Neuron® C Reference Guide

Revision 5

ECHELON®

Corporation

078-0140-02E

Document No. 29350

Printed in the United States of America.
Copyright ©1990-2003 by Echelon Corporation

Echelon Corporation
550 Meridian Avenue
San Jose, CA. USA
95126

www.echelon.com

Preface

This manual describes the Neuron® C Version 2.1 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 Neuron Chips and Smart Transceivers. 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.

Audience

The *Neuron C Reference 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:

- American National Standard X3.159-1989, Programming Language C, D.F. Prosser, American National Standards Institute, 1989.
- Standard C: Programmer's Quick Reference, P. J. Plauger and Jim Brodie, Microsoft Press, 1989.
- C: A Reference Manual, Samuel P. Harbison and Guy L. Steele, Jr., 4th edition, Prentice-Hall, Inc., 1994.
- The C Programming Language, Brian W. Kernighan and Dennis M. Ritchie, 2nd edition, Prentice-Hall, Inc., 1988.

Content

This guide provides a complete reference for the Neuron C Version 2 programming language.

Related Manuals

The *NodeBuilder® User's Guide* lists and describes all tasks related to LONWORKS application development using the NodeBuilder Development Tool. Refer to that guide for detailed information on the user interface and features of the NodeBuilder tool.

The *LonBuilder® User's Guide* lists and describes all tasks related to LONWORKS application development using the LonBuilder Development Tool. Refer to that guide for detailed information on the user interface and features of the LonBuilder tool.

The *Neuron C Programmer's Guide* outlines and discusses the key aspects of developing a LONWORKS application and explains the key concepts of programming in Neuron C Version 2 through the use of code fragments and examples.

The *NodeBuilder Errors Guide* lists and describes all warning and error messages related to the NodeBuilder software.

The *LonMaker® User's Guide* lists and describes all tasks related to LONWORKS network development and maintenance using the LonMaker Integration Tool. Refer to that guide for detailed information on the user interface and features of the LonMaker tool.

The *Gizmo 4 User's Guide* describes the Gizmo 4 I/O board hardware and software. Refer to that guide for detailed information on the hardware and software interface of the Gizmo 4.

The *FT 3120® and FT 3150® Smart Transceivers Databook* and the *PL 3120®/PL 3150® Power Line Smart Transceiver Data Book* describe the hardware and architecture for Echelon's Smart Transceivers. (These books are also called the Smart Transceivers databooks elsewhere in this manual.) Other Neuron Chip information is available from the respective manufacturers of those devices.

The *LTS-20 LonTalk Serial Adapter and PSG-20 User's Guide* describes how to use an LTS-20 LonTalk Serial Adapter and a host processor with EIA-232 (formerly RS-232) serial interface with a LONWORKS network.

The *Power Line SLTA Adapter and Power Line PSG/3 User's Guide* describes how to use the PL-SLTA Serial LonTalk Adapter to connect a host processor with an EIA-232 (formerly RS-232) serial interface to a LONWORKS network. The connection to the host processor can be made either directly or remotely through a pair of modems.

Typographic Conventions for Syntax

This manual uses the following typographic conventions for syntax:

Type	*Used For*	*Example*
boldface type	keywords literal characters	**network** **{**
Italic type	abstract elements	*identifier*
square brackets	optional fields	[*bind-info*]
vertical bar	a choice between two elements	**input | output**

For example, the syntax for declaring a network variable is shown below:

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

Punctuation other than square brackets and vertical bars must be used where shown (quotes, parentheses, semicolons, etc.).

Code examples appear in the `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
}
```

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, I/O, and event-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 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. 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.

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.

The rest of this reference guide will discuss these various aspects of Neuron C in much greater detail, accompanied by examples.

1
Predefined Events

This chapter provides reference information on predefined events.

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. Neuron C defines a number of predefined events for events that are managed by the Neuron firmware. Predefined events are represented by unique keywords, listed in the table below. Some predefined events, such as the I/O events, may be followed by a modifier that narrows the scope of the event. If the modifier is optional and not supplied, any event of that type qualifies. The following table lists events by functional group.

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

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

flush_completes EVENT

flush_completes

The **flush_completes** event evaluates to TRUE 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.*

EXAMPLE:

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

io_changes <span>EVENT</span>

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

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 (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, a state change occurs 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 **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 may not persist until the **io_changes** event can be processed.

Following are more detailed descriptions of the elements of the above syntax:

io-object-name	The I/O object name (see Chapter 8, *I/O Objects*). I/O objects of the following input object types can be used in an unqualified change event. The **by** and **to** options may 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

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

io-object-name	The I/O object name (see Chapter 8, *I/O Objects*).

The **io_in_ready** event evaluates to TRUE when a block of data is available to be read on the **parallel** I/O interface. The application then calls **io_in()** to retrieve the data. (See also the *Parallel I/O Interface to the Neuron Chip* engineering bulletin and the *Parallel I/O Object* in Chapter 8.)

The **io_in_ready** event is also used with the **sci** and **spi** objects to determine when the transfer is complete.

EXAMPLE:

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

io_out_ready EVENT

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

io-object-name The I/O object name (see Chapter 8, *I/O Objects*).

The **io_out_ready** event evaluates to TRUE whenever the parallel I/O interface is in a state where it can be written to and the **io_out_request()** function has been previously invoked. (See also the *Parallel I/O Interface to the Neuron Chip* engineering bulletin and the *Parallel I/O Object* description in Chapter 8.)

The **io_out_ready** event is also used with the **sci** and **spi** objects to determine when the transfer is complete.

EXAMPLE:

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

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

io_update_occurs EVENT

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

io-object-name The I/O object name (see Chapter 8, *I/O Objects*).

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 as shown below:

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.

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

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.

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.

EXAMPLE:

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

msg_completes EVENT

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

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

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**.

EXAMPLE:

```
msg_tag tag_out;
...
msg_out.tag = tag_out;
msg_send();
...
when (msg_completes(tag_out))
{
   ...
}
```

msg_fails EVENT

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

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

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**.

EXAMPLE:

```
msg_tag tag_out;
...
msg_out.tag = tag_out;
msg_send();
...
when (msg_fails(tag_out))
{
   ...
}
```

msg_succeeds EVENT

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

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

The **msg_succeeds** event evaluates to TRUE when a message is successfully sent (see Table 6.2 in 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**.

EXAMPLE:

```
msg_tag tag.out;
...
msg_out.tag = tag_out;
msg_send();
...
when (msg_succeeds(tag_out))
{
   ...
}
```

nv_update_completes EVENT

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). If the parameter is omitted, the event is TRUE when any network variable update completes.

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 will be checked for completion. The event will occur once for each element that experiences a completion event. An individual element may be checked with use of an array index. When **nv_update_completes** is TRUE for an event qualified by the name of an entire NV array, you may 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 will be checked for completion until the first such network variable with an event is found. The event will occur 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.

EXAMPLE 1 – EVENT FOR A SINGLE NETWORK VARIABLE:

```
network output int humidity;
...
humidity = 32;     // This initiates an NV update
...
when (nv_update_completes(humidity))
{
   ...
}
```

EXAMPLE 2 – EVENT FOR A NETWORK VARIABLE ARRAY:

```
network output int humidity[4];
...
humidity[1] = 32; // This initiates an NV update
...
when (nv_update_completes(humidity))
{
   ...
}
```

EXAMPLE 3 – EVENT FOR A RANGE OF NETWORK VARIABLES:

```
network output int humidity1, humidity2, humidity3;
...
humidity2 = 32;          // This initiates an NV update
...
when (nv_update_completes(humidity1 .. humidity3))
{
  ...
}
```

nv_update_fails EVENT

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). If the parameter is omitted, the event is TRUE when any network variable update fails.

The **nv_update_fails** event evaluates to TRUE when an output network variable update or poll fails (see Table 6-2 in the *Neuron C Programmer's Guide* for the definition of success).

If an array name is used, then each element of the array will be checked for failure. The event will occur once for each element that experiences a failure event. An individual element may 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 will be checked for failure until the first such network variable with an event is found. The event will occur 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.

EXAMPLE 1 – EVENT FOR A SINGLE NETWORK VARIABLE:

```
network output int humidity;
...
humidity = 32;
...
when (nv_update_fails(humidity))
{
   ...
}
```

EXAMPLE 2 – EVENT FOR A NETWORK VARIABLE ARRAY:

```
network output int humidity[4];
...
humidity[1] = 32;
...
when (nv_update_fails(humidity))
{
   ...
}
```

EXAMPLE 3 – EVENT FOR A RANGE OF NETWORK VARIABLES:

```
network output int humidity1, humidity2, humidity3;
...
humidity2 = 32;
...
when (nv_update_fails(humidity1 .. humidity3))
{
   ...
}
```

nv_update_occurs EVENT

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). If the parameter is omitted, the event is TRUE for any network variable update.

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 will be checked to see if a value has been received. The event will occur once for each element that receives an update. An individual element may 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**) may 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 will be checked to see if a value has been received. The event will occur once for each network variable in the range that receives an update.

EXAMPLE 1 – EVENT FOR A SINGLE NETWORK VARIABLE:

```
network input boolean switch_state;

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

EXAMPLE 2 – EVENT FOR A NETWORK VARIABLE ARRAY:

```
network input boolean switch_state[4];

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

EXAMPLE 3 – EVENT FOR A RANGE OF NETWORK VARIABLES:

```
network input boolean switch1, switch2, switch3;

when (nv_update_occurs(switch1 .. switch3))
{
   ...
}
```

nv_update_succeeds EVENT

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). If the parameter is omitted, the event is TRUE when any network variable update succeeds.

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 Table 6-2 in the *Neuron C Programmer's Guide* for the definition of success).

If an array name is used, then each element of the array will be checked for success. The event will occur once for each element that experiences a success completion event. (An individual element may be checked with use of 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 will be checked to see if a value has been received. The event will occur 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.

EXAMPLE 1 – EVENT FOR A SINGLE NETWORK VARIABLE:

```
network output int humidity;
...
humidity = 32;
...
when (nv_update_succeeds(humidity))
{
   ...
}
```

EXAMPLE 2 – EVENT FOR A NETWORK VARIABLE ARRAY:

```
network output int humidity[4];
...
humidity[1] = 32;
...
when (nv_update_succeeds(humidity))
{
   ...
}
```

EXAMPLE 3 – EVENT FOR A RANGE OF NETWORK VARIABLES:

```
network output int humidity1, humidity2, humidity3;
...
humidity2 = 32;
...
when (nv_update_succeeds(humidity1 .. humidity3))
{
   ...
}
```

offline EVENT

offline

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** event can be used to place a device offline 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. Once offline, a device responds only to the *Reset* or *Online* messages from a network tool. 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. Configuration properties implemented within a configuration file cannot be updated while a device is offline if the file transfer protocol (FTP) is used as the access method for the configuration file. Configuration properties implemented within a configuration file can be updated while a device is offline if the direct memory read/write access method is used.

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.

EXAMPLE:

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


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

online — EVENT

online

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.

EXAMPLE:

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


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

reset EVENT

reset

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.

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 error 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.

EXAMPLE:

```
when (reset)
{
   // poll state of all inputs
}
```

resp_arrives EVENT

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.

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

EXAMPLE:

