 Smart Transceiver

0x25	for a Neuron 6050 Chip
0x26	for an FT 6010 Smart Transceiver

Example

For an example of the use of this function, see Chapter 7, *Additional Features,* of the *Neuron C Programmer's Guide*.

reverse( ) *Built-in Function*

The **reverse( )** built-in function reverses the bits in *a*.

Syntax

unsigned int reverse (unsigned int *a***);**

Example

```
void f(void)
{
   unsigned value = reverse(0xE3);
   // now value is 0xC7
}
```

rotate_long_left( ) *Function*

The **rotate_long_left( )** function returns the bit-rotated value of *arg*. The bit positions are rotated the number of places determined by the *count* argument. The signedness of the argument does not affect the result. Bits that are rotated out from the upper end of the value are rotated back in at the lower end. See also **rotate_long_right( )**, **rotate_short_left( )**, and **rotate_short_right( )**.

Syntax

#include <byte.h>
long rotate_long_left (long *arg***, unsigned** *count***);**

Example

```
#include <byte.h>

void f(void)
{
   long k = rotate_long_left(0x3F00, 3);
   // k now contains 0xF801
}
```

rotate_long_right( ) *Function*

The **rotate_long_right( )** function returns the bit-rotated value of *arg*. The bit positions are rotated the number of places determined by the *count* argument. The signedness of the argument does not affect the result. Bits that are rotated

out from the lower end of the value are rotated back in at the upper end. See also **rotate_long_left( )**, **rotate_short_left( )**, and **rotate_short_right( )**.

Syntax

#include <byte.h>
long rotate_long_right (long *arg***, unsigned** *count***);**

Example

```
#include <byte.h>

void f(void)
{
   long k = rotate_long_right(0x3F04, 3);
   // k now contains 0x87E0
}
```

rotate_short_left( ) *Function*

The **rotate_short_left( )** function returns the bit-rotated value of *arg*. The bit positions are rotated the number of places determined by the *count* argument. The signedness of the argument does not affect the result. Bits that are rotated out from the upper end of the value are rotated back in at the lower end. See also **rotate_long_left( )**, **rotate_long_right( )**, and **rotate_short_right( )**.

Syntax

#include <byte.h>
short rotate_short_left (short *arg***, unsigned** *count***);**

Example

```
#include <byte.h>

void f(void)
{
   short s = rotate_short_left(0x3F, 3);
   // s now contains 0xF9
}
```

rotate_short_right( ) *Function*

The **rotate_short_right( )** function returns the bit-rotated value of *arg*. The bit positions are rotated the number of places determined by the *count* argument. The signedness of the argument does not affect the result. Bits that are rotated out from the lower end of the value are rotated back in at the upper end. See also **rotate_long_left( )**, **rotate_long_right( )**, and **rotate_short_left( )**.

Syntax

#include <byte.h>
short rotate_short_right (short *arg*, **unsigned** *count*);

Example

```
#include <byte.h>

void f(void)
{
   short s = rotate_short_right(0x3F, 3);
   // s now contains 0xE7
}
```

scaled_delay( ) *Function*

The **scaled_delay( )** function generates a delay that scales with the input clock for the Neuron Chip or the Smart Transceiver.

The formula for determining the duration of the delay is the following:

delay = (25.2 **count* + 7.2) **S* (*delay* is in microseconds)

In the formula above, the scaling factor *S* is determined by the input clock, as shown in the following table.

Table 31. 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	—

See also the **delay( )** and **msec_delay( )** functions. The **delay( )** function generates a delay that is not scaled and is only minimally dependent on the input

clock. The **msec_delay( )** function provides a scaled delay of up to 255 milliseconds.

Syntax

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

count	A delay value between 1 and 33333. The formula for determining the duration of the delay is based on count and the Neuron input clock (see above).

Example

```
IO_2 output bit software_one_shot;

void f(void)
{
   io_out(software_one_shot, 1);
      //turn it on
   scaled_delay(4);
      //approx. 108 µsec at 10MHz
   io_out(software_one_shot, 0);
      //turn it off
}
```

sci_abort( ) *Built-in Function*

The **sci_abort( )** built-in function terminates any outstanding SCI I/O operation in progress.

Syntax

void sci_abort (*io_object_name***);**

Example

```
IO_8 sci twostopbits baud(SCI_2400) iosci;

when (...)
{
   sci_abort(iosci);
}
```

sci_get_error( ) *Built-in Function*

The **sci_get_error( )** built-in function returns a cumulative OR of the bits shown below to specify data errors. Calling this function clears the SCI error state.

0x04	Framing error
0x08	Noise detected
0x10	Receive overrun detected

Syntax

unsigned short sci_get_error (*io_object_name***);**

Example

```
IO_8 sci twostopbits baud(SCI_2400) iosci;

when (io_out_ready(iosci)) {
  unsigned short sci_error;
  sci_error = sci_get_error(iosci));
  if (sci_error) {
    // Process SCI error
  }
  else {
    // Process end of SCI transmission ...
  }
}
```

service_pin_msg_send( ) *Function*

The **service_pin_msg_send( )** function attempts to send a service pin message. It returns non-zero if it is successful (queued for transmission in the network processor) and zero if not. This function is useful for automatic installation scenarios. For example, a device can automatically transfer its service pin message a random amount of time after powering up. This is also useful for devices that do not have a physical service pin, but have some other method for an installer to request a service pin message.

Syntax

#include <control.h>
int service_pin_msg_send (void);

Example

```
#include <control.h>

when ( ... )
{
   ...
   service_pin_msg_send();
}
```

service_pin_state( ) *Function*

The **service_pin_state( )** function allows an application program to read the service pin state. A state of 0 or 1 is returned. A value of 1 indicates the service pin is at logic zero. This function is useful for improving ease of installation and maintenance. For example, an application can check for the service pin being held low for three seconds following a reset, and then go unconfigured (for ease of re-installation in a new network).

Syntax

#include <control.h>
int service_pin_state (void);

Example

```
#include <control.h>

stimer three_sec_timer;

when (reset)
{
   if (service_pin_state()) three_sec_timer = 3;
}

when (timer_expires(three_sec_timer))
{
   if (service_pin_state()) {
      // Service pin still depressed
      // go to unconfigured state
      go_unconfigured();
   }
}
```

Note this example is functional, but incomplete: the device would go unconfigured if the service pin is still being pressed after three seconds, or when it has been released and, by chance, is pressed again when the three seconds expire. A more robust device would stop the timer when the service pin is being released, to facilitate the press-and-hold scheme.

---

set_bit( ) *Function*

The **set_bit( )** function sets a bit in a bit array pointed to by *array*. Bits are numbered from left to right in each byte, so that the first bit in the array is the most significant bit of the first byte in the array. Like all arrays in C, this first element corresponds to index 0 (*bitnum* 0). When managing a number of bits that are all similar, a bit array can be more code-efficient than a series of bitfields because the array can be accessed using an array index rather than separate lines of code for each bitfield. See also **clr_bit( )** and **tst_bit( )**.

Syntax

#include <byte.h>
void set_bit (void ***_array_, unsigned** _bitnum_**);**

Example

```
#include <byte.h>

unsigned short a[4];

void f(void)
{
```

```
   memset(a, 0, 4);     // Clears all bits at once
   set_bit(a, 4); // Sets a[0] to 0x08 (5th bit)
}
```

set_eeprom_lock( ) *Function*

The **set_eeprom_lock( )** function allows the application to control the state of the EEPROM lock. This feature is available only for:

- Neuron 3120xx Chips or FT 3120 Smart Transceivers with system firmware Version 4 or later
- Neuron 3150 Chips and FT 3150 Smart Transceivers with system firmware Version 6 or later
- Series 5000 chips and Series 6000 chips

The function enables or disables the lock (with a TRUE or FALSE argument, respectively).

The EEPROM lock feature reduces the chances that a hardware failure or application anomaly can lead to a corruption of checksummed onchip EEPROM or offchip EEPROM or flash memory. The lock is automatically suspended while a device is offline to allow network management operations to occur. The application must release the lock prior to performing self-configuration. Application EEPROM variables are not locked. For more information, including a discussion of the drawbacks of using this feature, see **#pragma eeprom_locked** in Chapter 2, *Compiler Directives*, on page 21.

Syntax

#include <control.h>
void set_eeprom_lock (boolean *lock***);**

Example

```
#include <control.h>

when (reset)
{
   // Lock the EEPROM to prevent accidental writes
   set_eeprom_lock(TRUE);
}
...
void f(void)
{
   // Unlock EEPROM for update
   set_eeprom_lock(FALSE);
   ...//Update EEPROM
   //Relock EEPROM
   set_eeprom_lock (TRUE)
   ...
}
```

Signed 32-bit Arithmetic Support Functions

void s32_abs (const s32_type *_arg1_, **s32_type** *_arg2_**);**

void s32_add (const s32_type *_arg1_, **const s32_type** *_arg2_, **s32_type** *_arg3_**);**

int s32_cmp (const s32_type *_arg1_, **const s32_type** *_arg2_**);**

void s32_dec (s32_type *_arg1_**);**

void s32_div (const s32_type *_arg1_, **const s32_type** *_arg2_, **s32_type** *_arg3_**);**

void s32_div2 (s32_type *_arg1_**);**

void s32_eq (const s32_type *_arg1_, **const s32_type** *_arg2_**);**

void s32_from_ascii (const char *_arg1_, **s32_type** *_arg2_**);**

void s32_from_slong (signed long _arg1_, **s32_type** *_arg2_**);**

void s32_from_ulong (unsigned long _arg1_, **s32_type** *_arg2_**);**

void s32_ge (const s32_type *_arg1_, **const s32_type** *_arg2_**);**

void s32_gt (const s32_type *_arg1_, **const s32_type** *_arg2_**);**

void s32_inc (s32_type *_arg1_**);**

void s32_le (const s32_type *_arg1_, **const s32_type** *_arg2_**);**

void s32_lt (const s32_type *_arg1_, **const s32_type** *_arg2_**);**

void s32_max (const s32_type *_arg1_, **const s32_type** *_arg2_, **s32_type** *_arg3_**);**

void s32_min (const s32_type *_arg1_, **const s32_type** *_arg2_, **s32_type** *_arg3_**);**

void s32_mul (const s32_type *_arg1_, **const s32_type** *_arg2_, **s32_type** *_arg3_**);**

void s32_mul2 (s32_type *_arg1_**);**

void s32_ne (const s32_type *_arg1_, **const s32_type** *_arg2_**);**

void s32_neg (const s32_type *_arg1_, **s32_type** *_arg2_**);**

void s32_rand (s32_type *_arg1_**);**

void s32_rem (const s32_type *_arg1_, **const s32_type** *_arg2_, **s32_type** *_arg3_**);**

int s32_sign (const s32_type *_arg1_**);**

void s32_sub (const s32_type *_arg1_, **const s32_type** *_arg2_, **s32_type** *_arg3_**);**

void s32_to_ascii (const s32_type *_arg1_, **char** *_arg2_**);**

signed long s32_to_slong (const s32_type ***_arg1_);**

unsigned long s32_to_ulong (const s32_type *_arg1_**);**

The signed 32-bit arithmetic support functions are part of the extended arithmetic library. See *Signed 32-Bit Integer Support Functions* on page 61 for a detailed explanation of the extended arithmetic support functions that are available.

sleep( ) Built-in Function

For Series 3100 devices, the **sleep( )** built-in function puts the Neuron Chip or Smart Transceiver in a low-power state. The processors are halted, and the

internal oscillator is turned off. Any of the three syntactical forms shown below can be used. The second form uses a declared I/O object's pin as a wakeup pin. The third form directly specifies a pin to be used for a wakeup event.

Series 5000 and 6000 devices do not support sleep mode.

The Neuron Chip or Smart Transceiver wakes up when any of the following conditions occurs:

- A message arrives (unless the COMM_IGNORE flag is set)
- The service pin is pressed
- The specified input object transition occurs (if one is specified)

See also Chapter 7, *Additional Features,* of the *Neuron C Programmer's Guide*.

Syntax

void sleep (unsigned int *flags***);**

void sleep (unsigned int *flags* **,** *io-object-name***);**

void sleep (unsigned int *flags* **,** *io-pin***);**

flags — One or more of the following three flags, or 0 if no flag is specified:

COMM_IGNORE	Causes incoming messages to be ignored
PULLUPS_ON	Enables all I/O pullup resistors (the service pin pullup is not affected)
TIMERS_OFF	Turns off all timers in the program

If two or more flags are used, they must be combined using either the **+** or the **|** operator.

io-object-name — Specifies an input object for any of the **IO_4** through **IO_7** pins. When any I/O transition occurs on the specified pin, the Neuron core wakes up. If neither this parameter nor the *io-pin* argument are specified, I/O is ignored after the Neuron core goes to sleep.

io-pin — Specifies one of the **IO_4** through **IO_7** pins directly instead of through a declared I/O object.

Example

```
IO_6 input bit wakeup;
...
when (flush_completes)
{
   sleep(COMM_IGNORE + TIMERS_OFF, wakeup);
}
```

spi_abort( ) *Function*

The **spi_abort( )** built-in function terminates any outstanding SPI I/O operation in progress.

Syntax

void spi_abort (*io-object-name***);**

Example

```
IO_8 spi master clock(4) iospi;

when (...)
{
   spi_abort(iospi);
}
```

spi_get_error( ) *Function*

The **spi_get_error( )** built-in function returns a cumulative OR of the bits shown below to specify data errors. Calling this function clears the SPI error state.

0x10	Mode fault occurred
0x20	Receive overrun detected

Syntax

unsigned short spi_get_error (*io-object-name***);**

Example

```
IO_8 spi master clock(4) iospi;

when (io_out_ready(iospi)) {
  unsigned short spi_error = spi_get_error(iospi));
  if (spi_error) {
    // Process SPI error
  } else {
    // Process end of SPI transmission ...
  }
}
```

strcat( ) *Function*

The **strcat( )** function appends a copy of the string *src* to the end of the string *dest*, resulting in concatenated strings (thus the name **strcat**, from string concatenate). The function returns a pointer to the string *dest*. See also **strchr( )**, **strcmp( )**, **strcpy( )**, **strlen( )**, **strncat( )**, **strncmp( )**, **strncpy( )**, and **strrchr( )**.

This function cannot be used to copy overlapping areas of memory, or to write into EEPROM memory or network variables.

Syntax

#include <string.h>
char *strcat (char *_dest_, const char *_src_);

Example

```
#include <string.h>

void f(void)
{
   char buf[40]

   strcpy(buf, "Hello");
   strcat(buf, " World");     // buf contains "Hello World"
   ...
}
```

strchr( ) *Function*

The **strchr( )** function searches the string *s* for the first occurrence of the character *c*. If the string does not contain *c*, the **strchr( )** function returns the null pointer. The NUL character terminator (**'\0'**) is considered to be part of the string, thus **strchr(s,'\0')** returns a pointer to the NUL terminator. See also **strcat( )**, **strcmp( )**, **strcpy( )**, **strlen( )**, **strncat( )**, **strncmp( )**, **strncpy( )**, and **strrchr( )**.

Syntax

#include <string.h>
char *strchr (const char *_s_, char _c_);

Example

```
#include <string.h>

void f(void)
{
   char buf[20];
   char *p;

   strcpy(buf, "Hello World");
   p = strchr(buf, 'o');      // Assigns &(buf[4]) to p
   p = strchr(buf, '\0');     // Assigns &(buf[11]) to p
   p = strchr(buf, 'x');      // Assigns NULL to p
}
```

strcmp( ) *Function*

The **strcmp( )** function compares the contents of the *s1* and *s2* strings, up to the NUL terminator character in the shorter string. The function performs a case-sensitive comparison. If the strings match identically, 0 is returned. When a mismatch occurs, the characters from both strings at the mismatch are

compared. If the first string's character is greater using an unsigned comparison, the return value is positive. If the second string's character is greater, the return value is negative.

The terminating NUL (**'\0'**) character is compared just as any other character. See also **strcat( )**, **strchr( )**, **strcpy( )**, **strlen( )**, **strncat( )**, **strncmp( )**, **strncpy( )**, and **strrchr( )**.

Syntax

#include <string.h>
int strcmp (const unsigned char *_s1_, const unsigned char *_s2_);

Example

```
#include <string.h>

void f(const unsigned char* test)
{
   int val = strcmp(test, "magic codeword");
   if (!val) {
      // Strings are equal
   } else if (val < 0) {
      // String test is less than the magic words
   } else {
      // String test is greater than the magic words
   }
}
```

---

strcpy( ) *Function*

The **strcpy( )** function copies the string pointed to by the parameter *src* into the string buffer pointed to by the parameter *dest*. The copy ends implicitly, when the terminating NUL (**'\0'**) character is copied—no string length information is available to the function. There is no attempt to ensure that the string can actually fit in the available memory. That task is left up to the programmer. See also **strcat( )**, **strchr( )**, **strcmp( )**, **strlen( )**, **strncat( )**, **strncmp( )**, **strncpy( )**, and **strrchr( )**.

This function cannot be used to copy overlapping areas of memory, or to write into EEPROM memory. Use of the compiler directive **#pragma relaxed_casting_on** is needed to copy to a network variable, and doing so does not automatically propagate the network variable update (see the **propagate( )** function).

Syntax

#include <string.h>
char *strcpy (char *_dest_, const char *_src_);

Example

```
#include <string.h>
```

```
void f(void)
{
   char s1[20], s2[20];

   strcpy(s1, "Hello World");
   strcpy(s2, s1);
}
```

strlen( ) *Function*

The **strlen( )** function returns the length of the string *s*, not including the terminating NUL (**'\0'**) character. See also **strcat( )**, **strchr( )**, **strcmp( )**, **strcpy( )**, **strncat( )**, **strncmp( )**, **strncpy( )**, and **strrchr( )**.