```
msg_tag tag_out;
...
msg_out.tag = tag_out;
msg_out.service = REQUEST;
msg_send();
...
when (resp_arrives(tag_out))
{
   ...
}
```

timer_expires

EVENT

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.

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 will remain TRUE as long as any timer object has expired. This event can be cleared only by checking for specific timer expiration events.

EXAMPLE:

```
mtimer countdown;
...
countdown = 100;
...
when (timer_expires(countdown))
{
   ...
}
```

wink

EVENT

wink

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.

EXAMPLE:

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

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.

Compiler Directives

ANSI C permits compiler extensions through the **#pragma** directive. These directives are implementation-specific. The ANSI standard states that a compiler may define any sort of language extensions through the use of these directives. Unknown directives may 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.

List of Directives

The following directives are defined in Neuron C Version 2.1:

#pragma all_bufs_offchip

This pragma is only used with the LonBuilder 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, since these are the only parts that support off-chip memory. See the *LonBuilder Microprocessor Interface Program (MIP) User's Guide* for more information.

#pragma allow_duplicate_events

This directive causes the compiler to issue an NCC#176 duplicate event message as a warning instead of 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 prevents 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*.

#pragma app_buf_in_count *count*

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 app_buf_in_size *size*

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 app_buf_out_count *count*

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 app_buf_out_priority_count *count*

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 app_buf_out_size *size*

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 codegen *option*

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

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
use_i2c_version_1

Some of these options are provided for compatibility with prior releases of the Neuron C Compiler and LonBuilder releases prior to Release 3. 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.

Although unlikely, it is possible that a program which compiled and linked for a Neuron 3120® Chip in LonBuilder releases prior to Release 3 would not fit if compiled under Release 3 or later, since some of the compiler optimizations introduced since then may, under certain circumstances, cause an increase in code size.

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* may achieve additional compaction beyond what is normally provided by automatic merging of whatever CP family members happen to be adjacent in the files. To enable this re-ordering feature, specify **#pragma codegen cp_family_space_optimization** in your program. 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. You cannot use both the **cp_family_space_optimization** option and the **no_cp_template_compression** option in the same application program.

WARNING: Configuration property re-ordering and merging may reduce the memory required for the template file, but may also result in slower access to the application's configuration properties by network tools. This may 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 **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 will be generated with no initial value, such that the Neuron loader will not load anything into the block of memory; instead, the contents prior to load will be 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, 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 insure that each statement contains at least two bytes of code by inserting a NOP instruction if necessary.

The automatic configuration property merging feature in NodeBuilder 3.1 may 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 may 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 **optimization_off** and **optimization_on** options discussed below.

The **noptropt** option may be desirable when debugging a program, since 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 may then use in subsequent statements. Use of this pragma will avoid the problem discussed above, but may also cause a substantial performance or size degradation in the generated code. This codegen option should not be used except while debugging.

The **optimization_off** and **optimization_on** options control optimization of generated executable code. Normally, optimization is on. To prevent the compiler's code optimizer from collapsing two or more statements together, and thus making it difficult to place breakpoints in a program being debugged, this option can be used to disable all compiler optimization. This option may also be useful if an optimization problem is suspected, as code can be generated without optimization, and its behavior compared.

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 may 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 may 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 **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, since 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 Chapter 8, *I/O Objects*.

#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 debugger is not part of the LonBuilder software and is not required to use the LonBuilder Neuron C debugger on a Neuron Emulator. 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 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.

#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 pragma, is slightly more efficient in memory usage, but may 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 shoehorn a program into a Neuron 3120xx Chip or 3120 Smart Transceiver. See the discussion on *What to Try When a Program Doesn't Fit on a 3120* in Chapter 8, *Memory Management,* of the *Neuron C Programmer's Guide.*

#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. Do not use this directive with a LonBuilder Neuron Emulator.

#pragma disable_snvt_si

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

#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 as attempts to write EEPROM as a result of wild jumps will fail. EEPROM variables are not protected. See the discussion of the **set_eeprom_lock( )** function in Chapter 3, *Functions* 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 will fail to change the configuration.

#pragma enable_io_pullups

Enables the internal pullup resistors on pins IO.4 through IO.7. 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, IO.11. On these models of Smart Transceiver, this directive also enables a pullup resistor for the IO.11 pin.

#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 (*e.g.* **io_in( )**, **io_out( )**).

#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. This pragma may only appear once in the source program. See *Standard Network Variable Types (SNVTs)* in Chapter 3, *How Devices Communicate Using Network Variables,* of the *Neuron C Programmer's Guide.*

#pragma explicit_addressing_off
#pragma explicit_addressing_on

These pragmas are only used with the Microprocessor Interface Program (MIP). See the *LONWORKS Microprocessor Interface Program (MIP) User's Guide* for more information.

#pragma fyi_off
#pragma fyi_on

Controls the compiler's printing of informational messages. Informational messages are less severe than warnings, yet may 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 may 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 MIPs. 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* for more information.

#pragma ignore_notused *symbol*

Requests the compiler to ignore the symbol-not-referenced flag for the named symbol. The compiler normally prints warning messages for any variables, functions, I/O objects, etc. 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 close 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 will always copy the contents such that the assembly code will not interfere with code being generated by the compiler. Echelon does not document or support the direct use of the Neuron assembler with user-written assembly code.

#pragma micro_interface

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

#pragma names_compatible

This pragma is useful in Neuron C Version 2 to force the compiler to treat names starting with SCPT*, UNVT*, UCPT*, SFPT*, and UFPT* as normal variable names instead of special symbols to be resolved via 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.

#pragma net_buf_in_count *count*

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.

#pragma net_buf_in_size *size*

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 net_buf_out_count *count*

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 net_buf_out_priority_count *count*

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 net_buf_out_size *size*

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.

#pragma no_hidden

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

#pragma num_addr_table_entries *num*

Sets the number of address table entries to *num*. Valid values for *num* are 0 to 15. The default number of address table entries is 15. You can use this pragma to trade EEPROM space for address table entries (see Chapter 8, *Memory Management,* of the *Neuron C Programmer's Guide*).

#pragma num_alias_table_entries *num*

Controls the number of alias table entries allocated by the compiler. This number must be chosen at compile time, it cannot be altered at run time. Valid values for *num* are 0 to 62. In Neuron C Version 2.1, there is no compiler default for this value. A Neuron C program **must** specify a value using this pragma. You can use this pragma to trade EEPROM space for alias table entries (see Chapter 8, *Memory Management,* of the *Neuron C Programmer's Guide*).

#pragma num_domain_entries *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*).

#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 **num_domain_entries** pragma instead. The default number of domain table entries is 2.

#pragma ram_test_off

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.

#pragma read_write_protect

Allows a device's program to be read and write protected to prevent copying or alteration via the network. This feature provides protection of a manufacturer's confidential algorithms. A device cannot be reloaded once 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. Once a device has been 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. For more information on the use of the EEPROM blanking programs, see the Smart Transceivers databooks.

#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.

#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 may 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 may 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 run_unconfigured

This pragma causes the application to run regardless of the device state, as long as the device is not applicationless. This means that even if the device is unconfigured or hard-offline, the application will run. You can use this directive to have an application perform some form of local control prior to or independent of being installed in a network.

Applications that use this pragma and will be 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 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 will not ever run when hard-offline and thus will not be vulnerable to this condition.

#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).

#pragma set_guidelines_version *string*

The Neuron C 2.1 compiler will generate LONMARK information in the device's XIF file and in the device's SIDATA (stored in device program memory). By default, the compiler uses **"3.3"** 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 may 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 will not change the syntax of SD strings generated.

#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 (e.g. 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 may 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). See the *LONWORKS Microprocessor Interface Program (MIP) User's Guide* for more information.

#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 2.1 compiler. This first part may be followed by a comment string that documents the purpose of the device. This comment string defaults to a `NULL` string and may 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 may 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. 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.

#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 *may* place the table in EEPROM or in ROM code space, as it determines. Placing this table in EEPROM ensures that it may be modified via *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 or 3150 Smart Transceiver.

#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 *may* place the table in EEPROM or in ROM code space, as it determines. Placing this table in RAM ensures that it may be modified via *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 or 3150 Smart Transceiver. See **#pragma snvt_si_eecode** for an example of usage.

#pragma specify_io_clock *string*

Specify this directive to inform the compiler of the value of the Neuron input clock speed. 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 "10 MHz", "6.5536 MHz", "5 MHz", and "2.5 MHz". The strings must appear exactly as shown, including spacing and capitalization.

#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 may continue to access the **nv_len** property as discussed in the *Neuron C Programmer's Guide*. However, the Neuron C Version 2.1 system image extension technique provides a more robust implementation and should therefore be used for all new designs.

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 get_nv_length_override(unsigned nvIndex);

The function returns the current length in bytes of the network variable with the given index, or 0xFF to signal that the length has not been changed. 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.

#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. The newer version of transaction tracking (the *on* option) is used by default when available, unless the device is a LONWORKS network interface (e.g. running the MIP), or the device's application program generates explicit destination addresses.

#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 into a warning. This 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 uses this directive to generate Neuron C source code that will compile, 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.

#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 may be intermixed multiple times throughout a program to turn warning message printing on and off as desired.

3
Functions

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

Introduction

The following pages list 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 may 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 listed below:

- <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> (automatically included)
- <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(s) 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 may require more EEPROM memory on a Neuron 3120 Chip or 3120 Smart Transceiver than on a Neuron 3150 Chip or 3150 Smart Transceiver. This is because more of the system functions are stored in the ROM firmware image on a Neuron 3150 Chip or a 3150 Smart Transceiver. 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 *LonBuilder User's Guide* and *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 and the PL 3120 Smart Transceiver.

Overview of Neuron C Functions

You can call the functions listed in the following tables 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.

Execution Control

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
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
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

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
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
update_program_id()	Update the device's program ID

Integer Math

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
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
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
s32_to_ulong()	Convert a 32-bit signed number into unsigned long
swap_bytes()	Exchange the two bytes of a 16-bit number

Floating-point Math

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

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

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
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**
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

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
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 signed 32-bit integers. However, there is a complete library of functions for 32-bit integer match. These functions are listed under *Integer Math* in the previous section. For example, in standard ANSI C, to evaluate **X = A + B * C** in long (32-bit) arithmetic, the '**+**' and '*****' infix operators may 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 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 may be assigned to each other with the '**=**' 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 may 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);
}
```

Since this function is fairly time consuming, it may 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* below.

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

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

then the output file will contain 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 LonBuilder's 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 you wish to display, and **Format** as **Dec**. The data will be displayed as unsigned, even if it is negative. To view the data as signed, click on the value field, and the Modify Variable window will show 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 may be reused as output arguments to facilitate operations in place.

Binary Arithmetic Operators

```
void s32_add(const s32_type *arg1, const s32_type *arg2,
     s32_type *arg3);
```

Adds two signed 32-bit integers. **(arg3 = arg1 + arg2)**

```
void s32_sub(const s32_type *arg1, const s32_type *arg2,
     s32_type *arg3);
```

Subtracts two signed 32-bit integers. **(arg3 = arg1 - arg2)**

```
void s32_mul(const s32_type *arg1, const s32_type *arg2,
     s32_type *arg3);
```

Multiplies two signed 32-bit integers. **(arg3 = arg1 *arg2)**

```
void s32_div(const s32_type *arg1, const s32_type *arg2,
     s32_type *arg3);
```

Divides two signed 32-bit integers. **(arg3 = arg1 / arg2)**

```
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.

```
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))**.

```
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

```
void s32_abs(const s32_type *arg1, s32_type *arg2);
```

Returns the absolute value of a signed 32-bit integer. **(arg2 = abs(arg1))**

```
void s32_neg(const s32_type *arg1, s32_type *arg2);
```

Returns the negative of a signed 32-bit integer. **(arg2 = - arg1)**

Comparison Operators

```
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)**

```
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)**

```
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)**

```
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)**

```
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)**

```
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)**

```
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

```
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.

```
void s32_inc(s32_type *arg);
```

Increments a signed 32-bit integer.

```
void s32_dec(s32_type *arg);
```

Decrements a signed 32-bit integer.

```
void s32_mul2(s32_type *arg);
```

Multiplies a signed 32-bit integer by two.

```
void s32_div2( s32_type *arg );
```

Divides a signed 32-bit integer by two.

```
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

```
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.

```
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.

```
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.

```
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

```
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

```
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

The following numbers are times in milliseconds for the various 32-bit functions. They were measured using a Neuron Chip with a 10MHz 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].

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
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 under *Floating-point Math* in the earlier section *Overview of Neuron C Functions*. For example, in standard ANSI C, to evaluate X = A + B * C in floating-point, the '+' and '*' infix operators may 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^{38} to +1*10^{38}, with a relative resolution of approximately ±1*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 may be assigned to each other with the '**=**' operator.

An **fl_error** global variable stores the last error detected by the floating-point functions. If error detection is desired in 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** may 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 may 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
}
```

Since this function is fairly time consuming, it may 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 the *Using the NXT Neuron C Extended Arithmetic Translator* section below.

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 will contain 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 may not be cast, but pointers to structures can. The following example shows how a local variable of type **float_type** may 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 may 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 LonBuilder 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 you wish 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 may match any of the input arguments to facilitate operations in place.

Binary Arithmetic Operators

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

Adds two floating-point numbers. (**arg3 = arg1 + arg2**)

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

Subtracts two floating-point numbers. (**arg3 = arg1 - arg2**)

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

Multiplies two floating-point numbers. (**arg3 = arg1 *arg2**)

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

Divides two floating-point numbers. (**arg3 = arg1 / arg2**)

```
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)**)

```
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

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

Returns the absolute value of a floating-point number.
(**arg2 = abs(arg1)**)

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

Returns the negative of a floating-point number. (**arg2 = - arg1**)

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

Returns the square root of a floating-point number.
(**arg2 = √ arg1**)

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

Returns the whole number part of a floating-point number. Truncation is towards zero. (**arg2 = trunc(arg1)**) For example, **trunc(-3.45) = -3.0**

```
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. Truncation is towards minus infinity. (**arg2 = floor(arg1)**)
For example, **floor(-3.45) = -4.0**

```
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. Truncation is towards plus infinity.
(**arg2 = ceil(arg1)**) For example, **ceil(-3.45) = -3.0**

```
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**

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

Multiplies a floating-point number by two.
(**arg2 = arg1 * 2.0**)

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

Divides a floating-point number by two.
(**arg2 = arg1 / 2.0**)

Comparison Operators

```
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**)

```
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**)

```
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**)

```
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**)

```
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**)

```
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**)

```
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, and 0 if it is equal.

Miscellaneous Floating-point Functions

```
int fl_sign(const float_type *arg);
```

Sign function, returns +1 if the argument is positive, 0 if the argument is zero, and -1 if the argument is negative.

```
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

```
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()**.

```
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()**.

```
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()**.

```
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.

```
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.

```
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

```
void fl_to_ascii(const float_type *arg1, char *arg2, int decimals,
unsigned buf_size);
```

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 will include a trailing decimal point. If decimals is -1, there will be no trailing decimal point. The number is rounded to the specified precision.

Example: Converting the number -12.34567, with a buf_size of 10.

decimals	***output string***
5	-12.34567
4	-12.3457
3	-12.346
2	-12.35
1	-12.3
0	-12.
-1	-12

```
void fl_to_ascii_fmt(const float_type *arg1, char *arg2, int decimals,
unsigned buf_size, format_type format);
```

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 may 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.

Conversion of ASCII String to Floating-point

```
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

The following numbers are times in milliseconds for the various functions in the floating-point library. They were measured using a Neuron Chip with a 10MHz 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.

Function	***Maximum***	***Average***
Add	0.56	0.36
Subtract	0.71	0.5
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 **struct**s and **typedef**s, the translator will be unable to 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

abs( ) BUILT-IN FUNCTION

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

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**.

EXAMPLE:

```
int i;
long l;

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

access_address( ) FUNCTION

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

The **access_address()** function returns a **const** pointer to the address structure which 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 ANSI/EIA/CEA-709.1 *Control Network Specification* for a description of the data structure.

EXAMPLE:

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

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

access_alias( ) FUNCTION

#include <access.h>
const alias_struct *access_alias (int *index*);

The **access_alias( )** function returns a **const** pointer to the alias structure which 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 ANSI/EIA/CEA-709.1 *Control Network Specification* for a description of the data structure.

EXAMPLE:

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

access_domain( ) FUNCTION

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

The **access_domain( )** function returns a **const** pointer to the domain structure which 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 ANSI/EIA/CEA-709.1 *Control Network Specification* for a description of the data structure.

EXAMPLE:

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

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

access_nv( ) FUNCTION

#include <access.h>
const nv_struct *access_nv (int *index***);**

The **access_nv()** function returns a **const** pointer to the network variable configuration structure which 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 ANSI/EIA/CEA-709.1 *Control Network Specification* for a description of the data structure.

EXAMPLE:

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

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

addr_table_index( ) BUILT-IN FUNCTION

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

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 will not permit this function to be used for a non-bindable message tag (i.e. a message tag declared with the **bind_info(nonbind)** option).

EXAMPLE:

```
int mt_index;
msg_tag my_mt;

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

ansi_memcpy( ) FUNCTION

#include <mem.h>
void *ansi_memcpy (void ***dest*, void ***src***, unsigned long** *len***);**

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()**.

EXAMPLE:

```
#include <mem.h>

unsigned buf[40];
unsigned *p;

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

ansi_memset( ) FUNCTION

#include <mem.h>
void *ansi_memset (void ****p*, int** *c***, unsigned long** *len***);**

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()**.

EXAMPLE:

```
#include <mem.h>

unsigned target[20];
unsigned *p;

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

application_restart( ) FUNCTION

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

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

EXAMPLE:

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

bcd2bin( )

unsigned long bcd2bin (struct bcd ****a*);**

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

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.

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( )

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

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

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

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

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

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

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

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

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.

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

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

The **crc8( )** function iteratively calculates an 8-bit CRC (cyclic redundancy check) over an array of data. This function is useful in conjunction with the support for Touch I/O object, but can also be used for any purposes where a CRC is needed.

EXAMPLE:

```
#include <stdlib.h>

unsigned data[SIZE];

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

crc16( )

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

The **crc16( )** function iteratively calculates a 16-bit CRC (cyclic redundancy check) over an array of data bytes. This function is useful in conjunction with the support for Touch I/O object, but can also be used for any purposes where a CRC is needed.

EXAMPLE:

```
#include <stdlib.h>

unsigned data[SIZE];

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

delay( ) FUNCTION

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 below).

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.

The formulas for determining the duration of the delay are listed in the following table:

Input Clock	***Delay in microseconds***
40 MHz	0.15*(max(1,min(65535,count*4))*42+179)
20 MHz	0.3*(max(1,min(65535,count*2))*42+159)
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, with a 10MHz 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 (at 10MHz) may 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, there is the potential for a watchdog timeout at 20MHz and 40MHz operation. The maximum inputs to **delay()** are 16666 at 20MHz and 8333 at 40MHz. Timing intervals greater than the watchdog interval must be done via software timers or via a user routine that calls **delay()** and **watchdog_update()** in a loop. Also see **msec_delay()**.

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

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

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()**.

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 will still occur 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

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

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

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 LonBuilder and NodeBuilder Neuron C debuggers maintain a log of the last 25 error messages. On a Neuron Emulator, the Neuron firmware adds a delay of up to 70ms between writes to the error log to give the PC time to retrieve the last value.

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

EXAMPLE:

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

fblock_director( ) BUILT-IN FUNCTION

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

index	A decimal number between 0 and 62 representing a functional block global index.
cmd	A decimal number between –128 and 127, interpreted as an application-specific command.

The **fblock_director()** built-in function is a special compiler function that 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, and passes the *cmd* parameter on to that director function.

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* earlier in this chapter.

flush( ) FUNCTION

#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.

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.

EXAMPLE:

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

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

flush_cancel( ) FUNCTION

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

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

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

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

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**, **nv_update_fails**) and pending response events (for example, **resp_arrives**, **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.

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

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

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

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

EXAMPLE:

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

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

The use of the **sizeof()** operator is recommended 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, owing to its efficiency.

get_fblock_count( ) BUILT-IN FUNCTION

unsigned int get_fblock_count (void);

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.

EXAMPLE:

```
unsigned numFBs;

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

get_nv_count( ) BUILT-IN FUNCTION

unsigned int get_nv_count (void);

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.

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

unsigned int get_tick_count (void);

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 50ms at 40MHz. The tick interval scales with the input clock.

EXAMPLE:

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

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

go_offline( ) FUNCTION

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

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.

EXAMPLE:

```
boolean nonrecoverable;
...
when (nonrecoverable)
{
   go_offline();
}

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

go_unconfigured( ) FUNCTION

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

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

EXAMPLE:

```
void f(void)
{
      if (
            (io_in(io_fast_for)==PUSHED) &&
            (io_in(io_set_time)==PUSHED) &&
            (io_in(io_chan_sel_9)==PUSHED))
      // erase network configuration info from this device
            go_unconfigured();

}
```

high_byte( ) BUILT-IN FUNCTION

unsigned short high_byte (unsigned long *a*);

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()**.

EXAMPLE:

```
short b;
long a;

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

io_change_init( ) BUILT-IN FUNCTION

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 may be **short** or **long** as needed.

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.

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

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

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

The **io_edgelog_preload()** built-in function is optionally used with the edgelog I/O object. The *value* parameter defines the maximum value, in units of the clock period, for each period measurement, and may 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 may cause an overflow, but this is normal.

EXAMPLE:

```
IO_4 input edgelog elog;

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

}
```

io_edgelog_single_preload( ) BUILT-IN FUNCTION

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

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

The **io_edgelog_single_preload()** built-in function is optionally used with the edgelog I/O object 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 may 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 may cause an overflow, but this is normal.