Syntax

#include <string.h>
unsigned long strlen (const char ****s*);**

Example

```
#include <string.h>

void f(void)
{
   unsigned long length = strlen("Hello, world!");
}
```

strncat( ) *Function*

The **strncat( )** function appends a copy of the first *len* characters from the string *src* to the end of the string *dest*, and then adds a NUL (**'\0'**) character, resulting in concatenated strings (thus the name **strncat**, from string concatenate). If the *src* string is shorter than *len*, no characters are copied past the NUL character. The function returns a pointer to the string *dest*. See also **strcat( )**, **strchr( )**, **strcmp( )**, **strcpy( )**, **strlen( )**, **strncmp( )**, **strncpy( )**, and **strrchr( )**.

This function cannot be used to copy overlapping areas of memory, or to write into EEPROM memory or network variables.

Syntax

#include <string.h>
char *strncat (char ****dest*, char ****src***, unsigned long** *len***);**

Example

```
#include <string.h>

void f(void)
{
   char buf[40]
```

```
   strncpy(buf, "Hello Beautiful", 16);
   strncat(buf, "World News Tonight", 5);
      // buf now contains "Hello Beautiful World"
}
```

strncmp( ) Function

The **strncmp( )** function compares the contents of the *s1* and *s2* strings, up to the NUL ('**\0**') terminator character in the shorter string, or until *len* characters have been compared, whichever occurs first. The function performs a case-sensitive comparison. If the strings match identically, 0 is returned.

When a mismatch occurs, the characters from both strings at the mismatch are compared. If the first string's character is greater using an unsigned comparison, the return value is positive. If the second string's character is greater, the return value is negative. The terminating NUL character is compared just as any other character. See also **strcat( )**, **strchr( )**, **strcmp( )**, **strcpy( )**, **strlen( )**, **strncat( )**, **strncpy( )**, and **strrchr( )**.

Syntax

#include <string.h>
int strncmp (const unsigned char * *s1***, const unsigned char *** *s2***,**
unsigned long *len***);**

Example

```
#include <string.h>

void f(const unsigned char* test)
{
   int val  = strncmp(test, ”magic “, 6);  // Compare first
6 chars
   if (!val) {
      // Strings are equal within the first 6 characters
   } else if (val < 0) {
      // String test is less than “magic “ (first 6 chars)
   } else {
      // String test is greater than “magic “ (first 6
chars)
   }
}
```

strncpy( ) Function

The **strncpy( )** function copies the string pointed to by the *src* parameter into the string buffer pointed to by the *dest* parameter. The copy ends either when the terminating NUL ('**\0**') character is copied or when *len* characters have been copied, whichever comes first.

The function returns *dest*.

If the copy is terminated by the length, a NUL character is *not* added to the end of the destination string. See also **strcat( )**, **strchr( )**, **strcmp( )**, **strcpy( )**, **strlen( )**, **strncat( )**, **strncmp( )**, and **strrchr( )**.

This function cannot be used to copy overlapping areas of memory, or to write into EEPROM memory or network variables.

Syntax

#include <string.h>
char *strncpy (char *_dest_, const char *_src_, unsigned long _len_);

Example

```
#include <string.h>

char s[20];

void f(char *p)
{
   strncpy(s, p, sizeof(s));  // Prevent overflow
   s[sizeof(s)-1] = '\0';     // Force termination
}
```

strrchr( ) *Function*

The **strrchr( )** function scans a string for the last occurrence of a given character. The function scans a string in the reverse direction (hence the extra 'r' in the name of the function), looking for a specific character. The **strrchr( )** function finds the last occurrence of the character *c* in string *s*. The NUL ('**\0**') terminator is considered to be part of the string. The return value is a pointer to the character found, otherwise null. See also **strcat( )**, **strchr( )**, **strcmp( )**, **strcpy( )**, **strlen( )**, **strncat( )**, **strncmp( )**, and **strncpy( )**.

Syntax

#include <string.h>
char *strrchr (const char *_s_, char _c_);

Example

```
#include <string.h>

void f(void)
{
   char buf[20];
   char *p;

   strcpy(buf, "Hello World");
   p = strrchr(buf, 'o');     // Assigns &(buf[7]) to p
   p = strrchr(buf, '\0');    // Assigns &(buf[11]) to p
   p = strrchr(buf, 'x');     // Assigns NULL to p
}
```

swap_bytes( ) *Built-in Function*

The **swap_bytes( )** built-in function returns the byte-swapped value of *a*. See also **high_byte( )**, **low_byte( )**, and **make_long( )**.

Syntax

unsigned long swap_bytes (unsigned long *a***);**

Example

```
long k;

void f(void)
{
   k = swap_bytes(0x1234);    // k now contains 0x3412L
}
```

timers_off( ) *Function*

The **timers_off( )** function turns off all software timers. This function could be called, for example, before an application goes offline.

Syntax

#include <control.h>
void timers_off (void);

Example

```
when (...)
{
   timers_off();
   go_offline();
}
```

touch_bit( ) *Built-in Function*

The **touch_bit( )** function writes and reads a single bit of data on a 1-Wire® bus. It can be used for either reading or writing. For reading, the *write-data* argument should be one (0x01), and the return value contains the bit as read from the bus. For writing, the bit value in the *write-data* argument is placed on the 1-Wire bus, and the return value normally contains that same bit value, and can be ignored. This function provides access to the same internal function that **touch_byte( )** calls.

Syntax

unsigned touch_bit(*io-object-name***, unsigned** *write-data***);**

Example

```
void f(void)
{
   unsigned dataIn, dataOut;
   ...
   dataOut = 42;
   dataIn = touch_bit(ioObj, dataOut);
}
```

touch_byte( ) *Built-in Function*

The **touch_byte( )** function sequentially writes and reads eight bits of data on a 1-Wire bus. It can be used for either reading or writing. For reading, the *write-data* argument should be all ones (0xFF), and the return value contains the eight bits as read from the bus. For writing, the bits in the *write-data* argument are placed on the 1-WIRE bus, and the return value normally contains those same bits.

Syntax

unsigned touch_byte(*io-object-name***, unsigned** *write-data***);**

Example

```
void f(void)
{
   unsigned dataIn, dataOut;
   ...
   dataOut = 42;
   dataIn = touch_byte(ioObj, dataOut);
}
```

touch_byte_spu( ) *Built-in Function*

This function applies to 1-Wire bus devices that require the bus to be actively held high during certain device operations. These devices require more current than a typical external pull-up resistor can provide for device operations. An example of such a device is the Maxim Integrated Products DS18S20 High-Precision 1-Wire Digital Thermometer. For other 1-Wire devices, use the standard **touch_byte( )** function.

The **touch_byte_spu( )** function writes eight bits of data on a 1-Wire bus. Unlike the standard **touch_byte( )** function, this function cannot be used for reading data. For writing, the bits in the *data* argument are placed on the 1-Wire bus, and the bus is left in the actively driven high state.

Syntax

extern void touch_byte_spu(unsigned *pinmask*, **unsigned** *data***);**

pinmask	Specifies a single-bit representation for which I/O pins IO0..IO7 to drive high when idle. Valid values are 0x01 (for IO0) to 0x08 (for IO7).

data Specifies the data to place on the 1-Wire bus.

Example

```
#define 1WIREPIN 0x02;

void f(void)
{  touch_byte_spu(1WIREPIN, 42);
}
```

touch_first( ) *Built-in Function*

The **touch_first( )** function executes the ROM Search algorithm as described in application note 937, *Book of iButton Standards*, from Maxim Integrated Products. Both functions make use of a **search_data_s** data structure for intermediate storage of a bit marker and the current ROM data. This data structure is automatically defined in Neuron C, regardless of whether a program references the touch I/O functions.

A return value of TRUE indicates whether a device was found, and if so, that the data stored at **rom_data[ ]** is valid. A FALSE return value indicates no device found. The **search_done** flag is set to TRUE when there are no more devices on the 1-Wire bus. The **last_discrepancy** variable is used internally, and should not be modified.

To start a new search, first call **touch_first( )**. Then, as long as the **search_done** flag is not set, call **touch_next( )** as many times as are required. For a Series 3100 device, each call to **touch_first( )** or **touch_next( )** takes 41 ms to execute at 10 MHz (63 ms at 5 MHz) when a device is being read. For a Series 5000 and 6000 device, each call to **touch_first( )** or **touch_next( )** takes 14 ms to execute at 80 MHz (29 ms at 10 MHz) when a device is being read.

Syntax

int touch_first(*io-object-name***, search_data ****sd***);**

Example

```
typedef struct search_data_s {
   int  search_done;
   int  last_discrepancy;
   unsigned rom_data[8];
} search_data;

search_data sd;

void f(void)
{
   sd.rom_data[0] = ...;
   sd.rom_data[1] = ...;
   ...
   sd.rom_data[7] = ...;

   if (touch_first(ioObj, &sd)) {
      // Found ...
```

```
   }
}
```

touch_next( ) *Built-in Function*

The **touch_next( )** function executes the ROM Search algorithm as described in application note 937, *Book of iButton Standards*, from Maxim Integrated Products. Both functions make use of a **search_data_s** data structure for intermediate storage of a bit marker and the current ROM data. This data structure is automatically defined in Neuron C, regardless of whether a program references the touch I/O functions.

A return value of TRUE indicates whether a device was found, and if so, that the data stored at **rom_data[ ]** is valid. A FALSE return value indicates no device found. The **search_done** flag is set to TRUE when there are no more devices on the 1-Wire bus. The **last_discrepancy** variable is used internally, and should not be modified.

To start a new search, first call **touch_first( )**. Then, as long as the **search_done** flag is not set, call **touch_next( )** as many times as are required. For a Series 3100 device, each call to **touch_first( )** or **touch_next( )** takes 41 ms to execute at 10 MHz (63 ms at 5 MHz) when a device is being read. For a Series 5000 and 6000 device, each call to **touch_first( )** or **touch_next( )** takes 14 ms to execute at 80 MHz (29 ms at 10 MHz) when a device is being read.

Syntax

int touch_next(*io-object-name***, search_data ****sd***);**

Example

```
typedef struct search_data_s {
   int  search_done;
   int  last_discrepancy;
   unsigned rom_data[8];
} search_data;

search_data sd;

void f(void)
{
   sd.rom_data[0] = ...;
   sd.rom_data[1] = ...;
   ...
   sd.rom_data[7] = ...;

   if (touch_first(ioObj, &sd)) {
      // Found ...
      while (!(sd.search_done)) {
         if (touch_next(ioObj, &sd)) {
            // Found another ...
         }
      }
   }
}
```

touch_read_spu( ) Built-in Function

This function applies to 1-Wire bus devices that require the bus to be actively held high during certain device operations. These devices require more current than a typical external pull-up resistor can provide for device operations. An example of such a device is the Maxim Integrated Products DS18S20 High-Precision 1-Wire Digital Thermometer. For other 1-Wire devices, use the standard **touch_byte( )** function.

The **touch_read_spu( )** function reads a specified number of bits of data on a 1-Wire bus. This function ensures that the bus is not in the actively driven high state prior to reading the data.

Syntax

extern void touch_read_spu(unsigned *pinmask*, **unsigned** *dp, **unsigned** *count***);**

pinmask	Specifies a single-bit representation for which I/O pins IO0..IO7 to drive high when idle. Valid values are 0x01 (for IO0) to 0x08 (for IO7).
dp	Specifies a pointer to the buffer into which the function stores the read data.
count	Specifies the number of bits to read.

Example

```
#define 1WIREPIN 0x02;

unsigned sensorData;

void f(void)
{
  ...

  touch_read_spu(1WIREPIN, *sensorData,
        sizeof(sensorData));
}
```

touch_reset( ) Built-in Function

The **touch_reset( )** function asserts the reset pulse and returns a one (1) value if a presence pulse was detected, or a zero (0) if no presence pulse was detected, or a minus-one (-1) value if the 1-Wire bus appears to be stuck low. The operation of this function is controlled by several timing constants. The first is the reset pulse period, which is 500 µs. Next, the Neuron Chip or Smart Transceiver releases the 1-Wire bus and waits for the 1-Wire bus to return to the high state. This period is limited to 275 µs, after which the **touch_reset( )** function returns a (-1) value with the assumption that the 1-Wire bus is stuck low. There also is a minimum value for this period: for a Series 3100 device, it must be >4.8 µs @10 MHz, or >9.6 µs @5 MHz; for a Series 5000 and 6000 device, it must be >0.3 µs @ 80 MHz, or >4.8 µs @5 MHz.

The **touch_reset( )** function does not return until the end of the presence pulse has been detected.

Syntax

int touch_reset (*io-object-name***);**

Example

```
void f(void)
{
   touch_reset(ioObj);
}
```

---

touch_reset_spu( ) *Built-in Function*

This function applies to 1-Wire bus devices that require the bus to be actively held high during certain device operations. These devices require more current than a typical external pull-up resistor can provide for device operations. An example of such a device is the Maxim Integrated Products DS18S20 High-Precision 1-Wire Digital Thermometer. For other 1-Wire devices, use the standard **touch_reset( )** function.

The **touch_reset_spu( )** function asserts the reset pulse and returns a one (1) value if a presence pulse was detected, or a zero (0) if no presence pulse was detected, or a minus-one (-1) value if the 1-Wire bus appears to be stuck low. The operation of this function is controlled by several timing constants. The first is the reset pulse period, which is 500 µs. Next, the Neuron Chip or Smart Transceiver releases the 1-Wire bus and waits for the 1-Wire bus to return to the high state. This period is limited to 275 µs, after which the **touch_reset_spu( )** function returns a (-1) value with the assumption that the 1-Wire bus is stuck low. There also is a minimum value for this period: for a Series 3100 device, it must be >4.8 µs @10 MHz, or >9.6 µs @5 MHz; for a Series 5000 and 6000 device, it must be >0.3 µs @ 80 MHz, or >4.8 µs @5 MHz.

The **touch_reset_spu( )** function does not return until the end of the presence pulse has been detected. This function ensures that the bus is not in the actively driven high state prior to asserting the reset pulse.

You can use this function to reset the state of the bus so that you can use the standard **touch_first( )** and **touch_next( )** functions.

Syntax

extern int touch_reset_spu(unsigned *pinmask***);**

pinmask	Specifies a single-bit representation for which I/O pins IO0..IO7 to drive high when idle. Valid values are 0x01 (for IO0) to 0x08 (for IO7).

Example

```
#define 1WIREPIN 0x02;

void f(void)
```

```
{
  ...
  int rc = touch_reset_spu(1WIREPIN);
}
```

touch_write_spu( ) Built-in Function

This function applies to 1-Wire bus devices that require the bus to be actively held high during certain device operations. These devices require more current than a typical external pull-up resistor can provide for device operations. An example of such a device is the Maxim Integrated Products DS18S20 High-Precision 1-Wire Digital Thermometer. For other 1-Wire devices, use the standard **touch_byte( )** function.

The **touch_write_spu( )** function writes a specified number of bits of data on a 1-Wire bus. This function ensures that the bus is in the actively driven high state after writing the data.

Syntax

extern void touch_write_spu(unsigned *pinmask*, **const unsigned** *dp, **unsigned** *count***);**

pinmask	Specifies a single-bit representation for which I/O pins IO0..IO7 to drive high when idle. Valid values are 0x01 (for IO0) to 0x08 (for IO7).
dp	Specifies a pointer to the buffer into which the function stores the read data.
count	Specifies the number of bits to write.

Example

```
#define 1WIREPIN 0x01;

const unsigned actuatorData;

void f(void)
{
  ...

  actuatorData = 168;
  touch_write_spu(1WIREPIN, &actuatorData,
           sizeof(actuatorData));
}
```

tst_bit( ) Function

The **tst_bit( )** function tests a bit in a bit array pointed to by *array*. Bits are numbered from left to right in each byte, so that the first bit in the array is the most significant bit of the first byte in the array. Like all arrays in C, this first element corresponds to index 0 (*bitnum* 0). The function returns a boolean value, TRUE if bit was set, FALSE if bit was not set. When managing a number of bits that are all similar, a bit array can be more code-efficient than a series of

bitfields because the array can be accessed using an array index rather than separate lines of code for each bitfield. See also **clr_bit( )** and **set_bit( )**.

Syntax

#include <byte.h>
boolean tst_bit (void ****array*, unsigned** *bitnum***);**

Example

```
#include <byte.h>

unsigned short a[4];

void f(void)
{
   memset(a, 0, sizeof(a));   // Clear all bits at once
   set_bit(a, 4); // Set a[0] to 0x08 (5th bit)

   if (tst_bit(a, 4)) {
      // Code executes here if bit was set
   }
}
```

update_address( ) *Function*

The **update_address( )** function copies from the structure referenced by the *address* pointer parameter to the address table entry specified by the *index* parameter.

Important: This function has a mechanism that ensures that a reset or power cycle during an EEPROM modification does not cause the device to go unconfigured. This mechanism uses the error log to serve as a semaphore. Thus, the error log is written to on every call to this function, even if the net effect of the function is to not modify or write to the configuration data at all (because the new contents match the old). Applications must minimize calls to this function to ensure that the maximum number of supported writes for EEPROM is not exceeded over the lifetime of the application.

See the ISO/IEC 14908 (ANSI/EIA/CEA-709.1) *Control Network Specification* for a description of the data structure.

Syntax

#include <access.h>
void update_address (const address_struct ****address*, unsigned** *index***);**

Example

```
#include <access.h>
address_struct address_copy;
msg_tag my_mt;

void f(void)
```

```
{
   address_copy = *access_address(
               addr_table_index(my_mt));
   // Modify the address_copy here as necessary
   ...
   update_address(&address_copy,
               addr_table_index(my_mt));
}
```

update_alias( ) *Function*

The **update_alias( )** function copies from the structure referenced by the *alias* pointer parameter to the alias table entry specified by the *index* parameter.