EXAMPLE:

```
IO_4 input edgelog single_tc elog;

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

io_idis( ) FUNCTION

void io_idis (void);

The **io_idis()** function disables the I/O interrupt used in the hardware support for the **sci** and **spi** I/O objects. 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.

EXAMPLE:

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

io_iena( ) FUNCTION

void io_iena (void);

The **io_iena()** function enables the I/O interrupt used in the hardware support for the **sci** and **spi** I/O objects. 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.

EXAMPLE:

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

io_in( )

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.

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**

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

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

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

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

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 will be returned. The unused bits are 0 if fewer than 16 bits are shifted in.

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

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.

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

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

buf	A (**void ***) pointer to a buffer.
addr	An **unsigned** short int I^2C device address.
count	The number of bytes to be transferred.

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

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

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 object description later in this chapter for more information.
v2	The threshold value (an **unsigned long**). See the infrared I/O object description later in this chapter for more information.

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

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

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.

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

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 will either be identical to the count argument, or a smaller number that indicates a timeout event occurred during the read.

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

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.

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

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

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

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

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 no. 005-0014-01) for more information.

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

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

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

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

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).

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

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.

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

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

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

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

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( )

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 objects.
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 for more information.

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

EXAMPLE:

```
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:

```
#pragma specify_io_clock "10 MHz"
IO_8 sci baud(SCI_2400) iosci;
unsigned short buf[20];

when (...)
{
      io_in_request(iosci, buf, 20);
}
```

io_out( )

void io_out (*output-io-object-name***,** *output-value* [**,** *args*]**);**

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.

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**
touch	(see below)
triac output	**unsigned long**
triggeredcount output	**unsigned long**

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

io_out (*output-obj*, *output-value***);**

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

For **bitshift** output objects, the syntax is the following:

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.

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

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

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).

For **infrared_pattern** output objects, the syntax is the following:

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 object in Chapter 8 for a detailed explanation of this restriction.

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

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

addr	An optional address to write (from 0 to 255). Omission of the address will cause 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.

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

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.

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

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

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

For **serial** output objects, the syntax is the following:

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).

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

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.

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

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

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.

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.

EXAMPLE:

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

The **io_out_request()** is used with a **sci** I/O object to start the serial transfer.

EXAMPLE:

```
#pragma specify_io_clock "10 MHz"
IO_8 sci baud(SCI_2400) iosci;
unsigned short buf[20];

when (...)
{
      io_out_request(iosci, buf, 20);
}
```

io_preserve_input( ) BUILT-IN FUNCTION

void io_preserve_input (*input-io-object-name***);**

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 only applicable to input timer/counter I/O objects.

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 will discard 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 since the data may 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, will override 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.

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

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.

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.

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( )

void io_set_baud (*io-object-name***,** *baud-rate***);**

io-object-name	The I/O object name, which 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.

The compiler directive **#pragma specify_io_clock** must be included to specify the Neuron input clock rate.

The **io_set_baud()** built-in function allows an application to optionally change the baud rate for a SCI device. The SCI device optionally has an initial bit rate setting from its declaration.

EXAMPLE:

```
#pragma specify_io_clock "10 MHz"
IO_8 sci baud(SCI_2400) iosci;

when (...)
{
      io_set_baud(iosci, SCI_38400);
      // Optional baud change
}
```

io_set_clock( )

BUILT-IN FUNCTION

void io_set_clock (*io-object-name***,** *clock-value***);**

io-object-name	The I/O object name, which 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	An optional clock, in the range 0 to 7, or a variable name for the clock. This value overrides a clock value specified in the object's declaration.

The **io_set_clock( )** built-in function allows an application to specify an alternate clock value for any input or output timer/counter object which permits a clock argument in its declaration syntax. The objects are listed below:

dualslope
edgelog
frequency
infrared
oneshot
ontime
period
pulsecount
pulsewidth
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.

EXAMPLE:

```
IO_1 output pulsecount clock(3)pcout;

when(...)
{
   io_set_clock(pcout, 5);
   ...
}
```

io_set_direction( ) BUILT-IN FUNCTION

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, which 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. Optional, if omitted, uses the declared direction of the I/O device to set the pin direction.

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 will be changed as specified.

In order 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 may trigger when the object is redirected as an output. This is because the Neuron firmware returns the last value output on an output object as the input value. Thus, the user may wish to qualify **io_changes** events with flags maintained by the program indicating the current direction of the device.

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);
}
```

is_bound( ) BUILT-IN FUNCTION

boolean is_bound (*net-object-name***);**

net-object-name	Either a network variable name or a message tag.

The **is_bound()** built-in function indicates whether the specified network variable or message tag is connected. The function returns TRUE if the network variable or message tag is connected, otherwise it returns FALSE. This function can be used to ensure that transactions are initiated only for connected network variables and message tags.

When an unconnected network variable is updated or a message is sent out on an unconnected message tag, success completion events are generated, even though no actual network communication takes place. In this instance, even if the unconnected message is a request and no response is received, the **message_succeeds** and **message_completes** events will be TRUE. Similarly, if an unconnected network variable is polled, no network variable update will occur, although the **nv_update_succeeds** event will be TRUE.

To avoid processing events for unconnected objects, the program can call **is_bound()** first to ensure that the network variable or message tag is actually connected before sending an update. In most cases, a program can simply ignore the fact that network variables and message tags are unconnected.

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.

EXAMPLE:

```
network input SNVT_color nviColor;
...
void f(void)
{
      // Poll temp if it is bound
      if (is_bound(nviColor)) {
            poll(nviColor);
      }
}
```

low_byte( ) BUILT-IN FUNCTION

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

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()**.

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

unsigned long make_long (unsigned short *low-byte***,**
unsigned short *high-byte***);**

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()**.

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( )

type **max (***a*, *b***);**

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 below.

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.

EXAMPLE:

```
int a, b, c;
long x, y, z;

void f(void)
{
      a = max(b, c);
      x = max(y, z);
}
```

The above discussion about the **max()** function result types and type promotion of arguments also applies equally to the **min()** function.

memccpy( ) FUNCTION

#include <mem.h>
int memccpy (void ****dest*, const void ****src***, int** *c***, unsigned long** *len***);**

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( )**.

EXAMPLE:

```
#include <mem.h>

unsigned array1[40], array2[40];

void f(void)
{
      // Copy up to 40 bytes from array2 to array1,
      // but stop if a 0xFF value is copied.
      unsigned *p;
      p = memccpy(array1, array2, 0xFF, 40);
}
```

memchr( ) FUNCTION

#include <mem.h>
void *memchr (const void ****buf*, int** *c***, unsigned long** *len***);**

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( )**.

EXAMPLE:

```
#include <mem.h>

unsigned array[40];

void f(void)
{
      unsigned *p;

      // Find the first 0xFF byte, if it exists
      p = memchr(array, 0xFF, 40);
}
```

memcmp( ) FUNCTION

#include <mem.h>
int memcmp (void ****buf1*, const void ****buf2***, unsigned long** *len***);**

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()**.

EXAMPLE:

```
#include <mem.h>

unsigned array1[40], array2[40];

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

memcpy( ) BUILT-IN FUNCTION

void memcpy (void ****dest*, void ****src***, unsigned long** *len***);**

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, as 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()**.

EXAMPLE:

```
void f(void)
{
      memcpy(msg_out.data, "Hello World", 11);
}
```

memset( ) BUILT-IN FUNCTION

void memset (void ****p*, int** *c***, unsigned long** *len***);**

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, as 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()**.

EXAMPLE:

```
unsigned target[20];

void f(void)
{
      memset(target, 0, 20);
}
```

min( ) BUILT-IN FUNCTION

type **min (***a***,** *b***);**

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 shown above for **max()**.

EXAMPLE:

```
int a, b, c;
long x, y, z;

void f(void)
{
      a = min(b, c);
      x = min(y, z);
}
```

msec_delay( ) FUNCTION

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

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

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 a easier way to specify millisecond delays than the **delay()** or **scaled_delay()** functions. See **delay()** and **scaled_delay()** for functions that will delay the application program for a longer duration.

EXAMPLE:

```
IO_4 input bit io_push_button;
boolean debounced_button_state;

when (io_changes(io_push_button))
{
   msec_delay(100); // Delay 100ms at any clock rate
   debounced_button_state = (boolean)io_in(io_push_button);
}
```

msg_alloc( ) BUILT-IN FUNCTION

boolean msg_alloc (void);

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.

EXAMPLE:

```
void f(void)
{
      if (msg_alloc()) {
         // OK.  Build and send message
         ...
      }
}
```

msg_alloc_priority( ) BUILT-IN FUNCTION

boolean msg_alloc_priority (void);

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.

EXAMPLE:

```
void f(void)
{
      if (msg_alloc_priority()) {
         // OK.  Build and send message
         ...
      }
}
```

msg_cancel( ) BUILT-IN FUNCTION

void msg_cancel (void);

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.

EXAMPLE:

```
void f(void)
{
      if (msg_alloc()) {
      ...
         if (offline()) {
         // Requested to go offline
            msg_cancel();
         } else {
            msg_send();
         }
      }
}
```

msg_free( ) BUILT-IN FUNCTION

void msg_free (void);

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.

EXAMPLE:

```
void f(void)
{
...
   if (msg_receive()) {
      // Process message
         ...
      msg_free();
   }
...
}
```

msg_receive( )

BUILT-IN FUNCTION

boolean msg_receive (void);

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 may 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 (*i.e.* it *can* be used in 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, such that the **online**, **offline**, and **wink** special events cannot be used. If the program handles any of these, 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.

EXAMPLE:

```
void f(void)
{
...
   if (msg_receive()){
      // Process message
         ...
      msg_free();
   }
...
}
```

msg_send( )

BUILT-IN FUNCTION

void msg_send (void);

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.

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

#include <stdlib.h>
unsigned long muldiv (unsigned long *A*, unsigned long *B*,
unsigned long *C*);

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.

EXAMPLE:

```
#include <stdlib.h>
unsigned long a, b, c, d;
...

void f(void)
{
      d = muldiv(a, b, c);   // d = (a*b)/c
}
```

muldiv24( ) FUNCTION

#include <stdlib.h>
unsigned long muldiv24 (unsigned long *A*, unsigned int *B*,
unsigned int *C*);

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 accuracy of the result is improved. 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.

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( )

#include <stdlib.h>
signed long muldiv24s (signed long *A*, signed int *B*, signed int *C*);

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 accuracy of the result is improved. 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.

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

#include <stdlib.h>
signed long muldivs (signed long *A*, signed long *B*, signed long *C*);

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.

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

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

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.

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

int nv_table_index (*netvar-name*);

netvar-name	A network variable name, possibly including an index expression.

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 in the range 0 to 61. The **global_index** property, introduced in Neuron C Version 2, is equivalent to the **nv_table_index()** built-in function.

EXAMPLE:

```
int nv_index;
network output SNVT_lux nvoLux;

void f(void)
{
      nv_index = nv_table_index(nvoLux);
      // Equivalent statement
      nv_index = nvoLux::global_index;
}
```

offline_confirm( ) FUNCTION

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

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 for 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.

EXAMPLE:

```
void f(void)
{
      ...
      if (offline){
            // Perform offline cleanup
            ...
            offline_confirm();
      }
}
```

poll( ) BUILT-IN FUNCTION

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.

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 will be polled. An individual element may be polled with use of an array index. When an event expression qualified by an unindexed network variable array name is TRUE, the **nv_array_index** built-in variable (type **short int**) may be examined to obtain the element's index to which the event applies. The network variable does not need to be declared as **polled**. 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 will be sent in response to the poll. The polling device cannot assume that all updates will be 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 may consume an address table entry when it is bound to any output network variables.

The device interface file must identify all polled network variables. This occurs automatically, however, any existing device templates must be updated if **poll()** calls are added or deleted from an application.

The **poll()** function may be used to obtain the initial values for input network variables after a device returns to an online state. This may cause excessive network traffic after a power outage for large networks. An alternative approach is to use heartbeat timers. For example, the **SCPTmaxSndT** configuration property can be used to provide a configurable heartbeat interval for an output network variable. Another alternative is to implement a random delay before an initial poll.

EXAMPLE:

```
network input SNVT_privacyzone nviZone;
...
// Poll temp if it is bound
if (is_bound(nviZone)) {
   poll(nviZone);
}
...
when (nv_update_occurs(nviZone))
{
   // New value of temp arrived
}
```

post_events( ) FUNCTION

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

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.

EXAMPLE:

```
boolean still_processing;
...
void f(void)
{
      while (still_processing) {
            post_events();
            ...
      }
}
```

power_up( ) FUNCTION

#include <status.h>
boolean power_up (void);

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)** clause becomes TRUE. This function can be used by the application to determine whether the start-up resulted from a power-up or not.

EXAMPLE:

```
when (reset)
{
   if (power_up())
      initialize_hardware();
   else {
      // hardware already initialized
      ...
   }
}
```

preemption_mode( ) FUNCTION

boolean preemption_mode (void);

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.*

EXAMPLE:

```
void f(void)
{
      if (preemption_mode()) {
            // Take some appropriate action
            ...
      }
}
```

propagate( )

BUILT-IN FUNCTION

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.

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 (i.e. connected) output network variable can be propagated at any time. If an array name is used, then each element of the array will be propagated. An individual element may be propagated with use of 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 via 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.

EXAMPLE 1:

```
network output const eeprom SNVT_address nvoAddress;

// Propagate nvoAddress on request
when (...)
{
      propagate(nvoAddress);
}
```

EXAMPLE 2:

```
// The pragma permits network variable addresses
// to be passed to functions with non-const pointers,
// with only a warning.
#pragma relaxed_casting_on

typedef struct { ... } struct_type;

network output UNVT_whatever nvoWhatever;

void f(struct_type *p);

when (...)
{
      f(&nvoWhatever);  // Process by address in function f
      propagate(nvoWhatever); // Cause NV to be sent out
}
```

random( ) FUNCTION

unsigned int random (void);

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 will be the same. However, if your device does anything different, based on I/O processing or messages received, or based on data changes, etc, the random number sequence will be different.

EXAMPLE:

```
void f(void)
{
      unsigned value;
      ...
      value = random();
}
```

resp_alloc( ) BUILT-IN FUNCTION

boolean resp_alloc (void);

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.

EXAMPLE:

```
when (...)
{
      if (resp_alloc()) {
            // OK.  Build and send message
            ...
      }
}
```

resp_cancel( ) BUILT-IN FUNCTION

void resp_cancel (void);

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 detailed information.

EXAMPLE:

```
void f(void)
{
      if (resp_alloc()) {
      ...
         if (offline()) {
            // Requested to go offline
            resp_cancel();
         } else {
            resp_send();
         }
      }
}
```

resp_free( ) BUILT-IN FUNCTION

void resp_free (void);

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*.

EXAMPLE:

```
void f(void)
{
...
   if (resp_receive()) {
      // Process message
      ...
      resp_free();
   }
...
}
```

resp_receive( ) BUILT-IN FUNCTION

boolean resp_receive (void);

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 may 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). Since this function defines a critical section boundary, it should never be used in a **when** clause (but it can be used within a task). Using it in a **when** clause 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 detailed information.

EXAMPLE:

```
void f(void)
{
...
   if (resp_receive()) {
      // Process message
         ...
      resp_free();
   }
...
}
```

resp_send( ) BUILT-IN FUNCTION

void resp_send (void);

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 detailed information.

EXAMPLE:

```
# define DATA_REQUEST 0
# define OK 1

when (msg_arrives(DATA_REQUEST)))
{
   int x, y;
   x = msg_in.data(0);
   y = get_response(x);
   resp_out.code = OK;
      // msg_in no longer available
   resp_out.data[0] = y;
   resp_send();
}
```

retrieve_status( ) FUNCTION

#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 at compile time (**#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 thrown away 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 at compile time (**#pragma net_buf_in_count**).

status_reset_cause Identifies the source of the most recent reset. The values for this byte are as follows (*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 as follows:

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 *NodeBuilder 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

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 above.

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

unsigned int reverse (unsigned int *a*);

The **reverse()** built-in function reverses the bits in *a*.

EXAMPLE:

```
void f(void)
{
      int value;
      ...
      value = 0xE3;
      ...
      value = reverse(value);
      // now value is 0xC7
}
```

rotate_long_left( ) FUNCTION

#include <byte.h>
long rotate_long_left (long *arg*, unsigned *count*);

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 which 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()**.

EXAMPLE:

```
#include <byte.h>

void f(void)
{
      long k;

      k = 0x3F00;
      k = rotate_long_left(k, 3);
// k now contains 0xF801
}
```

rotate_long_right( ) FUNCTION

#include <byte.h>
long rotate_long_right (long *arg*, **unsigned** *count***);**

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 which 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()**.

EXAMPLE:

```
#include <byte.h>

void f(void)
{
      long k;

      k = 0x3F04;
      k = rotate_long_right(k, 3);
// k now contains 0x87E0
}
```

rotate_short_left( ) FUNCTION

#include <byte.h>
short rotate_short_left (short *arg*, **unsigned** *count***);**

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 which 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()**.

EXAMPLE:

```
#include <byte.h>

void f(void)
{
      short s;

      s = 0x3F;
      s = rotate_short_left(s, 3);
// s now contains 0xF9
}
```

rotate_short_right( ) FUNCTION

#include <byte.h>
short rotate_short_right (short *arg***, unsigned** *count***);**

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 which 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()**.

EXAMPLE:

```
#include <byte.h>

void f(void)
{
      short s;

      s = 0x3F;
      s = rotate_short_right(s, 3);
// s now contains 0xE7
}
```

scaled_delay( )

FUNCTION

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 below).

The **scaled_delay()** function generates a delay that scales with the input clock for the Neuron Chip or the Smart Transceiver.

In the formula shown below, the scaling factor *S* is determined by the input clock:

0.25 = 40MHz input clock
0.5 = 20MHz input clock
1 = 10MHz input clock
1.5259 = 6.5536 MHz input clock
2 = 5MHz input clock
4 = 2.5MHz input clock
8 = 1.25MHz input clock
16 = 625kHz input clock

The formula for determining the duration of the delay is the following:

delay = (25.2 **count* + 7.2) **S*   (*delay* is in microseconds)

(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.)

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

void sci_abort (void);

The **sci_abort()** built-in function terminates any outstanding SCI I/O operation in progress.

EXAMPLE:

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

sci_get_error( ) BUILT-IN FUNCTION

unsigned short sci_get_error (void);

The **sci_get_error()** built-in function will return a cumulative OR of the bits shown below that reflect data errors. Calling this function will clear the SCI error state.

0x04	Framing error
0x08	Noise detected
0x10	Receive overrun detected

EXAMPLE:

```
unsigned short sci_error_value;

void f(void)
{
      sci_error_value = sci_get_error();
}
```

service_pin_msg_send( ) FUNCTION

#include <control.h>
int service_pin_msg_send (void);

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 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 service pin, but have some other method for an installer to request a service pin message.

EXAMPLE:

```
#include <control.h>

when ( ... )
{
      int tries;

      ...

      for (tries = 3;  tries > 0;  --tries) {
            if (service_pin_msg_send()) break;
      }
}
```

service_pin_state( ) FUNCTION

#include <control.h>
int service_pin_state (void);

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 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 go unconfigured (for ease of re-installation in a new network).

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();
      }
}
```

set_bit( ) FUNCTION

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

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()**.

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

#include <control.h>
void set_eeprom_lock (boolean *lock***);**

The **set_eeprom_lock()** function allows the application to control the state of the EEPROM lock. This feature is only available in Version 6 and later of the Neuron 3150 Chip and FT 3150 Smart Transceiver firmware, and Version 4 and later of the Neuron 3120xx Chip or FT 3120 Smart Transceiver firmware. 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 will 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*.

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)
      ...
}
```

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 s**32_type** ******arg1***);**

The signed 32-bit arithmetic support functions are part of the extended arithmetic library. See *Signed 32-bit Integer Support Functions*, prior to this function directory, for a detailed explanation of the extended arithmetic support functions that are available.

sleep( ) BUILT-IN FUNCTION

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 this 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 via a declared I/O object.

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 above may 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.

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*.)

EXAMPLE:

```
IO_6 input bit wakeup;
...
when (flush_completes)
{
   sleep(COMM_IGNORE + TIMERS_OFF, wakeup);
}
```

spi_abort( ) FUNCTION

void spi_abort (void);

The **spi_abort()** built-in function terminates any outstanding SPI I/O operation in progress.

EXAMPLE:

```
void f(void)
{
      spi_abort();
}
```

spi_get_error( ) FUNCTION

unsigned short spi_get_error (void);

The **spi_get_error()** built-in function returns a cumulative OR of the bits show below that reflect data errors. Calling this function will clear the SPI error state.

0x10	Mode fault occurred
0x20	Receive overrun detected

EXAMPLE:

```
unsigned short spi_error_value;

void f(void)
{
      spi_error_value = spi_get_error();
}
```

strcat( ) FUNCTION

#include <string.h>
char *strcat (char ****dest*, const char ****src***);**

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.

EXAMPLE:

```
#include <string.h>

void f(void)
{
      char buf[40]

      strcpy(buf, "Hello");
      strcat(buf, " World");  // buf contains "Hello World"
      ...
}
```

strchr( ) FUNCTION

#include <string.h>
char *strchr (const char ****s*, char** ***c***);**

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()**.

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

#include <string.h>
int strcmp (const unsigned char ****s1*, const unsigned char ****s2***);**

The **strcmp()** function compares the contents of the *s1* and *s2* strings, up until 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()**.

EXAMPLE:

```
#include <string.h>

void f(void)
{
      int val;
      char s1[20], s2[20];

      val = strcmp(s1, s2);
      if (!val) {
            // Strings are equal
      } else if (val < 0) {
            // String s1 is less than s2
      } else {
            // String s1 is greater than s2
      }
}
```

strcpy( ) FUNCTION

#include <string.h>
char *strcpy (char ****dest*, const char ****src***);**

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 insure that the string will 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 will not automatically propagate the network variable update (see the **propagate()** function).

EXAMPLE:

```
#include <string.h>

void f(void)
{
      char s1[20], s2[20];

      strcpy(s1, "Hello World");
      strcpy(s2, s1);
}
```

strlen( ) FUNCTION

#include <string.h>
unsigned long strlen (const char ****s*);**

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()**.

EXAMPLE:

```
#include <string.h>

void f(void)
{
      unsigned long length;

      length = strlen("Hello, world!");
}
```

strncat( ) FUNCTION

#include <string.h>
char *strncat (char *_dest_, char *_src_, unsigned long _len_);

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.

EXAMPLE:

```
#include <string.h>

void f(void)
{
      char buf[40]

      strncpy(buf, "Hello There", 6);
      strncat(buf, "World News Tonight", 5);
            // buf now contains "Hello World"
}
```

strncmp( ) FUNCTION

#include <string.h>
int strncmp (const unsigned char ****s1*, const unsigned char ****s2***,**
unsigned long ***len*****);**

The **strncmp()** function compares the contents of the *s1* and *s2* strings, up until 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()**.