The Neuron 3120 Chip with version 4 firmware does not support aliasing.

Important: This function has a mechanism that ensures that a reset or power cycle during an EEPROM modification does not cause the device to go unconfigured. This mechanism uses the error log to serve as a semaphore. Thus, the error log is written to on every call to this function, even if the net effect of the function is to not modify or write to the configuration data at all (because the new contents match the old). Applications must minimize calls to this function to ensure that the maximum number of supported writes for EEPROM is not exceeded over the lifetime of the application.

See the ISO/IEC 14908 (ANSI/EIA/CEA-709.1) *Control Network Specification* for a description of the data structure.

Series 6000 chips and version 21 Neuron firmware introduce support for an extended address table, which requires an extended alias configuration structure to accommodate the potentially larger address table index associated with the alias. The standard Neuron C *access.h* include file defines both the traditional alias configuration structure and the extended form (alias_struct, alias_struct_ex). An *ALIAS_STRUCT_TYPE* preprocessor definition is supplied which equates to the correct type of the alias configuration structure for the current compilation target.

Syntax

#include <access.h>
void update_alias (const *ALIAS_STRUCT_TYPE*******alias***, unsigned** *index***);**

Example

```
#include <access.h>
ALIAS_STRUCT_TYPE alias_copy;


void f(unsigned index)
{
   alias_copy = *(access_alias(index));
   // Modify the alias_copy here as necessary
   …
   update_alias(&alias_copy, index);
}
```

update_clone_domain( ) *Function*

The **update_clone_domain( )** function copies from the structure referenced by the *domain* pointer parameter to the domain table entry specified by the *index* parameter.

This function differs from **update_domain( )** in that it is only used for a cloned device. A cloned device is a device that does not have a unique domain/subnet/node address on the network. Typically, cloned devices are intended for low-end systems where network tools are not used for installation. The LonTalk protocol inherently disallows this configuration because devices reject messages that have the same source address as their own address. The **update_clone_domain( )** function enables a device to receive a message with a source address equal to its own address. There are several restrictions when using cloned devices; see the *NodeBuilder FX User's Guide*.

Important: This function has a mechanism that ensures that a reset or power cycle during an EEPROM modification does not cause the device to go unconfigured. This mechanism uses the error log to serve as a semaphore. Thus, the error log is written to on every call to this function, even if the net effect of the function is to not modify or write to the configuration data at all (because the new contents match the old). Applications must minimize calls to this function to ensure that the maximum number of supported writes for EEPROM is not exceeded over the lifetime of the application.

More information about cloned devices can be found in the ISO/IEC 14908 (ANSI/EIA/CEA-709.1) *Control Network Specification*.

Syntax

#include <access.h>
void update_clone_domain (domain_struct ****domain*, unsigned** *index***);**

Example

```
#include <access.h>
domain_struct domain_copy;

void f(void)
{
   domain_copy = *(access_domain(0));
   // Modify the domain copy as necessary
   update_clone_domain(&domain_copy, 0);
}
```

update_config_data( ) *Function*

The **update_config_data( )** function copies from the structure referenced by the *p* configuration data pointer parameter to the **config_data** variable. The **config_data** variable is declared **const**, but can be modified through this function. The **config_data** variable is automatically defined for every program in the **<echelon.h>** file.

Important: This function has a mechanism that ensures that a reset or power cycle during an EEPROM modification does not cause the device to go

unconfigured. This mechanism uses the error log to serve as a semaphore. Thus, the error log is written to on every call to this function, even if the net effect of the function is to not modify or write to the configuration data at all (because the new contents match the old). Applications must minimize calls to this function to ensure that the maximum number of supported writes for EEPROM is not exceeded over the lifetime of the application.

See the ISO/IEC 14908 (ANSI/EIA/CEA-709.1) *Control Network Specification* for a description of the data structure.

Syntax

#include <access.h>
void update_config_data (const config_data_struct ****p*);**

Example

```
#include <access.h>
config_data_struct  config_data_copy;

void f(void)
{
   config_data_copy = config_data;
   // Modify the config_data_copy as necessary
   update_config_data(&config_data_copy);
}
```

update_domain( ) *Function*

The **update_domain( )** function copies from the structure referenced by the *domain* pointer parameter to the domain table entry specified by the *index* parameter.

Important: This function has a mechanism that ensures that a reset or power cycle during an EEPROM modification does not cause the device to go unconfigured. This mechanism uses the error log to serve as a semaphore. Thus, the error log is written to on every call to this function, even if the net effect of the function is to not modify or write to the configuration data at all (because the new contents match the old). Applications must minimize calls to this function to ensure that the maximum number of supported writes for EEPROM is not exceeded over the lifetime of the application.

See the ISO/IEC 14908 (ANSI/EIA/CEA-709.1) *Control Network Specification* for a description of the data structure.

Syntax

#include <access.h>
void update_domain (domain_struct ****domain*, unsigned** *index***);**

Example

```
#include <access.h>
domain_struct domain_copy;
```

```
void f(void)
{
   domain_copy = *access_domain(0);
   // Modify the domain_copy as necessary
   ...
   update_domain(&domain_copy, 0);
}
```

update_nv( ) *Function*

The **update_nv( )** function copies from the structure referenced by the *nv-entry* pointer parameter to the network variable configuration table entry as specified by the *index* parameter.

Important: This function has a mechanism that ensures that a reset or power cycle during an EEPROM modification does not cause the device to go unconfigured. This mechanism uses the error log to serve as a semaphore. Thus, the error log is written to on every call to this function, even if the net effect of the function is to not modify or write to the configuration data at all (because the new contents match the old). Applications must minimize calls to this function to ensure that the maximum number of supported writes for EEPROM is not exceeded over the lifetime of the application.

See the ISO/IEC 14908 (ANSI/EIA/CEA-709.1) *Control Network Specification* for a description of the data structure.

Series 6000 chips and version 21 Neuron firmware introduce support for an extended address table, which requires a extended network variable configuration structure to accommodate the potentially larger address table index associated with the network variable. The standard Neuron C *access.h* include file defines both the traditional network variable configuration structure and the extended form (nv_struct, nv_struct_ex). An *NV_STRUCT_TYPE* preprocessor definition is supplied which equates to the correct type of the network variable configuration structure for the current compilation target.

Syntax

#include <access.h>
void update_nv (const *NV_STRUCT_TYPE* ******nv-entry***, unsigned*** *index***);**

Example

```
#include <access.h>
NV_STRUCT_TYPE nv_copy;
network output SNVT_switch nvoSwitch;

void f(void)
{
   nv_copy = *access_nv(nvoSwitch::global_index);
   // Modify the nv_copy here as necessary
   ...
   update_nv(&nv_copy, nvoSwitch::global_index);
}
```

update_program_id( ) *Function*

The **update_program_id( )** function copies the 8-byte array referenced by the *pid_p* pointer parameter to the program ID stored in the device's EEPROM.

Important: This function has a mechanism that ensures that a reset or power cycle during an EEPROM modification does not cause the device to go unconfigured. This mechanism uses the error log to serve as a semaphore. Thus, the error log is written to on every call to this function, even if the net effect of the function is to not modify or write to the configuration data at all (because the new contents match the old). Applications must minimize calls to this function to ensure that the maximum number of supported writes for EEPROM is not exceeded over the lifetime of the application.

Syntax

#include <access.h>
void update_program_id (unsigned char * *pid_p***);**

Example

```
#include <access.h>
unsigned char progID_copy[8];

void f(void)
{
   update_program_id(progID_copy);
}
```

watchdog_update( ) *Function*

The **watchdog_update( )** function updates the watchdog timer. For Series 3100 devices, the watchdog timer times out in the range of .84 to 1.68 seconds with a 10 MHz Neuron input clock. The watchdog timer period scales inversely with the input clock frequency. 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 scheduler updates the watchdog timer before entering each critical section. To ensure that the watchdog timer does not expire, call the **watchdog_update( )** function periodically within long tasks (or in bypass mode). The **post_events( )**, **msg_receive( )**, and **resp_receive( )** functions also update the watchdog timer, as does the **pulsecount** output object.

Within long tasks when the scheduler does not run, the watchdog timer could expire, causing the device to reset. To prevent the watchdog timer from expiring, an application program can call the **watchdog_update( )** function periodically.

Syntax

#include <control.h>
void watchdog_update (void);

Example

```
void f(void)
{
   boolean still_processing;
   ...
   while (still_processing) {
      watchdog_update();
      ...
   }
}
```

4

Timer Declarations

This chapter provides reference information for declaring and using Neuron C timers.

Neuron C timers are an application development feature for Neuron-hosted devices and do not apply to model files.

Timer Object

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. Assigning a value of zero to this name turns the timer off. The value of a timer object is an **unsigned long** (0..65535); however, the maximum value assigned to a millisecond timer cannot exceed 64000. A timer that is running or has expired can be restarted by assigning a new value to this object. The timer object can be evaluated while the timer is running, and it indicates the time remaining. Up to 15 timer objects can be declared in an application.
initial-value	An optional initial value to be loaded into the timer on power-up or reset. Zero is loaded if no initial-value is supplied (and therefore the timer is off).

When a timer expires, the **timer_expires** event becomes TRUE. The **timer_expires** event returns to FALSE after the **timer_expires** expression is read, or when the timer is set to zero.

Example:

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

when (timer_expires(led timer))
{
   toggle_led();
   led_timer = 2;  // restart timer with value of 2 sec
}
```

The **timers_off( )** function can be used to turn off all application timers – for example, before an application goes offline. See Chapter 2, *Focusing on a Single Device,* of the *Neuron C Programmer's Guide* for a discussion of timer accuracy.

5

Network Variable, Configuration Property, and Message Tag Declarations

This chapter describes the network variable, configuration property, and application message tag declarations for use in Neuron C programs. It also describes how configuration properties are associated with a device, with a functional block on the device, or with a network variable on the device. Finally, this chapter describes the syntax for accessing the configuration properties from the device's program.

Network variables, configuration properties, and message tags are part of a device's interface. The discussion in this chapter applies to development of both Neuron-hosted applications and host-based applications with model files.

Introduction

The external application interface of a LONWORKS device consists of its functional blocks, network variables, and configuration properties. The *network variables* are the device's means of sending and receiving data using interoperable data types and using an event-driven programming model. The *configuration properties* are the device's means of providing externally exposed configuration data, again using interoperable data types. The configuration data items can be read and written by a network tool. The device interface is organized 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*.

Configuration properties can be implemented using two different techniques. The first, called a *configuration network variable*, uses a network variable to implement a configuration property. This has the advantage of enabling the configuration property to be modified by another LONWORKS device, just like any other network variable. It also has the advantage of having the 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 of a network variable. Network variables are limited to 31 bytes each on series 3100 and 5000 Neuron Chips and Smart Transceivers. Neuron C Version 2.3 and Series 6000 chips support network variables up to 228 bytes each. Model files, such as those used with the ShortStack Development Kit, also support network variables up to 228 bytes.

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 of the device that is accessible by the application. When there are two value files, one contains writeable configuration properties and the second contains read-only data. To permit a network tool to access 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 other than the limitations on the size of a file. The disadvantages are that other devices cannot connect to or poll a configuration property implemented within a configuration file; requiring a network tool to modify a configuration property implemented within a configuration file; and, no events are automatically generated upon an update of a configuration property implemented within a configuration file. The application can force notification of updates by requiring network tools to disable a functional block or take a device offline when a configuration property is updated, and then re-enable or put the device back online.

You can declare functional blocks, network variables, and configuration properties using the Neuron C Version 2 syntax. You can declare configuration properties that are implemented within configuration files or configuration

network variables. The Neuron C Version 2 compiler uses these declarations to generate the value files, template file, all required self-identification and self-documentation data, and the device interface file (**.xif** extension) for a Neuron C application.

Network Variable Declarations Syntax

The complete syntax for declaring a network variable is one of the following:

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

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

The brackets around *array-bound* are shown in **bold** type. The brackets do not, in this case, indicate an optional field. They are a required part of the syntax of declaring an array, and must be entered into the program code.

The maximum number of network variables available to a device depends on the target chip type, firmware version, and the device development platform used. For all chips, firmware, and development platforms, each element of a network variable array counts as a separate network variable relative to the maximum number and configuration network variables count towards that maximum number.

Network Variable Modifiers (netvar-modifier)

One or more of the following optional modifiers can be included in the declaration of each network variable:

sync | **synchronized** Specifies that all values assigned to this network variable must be propagated, and in their original order. Mutually exclusive with the **polled** modifier.

polled Specifies that the value of the output network variable is to be sent *only* in response to a poll request from a device that reads this network variable. When this keyword is omitted, the value is propagated over the network every time the variable is assigned a value and also when polled. Mutually exclusive with the **sync** modifier.

Normally only used for output network variables. Can be used with input network variables in model files.

changeable_type Specifies that the network variable type can be changed at runtime. If either the **sync** or **polled** keyword is used (these two keywords are mutually exclusive) along with the **changeable_type** keyword, then the **changeable_type** keyword must follow the other keyword. For more information on changeable type network variables, see *Changeable Type Network Variables* in *How Devices Communicate Using Network Variables* in the *Neuron C Programmer's Guide*.

The **changeable_type** keyword requires the program ID to be specified, and requires the Changeable Interface flag to be set in that program ID. A compilation error occurs otherwise.

sd_string (*concatenated-string-constant* **)**

Sets a network variable's self-documentation (SD) string of up to 1023 characters. This modifier can only appear once per network variable declaration. If any of the **sync**, **polled,** or **changeable_type** keywords are used, then the **sd_string** modifier must follow these other keywords. Concatenated string constants are permitted. Each variable's SD string can have a maximum length of 1023 bytes.

The use of any of the following Neuron C Version 2 keywords causes the compiler to take control of the generation of self-documentation strings: **fblock**, **config_prop**, **cp**, **device_properties**, **nv_properties**, **fblock_properties**, or **cp_family**.

In an application that uses compiler-generated SD data, additional SD data can still be specified with the **sd_string( )** modifier. The compiler appends this additional SD information to the compiler-generated SD data, but it is separated from the compiler-generated information with a semicolon.

Network Variable Classes (class)

Network variables constitute one of the storage classes in Neuron C. They can also be combined with one or more of the following classes:

config This variable class is equivalent to the **const** and **eeprom** classes, except that the variable is also identified as a configuration variable to network tools which access the device's interface information. The **config** keyword 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 explained in *Configuration Network Variables* on page 177.

const The network variable is of **const** type. The Neuron C compiler does not allow modifications of **const** type variables by the device's program. However, a **const network input** variable is still placed in modifiable memory and the value can change as a result of a network variable update from another device.

When used with the declaration of a configuration network variable, the **const** storage class prevents both the Neuron C application and network tools from writing to the configuration network variable. The application can cast away the const-ness of the property to implement device-specific configuration properties as configuration network variables. However, because the network variable is placed in modifiable memory, network variable

connections can still cause changes to such a configuration network variable.

eeprom The network variable is placed in EEPROM or flash memory instead of RAM. All variables are placed in RAM by default. EEPROM and flash memory are only appropriate for variables that change infrequently, due to the overhead and execution delays inherent in writing such memory, and due to the limited number of writes for such memory devices.

far The network variable is placed in the *far* section of the variable space. In Neuron C, variables are placed in *near* memory by default, but the *near* memory areas are limited in space. The maximum size of *near* memory areas is 256 bytes of RAM and 255 bytes of EEPROM, but could be less in some circumstances.

offchip This keyword places the variable in the *off-chip* portion of the variable space. By default, the linker places variables in either space as it chooses, depending on availability. If the requested memory is not available, the link fails.

onchip This keyword places the variable in the *on-chip* portion of the variable space. By default, the linker places variables in either space as it chooses, depending on availability. If the requested memory is not available, the link fails.

uninit This keyword prevents compile-time initialization of variables. This is useful for **eeprom** variables that should not or need not be written by program load or reload.

A different mechanism, subject to your network management tool, is used to determine whether configuration properties, including configuration network variables, are initialized after loading or commissioning the device. The **uninit** keyword cannot be used to prevent configuration network variables from being initialized by the network management tool. See your network tool's documentation for details.

Network Variable Types (type)

A network variable can be declared using any of the following types:

- A standard network variable type (SNVT) as described in Chapter 3, *How Devices Communicate Using Network Variables,* of the *Neuron C Programmer's Guide*. Use of a SNVT promotes interoperability. See types.lonmark.org for a list of SNVTs.
- A user network variable type (UNVT) as described in Chapter 3, *How Devices Communicate Using Network Variables,* of the *Neuron C Programmer's Guide*. UNVTs are defined using the NodeBuilder Resource Editor as described in the *NodeBuilder FX User's Guide*.
- Any of the variable types specified in Chapter 1, *Overview,* of the *Neuron C Programmer's Guide*, except for pointers. The types are those

listed below:

[**signed**] **long** [**int**]
unsigned long [**int**]
signed char
[**unsigned**] **char**
[**signed**] [**short**] [**int**]
unsigned [**short**] [**int**]
enum (An **enum** is **int** type)

Structures and unions of the above types. Structures and unions cannot exceed the network variable size limit when used as the type of a network variable. Network variables are limited to 31 bytes with Series 3100 and 5000 Neuron chips and Smart Transceivers, and support up to 228 bytes on most other platforms.

Single-dimension arrays of the above types.

For interoperability, SNVTs and UNVTs defined in resource files should be used for network variables instead of these base types.

- A **typedef**. Neuron C provides some predefined type definitions, for example:

  typedef enum {FALSE, TRUE} boolean;

  The user can also define other type definitions and use these for network variable types.

  For interoperability, SNVTs and UNVTs defined in resource files should be used for network variables instead of typedefs.

Configuration Network Variables

The syntax for network variable declarations above includes the following syntax fragment for declaring the network variable as a configuration property:

network ... [**config_prop** [*cp-modifiers*]] **...**

The **config_prop** keyword (which can also be abbreviated as **cp**) is used to specify that the network variable (or array) is a configuration property (or array of configuration properties).

If you declare a configuration network variable as **const**, the compiler issues a warning message (**NCC#599**). An application can update a constant configuration network variable as it would any network variable.

The *cp-modifiers* for configuration network variables are identical to the *cp-modifiers* described in *Configuration Property Modifiers (cp-modifiers)* on page 185.

Network Variable Property Lists (nv-property-list)

A network variable property list declares instances of configuration properties defined by CP family statements and configuration network variable declarations

that apply to a network variable. The syntax for a network variable's property list is:

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

property-reference-list :
property-reference-list **,** *property-reference*
property-reference

property-reference :
property-identifier [**=** *initializer*] [*range-mod*]
property-identifier [*range-mod*] [**=** *initializer*]

range-mod : **range_mod_string (** *concatenated-string-constant* **)**

property-identifier :
[*property-qualifier*] *cpnv-prop-ident*
[*property-qualifier*] *cp-family-prop-ident*

property-qualifier : **static** | **global**

cpnv-prop-ident : *identifier* **[** *constant-array-index-expr* **]**
identifier

cp-family-prop-ident : *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,
      // 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 can occur in either order if both are specified. If present, the instantiation initializer for a CP family member overrides any initializer provided at the time of declaration of the CP family; thus, using this mechanism, some CP family members can be initialized specially, with the remaining CP family members having a more generic initial value. If a network variable is initialized in multiple places (in other words, in its declaration as well as in its use in a property list), the initializations must match.

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.

Finally, each property instantiation can have a range-modification string following the property identifier. The range-modification string works identically to the *range-mod* described in *Configuration Property Modifiers (cp-modifiers)* on page 185. A range-modification string provided in the instantiation of a CP family member overrides any range-modification string provided in the declaration of a CP family.

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 the discussion of network variable properties in the *Neuron C Programmer's Guide* for more information.

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.

Configuration Network Variable Arrays

A configuration network variable array that is a configuration property can be used in one of two ways. Each element of the array can be treated as a separate configuration property, or all elements of the array can be treated as a single configuration property taken together.

To use each network variable array element as a separate, scalar configuration property, specify the starting index of the first array element in the properties list, as in Example 1 below. The example shows elements **[2]** through **[5]** of the **cpMaxSendT** array used as properties for **nvoValue[0]** through **nvoValue[3]**, respectively, with the remaining elements of **cpMaxSendT** being unused.

Example 1:

```
network input cp SCPTmaxSendT cpMaxSendT[10];
network output SNVT_lev_percent nvoValue[4]
   nv_properties {
      cpMaxSendT[2]
};
```

To use the entire network variable array as a single property, do not specify any index in the properties list, as in Example 2 below. The entire array **cpMaxSendT** becomes a single property of **nvoValue**.

Example 2:

```
network input cp SCPTmaxSendT cpMaxSendT[10];
network output SNVT_lev_percent nvoValue
   nv_properties {
      cpMaxSendT
};
```

Similarly, a single network variable array element, or the entire network variable array can be used as a device property (see *Device Property Lists* on page 188).

A configuration network variable array must be shared with the static or global keyword if it applies to a network variable array.

Example 3:

```
network input cp SCPTmaxSendT
cpMaxSendT[10];
network output SNVT_lev_percent nvoValue[4]
nv_properties {
   static cpMaxSendT  // MUST be shared
};
```

Network Variable Connection Information (connection-info)

The following optional fields can be included in the declaration of each network variable. The fields can be specified in any order. This information can be used by a network tool, as described in the *NodeBuilder FX User's Guide*. These connection information assignments can be overridden by a network tool after a device is installed, unless otherwise specified using the **nonconfig** option.

bind_info (
[**expand_array_info**]
[**offline**]
[**unackd** | **unackd_rpt** | **ackd** [**(config** | **nonconfig)**]]
[**authenticated** | **nonauthenticated** [**(config** | **nonconfig)**]]
[**priority** | **nonpriority** [**(config** | **nonconfig)**]]
[**rate_est (***const-expr***)**]
[**max_rate_est (***const-expr***)**]
)

expand_array_info Applies to a network variable array. This option is used to tell the compiler that, when publishing the device interface in the SI and SD data and in the device interface file, each element of a network variable array should be treated as a separate network variable for naming purposes. The names of the array elements have unique identifying characters postfixed. These identifying characters are typically the index of the array element. Thus, a network variable array **xyz[4]** would become the four separate network variables **xyz0**, **xyz1**, **xyz2**, and **xyz3**.

offline Specifies that a network tool must take this device offline, or ensure that the device is already offline, before updating the network variable. This option is commonly used with a **config** class network variable (this is an obsolete usage, but is supported for legacy applications).

Do not use this feature in the **bind_info** for a configuration network variable that is declared using the **config_prop** or **cp** keyword. Use the **offline** option in the **cp_info**, instead.

unackd | unackd_rpt | ackd [**(config | nonconfig)**]

Selects the LonTalk protocol service to use for updating this network variable. The allowed protocol service options are:

unackd — unacknowledged service; the update is sent once and no acknowledgment is expected.

unackd_rpt — repeated service; the update is sent multiple times and no acknowledgments are expected.

ackd (the default) — acknowledged service; with retry; if acknowledgments are not received from all receiving devices before the layer 4 retransmission timer expires, the message is sent again, up to the retry count.

An unacknowledged (**unackd**) network variable uses minimal network resources to propagate its values to other devices. As a result, propagation failures are more likely to occur, and failures are not detected by the sending device. This class might be used for variables that are updated on a frequent, periodic basis, where loss of an update is not critical, or in cases where the probability of a collision or transmission error is extremely low.

The repeated (**unackd_rpt**) service is typically used when a message is propagated to many devices, and a reliable delivery is required. This option reduces the network traffic caused by a large number of devices sending acknowledgements simultaneously and can provide the same reliability as the acknowledged service by using a repeat count equal to the retry count.

The **config** keyword, the default, indicates that this service type can be changed by a network tool. This option allows a network tool to change the service specification at installation time.

The **nonconfig** keyword indicates that this service cannot be changed by a network tool.

authenticated | nonauthenticated [**(config | nonconfig)**]

Specifies whether a network variable update requires authentication. With authentication, the identity of the sending device is verified by all receiving devices. Abbreviations for **authentication** are **auth** and **nonauth**. The **config** and **nonconfig** keywords specify whether the authentication designation can be changed by a network tool.

A network variable connection is authenticated only if the readers *and* writers have the **authenticated** keywords specified. However, if only the originator of a network variable update or poll has used the keyword, the connection is not authenticated (although the update does take place). See also the *Authentication* section in Chapter 3, *How Devices Communicate Using Network Variables,* of the *Neuron C Programmer's Guide*.

The default is **nonauth** (**config**).

Note: Use only the acknowledged service with authenticated updates. Do **not** use the unacknowledged or repeated services.

priority | **nonpriority** [**(config** | **nonconfig)**]

Specifies whether a network variable update has priority access to the communications channel. This field specifies the default value. The **config** and **nonconfig** keywords specify whether the priority designation can be changed by a network tool. The default is **config**. All priority network variables in a device use the same priority time slot because each device is configured to have no more than one priority time slot.

The default is **nonpriority** (**config**).

The **priority** keyword affects output or polled input network variables. When a priority network variable is updated, its value is propagated over the network within a bounded amount of time as long as the device is configured to have a priority slot by a network tool. The exact bound is a function of the bit rate and priority. This is in contrast to a **nonpriority** network variable update, whose delay before propagation is unbounded.

rate_est (*const-expr*) The estimated sustained update rate, in tenths of messages per second, that the associated network variable is expected to transmit. The allowable value range is from 0 to 18780 (0 to 1878.0 network variable updates per second).

max_rate_est (*const-expr*)

The estimated maximum update rate, in tenths of messages per second, that the associated network variable is expected to transmit. The allowable value range is from 0 to 18780 (0 to 1878.0 network variable updates per second).

Note: It might not always be possible to determine **rate_est** and **max_rate_est**. For example, update rates are often a function of the particular network where the device is installed. These values can be used by a network tool to perform network load analysis and are optional.

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 value *a*, produced by the formula, is in units of messages per second.

Configuration Property Declarations

You can implement a configuration property as a configuration network variable or as part of a configuration file. To implement a configuration property as a configuration network variable, declare it using the **network** … **config_prop** syntax described in *Network Variable Declarations Syntax* on page 173. To implement a configuration property as a part of a configuration file, declare it with the **cp_family** syntax described in this section.

The syntax for declaring a configuration property family implemented as part of a configuration file is the following:

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

The brackets around *array-bound* are shown in **bold** type. The brackets do not, in this case, indicate an optional field. They are a required part of the syntax of declaring an array, and must be entered into the program code.

Example 1 – Declaring a CP family for a singular CP:

```
SCPTlocation cp_family cpLocation = "";
```

Example 2 – Declaring a CP family for a CP-array:

```
SCPTbrightness cp_family cpBrightness[3];
```

Example 3 – Declaring a CP family for CP-array with explicit initial values:

```
SCPTbrightness cp_family cpBrightness[3] = {
   {    0, ST_OFF },
   { 100u, ST_ON },
   { 200u, ST_ON }
};
```

Any number of CP families can be declared in a Neuron C program. Declarations of CP families do not result in any data memory being used until a family member is created through the instantiation process. In this regard, the CP family is similar to an ANSI C **typedef**, but it is more than just a type definition.

A configuration property type is also similar to an ANSI C **typedef**, but it is also much more. The configuration property type also defines a standardized semantic meaning 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.

CP families that are declared using the **const** keyword have their family members placed in the read-only value file. All other CP families have their family members placed in the *writeable* value file (this file is also called the *modifiable* value file).

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 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 could be many similar resource files containing UCPT definitions, and these are managed on the computer by the NodeBuilder Resource Editor as described in the *NodeBuilder FX User's Guide*.

A configuration property family can be declared with an optional *array-bound*. This declares the family such that *each* member of the configuration property family is a separate array (of identical size). Each instantiation of a member of the configuration property family becomes a separate array. All elements of the array are part of the *single* configuration property that instantiates a member of such a family.

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. For more information about CP families and instantiated members, see the discussion in Chapter 4, *Using Configuration Properties to Configure Device Behavior,* of the *Neuron C Programmer's Guide*.

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 these as a single declaration.

Example 1 – Repeated family declaration:

```
SCPTbrightness cp_family cpBrightness;
SCPTbrightness cp_family cpBrightness;
```

In Example 1, the compiler treats the two families as one. One of the two declarations can be omitted. Note the CP family declaration is similar to a C language typedef in that no memory is allocated; the repeated declaration simply has no effect.

Example 2 – Repeated family declaration:

```
SCPTbrightness cp_family cpBrightness;
SCPTbrightness cp_family cpDarkness;
```

In Example 2, the compiler treats the two families as two distinct families, because of the different family names.

Example 3 – Invalid re-use of family name:

```
SCPTbrightness cp_family cpBrightness = {100, ST_ON};
SCPTbrightness cp_family cpBrightness = {0, ST_OFF};
```

The declaration in Example 3 causes a compile-time error, because of the fact that the two families have different properties (the default value) yet are declared using the same family name.

Configuration Property Modifiers (cp-modifiers)

The configuration property modifiers are an optional part of the CP family declaration discussed above, as well as the configuration network variable declaration discussed later.

The syntax for the configuration property modifiers is shown below:

cp-modifiers : [**cp_info (** *cp-option-list* **)**] [*range-mod*]

cp-option-list : *cp-option-list* **,** *cp-option*
cp-option

cp-option : **device_specific | manufacturing_only | object_disabled | offline | reset_required**

range-mod : **range_mod_string (** *concatenated-string-constant* **)**

There must be at least one keyword in the option list. For multiple keywords, the keywords can occur in any order, but the same keyword must not appear more than once. Keywords must be separated by commas.

You can specify the following configuration property options:

device_specific Specifies a configuration property that should always be read from the device instead of relying upon the value in the device interface file or a value stored in a network database. This mechanism is used for configuration properties that must be managed by the device or by a passive configuration tool that does not have access to the network database. An example of such a configuration property is a setpoint that is updated by a local operator interface on the device.

Recommendation: Declare a **device_specific** CP family or configuration network variable as **const**. If the CP family or configuration property network variable is defined within a functional profile and the declaration does not match the profile, the compiler issues a warning message.

manufacturing_only Specifies a factory setting that can be read or written when the device is manufactured, but is not normally (or ever) modified in the field. In this way a standard network tool can be used to calibrate the device when a device is manufactured, while a field installation tool would observe the flag in the field and prevent updates or require a password to modify the value.

object_disabled Specifies that a network tool must disable the functional block containing the configuration property, take the device offline, or ensure that the functional block is already disabled or the device is already offline, before modifying the configuration property.

offline Specifies that a network tool must take this device offline, or ensure the device is already offline, before modifying the configuration property.

A configuration property can be declared as both **offline** and **object_disabled**. In this case, the **offline** declaration takes precedence.

reset_required Specifies that a network tool must reset the device after changing the value of the configuration property.

The optional *range-mod* modifier allows you to specify a range-modification string that modifies the valid range for the configuration property defined by the resource file. The range-modification string can only be used with fixed-point

and floating-point types, and consists of a pair of either fixed-point or floating-point numbers delimited by a colon. The first number is the lower limit and the second number is the high limit. If either the high limit or the low limit should be the maximum or minimum specified in the configuration property type definition, then the field should be empty. In the case of a structure or an array, if one member of the structure or array has a range modification, then all members must have a range modification specified. In this case, each range modification pair is delimited by the ASCII '|'. To specify no range modification for a member of a structure (that is, revert to the default for that member), encode the field as '|'. Use the same encoding for structure members that cannot have their ranges modified due to their data type. The '|' encoding is only allowed for members of structures.

Whenever a member of a structure is not a fixed or floating-point number, its range cannot be restricted. Instead, the default ranges must be used. In the case of an array, the specified range modifications apply to all elements of the array. For example, to specify a range modification for a 3-member structure where the second member has the default ranges, and the third member only has an upper limit modification, the range modification string is encoded as: **"n:m||:m;"**. Positive values for range modifications and their exponents (if any) are implicit, while negative numbers and negative exponents must be explicitly designated as such with a preceding '**-**' character. Floating-point numbers use a '**.**' character for the decimal point. Fixed-point numbers must be expressed as a signed 32-bit integer. Floating-point numbers must be within the range of an IEEE 32-bit floating-point number. To express an exponent, precede the exponent by an '**e**'or an '**E**' and then follow with an integer value.

Configuration Property Instantiation

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 computer storage is assigned for the variables. The variables can then be used in later expressions in the executable code of the program.

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 a device are explained in the next section, property lists for network variables are explained later in this chapter, and property lists for functional blocks is described in Chapter 6, *Functional Block Declarations*, on page 193.

The instantiation of CP family members occurs when the CP family declaration's identifier is used in a property list. 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 inform the compiler of 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 statements and configuration network variable declarations that apply to a device. The complete syntax for a device property list is:

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

property-reference-list :
property-reference-list **,** *property-reference*
property-reference

property-reference :
property-identifier [**=** *initializer*] [*range-mod*]
property-identifier [*range-mod*] [**=** *initializer*]

range-mod : **range_mod_string (** *concatenated-string-constant* **)**

property-identifier :
cpnv-prop-ident
cp-family-prop-ident

cpnv-prop-ident : *identifier* **[** *constant-array-index-expr* **]**
identifier

cp-family-prop-ident : *identifier*

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, and a single element of that array is to be used as a property for the device, specify that element with an index expression (such as **var[4]**) in the **device_properties** clause. On the other hand, if the property is itself the entire network variable array, specify just the array name without an index expression (such as **var**, where **var** is declared as an array) in the **device_properties** clause.

Following the *property-identifier*, there can be an optional *initializer*, and an optional *range-mod*. These optional elements can occur in either order if both are given. If present, the instantiation initializer for a CP family member overrides any initializer provided at the time of declaration of the CP family; thus, using this mechanism, some CP family members can be initialized specially, with the remaining CP family members having a more generic initial value. If a network variable is initialized in multiple places (in other words, in its declaration as well as in its use in a property list), the initializations must be identical in type and value.

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.

A Neuron C program can have multiple device property lists. These lists are merged together by the compiler to create one combined device property list. This feature is provided for modularity in the program (different modules can specify certain properties for the device, but the list is combined by the compiler). 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 of the same type in the device property lists, this situation causes a compilation error.

Finally, each property instantiation can have a range-modification string following the property identifier. The range-modification string works identically to the *range-mod* described *Configuration Property Modifiers (cp-modifiers)* on page 185. A range-modification string provided in the instantiation of a CP family member overrides any range-modification string provided in the declaration of the CP family.

Example:

```
UCPTsomeDeviceCp cp_family cpSomeDeviceCp;
SCPTupdateRate cp_family cpUpdateRate = {3};
SCPTlocation cp_family cpLocation;

device_properties {
   cpSomeDeviceCp,
   cpUpdateRate
   range_mod_string(":180"),
   cpLocation = { "Unknown" }
};
```

This example implements three device properties: **cpSomeDeviceCp** implements a UCPT with a default value as defined in the user-defined resource file. **cpUpdateRate** implements **SCPTupdateRate** with a maximum value of 180 seconds (the SCPT supports up to 65,535 seconds). Note that the entire **cpUpdateRate** configuration property family, not just the **cpUpdateRate** device property, uses an implementation-specific default value of 3 seconds (the SCPT is defined with a default of 0 seconds). Finally, **cpLocation** shows the declaration of a **SCPTlocation**-typed device property with a device-specific default value ("Unknown").

Accessing Property Values from a Program

Configuration properties can be accessed from a program just as any other variable can be accessed. 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, the compiler must know which family member is being accessed, because there could be more than one member of the same CP family with the same name, but applying to different network variables. The syntax for accessing a configuration property from a network variable's property list is:

nv-context **::** *property-identifier* **[** *index-expr* **]**
nv-context **::** *property-identifier*

nv-context : *identifier* **[** *index-expr* **]**
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,
      // Override default for minSendT to 30 seconds
      // for this family member, only:
      cpMinSendT = { 0, 0, 0, 30, 0 }
};

void f(void)
{
   ...
   if (nvoValue::cpMaxSendT.seconds > 0) {
      ...
   }
}
```

The particular 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. 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, **nva**, with 10 elements, could be declared with a property list referencing a CP family named **xyz**. There would then be 10 different members of the **xyz** CP family, all with the same name. However, adding the context, such as **nva[4]::xyz**, or **nva[j]::xyz**, uniquely identifies the family member.