EXAMPLE:

```
#include <string.h>

void f(void)
{
      int val;
      char s1[20], s2[20];

      val = strncmp(s1, s2, 10);  // Compare first 10 chars
      if (!val) {
            // Strings are equal
      } else if (val < 0) {
            // String s1 is less than s2
      } else {
            // String s1 is greater than s2
      }
}
```

strncpy( ) FUNCTION

#include <string.h>
char *strncpy (char ***dest*, const char ***src***, unsigned long** *len***);**

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.

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.

EXAMPLE:

```
#include <string.h>

char s[20];

void f(char *p)
{
      strncpy(s, p, 19);       // Prevent overflow
      s[19] = '\0';      // Force termination
}
```

strrchr( ) FUNCTION

#include <string.h>
char *strrchr (const char *_s_, char _c_);

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()**.

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

unsigned long swap_bytes (unsigned long *a***);**

The **swap_bytes()** built-in function returns the byte-swapped value of *a*. See also **high_byte()**, **low_byte()**, and **make_long()**.

EXAMPLE:

```
long k;

void f(void)
{
      k = 0x1234L;
      k = swap_bytes(k);      // k now contains 0x3412L
}
```

timers_off( ) FUNCTION

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

The **timers_off()** function turns off all software timers. This function could be called, for example, before an application goes offline.

EXAMPLE:

```
when (...)
{
      timers_off();
      go_offline();
}
```

touch_bit( ) BUILT-IN FUNCTION

unsigned touch_bit(*io-object-name***, unsigned** *write-data***);**

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 will contain 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 will normally contain that same bit value, and can be ignored. This function provides access to the same internal function that **touch_byte()** calls.

EXAMPLE:

```
void f(void)
{
      unsigned dataIn, dataOut;
      ...
      dataOut = 42;
      dataIn = touch_bit(ioObj, dataOut);
}
```

touch_byte( ) BUILT-IN FUNCTION

unsigned touch_byte(*io-object-name***, unsigned** *write-data***);**

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 will contain 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 will normally contain those same bits.

EXAMPLE:

```
void f(void)
{
      unsigned dataIn, dataOut;
      ...
      dataOut = 42;
      dataIn = touch_byte(ioObj, dataOut);
}
```

touch_first( )

BUILT-IN FUNCTION

int touch_first(*io-object-name***, search_data ****sd***);**

The **touch_first()** function executes the ROM Search algorithm as described in *Book of DS19xx Touch Memory Standards*, Dallas Semiconductor, Edition 2.0. 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. Each call to **touch_first()** or **touch_next()** will take 41ms to execute at 10MHz (63ms at 5MHz) when a device is being read.

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( )

int touch_next(*io-object-name***, search_data ****sd***);**

The **touch_next()** function executes the ROM Search algorithm as described in *Book of DS19xx Touch Memory Standards*, Dallas Semiconductor, Edition 2.0. 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. Each call to **touch_first()** or **touch_next()** will take 41ms to execute at 10MHz (63ms at 5MHz) when a device is being read.

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_reset( ) BUILT-IN FUNCTION

int touch_reset (*io-object-name*);

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 500µs. 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 275µs, after which the **touch_reset()** function will return a (-1) value with the assumption that the 1-WIRE bus is stuck low. There also is a minimum value for this period, it must be >4.8µs @10MHz, or 9.6µs @5MHz.

The **touch_reset()** function does not return until the end of the presence pulse has been detected.

EXAMPLE:

```
void f(void)
{
      touch_reset(ioObj);
}
```

tst_bit( ) FUNCTION

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

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()**.

EXAMPLE:

```
#include <byte.h>

unsigned short a[4];

void f(void)
{
      memset(a, 0, 4);  // 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

#include <access.h>
void update_address (const address_struct ****address*, int** *index***);**

The **update_address()** function copies from the structure referenced by the *address* pointer parameter to the address table entry specified by the *index* parameter.

WARNING: 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 ANSI/EIA/CEA-709.1 *Control Network Specification* for a description of the data structure.

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

#include <access.h>
void update_alias (const alias_struct ****alias*, int** ***index*****);**

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.

WARNING: 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 ANSI/EIA/CEA-709.1 *Control Network Specification* for a description of the data structure.

EXAMPLE:

```
#include <access.h>
alias_struct alias_copy;
unsigned int index;

void f(void)
{
      alias_copy = *(access_alias(index));
      // Modify the alias_copy here as necessary
      update_alias(&alias_copy, index);
}
```

update_clone_domain( ) FUNCTION

#include <access.h>
void update_clone_domain (domain_struct ****domain*, int** ***index*****);**

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 which 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 which 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 *LonBuilder User's Guide* or *NodeBuilder User's Guide*.

WARNING: 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 ANSI/EIA/CEA-709.1 *Control Network Specification*.

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

#include <access.h>
void update_config_data (const config_data_struct ****p*);**

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 via this function. The **config_data** variable is automatically defined for every program by the **<echelon.h>** file.

WARNING: 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 ANSI/EIA/CEA-709.1 *Control Network Specification* for a description of the data structure.

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

#include <access.h>
void update_domain (domain_struct ****domain*, int** ***index***);**

The **update_domain()** function copies from the structure referenced by the *domain* pointer parameter to the domain table entry specified by the *index* parameter.

WARNING: 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 ANSI/EIA/CEA-709.1 *Control Network Specification* for a description of the data structure.

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

#include <access.h>
void update_nv (const nv_struct ****nv-entry*, int** *index***);**

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.

WARNING: 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 ANSI/EIA/CEA-709.1 *Control Network Specification* for a description of the data structure.

EXAMPLE:

```
#include <access.h>
nv_struct nv_copy;
network output SNVT_switch nvoSwitch;

void f(void)
{
      nv_copy = *(access_nv(nv_table_index(nvoSwitch)));
      // Modify the nv_copy here as necessary
      update_nv(&nv_copy,nv_table_index(nvoSwitch));
}
```

update_program_id( ) FUNCTION

#include <access.h>
void update_program_id (unsigned char * ***pid_p*****);**

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.

WARNING: 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.

EXAMPLE:

```
#include <access.h>
unsigned char progID_copy[8];

void f(void)
{
      update_program_id(progID_copy);
}
```

watchdog_update( ) FUNCTION

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

The **watchdog_update()** function updates the watchdog timer. The watchdog timer times out in the range of .84 to 1.68 seconds with a 10MHz Neuron input clock. The watchdog timer period scales inversely with the input clock frequency. 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 when 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 may expire, causing the device to reset. To prevent the watchdog timer from expiring, an application program can call the **watchdog_update()** function periodically.

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.

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 started over by assigning a new value to this object. The timer object can be evaluated while the timer is running, and it will indicate the time remaining. Up to 15 timer objects may 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

Configuration Property and Network Variable Declarations

This chapter describes the configuration property and network variable 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.

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 31 bytes each, and a Neuron Chip or Smart Transceiver hosted device is limited to a maximum of 62 network variables.

The second method of implementing configuration properties uses configuration files to implement the configuration properties for a device. Rather than being separate externally-exposed 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.

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 the next section on *Network Variable Declarations*. 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 may 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.

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 may either be a standard configuration property type (SCPT) or a user configuration property type (UCPT). There are over 200 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 may 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 User's Guide*.

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.

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 will *replicate* 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 may be repeated two or more times, and, as long as the duplicated declarations match in every regard, the compiler will treat these as a single declaration.

EXAMPLE 1 – REPEATED FAMILY DECLARATION:

```
SCPTbrightness cp_family cpBrightness;
SCPTbrightness cp_family cpBrightness;
```

In example 1, the compiler will treat the two families as one. One of the two declarations may 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 will treat the two families as two distinct families, owing to 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};
```

This declaration in example 3 causes a compile-time error, owing to 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 will 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 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. Use of this option requires the CP family or configuration property network variable to be declared as **const**.
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 may 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.

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 while the second number is the high limit. If either the high limit or the low limit should be the maximum or minimum specified in the configuration property type definition, then the field is empty to specify this. In the case of a structure or an array, if one member of the structure or array has a range modification, then all members must have a range modification specified. In this case, each range modification pair is delimited by 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 may not be restricted. Instead, the default ranges must be used. In the case of an array, the specified range modifications apply to all elements of the array. For example, to specify a range modification for a 3-member structure where the second member has the default ranges, and the third member only has an upper limit modification, the range modification string is encoded as: **"n:m||:m;"**. Positive values for range modifications and their exponents (if any) are implicit, while negative numbers and negative exponents must be explicitly designated as such with a preceding '**-**' 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 may 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 will be explained in the next section, property lists for network variables will be explained later in this chapter, and property lists for functional blocks will be explained in the chapter on *Functional Block Declarations*.

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 as follows:

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 may be an optional *initializer*, and an optional *range-mod*. These optional elements may 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 may have multiple device property lists. These lists will be 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 will be 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 will cause a compile-time error.

Finally, each property instantiation may have a range modification string following the property identifier. The range modification string works identically to the *range-mod* described above in *Configuration Property Modifiers (cp-modifiers).* 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 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").

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.

Up to 62 network variables (counting each array element as a separate network variable), including configuration network variables, may be declared in a Neuron C program.

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. Used only for output network variables.

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 will occur 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 may have a maximum length of 1023 bytes.

The use of any of the following Neuron C Version 2.1 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 may still be specified with the **sd_string()** modifier. The compiler will append this additional SD information to the compiler-generated SD data, but it will be 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 the variable is also identified as a configuration variable to network tools which access the device's interface information. <u>The **config** keyword is obsolete and is included only for legacy applications</u>. The Neuron C compiler will not generate self-documentation data for **config** class network variables. New applications should use the configuration network variable syntax explained in *Configuration Network Variables* below.

const The network variable is of **const** type. The Neuron C compiler will not allow modifications of **const** type variables by the device's program. However, a **const network input** variable will still be placed in modifiable memory and the value may 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 may cast away the const-ness of the

property to implement device-specific configuration properties as configuration network variables. However, since the network variable will be placed in modifiable memory, network variable connections may 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 is only appropriate for variables which 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 may 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, will be 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 http://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 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 up to 31 bytes long. Structures and unions may not exceed 31 bytes in length when used as the type of a network variable.

  Single-dimension arrays of the above types, up to 62 elements.

  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).

The *cp-modifiers* for configuration network variables are identical to the *cp-modifiers* described in *Configuration Property Modifiers (cp-modifiers)* earlier in this chapter.

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 as follows:

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 may be an optional *initializer*, and an optional *range-mod*. These optional elements may 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 match.

You cannot have more than one configuration property of any given SCPT or UCPT type that applies to the same network variable. A compile-time error will occur when a particular configuration property type is used for more than one property in the network variable's property list.

Finally, each property instantiation may have a range-modification string following the property identifier. The range-modification string works identically to the *range-mod* described above in *Configuration Property Modifiers (cp-modifiers).* 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 may 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 may not, 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 network variable array that is a configuration property may be used in one of two ways. Each element of the array may be treated as a separate configuration property, or all elements of the array may 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 (device properties were explained in an earlier section).

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 *LonBuilder User's Guide* and *NodeBuilder 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, as detailed below.

> **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 the following:

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 will be 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 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 may not 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 may be changed by a network tool.

A network variable connection will be 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 will not be authenticated (although the update will 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 since 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 will be propagated on 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 may 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 may be used by a network tool to perform network load analysis and are optional.

Although any value in the range 0..18780 may 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.