Because 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 two consecutive colon characters without a preceding context identifier.

If accessing a CP family or network variable CP where each member is an array, you can add an array index expression to the end of the context/property reference expression, just as you would add an array index expression to any other array in C.

Using the example above with **xyz** being a name of a configuration property array, the expression **nva[4]::xyz** evaluates to the entire configuration property array (the expression returns the address of the array's first element), whereas **nva[4]::xyz[2]** returns the third element of the configuration property array that applies to the fifth element of the **nva** network variable array.

Finally, even though a configuration network variable can be uniquely accessed through its variable identifier, it can also be accessed through the context expression, just like the CP family members.

When accessing a member of a configuration property family that implements a device property, the context expression is an empty string. For example, **::cpXyz** refers to a device property **cpXyz**.

For more information about accessing configuration properties, including examples, see *Configuration Properties* in the *Neuron C Programmer's Guide*.

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.

6

Functional Block Declarations

This chapter provides reference information for functional block declarations. The Neuron C language allows creation of functional blocks to group network variables and configuration properties that perform a single task together.

Functional blocks are an important part of a device's interface definition. Functional block declarations apply to both Neuron-hosted applications and host-based applications with a model file.

Introduction

The external application interface of 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 configuration properties. Each functional block implements an instance of a functional profile. A functional block must implement all the mandatory network variables and configuration properties defined by the functional profile, and can implement any of the optional network variables and 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.

Functional profiles are defined in *resource files*. You can use standard functional profiles (SFPT) defined in the standard resource file set, and you can define your own functional profiles (UFPT) in your own resource file sets. Functional blocks based on standard functional profiles are also called *LonMark objects*. A functional profile defined in a resource file is also called a *functional profile template* (FPT). See types.lonmark.org for a list of standard functional profiles.

You can declare functional blocks in your Neuron C applications using **fblock** declarations. These declarations are described in this chapter.

A functional block declaration does not cause the compiler to generate any executable code, although the compiler does create some data structures as described in *Related Data Structures* on page 199. These data structures are used to implement various functional block features.

Principally, the functional block declaration creates associations among network variables and configuration properties. The compiler then uses these associations to create the self-documentation (SD) and self-identification (SI) data in the device and in its associated device interface file (**.xif** extension).

The functional block information in the device interface file or the SD and SI data communicates the presence and names of the functional blocks contained in the device to a network tool. The information also communicates which network variables and configuration properties in the device are members of each functional block.

Functional Block Declarations Syntax

The complete syntax for declaring a functional block is:

fblock *FPT-identifier* **{** *fblock-body* **}** *identifier* [*array-bounds*]
[*ext-name*] [*fb-property-list*] **;**

array-bounds : **[** *const-expr* **]**

ext-name :	**external_name (** *C-string-const* **)** **external_resource_name (** *C-string-const* **)** **external_resource_name (** *const-expr* **:** *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* **;**

Example:

```
// Prototype for director function
extern void MyDirector (unsigned uFbIdx, int nCmd);

// Network variables referenced by this fblock:
network output SNVT_lev_percent nvoValue;
network input  SNVT_count nviCount;

// The functional block itself ...
fblock SFPTanalogInput {
   nvoValue implements nvoAnalog;
   nviCount implementation_specific(128) nviCount;
   director myDirector;
} MyAnalogInput external_name("AnalogInput");
```

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. The functional profile defines the network variable and configuration property members, a unique key called the *functional profile number* (also called the *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 need not be implemented.

The functional block declaration then proceeds with a member list. In this member list, network variables are associated with the abstract network variable members of the profile. These network variables must have previously been declared in the program. The association between the members of the functional block declaration and the profile's abstract network variable members is performed with the **implements** keyword. At a minimum, every *mandatory* profile network variable member 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.

If allowed by the profile, you can have an empty member list. Such a functional block is useful as a collection of related configuration properties.

A Neuron C program can also implement *additional* network variables in the functional block that are not in the list of optional members of the profile. Such additional network variable members beyond the profile are called *implementation-specific* members. These extra members are declared 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 abstract network variable member of the profile, and the implementation-specific member cannot use any of the index numbers or member names that the profile already uses.

Note that implementation-specific member network variables or configuration properties can prevent device certification. Instead of adding implementation-specific member network variables or configuration properties, consider removing those items from the interoperable interface, or defining a user-defined functional profile, possibly inheriting from a standard functional profile, and adding the desired new members to that UFPT.

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. See the chapter on functional blocks in the *Neuron C Programmer's Guide* for more explanation and examples of functional block members and the director function.

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 **fblock** is implemented as an array, as shown in the example below, then each network variable that is to be referenced by that **fblock** must be declared as an array of at least the same size. When implementing an **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, which
// have members nvoValue[2]..nvoValue[5], respectively
fblock SFPTanalogInput {
   nvoValue[2] implements nvoAnalog;
} myFB[4];
```

An optional external name can be provided for each functional block. An external name can be specified with an **external_name** keyword, followed by a string in parentheses. The string becomes part of the device interface that is exposed to network tools. The external name is limited to 16 characters. If the **external_name** feature is not used, nor the **external_resource_name** feature described below, the functional block identifier (supplied in the declaration) is also used as the default external name. In this case, there is a limitation of 16 characters applying to the functional block identifier.

An external name can optionally be specified using a reference to a resource file. The reference is specified using the **external_resource_name** keyword, instead

of the **external_name** string described above. In this case, the device interface information contains a scope and index pair (the first number is a scope, then a colon character, and then the second number is an index). The scope and index pair identifies a language string in the resource files, 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.

Alternatively, a string argument can be supplied to the **external_resource_name** keyword. The compiler takes this string and uses it to look up the appropriate string in the resource files that apply to the device. This mechanism is provided as a convenience to the programmer, so the compiler can look up the scope and index; but the result is the same, the scope and index pair is used in the external interface information, rather than a string. The string *must* exist in an accessible resource file for the compiler to properly perform the lookup.

Functional Block Property Lists (fb-property-list)

You can include a property list at the end of the functional block declaration, 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. Implementation-specific properties can be added 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*.

The complete syntax for a functional block's property list is:

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

property-reference-list :
property-reference-list **,** *property-reference*
property-reference

property-reference :
property-identifier [**=** *initializer*] [*range-mod*]
property-identifier [*range-mod*] [**=** *initializer*]

range-mod : **range_mod_string (** *C-string-constant* **)**

property-identifier :
[*property-qualifier*] *cpnv-prop-ident*
[*property-qualifier*] *cp-family-prop-ident*

property-qualifier : **static** | **global**

cpnv-prop-ident:
identifier **[***constant-array-index-expr***]**
identifier

cp-family-prop-indent: *identifier*

The functional block property list begins with the **fb_properties** keyword. It then contains a list of property references, separated by commas, exactly like the device property list and the network variable 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 network variable property list syntax discussed in the previous chapter.

Following the *property-identifier*, there can be an optional *initializer*, and an optional *range-mod*. These optional elements can occur in either order if both are specified. If present, the instantiation initializer for a CP family member overrides any initializer provided at the time of declaration of the family; thus, using this mechanism, some CP family members can be initialized specially, with the remaining family members having a more generic initial value. If a network variable is initialized in multiple places (in other words, in its declaration as well as in its use in a property list), the initializations must match.

Each property instantiation can have a range-modification string following the property identifier. The range-modification string works identically to the *range-mod* described in *Configuration Property Modifiers (cp-modifiers)* on page 185. A range-modification string provided in the instantiation of a CP family member overrides any range modification string provided in the declaration of the CP family.

The elements of an **fblock** array all share the same set of configuration properties as listed in the associated *fb-property-list*. Without special keywords, each element of the **fblock** array obtains its own set of configuration properties. Special modifiers can be used to *share* individual properties among members of the same **fblock** array (through use of the **static** keyword), or among all the functional blocks on the device that have the particular property (through use of the **global** keyword).

Example:

```
// CP Family Declarations:
SCPTgain cp_family cpGain;
SCPTlocation cp_family cpLocation;
SCPToffset cp_family cpOffset;
SCPTmaxSndT cp_family cpMaxSendT;
SCPTminSndT cp_family cpMinSendT;

// NV Declarations:
network output SNVT_lev_percent nvoData[4]
   nv_properties {
      cpMaxSendT,  // throttle interval
      cpMinSendT   // heartbeat interval
};

// four open loop sensors, implemented as two arrays of
// two sensors, each.  This might be beneficial in that
// this software layout might meet the hardware design
// best, for example with regards to shared and individual
// properties.

fblock SFPTopenLoopSensor {
   nvoData[0] implements nvoValue;
} MyFb1[2]
```

```
   fb_properties {
      cpOffset,         // offset for each fblock
      static cpGain,    // gain shared in MyFb1
      global cpLocation // location shared in all 4
};

fblock SFPTopenLoopSensor {
   nvoData[2] implements nvoValue;
} MyFb2[2]
   fb_properties {
      cpOffset,         // offset for each fblock
      static cpGain,    // gain shared in MyFb2
      global cpLocation // location shared in all 4
};
```

Like network variable properties, functional block properties can be shared between two or more functional blocks. The use of the **global** keyword creates a CP family member that is shared among two or more functional blocks. This global member is a *different* member than a global member that would be shared among network variables, because no single configuration property can apply to both network variables and functional blocks.

The use of the **static** keyword creates a CP family member that is shared among all the members of a functional block array, but not with any other functional blocks outside the array. See the discussion of functional block properties in the *Neuron C Programmer's Guide* for more information on this topic.

Consequently, the example shown above instantiates four heartbeat (**SCPTminSndT**) and four throttle (**SCPTmaxSndT**) CP family members (one pair for each member of the **nvoData** network variable array), and four offset CP family members (**SCPToffset**), one for each member of each **fblock** array. It also instantiates a *total of two* gain control CP family members (**SCPTgain**), one for **MyFb1**, and one for **MyFb2**. Finally, it instantiates a *single* location CP family member (**SCPTlocation**), which is shared by **MyFb1** *and* **MyFb2**.

Just as for properties of network variables, you can treat a network variable array that is a configuration property either as a collection of separate properties where each element is a separate property, or as a single configuration property that is an array. In the former case, specify the network variable name with an array index representing the starting index for the element of the network variable array that is to be the first property used. In the latter case, specify the network variable name without an index to treat the entire network variable array as a single property.

Related Data Structures

Each functional block is assigned a global index (from 0 to *n-1*) by the compiler. In the case of an array of functional blocks, each element is assigned a consecutive index (but because these indices are global, they do not necessarily start at zero). An application can get the global index for a functional block using the **global_index** property as described in *Accessing Members and Properties of a Functional Block from a Program* on page 200.

If one or more functional blocks are declared in a Neuron C program, the compiler creates an array of values that can be accessed from the program. This array is named **fblock_index_map**, and it has one element per *network variable*

in the program. The array entry is an **unsigned short**. Its declaration, in the **<echelon.h>** file, is:

extern const unsigned short fblock_index_map[ ];

The value for each network variable is set to the global index of the functional block of which it is a member. If the network variable is not a member of any functional block, the value for its entry in the **fblock_index_map** array is set to the value 0xFF.

Accessing Members and Properties of a Functional Block from a Program

The network variable members and configuration property (implemented as network variable) members of a functional block can be accessed from a program just as any other variable can be accessed. For example, they 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 is:

fb-context **::** *property-identifier* **[** *index-expr* **]**
fb-context **::** *property-identifier*

fb-context : *identifier* **[** *index-expr* **]**
identifier

The particular 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. Because there cannot be two or more properties with the same SCPT or UCPT type that apply to the same functional block, this restriction means that each property is unique within a particular context. 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 **xyz**. There would then be 10 different members of the CP family **xyz**, all with the same name. However, adding the context, such as **fba[4]::xyz**, or **fba[j]::xyz**, uniquely identifies the CP family member.

Example:

```
// Continuing from the example earlier in the chapter
// that declared MyFb1[2] and MyFb2[2] ...

void f(void)
{
   MyFb1[0]::nvoData = muldiv(rawData,
         MyFb1[0]::cpGain.multiplier,
         MyFb1[0]::cpGain.divider);
}
```

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 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*

Example:

```
if (MyFb1[0]::cpGain.divider == 0) {
   // flag error indicating division by zero
}
```

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:

```
MyTimer = MyFb1[0]::nvoValue::cpMaxSendT;
```

Neuron C provides the following built-in properties for a functional block (the *fb-context* syntactical element is defined above):

fb-context **:: global_index**

The **global_index** property is an **unsigned short** value that provides the global index assigned by the compiler. The global index is a read-only value.

fb-context **:: director (** *expr* **)**

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.

For more information about functional blocks and accessing their members and properties, including examples, see Chapter 5, *Using Functional Blocks to Implement a Device Interface*, in the *Neuron C Programmer's Guide*.

7

Built-In Variables, Objects, Symbols, and Semaphore

This chapter provides reference information about the built-in variables, objects, symbols, and semaphore in Neuron C.

Built-in variables, objects, and semaphores apply only to Neuron-hosted application development, and are ignored in model files. Built-in symbols apply to both Neuron-hosted and host-based development.

Introduction

Neuron C Version 2 provides built-in variables and built-in objects. The term "built-in" means that the definition is part of the Neuron C language, and is directly generated by the compiler, rather than being a reference to a normal variable.

The built-in variables are:

activate_service_led
config_data
cp_modifiable_value_file
cp_modifiable_value_file_len
cp_readonly_value_file
cp_readonly_value_file_len
cp_template_file
cp_template_file_len
fblock_index_map
input_is_new
input_value
msg_tag_index
nv_array_index
nv_in_addr
nv_in_index
read_only_data
read_only_data_2

The built-in objects are:

msg_in
msg_out
resp_in
resp_out

The built-in, predefined pre-processor symbols are:

_ECHELON
FAMILY(3100|5000|6000)
_FTXL
_ILON
_LID3
_MINIKIT
_MODEL_FILE
_NCC5
_NCC6
_NEURONC
_NODEBUILDER
_PUREC
_SHORTSTACK
_SUPPORT_* (various)

For Series 5000 and 6000 chips, you can control access to shared data through the use of the built-in semaphore. The following keyword defines a locked section controlled by the semaphore:

__lock { }

The following sections describe these built-in elements.

Built-In Variables

The following sections list the Neuron C built-in variables alphabetically, providing relevant syntax information and a detailed description of each function.

activate_service_led *Variable*

The **activate_service_led** variable can be assigned a value by the application program to control the service LED status. Assign a non-zero value to **activate_service_led** to turn the service LED on. Assign a zero value to turn the service LED off. The **<control.h>** include file contains the definition for the variable as follows:

extern system int activate_service_led;

This variable is located in RAM space belonging to the Neuron firmware. Its value is not preserved after a reset.

There can be a delay of up to one second between the time that the application program sets this variable and the time that its new value is sensed and acted upon by the Neuron firmware. Therefore, attempts to flash the service LED are limited to a minimum period of one second.

Example:

```
// Turn on service LED
activate_service_led = TRUE;

// Turn off service LED
activate_service_led = FALSE;
```

config_data *Variable*

The **config_data** variable defines the hardware and transceiver properties of this device. It is located in EEPROM, and parts of it belong to the application image written during device manufacture, and to the network image written during device installation. The type is a structure declared in **<access.h>** as follows:

```
#define LOCATION_LEN  6
#define NUM_COMM_PARAMS  7
typedef struct {
      // This embedded struct starts at
      // offset 0x11 when placed in outer struct
   unsigned collision_detect        : 1;
   unsigned bit_sync_threshold      : 2;
   unsigned filter                  : 2;
   unsigned hysteresis              : 3;
      // offset 0x12 starts here when it is nested
      // in the outer struct below
   unsigned cd_to_end_packet        : 6;
   unsigned cd_tail                 : 1;
   unsigned cd_preamble             : 1;
} direct_param_struct;
typedef struct {  // This is the outer struct
```

```
   unsigned long    channel_id;     // offset 0x00
   char location[LOCATION_LEN];     // offset 0x02
   unsigned comm_clock         : 5; // offset 0x08
   unsigned input_clock        : 3;
   unsigned comm_type          : 3; // offset 0x09
   unsigned comm_pin_dir       : 5;
   unsigned preamble_length;        // offset 0x0A
   unsigned packet_cycle;           // offset 0x0B
   unsigned beta2_control;          // offset 0x0C
   unsigned xmit_interpacket;       // offset 0x0D
   unsigned recv_interpacket;       // offset 0x0E
   unsigned node_priority;          // offset 0x0F
   unsigned channel_priorities;     // offset 0x10
   union {                          // offset 0x11
      unsigned xcvr_params[NUM_COMM_PARAMS];
      direct_param_struct dir_params;
   } params;
   unsigned non_group_timer    : 4; // offset 0x18
   unsigned nm_auth            : 1;
   unsigned preemption_timeout : 3;
} config_data_struct;
const config_data_struct config_data;
```

The application program can read this structure, but cannot write it, using the **config_data** global declaration. The structure is 25 bytes long, and it can be read and written over the network using the *read memory* and *write memory* network management messages with **address_mode=2**. For detailed descriptions of the individual fields, see the ISO/IEC 14908 (ANSI/EIA/CEA-709.1) *Control Network Specification*. To write this structure, use the **update_config_data( )** function described on page 164.

cp_modifiable_value_file — Variable

The **cp_modifiable_value_file** variable contains the writeable configuration property value file. This block of memory contains the values for all writeable configuration properties implemented as CP family members. It is defined as an **unsigned short** array. See Chapter 5, *Network Variable, Configuration Property, and Message Tag Declarations*, on page 171, for more information about configuration properties.

cp_modifiable_value_file_len — Variable

The **cp_modifiable_value_file_len** variable contains the length of the **cp_modifiable_value_file** array. It is defined as an **unsigned long**. See Chapter 5, *Network Variable, Configuration Property, and Message Tag Declarations*, on page 171, for more information about configuration properties.

cp_readonly_value_file — Variable

The **cp_readonly_value_file** variable contains the read-only configuration property value file. This block of memory contains the values for all read-only configuration properties implemented as CP family members. The type is an **unsigned short** array. See Chapter 5, *Network Variable, Configuration*

Property, and Message Tag Declarations, on page 171, for more information about configuration properties.

cp_readonly_value_file_len Variable

The **cp_readonly_value_file_len** variable contains the length of the **cp_readonly_value_file** array. The type is **unsigned long**. See Chapter 5, *Network Variable, Configuration Property, and Message Tag Declarations*, on page 171, for more information about configuration properties.

cp_template_file Variable

The **cp_template_file** variable contains the configuration property template file. The configuration template file contains a definition of all configuration properties implemented as CP family members. This is an **unsigned short** array. See Chapter 5, *Network Variable, Configuration Property, and Message Tag Declarations*, on page 171, for more information about configuration properties.

cp_template_file_len Variable

The **cp_template_file_len** variable contains the length of the **cp_template_file** array. The type is an **unsigned long**. See Chapter 5, *Network Variable, Configuration Property, and Message Tag Declarations*, on page 171, for more information about configuration properties.

fblock_index_map Variable

The **fblock_index_map** variable contains the functional block index map. The functional block index map provides a mapping of each network variable (or, each network variable array element in case of an array) to the functional block that contains it, if any. The type is an **unsigned short** array. The length of the array is identical to the number of network variables (counting each network variable array element separately) in the Neuron C program.

For each network variable, the mapping array entry corresponding to that variable's global index (or that element's global index) is either set to **0xFF** by the compiler if the variable (or element) is not a member of a functional block, or it is set to the functional block global index that contains the network variable (or element). The functional block global indices range from 0 to *n*-1 consecutively, for a program containing *n* functional blocks. See Chapter 6, *Functional Block Declarations*, on page 193, for more information about functional blocks.

input_is_new Variable

The **input_is_new** variable is set to TRUE for all timer/counter input objects whenever a call to the **io_in( )** function returns an updated value. The type of the **input_is_new** variable is **boolean**.

input_value Variable

The **input_value** variable contains the input value for an **io_changes** or **io_update_occurs** event. When the **io_changes** or **io_update_occurs** event is evaluated, an implicit call to the **io_in( )** function occurs. This call to **io_in( )** obtains an input value for the object, which can be accessed using the **input_value** variable. The type of **input_value** is a **signed long**.

Example 1:

```
signed long switch_state;

when (io_changes(switch_in))
{
   switch_state = input_value;
}
```

Here, the value of the network variable **switch_state** is set to the value of **input_value** (the switch value that was read in the **io_changes** clause).

However, there are some I/O models, such as **pulsecount**, where the true type of the input value is an **unsigned long**. An explicit cast should be used to convert the value returned by **input_value** to an **unsigned long** variable in this case.

Example 2:

```
unsigned long last_count;
IO_7 input pulsecount count;

when (io_update_occurs(count))
{
   last_count = (unsigned long)input_value;
}
```

msg_tag_index Variable

The **msg_tag_index** variable contains the message tag for the last **msg_completes**, **msg_succeeds**, **msg_fails**, or **resp_arrives** event. When one of these events evaluates to TRUE, **msg_tag_index** contains the message tag index to which the event applies. The contents of **msg_tag_index** is undefined if no input message event has been received. The type is **unsigned short**.

nv_array_index Variable

The **nv_array_index** variable contains the array index for a **nv_update_occurs, nv_update_completes, nv_update_fails, nv_update_succeeds** event. When one of these events, qualified by an unindexed network variable array name, evaluates to TRUE, **nv_array_index** contains the index of the element within the array to which the event applies. The contents of **nv_array_index** will be undefined if no network variable array event has occurred. The type is **unsigned int**.

nv_in_addr Variable

The **nv_in_addr** variable contains the source address for a network variable update. This value can be used to process inputs from a large number of devices

that fan-in to a single input on the monitoring device. When the devices being monitored have the same type of output, a single input network variable can be used on the monitoring device. The connection would likely include many output devices (the sensors) and a single input device (the monitor). However, the monitoring device in this example must be able to distinguish between the many sensor devices. The **nv_in_addr** variable can be used to accomplish this.

When an **nv_update_occurs** event is TRUE, the **nv_in_addr** variable is set to contain the LONWORKS addressing information of the sending device. The type is a structure predefined in the Neuron C language as shown below:

```
typedef struct {
   unsigned domain            : 1;
   unsigned flex_domain       : 1;
   unsigned format            : 6;
   struct {
      unsigned subnet;
      unsigned                : 1;
      unsigned node           : 7;
   } src_addr;
   struct {
      unsigned group;
   } dest_addr;
} nv_in_addr_t;
const nv_in_addr_t nv_in_addr;
```

The various fields of the network variable input address structure are:

domain Domain index of the network variable update.

flex_domain Always 0 for network variable updates.

format Addressing format used by the network variable update. Contains one of the following values:

0	Broadcast
1	Group
2	Subnet/Node
3	Neuron ID
4	Turnaround

src_addr Source address of the network variable update. The **subnet** and **node** fields in the **src_addr** are both zero (0) for a turnaround network variable.

dest_addr Destination address of the network variable update if group addressing is used as specified by the format field.

When the **nv_in_addr** variable is used in an application, its value corresponds to the last input network variable updated in the application. The contents of **nv_in_addr** are undefined if no network variable update event has occurred. Updates occur when network variable events are checked or when **post_events( )** is called (either explicitly from the program or by the scheduler between tasks) and events arrive for network variables for which there is no corresponding event check.

See *Monitoring Network Variables* in Chapter 3, *How Devices Communicate Using Network Variables,* of the *Neuron C Programmer's Guide* for more description of how **nv_in_addr** is used.

Use of **nv_in_addr** enables explicit addressing for the application, and affects the required size for input and output application buffers. See Chapter 8, *Memory Management,* of the *Neuron C Programmer's Guide* for more information about allocating buffers.

nv_in_index *Variable*

The **nv_in_index** variable contains the network variable global index for an **nv_update_completes**, **nv_update_fails**, **nv_update_succeeds**, or **nv_update_occurs** event. When one of these events evaluates to TRUE, **nv_in_index** contains the network variable global index to which the event applies. The contents of **nv_in_index** are undefined if no network variable events have occurred. Updates occur when one of the above events are checked or when **post_events( )** is called (either explicitly from the program or by the scheduler between tasks) and events arrive for network variables for which there is no corresponding event.

The global index of a network variable is set during compilation and depends on the order of declaration of the network variables in the program. The type of **nv_in_index** is **unsigned short**. The global index of a network variable can be accessed using either the **global_index** property, or using the **nv_table_index( )** function.

read_only_data *Variable*

read_only_data_2 *Variable*

read_only_data_3 *Variable*

read_only_data_4 *Variable*

The **read_only_data**, **read_only_data_2**, **read_only_data_3** and **read_only_data_4** variables contain the read-only data stored in the Neuron Chip or Smart Transceiver on-chip EEPROM, at location **0xF000**. The secondary part (**read_only_data_2**) is immediately following, but only exists on Neuron Chips or Smart Transceivers with version 6 firmware or later. The tertiary part (**read_only_data_3**) is immediately following, but only exists on Neuron Chips or Smart Transceivers with version 16 firmware or later. The **read_only_data_4** variable immediately follows read_only_data_3, but only exists on Series 6000 chips with version 21 system firmware or later.

This data defines the Neuron identification, as well as some of the application image parameters. The types are structures, declared in **<access.h>** as follows:

```
#define NEURON_ID_LEN   6
#define ID_STR_LEN      8

typedef struct read_only_data_struct {
   unsigned neuron_id [NEURON_ID_LEN];
   unsigned model_num;
   unsigned                         : 4;
```

```
   unsigned minor_model_num         : 4;
   const nv_fixed_struct * nv_fixed;
   unsigned read_write_protect      : 1;
   unsigned                         : 1;
   unsigned nv_count                : 6;
   const snvt_struct     * snvt;
   unsigned id_string [ID_STR_LEN];
   unsigned NV_processing_off       : 1;
   unsigned two_domains             : 1;
   unsigned explicit_addr           : 1;
   unsigned                         : 0;
   unsigned address_count           : 4;
   unsigned                         : 0;
   unsigned                         : 4;
   unsigned receive_trans_count     : 4;
   unsigned app_buf_out_size        : 4;
   unsigned app_buf_in_size         : 4;
   unsigned net_buf_out_size        : 4;
   unsigned net_buf_in_size         : 4;
   unsigned net_buf_out_priority_count    : 4;
   unsigned app_buf_out_priority_count    : 4;
   unsigned app_buf_out_count       : 4;
   unsigned app_buf_in_count        : 4;
   unsigned net_buf_out_count       : 4;
   unsigned net_buf_in_count        : 4;
   unsigned reserved1 [6];
   unsigned                         : 6;
   unsigned tx_by_address           : 1;
   unsigned idempotent_duplicate    : 1;
} read_only_data_struct;

typedef struct read_only_data_struct_2 {
   unsigned read_write_protect_2    : 1;
   unsigned custom_mac              : 1;
   unsigned alias_count             : 6;
   unsigned msg_tag_count           : 4;
   unsigned has_ecs_flag            : 1;
   unsigned prt_count               : 3;
   int      reserved2 [3];
} read_only_data_struct_2;

typedef struct read_only_data_struct_3 {
    unsigned nvCount;
    unsigned aliasCount;
    unsigned long siDataEx;
    unsigned dmf                    : 1;
    unsigned                        : 7;
} read_only_data_struct_3;

typedef struct read_only_data_struct_4 {
    unsigned    address_count;
} read_only_data_struct_4;

const read_only_data_struct   read_only_data;
const read_only_data_struct_2 read_only_data_2;
const read_only_data_struct_3 read_only_data_3;
const read_only_data_struct_4 read_only_data_4;
```

The application program can read these structures, but cannot write them, using **read_only_data**, **read_only_data_2**, **read_only_data_3 and read_only_data_4**. The first structure is 41 bytes, and it can be read and mostly written (except for the first eight bytes) over the network using the *read memory* and *write memory* network management messages with **address_mode=1**. The second structure is seven bytes, the third structure is five bytes and the forth is one byte. The structures are written during the process of downloading a new application image into the device.

For more information about the individual fields of the read-only data structures, see the ISO/IEC 14908 (ANSI/EIA/CEA-709.1) *Control Network Protocol* specification.

Built-In Objects

The following sections list the Neuron C built-in objects alphabetically, providing relevant syntax information and a detailed description of each function.

msg_in *Object*

The **msg_in** object contains an incoming application or foreign-frame message. The type is a structure predefined in Neuron C as shown below:

```
typedef enum {ACKD, UNACKD_RPT,
              UNACKD, REQUEST} service_type;

struct  {
   int          code;
   int          len;
   int          data[MAXDATA];
   boolean      authenticated;
   service_type service;
   msg_in_addr  addr;
   boolean      duplicate;
   unsigned     rcvtx;
} msg_in;
```

The various fields of the **msg_in** object are:

code	Message code for the incoming message.
len	Length of message data in bytes.
data	Message data.
authenticated	TRUE if authenticated, message has passed challenge.
service	Service type for the incoming message.
addr	Source address of this message, and address through which the message was received. See **<msg_addr.h>** include file.
duplicate	Message is a duplicate request. See *Idempotent Versus Non-Idempotent Requests* in the *Neuron C Programmer's Guide*.

rcvtx The index into the receive transaction database for this message.

See *Format of an Incoming Message* in Chapter 6, *How Devices Communicate Using Application Messages,* of the *Neuron C Programmer's Guide* for more information about this structure.

msg_out *Object*

The **msg_out** object contains an outgoing application or foreign frame message. The type is a structure predefined in the Neuron C as shown below:

```
typedef enum {FALSE, TRUE} boolean;
typedef enum {ACKD, UNACKD_RPT,
              UNACKD, REQUEST} service_type;

struct  {
   boolean      priority_on;
   msg_tag      tag;
   int          code;
   int          data[MAXDATA];
   boolean      authenticated;
   service_type service;
   msg_out_addr dest_addr;
} msg_out;
```

The various fields of the **msg_out** object are:

priority_on TRUE if a priority message. Defaults to FALSE.

tag Message tag of the outgoing message. This field must be set.

code Message code of the outgoing message. This field must be set.

data Message data.

authenticated Specifies message is to be authenticated. Defaults to FALSE.

service Service type of the outgoing message. Defaults to ACKD.

dest_addr Optional, see the **<msg_addr.h>** include file.

See *msg_out Object Definition* in Chapter 6, *How Devices Communicate Using Application Messages,* of the *Neuron C Programmer's Guide* for a more detailed description of this structure.

resp_in *Object*

The **resp_in** object contains an incoming response to a request message. The type is a structure predefined in Neuron C as shown below:

```
struct  {
   int           code;
   int           len;
   int           data[MAXDATA];
   resp_in_addr  addr;
} resp_in;
```

The various fields of the **resp_in** object are:

code	Message code for the incoming response message.
len	Length of the message data in bytes.
data	Message data.
addr	Source address of this response, and address through which this response was received. See the **<msg addr.h>** include file.

See *Receiving a Response* in Chapter 6, *How Devices Communicate Using Application Messages,* of the *Neuron C Programmer's Guide* for a more detailed description of this structure.

resp_out *Object*

The **resp_out** object contains an outgoing response message to be sent in response to an incoming request message. The response message 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. The type is a structure predefined in Neuron C as shown below:

```
struct  {
   int   code;
   int   data[MAXDATA];
} resp_out;
```

The various fields of the **resp_out** object are:

code	Message code of the outgoing response message.
data	Message data.

See *Constructing a Response* in Chapter 6, *How Devices Communicate Using Application Messages,* of the *Neuron C Programmer's Guide* for a more detailed description of this structure.

Built-In Symbols

The Neuron C compiler defines a number of pre-processor symbols, listed in **Table 32**, that allow you to share common code with different versions of the compiler or between Neuron-hosted applications and host-based applications that use a model file.

Table 32. Built-In Symbols

Symbol	Description
_ECHELON	Defined for all versions of the Neuron C compiler in the **<echelon.h>** file.
_FAMILY_3100 **_FAMILY_5000** **_FAMILY_6000**	Defined if the Neuron C compilation target is member of the respective chip family.

Symbol	Description
_FTXL	Defined for the Neuron C compiler that is included with the FTXL LonTalk Interface Developer utility for model files.
_ILON	Defined for the Neuron C compiler that is included with the *i*.LON SmartServer LonTalk Interface Developer utility for model files.
_LID3	Defined for the Neuron C compiler that is included with version 3 of the LonTalk Interface Developer utility for model files.
_MODEL_FILE	Defined for the Neuron C compiler that is included with all versions of the LonTalk Interface Developer utility for model files.
_NCC5	Defined for version 5 of the Neuron C compiler. This version corresponds to Version 2.2 of the Neuron C language.
_NCC6	Defined for version 6 of the Neuron C compiler. This version corresponds to Version 2.3 of the Neuron C language.
_NEURONC	Defined for all versions of the Neuron C compiler.
_NODEBUILDER	Defined for the Neuron C compiler that is included with the NodeBuilder FX Development Tool, and later versions.
_PUREC	Defined when compiling in *Pure C Mode*. In Pure C mode, most of the Neuron C advanced features, such as built-in support for network variables or message objects, timers and similar features, is disabled.
_RESIDENT	Expands to the *__resident* keyword for versions of the Neuron C Compiler which support resident and transient function definitions. Expands to nothing with older versions of the compiler.
_SHORTSTACK	Defined for the Neuron C compiler that is included with the ShortStack LonTalk Interface Developer utility for model files.
_SUPPORT_EAT	Defined if the compilation target supports an extended address table exceeding 15 address table entries.

Symbol	Description
_SUPPORT_TRANSIENCE	Defined if the compilation target supports transient functions.

Built-In Semaphore

For Series 5000 and 6000 chips, you use the Neuron C **__lock { }** keyword, followed by a code block, within the application or the interrupt service routine (the interrupt task) to acquire the built-in binary semaphore and synchronize access to shared resources. The **__lock** keyword precedes a block of code (within a pair of curly braces) that is controlled by the semaphor.

The semaphore is not supported for the lowest system clock rates (5 MHz and 10 MHz).

You can use the **__lock** keyword 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**. While the compiler detects and prevents direct nesting of **__lock{ }** constructs within the same function, when task, or interrupt task, the compiler does not detect runtime nesting.

Example: The following example illustrates nesting of locks at runtime.