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 may 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 as follows:

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. Since there cannot be two or more properties with the same configuration property type that apply to the same network variable, this means that 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.

Since the same CP family could also be used as a device property, there is a special context defined for the device. The device's context is just 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** will evaluate to the entire configuration property array (the expression will return the address of the array's first element), whereas **nva[4]::xyz[2]** will return 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 via its variable identifier, it can also be accessed equally well 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*.

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.

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 may implement any of the optional network variables and configuration properties defined by the functional profile. A functional block may 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, though the compiler does create some data structures as described in *Related Data Structures* later in this chapter. These data structures are used to accomplish 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 the following:

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 may or may 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 may have an empty member list. Such a functional block is useful as a collection of related configuration properties.

A Neuron C program may 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 has already used.

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 may be provided for each functional block. An external name may 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 may 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 then takes this string and uses it to look up the appropriate string in the resource files that apply to the device. This is provided as a convenience to the programmer, so the compiler will 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 may 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 as follows:

> **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 may be an optional *initializer*, and an optional *range-mod*. These optional elements may 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 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 may have a range modification string following the property identifier. The range modification string works identically to the *range-mod* described above in *Configuration Property Modifiers (cp-modifiers)* in the previous chapter. 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 will obtain 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 may 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 in properties of network variables, you may 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 since 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*, below.

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, appears as follows:

```
extern const unsigned short fblock_index_map[ ];
```

The value for each network variable is set to the global index of the functional block that it is a member of. 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 as follows:

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. Since there cannot be two or more properties with the same SCPT or UCPT type that apply to the same functional block, this 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)
{
      z = muldiv( rawData,
                  MyFb1[0]::cpGain.multiplier,
                  MyFb1[0]::cpGain.divider );
      MyFb1[0]::nvoData = z;
}
```

Just like for network variable properties, even though a configuration network variable can be uniquely accessed via its variable identifier, it can also be accessed equally well through the context expression, just like the CP family members.

Also, the network variable members of the functional block can be accessed through a similar syntax. The syntax for accessing a functional block member is shown below (the *fb-context* syntactical element is defined above):

fb-context **::** *member-identifier*

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 and Objects

This chapter provides reference information on the built-in variables and objects in Neuron C.

Introduction to Built-in Variables and Objects

Neuron C Version 2.1 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 listed below:

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 shown below:

msg_in
msg_out
resp_in
resp_out

The following sections describe these built-in elements.

Built-in Variables

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 may 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 may read this structure, but not write it, using the **config_data** global declaration. The structure is 25 bytes long, and it may 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 ANSI/EIA/CEA-709.1 *Control Network Specification*. To write this structure, use the **update_config_data()** function described in Chapter 3, *Functions*.

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, *Configuration Property and Network Variable Declarations,* 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, *Configuration Property and Network Variable Declarations,* 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, *Configuration Property and Network Variable Declarations,* 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, *Configuration Property and Network Variable Declarations,* 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, *Configuration Property and Network Variable Declarations,* 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, *Configuration Property and Network Variable Declarations,* 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,* 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:

```
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:

```
unsigned long last_count;
IO_7 input pulsecount count;

when (io_update_occurs(count))
{
      save_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 may 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 may 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 following is a detailed explanation of the various fields of the network variable input address structure:

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** will be 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** will be 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 may be accessed using either the **global_index** property, or using the **nv_table_index()** function.

read_only_data VARIABLE
read_only_data2 VARIABLE

The **read_only_data** and **read_only_data2** 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. 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 {
      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;

const read_only_data_struct read_only_data;

typedef struct {
      unsigned                           : 2;
      unsigned    alias_count            : 6;
      unsigned    msg_tag_count          : 4;
      unsigned                           : 4;
      unsigned    reserved2 [3];
} read_only_data_struct_2;

const read_only_data_struct_2 read_only_data_2;
```

The application program may read these structures, but not write them, using **read_only_data** and **read_only_data_2**. The first structure is 36 bytes long, and it may 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 five bytes long. 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 ANSI/EIA/CEA-709.1 *Control Network Specification.*

Built-in Objects

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 following is a detailed explanation of the various fields of the **msg_in** object:

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 a more detailed description of 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 following is a detailed explanation of the various fields of the **msg_out** object:

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 following is a detailed explanation of the various fields of the **resp_in** object:

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 it is replying to. 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 following is a detailed explanation of the various fields of the **resp_out** object:

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.

8
I/O Objects

This chapter provides reference information for the Neuron C I/O objects.

I/O Objects Syntax and Usage

The syntax and usage for each of the specific I/O object types is described in the following sections. Option keywords such as **clockedge**, **baud**, **numbits**, **select**, and **clock** may appear in any order. Each description also lists the data type of *return-value* for **io_in()** and *output-value* for **io_out()**.

I/O Object Types

- Bit Input/Output
- Bitshift Input/Output
- Byte Input/Output
- Dualslope Input
- Edgedivide Output
- Edgelog Input
- Frequency Output
- I2C Input/Output
- Infrared Input
- Infrared-Pattern Output
- Leveldetect Input
- Magcard Input
- Magcard-Bitstream Input
- Magtrack1 Input
- Muxbus Input/Output
- Neurowire Input/Output
- Nibble Input/Output
- Oneshot Output
- Ontime Input
- Parallel Input/Output
- Period Input
- Pulsecount Input
- Pulsecount Output
- Pulsewidth Output
- Quadrature Input
- SCI Input/Output
- SPI Input/Output
- Serial Input/Output
- Totalcount Input
- Touch Input/Output
- Triac Output
- Triggeredcount Output
- Wiegand Input

See the Neuron Chip and Smart Transceivers databooks for more information on the hardware interfaces provided by the Neuron core for these I/O objects.

Bit Input/Output

The **bit** I/O object type is used to read or control the logical state of a single pin, where 0 equals low and 1 equals high. For bit input, the data type of the return value for **io_in()** is an **unsigned short**. For bit output, the output value is treated as a boolean, so any non-zero value is treated as a 1. Add a **#pragma enable_io_pullups** directive to enable the Neuron Chip or Smart Transceiver's built-in pull-up resistors on pins IO4 through IO7; and, on IO11 on a PL 3120 Smart Transceiver and on a PL 3150 Smart Transceiver. (See Chapter 2, *Compiler Directives* for more details.)

Syntax

pin **input bit** *io-object-name***;**

pin **output bit** *io-object-name* [**=***initial-output-level*]**;**

pin	Specifies one of the eleven I/O pins, **IO_0** through **IO_10**. Bit input/output can be used on any pin. A twelfth pin, **IO_11**, may be used for bit input and/or output on a PL 3120 Smart Transceiver and on a PL 3150 Smart Transceiver.
io-object-name	A user-specified name for the I/O object, in the ANSI C format for variable identifiers.
initial-output-level	A constant expression, in ANSI C format for initializers, used to set the state of the output pin of the I/O object at initialization. The initial state can be 0 or 1. The default is 0.

Usage

unsigned int *input-value***;**
unsigned int *output-value***;**

input-value **= io_in(***io-object-name***);**
io_out(*io-object-name, output-value***);**

Bit Input Example

```
IO_1 input bit io_switch_1;   // declares pin IO1 as a
            // bit input object named io_switch_1
unsigned int switch_on_off;
...
when (reset)
{
      io_change_init(io_switch_1);
}
when (io_changes(io_switch_1))
{
      switch_on_off = input_value;
}
```

Bit Output Example

```
IO_2 output bit io_LED;
unsigned int led_on_off;
...
when(...)
{
      io_out(io_LED, led_on_off);
}
```

Bitshift Input/Output DIRECT I/O OBJECT

The **bitshift** I/O object type is used to shift a data word of up to 16 bits into or out of the Neuron Chip or Smart Transceiver. Data is clocked in and out by an internally generated clock. For bitshift input/output, the data type of the return value for **io_in( )**, and the data type of the output value for **io_out( )**, is an **unsigned long**.

When using multiple serial I/O devices that have differing bit rates, the following pragma must be used:

#pragma enable_multiple_baud

This pragma must appear prior to the use of any I/O function (*e.g.* **io_in( )**, **io_out( )**).

Syntax

pin **input bitshift** [**numbits (***const-expr***)**] [**clockedge (+|-)**]
[**kbaud (***const-expr***)**] *io-object-name***;**

pin **output bitshift** [**numbits (***const-expr***)**] [**clockedge (+|-)**]
[**kbaud (***const-expr***)**] *io-object-name* [**=***initial-output-level*]**;**

pin	An I/O pin. Bitshift input/output requires adjacent pins. The Clock pin is the pin specified, and the Data pin is the following pin. The pin specification denotes the lower-numbered pin of the pair and can be **IO_0** through **IO_6**, **IO_8**, or **IO_9**.
numbits (*const-expr***)**	Specifies the number of bits to be shifted in or out. The *const-expr* expression can evaluate to any number from 1 to 31. The default is 16. Data is shifted in and out with the most significant bit of data first. For **io_in( )**, only the last 16 bits shifted in will be returned. For **io_out( )**, after 16 bits, zeros are shifted out. The number of bits to be shifted can also be specified in the **io_in( )** or **io_out( )** call (for detailed description of these two calls, see Chapter 3, *Functions*). This temporarily overrides the number specified in the device declaration, for that one call only.

clockedge (+|-)	For inputs, this option specifies whether the data is read on the positive-going or negative-going edge of the clock. For outputs, it specifies whether the data is stable on the positive-going or negative-going edge of the clock. The default value is [**+**].
kbaud (*const-expr***)**	Specifies the bit rate. The expression *const-expr* can be 1, 10, or 15. The default is 15kbps with a 10MHz input clock. The bit rate scales proportionally to the input clock.
io-object-name	A user-specified name for the I/O object, in the ANSI C format for variable identifiers.
initial-output-level	A constant expression, in ANSI C format for initializers, used to set the state of the clock pin at initialization. The initial state can be 0 or 1; this applies to the clock pin only. The default is 0.

Usage

unsigned long *input-value***;**
unsigned long *output-value***;**

input-value **= io_in(***input-object* [**,** *numbits*]**);**
io_out(*output-object***,** *output-value*[**,** *numbits*]**);**

Bitshift Input Example

```
IO_6 input bitshift numbits(8) io_shiftreg_keyboard;
unsigned long keyed_in_data;
...
when (...)
{
      keyed_in_data = io_in(io_shiftreg_keyboard);
}
```

Bitshift Output Example

```
IO_8 output bitshift numbits(5)
            clockedge(+) io_adc_1_2_control;
...
when (...)
{
      io_out(io_adc_1_2_control, 0b10010UL);
}
```

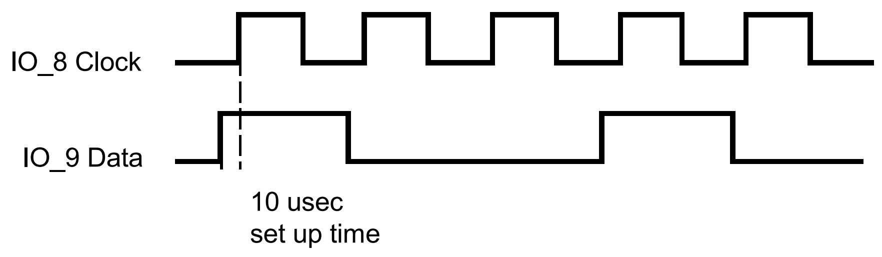

Figure 8.1 Bitshift Output

Byte Input/Output DIRECT I/O OBJECT

The **byte** I/O object type is used to read or control eight pins simultaneously. For byte input/output, the data type of the return value for **io_in( )**, and the data type of the output value for **io_out( )**, is an **unsigned short**.