```
void f() {
   __lock {
      ...  // do something
   }
}

interrupt (IO_0) {

   __lock { // acquire semaphore
      f();
   }        // release semaphore

   ...      // more unguarded code
}
```

This code results in a deadlock because the interrupt task owns the semaphore, thus function **f()** can never succeed in acquiring it. The Neuron firmware

resolves the deadlock by resetting the chip after the watchdog timer expires, but you must be sure to release the semaphore before acquiring it again.

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, access is granted immediately, but a system error is logged.

A

Syntax Summary

This appendix provides a summary of Neuron C Version 2.3 syntax, with some explanatory material interspersed. In general, the syntax presentation starts with the highest, most general level of syntactic constructs and works its way down to the lowest, most concrete level as the appendix progresses. The syntax is divided into sections for ease of use, with declaration syntax first, statement syntax next, and expression syntax last.

Syntax Conventions

In this syntax section, syntactic categories (nonterminals) are indicated by *italic* type, and literal words and character set members (terminals) by **bold** type. In the example below, *basic-net-var* is a nonterminal, meaning it represents a syntactic category, or construct, rather than a literal string of characters to be typed. The symbols **network**, **input**, and **output** are terminals, meaning they are to be typed in exactly as shown.

basic-net-var :
network input
network output

A colon (:) following a nonterminal introduces its definition. Alternative definitions for a nonterminal are listed on separate, consecutive lines, except when prefaced by the phrase "one of," and the alternatives are then shown separated by a vertical bar. The example above shows two alternative definitions on separate lines. The example below shows two alternative definitions using the "one of" notation style.

assign-op :
one of **= | |= | ^= | &= | <<= | >>=**
/= | *= | %= | += | -=

When a definition of a nonterminal has an optional component, that optional component is shown inside square brackets, like this: [*optional-component*]. The following example demonstrates this concept. The square brackets are not to be typed, and are not part of the syntax. They merely indicate that the keyword **repeating** is optional, rather than required.

timer-type :
mtimer [**repeating**]
stimer [**repeating**]

Neuron C External Declarations

The language consists of basic blocks, called "external declarations".

Neuron-C-program :
Neuron-C-program external-declaration
external-declaration

The external declarations are ANSI C declarations like data and function declarations, and Neuron C extensions like I/O object declarations, functional block declarations, and task declarations.

external-declaration :
ANSI-C-declaration
Neuron-C-declaration

ANSI-C-declaration :
; (C language permits extra semicolons)
data-declaration **;**
function-declaration

Neuron-C-declaration :
task-declaration
io-object-declaration **;**
functional-block-declaration **;**
device-property-list-declaration

A data declaration is an ANSI C variable declaration.

data-declaration :
variable-declaration
variable-list

Variable Declarations

The following is ANSI C variable declaration syntax.

variable-declaration-list :
variable-declaration-list variable-declaration **;**
variable-declaration **;**

variable-declaration :
declaration-specifier-list variable-list
declaration-specifier-list

The variable declaration can declare more than one variable in a comma-separated list. A network variable can also optionally include a property list declaration after the variable name (and the variable initializer, if present).

variable-list :
variable-list **,** *extended-variable*
extended-variable

extended-variable :
variable nv-property-list-declaration
variable

variable :
declarator **=** *variable-initializer*
declarator

variable-initializer :
{ *variable-initializer-list* **, }**
{ *variable-initializer-list* **}**
constant-expr

variable-initializer-list :
variable-initializer-list **,** *variable-initializer*
variable-initializer

Declaration Specifiers

The ANSI C declaration specifiers are augmented in Neuron C by adding the connection information, the message tag specifier, configuration property specifiers, network variable specifiers, and timer type specifiers.

declaration-specifier-list :
declaration-specifier-list declaration-specifier
declaration-specifier

declaration-specifier :
timer-type
type-specifier
storage-class-specifier
cv-type-qualifier
configuration-property-specifier
msg_tag
net-var-types
connection-information

type-specifier :
type-identifier
type-keyword
struct-or-union-specifier
enum-specifier

Timer Declarations

Timer objects are declared with one of the following sequences of keywords. Timer objects are specific to Neuron C.

timer-type :
mtimer [**repeating**]
stimer [**repeating**]

Type Keywords

The data type keywords can appear in any order. Floating-point types (**double** and **float**) are not supported in Neuron C.

type-keyword :

char	
double	(Reserved for future implementations)
float	(Reserved for future implementations)
int	
long	
quad	(Reserved for future implementations)
short	
signed	
unsigned	
void	

In addition to the above type keywords, the extended arithmetic library defines two data types as structures, and these can be used as if they were also a *type-keyword*. The **s32_type** is a signed 32-bit integer, and the **float_type** is an IEEE754 single precision floating-point value.

s32_type
float_type

Storage Classes

The ANSI C storage classes are augmented in Neuron C with the additional classes **config**, **eeprom**, **far**, **fastaccess**, **offchip**, **onchip**, **ram**, **system**, and **uninit**. The ANSI C **register** storage class is not supported in Neuron C (it is ignored by the compiler).

class-keyword :

auto
config
eeprom
extern
far
fastaccess
offchip
onchip
ram
register
static
system
typedef
uninit

Type Qualifiers

The ANSI C language also defines type qualifiers for declarations. Although the type qualifier **volatile** is not useful in Neuron C (it is ignored by the compiler), the type qualifier **const** is quite important in Neuron C.

cv-type-qualifiers :

cv-type-qualifiers cv-type-qualifier
cv-type-qualifier

cv-type-qualifier :

const
volatile

Enumeration Syntax

The following is ANSI C enum type syntax.

enum-specifier :

enum *identifier* **{** *enum-value-list* **}**
enum { *enum-value-list* **}**
enum *identifier*

enum-value-list :

enum-const-list **,**
enum-const-list

enum-const-list :
enum-const-list **,** *enum-const*
enum-const

enum-const :
variable-identifier **=** *constant-expr*
variable-identifier

Structure/Union Syntax

The following is ANSI C struct/union type syntax.

struct-or-union-specifier :
aggregate-keyword identifier **{** *struct-decl-list* **}**
aggregate-keyword **{** *struct-decl-list* **}**
aggregate-keyword identifier

aggregate-keyword :
struct
union

struct-decl-list :
struct-decl-list struct-declaration
struct-declaration

struct-declaration :
abstract-decl-specifier-list struct-declarator-list **;**

struct-declarator-list :
struct-declarator-list **,** *struct-declarator*
struct-declarator

struct-declarator :
declarator
bitfield

bitfield :
declarator **:** *constant-expr*
: *constant-expr*

Configuration Property Declarations

Configuration properties are declared with one of the following sequences of keywords. Configuration properties are specific to Neuron C, and were introduced in Neuron C Version 2. The first syntax alternative is used to declare configuration properties implemented as configuration network variables, and the second alternative is used to declare configuration properties implemented in configuration files.

configuration-property-specifier :
cp [*cp-info*] [*range-mod*] (for configuration NVs (CPNVs))
cp_family [*cp-info*] [*range-mod*] (for CPs implemented in

files)

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

cp-option-list :
cp-option-list **,** *cp-option*
cp-option-list *cp-option*
cp-option

cp-option :
one of **device_specific** | **manufacturing_only** | **object_disabled** | **offline** | **reset_required**

range-mod :
range_mod_string (*concatenated-string-constant* **)**

Network Variable Declarations

Network variables are declared with one of the following sequences of keywords. Network variables are specific to Neuron C. The changeable type network variable was introduced in Neuron C Version 2.

net-var-types :
basic-net-var [*net-var-modifier*] [*changeable-net-var*]

basic-net-var :
network input
network output

net-var-modifier :
one of **polled** | **sync** | **synchronized**

changeable-net-var :
changeable_type

Connection Information

The *connection-information* feature (**bind_info**) is Neuron C specific. It allows the Neuron C programmer to communicate specific options directly to the network management tool for individual message tags and network variables. Connection information can only be part of a *declaration-specifier-list* that also contains either the **msg_tag** or *net-var-type declaration-specifier*.

connection-information :
bind_info (*bind-info-option-list* **)**
bind_info ( )

bind-info-option-list :
bind-info-option-list *bind-info-option*
bind-info-option

bind-info-option :

auth (*configurable-keyword* **)**
authenticated (*configurable-keyword* **)**
auth
authenticated
bind
nonbind
offline
priority (*configurable-keyword* **)**
priority
nonpriority (*configurable-keyword* **)**
nonpriority
rate-est-keyword **(** *constant-expr* **)**
service-type-keyword **(** *configurable-keyword* **)**
service-type-keyword

rate-est-keyword :

max_rate_est
rate_est

service-type-keyword :

ackd
unackd
unackd_rpt

configurable-keyword :

config
nonconfig

Declarator Syntax

The following is ANSI C declarator syntax. Pointers are not supported within network variables.

declarator :

***** *type-qualifier declarator*
***** *declarator*
sub_declarator

sub-declarator :

sub-declarator array-index-declaration
sub-declarator function-parameter-declaration
(*declarator* **)**
variable-identifier

array-index-declaration :

[*constant-expr* **]**
[ ]

function-parameter-declaration :
formal-parameter-declaration
prototype-parameter-declaration

formal-parameter-declaration :
(*identifier-list* **)**
( )

identifier-list :
identifier-list **,** *variable-identifier*
variable-identifier

prototype-parameter-declaration :
(*prototype-parameter-list* **)**
(*prototype-parameter-list* **, ...)** (not supported in Neuron C)

prototype-parameter-list :
prototype-parameter-list **,** *prototype-parameter*
prototype-parameter

prototype-parameter :
declaration-specifier-list prototype-declarator
declaration-specifier-list

prototype-declarator :
declarator
abstract-declarator

Abstract Declarators

The following is ANSI C abstract declarator syntax.

abstract-declarator :
*****
***** *cv-type-qualifier abstract-declarator*
***** *abstract-declarator*
***** *cv-type-qualifiers*
abstract-sub-declarator

abstract-sub-declarator :
(*abstract-declarator* **)**
abstract-sub-declarator **( )**
abstract-sub-declarator prototype-parameter-declaration
abstract-sub-declarator array-index-declaration **( )**
prototype-parameter-declaration
array-index-declaration

abstract-type :
abstract-decl-specifier-list abstract-declarator
abstract-decl-specifier-list

abstract-decl-specifier-list :
abstract-decl-specifier-list abstract-decl-specifier
abstract-decl-specifier

abstract-decl-specifier :
type-specifier
cv-type-qualifier

Task Declarations

Neuron C contains task declarations. Task declarations are similar to function declarations. A task declaration consists of a **when** or an **interrupt** clause list, followed by a task. A task is a compound statement (like an ANSI C function body).

task-declaration :
when-clause-list task

interrupt-clause task

when-clause-list :
when-clause-list when-clause
when-clause

when-clause :
priority preempt_safe when *when-event*
priority when *when-event*
preempt_safe when *when-event*
when *when-event*

interrupt-clause :

interrupt *interrupt-event*

task :
compound-stmt

Function Declarations

The following is ANSI C function declaration syntax.

function-declaration :
function-head compound-stmt

function-head :
function-type-and-name parm-declaration-list
function-type-and-name

function-type-and-name :
declaration-specifier-list declarator

parm-declaration-list :
parm-declaration-list parm-declaration
parm-declaration

parm-declaration :
declaration-specifier-list *parm-declarator-list* **;**

parm-declarator-list :
parm-declarator-list **,** *declarator*
declarator

Conditional Events

In Neuron C, an event is an expression which can evaluate to either TRUE or FALSE. Neuron C extends the ANSI C concept of conditional expressions through special built-in functions that test for the presence of special Neuron firmware events. The Neuron C compiler has many useful built-in events that cover all the common cases encountered in Neuron programming. However, a Neuron C programmer can also create custom events by using any parenthesized expression as an event, including one or more function calls, and so on.

when-event :
(reset)
predefined-event
parenthesized-expr

interrupt-event :
(repeating [, *frequency-value*] **)**
(*io-object-declarator* **)**
(*io-object-pin-name, interrupt-condition* **)**

predefined-event :
(flush_completes)
(offline)
(online)
(wink)
(*complex-event* **)**

Complex Events

All of the predefined events shown above can be used not only in the *when-clause* portion of the task declaration but also in any general expression in executable code. The complex events below use a function-call syntax, instead of the keyword syntax of the special events above.

complex-event :
io-event
message-event
net-var-event
timer-event

io-event :
io_update_occurs (*variable-identifier* **)**
io_changes (*variable-identifier* **)**
io_changes (*variable-identifier* **) by** *shift-expr*
io_changes (*variable-identifier* **) to** *shift-expr*

message-event :
message-event-keyword **(** *expression* **)**
message-event-keyword

message-event-keyword :
msg_arrives
msg_completes
msg_fails
msg_succeeds
resp_arrives

net-var-event :
nv-event-keyword **(** *net-var-identifier* **..** *net-var-identifier* **)**
nv-event-keyword **(** *variable-identifier* **)**
nv-event-keyword

net-var-identifier :
variable-identifier **[** *expression* **]**
variable-identifier

nv-event-keyword :
nv_update_completes
nv_update_fails
nv_update_occurs
nv_update_succeeds

timer-event :
timer_expires (*variable-identifier* **)**
timer_expires

interrupt-condition :
clock-edge
clockedge (*clock-edge* **)**

frequency-value :
A quoted string value from 2,441.406 to 625,000, in 256 steps. The default *frequency-value* is "8kHz". You can append an optional multiplier: "Hz" (hertz), "kHz" (kilohertz), "MHz" (megahertz), or "GHz" (gigahertz). See the *Neuron C Programmer's Guide* for more information about specifying the *frequency-value*.

I/O Object Declarations

An I/O object declaration is similar to an ANSI C variable declaration. It can contain an initialization.

io-object-declaration:
modified-io-object-declarator variable-identifier **=** *assign-expr*
modified-io-object-declarator variable-identifier

The I/O object declaration begins with an I/O object declarator, possibly followed by one or more I/O object option clauses.

modified-io-object-declarator :
io-object-declarator [*io-option-list*]

io-option-list :
io-option-list io-option
io-option

The I/O object declarator begins with a pin name, followed by the I/O object type.

io-object-declarator :
io-object-pin-name [*io-object-direction*] *io-object-type*

io-object-pin-name :
one of **IO_0 | IO_1 | IO_2 | IO_3 | IO_4 | IO_5**
IO_6 | IO_7 | IO_8 | IO_9 | IO_10 | IO_11

io-object-direction :
one of **input | output**

io-object-type :
one of **bit | bitshift | byte**
dualslope
edgedivide | edgelog
frequency
i2c | infrared | infrared_pattern
leveldetect
magcard | magcard_bitstream| magtrack1
muxbus
neurowire | nibble
oneshot | ontime
parallel | period | pulsecount | pulsewidth
quadrature | sci | serial | spi
totalcount | touch | triac | stretchedtriac
triggeredcount
wiegand

I/O Options

Most I/O options only apply to a few specific object types. The detailed reference documentation in the *I/O Model Reference* explains each option that applies for that I/O object.

io-option :
baud (*constant-expr* **)**
clock (*constant-expr* **)**
clockedge (*clock-edge* **)**
ded
__fast
invert
kbaud (*constant-expr* **)**
long
master
mux
numbits (*constant-expr* **)**
__parity (*constant-expr* **)**

select (*io-object-pin-name* **)**
short
single_tc
slave
slave_b
__slow
sync (*io-object-pin-name* **)**
synchronized (*io-object-pin-name* **)**
timing(*constant-expr* **,** *constant-expr* **,** *constant-expr* **)**
twostopbits
use_stop_condition

The clock-edge option is specified using either the plus or the minus character, or both characters in the case of a dual-edge clock. The dual-edge clock (**+-**) is not available on minor model 0 of the Neuron 3150 Chip.

clock-edge :
one of **+** | **-** | **+-**

Functional Block Declarations

The following is Neuron C syntax for functional block declarations. The functional block is based on a functional profile definition from a resource file.

functional-block-declaration :
fblock-main fblock-name-section fblock-property-list-declaration
fblock-main fblock-name-section

fblock-main :
fblock *FPT-identifier* **{** *fblock-body* **}**
fblock *FPT-identifier* **{ }**

FPT-identifier :
variable-identifier

The body of the functional block declaration consists of a list of network variable members that the functional block implements. At the end of the list, the functional block declaration can optionally declare a director function.

fblock-body :
*fblock-member-list fblock-director-declaratio*n
fblock-member-list
fblock-director-declaration

fblock-member-list :
fblock-member-list fblock-member **;**
fblock-member **;**

fblock-member :
net-var-identifier member-implementation

member-implementation :
implements *variable-identifier*
implementation_specific (*constant-expr* **)** *variable-identifier*

The functional block name can specify either a scalar or a single-dimensioned array (like a network variable declaration). The functional block can also optionally have an external name, and this external name can either be a string constant or a resource file reference.

fblock-name-section :
fblock-name *fblock-external-name*
fblock-name

fblock-name :
variable-identifier **[** *constant-expr* **]**
variable-identifier

fblock-external-name :
external_name (*concatenated-string-constant* **)**
external_resource_name (*concatenated-string-constant* **)**
external_resource_name (*constant-expr* **:** *constant-expr* **)**

Property List Declarations

The following is Neuron C syntax for property declarations. The property declarations for the device, for a network variable, and for a functional block are identical in syntax except for the introductory keyword. The keywords were designed to be different to promote readability of the Neuron C code. Although a network variable or a functional block can only have at most one property list, there can be any number of device property list declarations throughout a program, and the lists are merged into a single property list for the device. This feature promotes modularity of code.

device-property-list-declaration :
device_properties **{** *property-instantiation-list* **}**

nv-property-list-declaration :
nv_properties **{** *property-instantiation-list* **}**

fblock-property-list-declaration :
fb_properties **{** *property-instantiation-list* **}**

The property instantiation list is a comma-separated list of one or more property instantiations. A property instantiation uses the name of a previously declared network variable configuration property or the name of a previously declared CP family. The instantiation can optionally be followed by either an initialization or a range-modification, or both, in either order.

property-instantiation-list :
property-instantiation-list **,** *complete-property-instantiation*
complete-property-instantiation

complete-property-instantiation :
property-instantiation [*property-initialization*] [*range-mod*]
property-instantiation [*range-mod*] [*property-initialization*]

property-initialization :
= *variable-initialization*

property-instantiation :
[*property-qualifier*] *cpnv-prop-ident*
[*property-qualifier*] *cp-family-prop-ident*

property-qualifier :
one of **global** | **static**

cpnv-prop-ident :
net-var-identifier **[** *constant-expression* **]**
net-var-identifier

cp-family-prop-ident :
variable-identifier

Statements

The following is ANSI C statement syntax. Compound statements begin and end with left and right braces, respectively. Compound statements contain a variable declaration list, a statement list, or both. The variable declaration list, if present, must precede the statement list.

compound-stmt :
{ [*variable-declaration-list*] [*statement-list*] **}**

statement-list :
statement-list statement
statement

In the C language, there is a grammatical distinction between a complete statement and an incomplete statement. This is basically done for one reason, and that is to permit the grammar to unambiguously decide which **if** statement goes with which **else** statement. An **if** statement without an **else** is called an incomplete statement.

statement :
complete-stmt
incomplete-stmt

complete-stmt :
compound-stmt
label **:** *complete-stmt*
break ;
continue ;
do *statement while-clause* **;**
for-head complete-stmt
goto *identifier* **;**
if-else-head complete-stmt
__lock *compound-statement*
switch-head complete-stmt
return ;
return *expression* **;**
while-clause complete-stmt
expression **;**
;

incomplete-stmt :
label **:** *incomplete-stmt*
for-head incomplete-stmt
if-else-head incomplete-stmt
if-head statement
switch-head incomplete-stmt
while-clause incomplete-stmt

These are the various pieces that make up the statement syntax from above.

label :
case *expression*
default
identifier

if-else-head :
if-head complete-stmt **else**

if-head :
if *parenthesized-expr*

for-head :
for ([*expression*] **;** [*expression*] **;** [*expression*] **)**

switch-head :
switch *parenthesized-expr*

while-clause :
while *parenthesized-expr*

Expressions

The following is expression syntax.

parenthesized-expr :
(*expression* **)**

constant-expr :
expression

expression :
expression **,** *assign-expr*
assign-expr

assign-expr :
choice-expr assign-op assign-expr
choice-expr

assign-op :
one of **=** | **|=** | **^=** | **&=** | **<<=** | **>>=**
/= | ***=** | **%=** | **+=** | **-=**

choice-expr :
logical-or-expr **?** *expression* **:** *choice-expr*
logical-or-expr

logical-or-expr :
logical-or-expr **||** *logical-and-expr*
logical-and-expr

logical-and-expr :
logical-and-expr **&&** *bit-or-expr*
bit-or-expr

bit-or-expr :
bit-or-expr **|** *bit-xor-expr*
bit-xor-expr

bit-xor-expr :
bit-xor-expr **^** *bit-and-expr*
bit-and-expr

bit-and-expr :
bit-and-expr **&** *equality-comparison*
equality-comparison

equality-comparison :
equality-comparison **==** *relational-comparison*
equality-comparison **!=** *relational-comparison*
relational-comparison

relational-comparison :
relational-comparison relational-op io-change-by-to-expr
io-change-by-to-expr

relational-op :
one of **<** | **<=** | **>=** | **>**

io-change-by-to-expr :
io_changes (*variable-identifier* **) by** *shift-expr*
io_changes (*variable-identifier* **) to** *shift-expr*
shift-expr

shift-expr :
shift-expr shift-op additive-expr
additive-expr

shift-op :
one of **<<** | **>>**

additive-expr :
additive-expr add-op multiplicative-expr
multiplicative-expr

add-op :

one of **+** | **-**

multiplicative-expr :

multiplicative-expr mul-op cast-expr
cast-expr

mul-op :

one of ***** | **/** | **%**

cast-expr :

(*abstract-type* **)** *cast-expr*
unary-expr

unary-expr :

unary-op cast-expr
sizeof *unary-expr*
sizeof (*abstract-type* **)**
predefined-event

unary-op :

one of ***** | **&** | **!** | **~** | **+** | **-** | **++** | **--**

postfix-expr :

postfix-expr **[** *expression* **]**
postfix-expr **->** *identifier*
postfix-expr **.** *identifier*
postfix-expr **++**
postfix-expr **--**
postfix-expr actual-parameters
primary-expr

actual-parameters :

(*actual-parameter-list* **)**
( )

actual-parameter-list :

actual-parameter-list **,** *assign-expr*
assign-expr

Primary Expressions, Built-in Variables, and Built-in Functions

In addition to the ANSI C definitions of a primary expression, Neuron C adds some built-in variables and built-in functions. Neuron C removes *float-constant* from the standard list of primary expressions.

primary-expr :

parenthesized-expr
integer-constant
concatenated-string-constant
variable-identifier
property-reference
builtin-variables
builtin-functions actual-parameters
msg-call-kwd **( )**

concatenated-string-constant :

concatenated-string-constant string-constant
string-constant

property-reference :

[*postfix-expr*] **::** *variable-identifier*
postfix-expr **:: director** *actual-parameters*
postfix-expr **:: global_index**
postfix-expr **:: nv_len**

network-variable-reference :

variable-identifier

variable-identifier :: global_index
variable-identifier :: nv_len
variable-identifier :: __type_index
variable-identifier :: __type_scope

builtin-variables :

one of **activate_service_led**
config_data
cp_modifiable_value_file
cp_modifiable_value_file_len
cp_readonly_value_file
cp_readonly_value_file_len
cp_template_file | cp_template_file_len
fblock_index_map
input_is_new | input_value
msg_tag_index
nv_array_index | nv_in_addr | nv_in_index

read_only_data | **read_only_data_2**
msg-name-kwd **.** *variable-identifier*

msg-name-kwd :

one of **msg_in** | **msg_out** | **resp_in** | **resp_out**

builtin-functions :

one of **abs** | **addr_table_index**
bcd2bin | **bin2bcd**
eeprom_memcpy
fblock_director
get_fblock_count | **get_nv_count**
get_tick_count
high_byte
interrupt_control
io_change_init
io_in | **io_in_ready** | **io_in_request**
io_out | **io_out_ready** | **io_out_request**
io_preserve_input
io_select
io_set_baud | **io_set_clock** | **io_set_direction**
io_set_terminal_count
is_bound
low_byte
make_long
max
memcpy | **memset**
min
nv_table_index
poll
propagate
sci_abort | **sci_get_error**
sleep
spi_abort | **spi_get_error**
swap_bytes
touch_bit | **touch_byte** | **touch_first**
touch_next | **touch_reset**
touch_byte_spu | **touch_read_spu**
touch_read_spu | **touch_write_spu**
touch_reset_spu

msg-call-kwd :

one of **msg_alloc** | **msg_alloc_priority**
msg_cancel | **msg_free** | **msg_receive**
msg_send | **resp_alloc** | **resp_cancel**
resp_free | **resp_receive** | **resp_send**

Implementation Limits

The contents of the standard include file **<limits.h>** are given below.

```
#define CHAR_BIT      8
#define CHAR_MAX      255
#define CHAR_MIN      0
```

```
#define SCHAR_MAX    127
#define SCHAR_MIN    ((signed char)(-128))

#define UCHAR_MAX    255

#define SHRT_MAX     127
#define SHRT_MIN     ((signed short)(-128))

#define USHRT_MAX    255

#define INT_MAX      127
#define INT_MIN      ((int)(-128))

#define UINT_MAX     255

#define LONG_MAX     32767
#define LONG_MIN     ((signed long)(-32768))

#define ULONG_MAX    65535

#define MB_LEN_MAX   2
```

B

Reserved Keywords

This chapter lists all Neuron C Version 2.3 reserved keywords, including the standard reserved keywords of the ANSI C language.

Reserved Words List

The following list of reserved words includes keywords in the Neuron C language as well as Neuron C built-in symbols. Each of these reserved words should be used only as it is defined elsewhere in this reference guide. A Neuron C programmer should avoid the use of any of these reserved words for other purposes, such as variable declarations, function definitions, or **typedef** names.

Following each reserved word is a code indicating the usage of the particular item:

“(c)”	indicates a keyword from the ANSI C language
“(1)”	indicates keywords from Neuron C Version 1
“(2)”	indicates a keyword introduced in Neuron C Version 2
"(2.1)"	indicates a keyword introduced in Neuron C Version 2.1
"(2.2)"	indicates a keyword introduced in Neuron C Version 2.2
“(2.3)”	indicates a keyword introduced in Neuron C Version 2.3

The remaining reserved words are built-in symbols in the Neuron C Compiler, many of which are found in the **<echelon.h>** file that is always included at the beginning of a Neuron C compilation. Various codes are used to indicate the type of built-in symbol:

"(et)”	indicates an **enum** tag
“(st)”	indicates a **struct** tag
“(t)”	indicates a **typedef** name
“(f)”	indicates a built-in function name
“(v)”	indicates a built-in variable name
“(e)"	indicates an enum value literal
“(d)”	indicates a built-in **#define** preprocessor symbol
“(w)”	denotes a built-in event name (as used in a when clause)
“(p)”	indicates a built-in property name (new to Neuron C Version 2)

abs (f)

ackd (1)

ACKD (e)

addr_table_index (f)

auth (1)

authenticated (1)

auto (c)

bank_index (f)

baud (1)

bcd (st)

bcd2bin (f)

bin2bcd (f)

bind (1)

bind_info (1)

bit (1)

bitshift (1)

boolean (et,t)

break (c)

interrupt_control (f, 2.2)

INTERRUPT_IO (d)

INTERRUPT_REPEATING (d)

INTERRUPT_TC (d)

invert (1)

IO_0 (1)

IO_1 (1)

IO_10 (1)

IO_11 (2.1)

IO_2 (1)

IO_3 (1)

IO_4 (1)

IO_5 (1)

IO_6 (1)

IO_7 (1)

IO_8 (1)

IO_9 (1)

io_change_init (f)

io_changes (w)

io_direction (et,t)

IO_DIR_IN (e)

IO_DIR_OUT (e)

io_edgelog_preload (1)

io_edgelog_single_preload (2.1)

io_in (f)

io_in_ready (f)

io_in_request (f)

io_out (f)

io_out_ready (f)

io_out_request (f)

io_preserve_input (f)

io_select (f)

io_set_baud (2.1)

io_set_clock (f)

io_set_direction (f)

io_set_terminal_count (f, 2.2)

io_update_occurs (w)

is_bound (f)

kbaud (1)

level (1)

leveldetect (1)

__lock (2.2)

long (c)

low_byte (f)

magcard (1)

magcard_bitstream (2.1)

magtrack1 (1)

make_long (f)

manufacturing_only (2)

master (1)

max (f)

max_rate_est (1)

memcpy (f)

memset (f)

min (f)

msg_alloc (f)

msg_alloc_priority (f)

msg_arrives (w)

msg_cancel (f)

msg_completes (w)

msg_fails (w)

msg_free (f)

msg_in (v)

msg_out (v)

msg_receive (f)

msg_send (f)

msg_succeeds (w)

msg_tag (1)

msg_tag_index (f)

mtimer (1)

select (1)
serial (1)
service_type (et,t)
short (c)
signed (c)
single_tc (2.1)
sizeof (c)
slave (1)
slave_b (1)
sleep (f)
sleep_flags (et,t)
__slow
spi (2.1)
spi_abort (2.1)
spi_get_error (2.1)
static (c)
stimer (1)
stretchedtriac (2.2)
struct (c)
swap_bytes (f)
switch (c)
sync (1)
synchronized (1)
system (1)
timeout (1)
timer_expires (w)
TIMERS_OFF (e)
timing (2.1)
to (1)
totalcount (1)
touch (1)
touch_bit (f)
touch_byte (f)
touch_byte_spu (f, 2.2)
touch_first (f)
touch_next (f)
touch_read_spu (f, 2.2)
touch_reset (f)
touch_reset_spu (f, 2.2)
touch_write_spu (f, 2.2)
triac (1)
triggeredcount (1)
TRUE (e)
twostopbits (2.1)
__type_index (2.3)
__type_scope (2.3)
typedef (c)
unackd (1)
UNACKD (e)
unackd_rpt (1)
UNACKD_RPT (e)
uninit (1)
union (c)
unsigned (c)
use_stop_condition (2.1)
void (c)
volatile (c)
when (1)
while (c)
wiegand (1)
wink (1)

Finally, and in addition to the restrictions imposed by the previous list, the compiler automatically recognizes names of standard network variable types (SNVT*), standard configuration property types (SCPT*), standard functional profiles (SFPT*), as well as the user types and functional profiles applicable to the current program ID.

The compiler does not permit the program to define any symbol starting with any of the following prefixes: SCPT, SFPT, UNVT, UCPT, or UFPT, unless the **#pragma names_compatible** directive is present in the program.

In addition to the restrictions imposed by the previous list of reserved words, the programmer cannot use the following reserved names at all; they are part of the compiler-firmware interface only, and are not permitted in a Neuron C program.

_bcd2bin
_bin2bcd
_bit_input
_bit_input_ext
_bit_output_ext
_bit_output_hi
_bit_output_lo1
_bit_output_lo2
_bitshift_input
_bitshift_output
_bound_mt
_bound_nv
burst_sequence_output
_byte_input
_byte_output
cp_modifiable_value_file_len_fake
cp_readonly_value_file_len_fake
cp_template_file_len_fake
crc16_ccitt
_dualslope_input
_dualslope_start
_edgelog_input
edgelog_input_single
edgelog_setup_single
ext_touch_bit
ext_touch_byte
ext_touch_first
ext_touch_next
ext_touch_read
ext_touch_reset
ext_touch_write
fblock_index_map
_flush_completes
_frequency_output
_i2c_read
_i2c_read0
_i2c_read8
_i2c_read_opt0
_i2c_read_opt8
_i2c_write
_i2c_write0
_i2c_write8
_i2c_write_opt0
_i2c_write_opt8
_i2cs_read0
_i2cs_read8
_i2cs_read_opt0
_i2cs_read_opt8
_i2cs_write0
_i2cs_write8
_i2cs_write_opt0
_i2cs_write_opt8
_init_baud
_init_timer_counter1
_init_timer_counter2
_io_abort_clear
_io_change_init
_io_changes
_io_changes_by

_io_changes_to
_io_direction_hi
_io_direction_lo
io_idis
io_iena
_io_input_value
_io_sci_baud
_io_sci_get_error
_io_sci_init
_io_sci_initram
_io_sci_set_buffer_in
_io_sci_set_buffer_out
_io_scispi_abort
_io_scispi_input_ready
_io_scispi_output_ready
_io_set_clock
_io_set_clock_x2
_io_spi_clock
_io_spi_init
_io_spi_initram
_io_spi_set_buffer
_io_update_occurs
_ir_input
_leveldetect_input
_magcard_input
_magt1_input
_magt2_input
magx_input
_memcpy
_memcpy16
_memcpy8
_memset
_memset16
_memset8
_msg_addr_blockget
_msg_addr_blockset
_msg_addr_get
_msg_addr_set
_msg_alloc
_msg_alloc_priority
_msg_arrives
_msg_auth_get
_msg_auth_set
_msg_cancel
_msg_code_arrives
_msg_code_get
_msg_code_set
_msg_completes
_msg_data_blockget
_msg_data_blockset
_msg_data_get
_msg_data_set
_msg_domain_get
_msg_domain_set
_msg_duplicate_get
_msg_fails
_msg_format_get
_msg_free
_msg_len_get
_msg_node_set
_msg_priority_set
_msg_rcvtx_get
_msg_receive
_msg_send
_msg_service_get
_msg_service_set
_msg_succeeds
_msg_tag_set
_muxbus_read
_muxbus_reread
_muxbus_rewrite
_muxbus_write

_neurowire_inv_master
_neurowire_inv_slave
_neurowire_master
_neurowire_slave
_nibble_input
_nibble_output
_nv_array_poll
_nv_array_update_completes
_nv_array_update_fails
_nv_array_update_occurs
_nv_array_update_request
_nv_array_update_succeeds
_nv_poll
_nv_poll_all
_nv_update_completes
_nv_update_fails
_nv_update_occurs
_nv_update_request
_nv_update_request_all
_nv_update_succeeds
_offline
_oneshot_output
_online
_parallel_input
_parallel_input_ready
_parallel_output
_parallel_output_ready
_parallel_output_request
_period_input
_pulsecount_output
_pulsewidth_output
_quadrature_input
_resp_alloc
_resp_arrives
_resp_cancel
_resp_code_set
_resp_data_blockset
_resp_data_set
_resp_free
_resp_receive
_resp_send
_select_input_fn
_serial_input
_serial_output
_sleep
_timer_expires
_timer_expires_any
_totalize_input
_touch_bit
_touch_byte
_touch_first
_touch_next
_touch_read
_touch_reset
_touch_write
_triac_level_output
_triac_pulse_output
_triacStrOut
_triacStrInit
_triacStrMax
_wiegand_input
_wink

Index

C

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W