Syntax

IO_0 input byte *io-object-name***;**

IO_0 output byte *io-object-name* [=*initial-output-level*]**;**

IO_0	Specifies pin **IO_0** as the least significant bit of the byte. Byte input/output uses pins **IO_0** through **IO_7**. The pin specification denotes the lowest numbered pin of the set and must be **IO_0**.
io-object-name	A user-specified name for the I/O object, in the ANSI C format for variable identifiers.
initial-output-level	A constant expression, in ANSI C format for initializers, used to set the state of the output pin of the I/O object at initialization. The initial state can be from 0 to 255. The default is 0.

Usage

unsigned int *input-value***;**
unsigned int *output-value***;**

input-value = **io_in(***io-object-name***);**
io_out(*io-object-name*, *output-value***);**

Byte Input Example

```
IO_0 input byte io_keyboard;
unsigned int character;
...
when (reset)
{
      io_change_init(io_keyboard);
}

when (io_changes(io_keyboard))
{
      character = input_value;
}
```

Byte Output Example

```
IO_0 output byte io_LED_display;
...
when (...)
{
      io_out(io_LED_display, '?');
}
```

Dualslope Input   TIMER/COUNTER I/O OBJECT

The **dualslope** I/O object type is used to control a timer/counter output pin based on a **control_value** argument and the state of a timer/counter input pin. In this configuration, the Neuron Chip or Smart Transceiver controls and measures the integration periods of a dual-slope integrating A/D converter. When combined with external analog circuitry, the Neuron Chip or Smart Transceiver performs A/D measurements with 16 bits of resolution for as little as a 3.278ms integration period with a 40MHz input clock (the period scales with the input clock). Faster conversion rates are attainable at the expense of bit resolution. The duration of the first integration period is a function of **control_value** and the selected clock value:

duration (ns) = control_value * 2000 * 2^(clock) / input_clock (MHz)

The value read back by this device reflects the length of the second integration period, and is also in units of the selected clock value:

2nd_integration (ns) = input_value * 2000 * 2^(clock) / input_clock (MHz)

A single timer/counter provides the control out signal and senses a comparator output signal. The control output signal controls an external analog multiplexer that switches between the unknown input voltage and a voltage reference. The timer/counter's input pin is driven by an external comparator that compares an integrator output with a voltage reference.

For dualslope input, the data type of **control_value** for the **io_in_request()** function is an **unsigned long**. The return value of the **io_in()** function is an **unsigned long**. Both the return value for **io_in()** and the value stored at **input_value** is a number biased negatively by the **control_value** used for the **io_in_request()** function, and may be corrected by adding the **control_value** value into it.

For additional information regarding dualslope A/D conversion, see the *Analog to Digital Conversion with the Neuron Chip* engineering bulletin (part no. 005-0019-02).

Neuron C Resources

The following functions and events are provided for use with the dualslope input object:

io_in_request()	Starts the first step of the integration process. The *control_value* argument controls the length of the first integration period.
io_update_occurs	Signals the end of the entire conversion process. The value at *input_value* now contains the new measurement data.

Syntax

pin [**input**] **dualslope** [**mux** | **ded**] [**invert**] [**clock** (*const-expr*)]
io-object-name;

pin	An I/O pin. Dualslope input can specify pins **IO_4** through **IO_7**.
mux | **ded**	Specifies whether the I/O object is assigned to the multiplexed or dedicated timer/counter. This field only applies, and must be used, when pin **IO_4** is the input pin. The **mux** keyword assigns the I/O object to the multiplexed timer/counter. The **ded** keyword assigns the I/O object to the dedicated timer/counter. When the dedicated timer/counter is used the control output pin will be **IO_1**. When the multiplexed timer/counter is used the control output pin will be **IO_0**. The multiplexed timer/counter is always used for pins **IO_5** through **IO_7**.

invert	Reverses the logical value of the input pin. Use this keyword if the comparator output is high when the converter is in the idle state.
clock (*const-expr***)**	Specifies a clock in the range 0 to 7, where 0 is the fastest clock and 7 is the slowest clock. The default clock for period input is clock 0. The **io_set_clock()** function can be used to change the clock. The clock values are as follows for a Neuron input clock of 10MHz (the values scale with the input clock):

Clock	***Range and Resolution of Period***
0 (default)	0 to 13.11ms in steps of 200 ns (0-65535)
1	0 to 26.21ms in steps of 400 ns
2	0 to 52.42ms in steps of 800 ns
3	0 to 104.86ms in steps of 1.6 µs
4	0 to 209.71ms in steps of 3.2 µs
5	0 to 419.42ms in steps of 6.4 µs
6	0 to 838.85ms in steps of 12.8 µs
7	0 to 1.677s in steps of 25.6 µs

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

Usage

unsigned long *input-value*, *control-value*;

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

input-value **= io_in(***io-object-name***);**

Example

```
IO_4 input dualslope ded clock(0) io_dsad_1;
mtimer repeating go_time;
unsigned long raw_ds;
. . .
when (reset)
{
      go_time = 500;    // Perform a measurement every 500ms
}

when (timer_expires(go_time))
{
      // Start the first integration period (9ms at 10MHz).
      io_in_request(io_dsad_1, 45000UL);
}

when (io_update_occurs(io_dsad_1))
{
      // The value at input_value is biased by the
      // negative value of the control value used.
      // Correct this by adding it back now.
      raw_ds = input_value + 45000UL;
}
```

Edgedivide Output DIRECT I/O OBJECT

The **edgedivide** I/O object type is used to control an output pin by toggling its logic state every **output_value** negative edges on an input pin. This results in a divide-by-n*2 counter where n is the value defined by the **output_value** argument.

For **edgedivide** output, the data type of the output value for **io_out()** is an **unsigned long**. Following reset of the Neuron Chip or Smart Transceiver, the divider will be disabled until the first call to **io_out()** is executed. The first call to **io_out()** for the edgedivide output object will set the output pin high and start the divider. Once the divider is running the function call to **io_out()** only sets the value used for the divider and does not affect the state of the output pin. The exception to this is when the output value is 0, in which case the output signal is forced to a low state and the divider is halted.

Syntax

pin **[output] edgedivide sync (***pin-nbr***)** [**invert**] *io-object-name*
[= *initial-output-level*]**;**

pin	An I/O pin. Edgedivide output can specify pins **IO_0** or **IO_1**. If **IO_0** is specified, the multiplexed timer/counter is used and the sync pin can be **IO_4** through **IO_7**. If **IO_1** is specified, the dedicated timer/counter is used and the sync pin must be **IO_4**.
sync (*pin-nbr***)**	Specifies the sync pin, which is the counting input signal. By default, the divider counts negative edges.

invert	This keyword causes positive edges at the sync pin input to be counted instead of the default negative edges.
io-object-name	A user-specified name for the I/O object, in the ANSI C format for variable identifiers.
initial-output-level	A constant expression, in ANSI C format for initializers, used to set the state of the output pin of the I/O object at initialization. The initial state can be 0 or 1. The default is 0.

Usage

unsigned long *output-value***;**

io_out(*io-object-name*, *output-value***);**

Example

```
IO_0 output edgedivide sync(IO_4) io_divider;
. . .
when (reset)
{
      // There is a 60Hz signal at pin IO_4.
      // Set up the divider to produce
      // a change on pin IO_0 once a minute.
      io_out(io_divider, 3600UL);
}
```

Edgelog Input — TIMER/COUNTER I/O OBJECT

The **edgelog** I/O object type is used to measure a series of both high and low input signal periods on a single input pin, **IO_4**, in units of the clock period:

time_on/time_off (ns) = value_stored * 2000 * 2^(clock) / input_clock (MHz)

Edgelog input can be used to capture complex waveforms such as infrared command input (see also the *Infrared Input* I/O object). For edgelog input, the **io_in()** function requires a pointer to a data buffer, into which the series of **unsigned long** values are stored, and a count argument, which controls the number of values to be stored. The values stored represent the units of clock period between input signal edges, rising or falling. The **io_in()** function returns an **unsigned short int** that contains the actual number of edge-to-edge periods stored. No input events are associated with the edgelog input object.

During the **io_in()** function call, the measurement process stops whenever the maximum period is exceeded. In this case, the value returned will not be equal to the count argument passed.

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

This I/O object uses both of the Neuron timer/counters.

Neuron C Resources

The following functions are provided specifically for use with the edgelog I/O object:

io_edgelog_preload() Changes the maximum value for each period measurement. The maximum value may range from 1 to 65,535; the default value is 65,535. This function is only used for an edgelog device that is *not* declared with the **single_tc** option keyword.

io_edgelog_single_preload() Changes the maximum value for each period measurement for an edgelog device declared with the **single_tc** option keyword. The maximum value may range from 1 to 65,535; the default value is 65,535.

For example, for a 10MHz input clock: an **edgelog** input object using **clock(3)** and the default maximum period would yield a 1.6µs resolution and would not overflow until 104.86ms had elapsed. Using a value of 7500 for **io_edgelog_preload()** would result in the **io_in()** function call terminating if 12ms had elapsed with no input edges.

Syntax

pin [**input**] **edgelog** [**single_tc**] [**mux** | **ded**] [**clock** **(***const-expr***)**]
*io-object-name***;**

pin Specifies a Neuron input pin for the edgelog input object. The input pin may be **IO_4** through **IO_7** if the **single_tc** option is specified, otherwise the input pin must be **IO_4**.

single_tc Optionally specifies that a single timer/counter should be used. Two timer/counters are used if not specified. If a single timer/counter is specified, the application can only be loaded on a device based on a Smart Transceiver, Neuron 3120 Chip, or Neuron 3150B1 Chip (or newer).

mux | ded Specifies whether the I/O object is assigned to the multiplexed or dedicated timer/counter. This option is only necessary with the **single_tc** option when the edgelog device is declared on pin **IO_4**. The multiplexed timer/counter is always used on pins **IO_5** through **IO_7**.

clock(*const-expr***)** Specifies a clock rate in the range 0 to 7, where 0 is the fastest and 7 is the slowest. The default clock rate for edgelog input is 2. The **io_set_clock()** function can be used to change the clock. The clock values are as follows for a Neuron input clock of 10MHz (the values scale with the input clock):

Clock	*Input Range and Resolution*
0	0 to 13.11ms in steps of 200 ns (0-65535)
1	0 to 26.21ms in steps of 400 ns
2 (default)	0 to 52.42ms in steps of 800 ns
3	0 to 104.86ms in steps of 1.6 µs
4	0 to 209.71ms in steps of 3.2 µs
5	0 to 419.42ms in steps of 6.4 µs
6	0 to 838.85ms in steps of 12.8 µs
7	0 to 1.677sec in steps of 25.6 µs

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

In Figure 8.2, an **io_in()** function call is executed sometime after the **IO_4** input signal is sensed as changing to high, but before it has changed back to low. The first period, Period [1], is stored as a value in the array pointed to by the buffer argument. If the **io_in()** function call occurs within the Period [2] time frame, the data for Period [1] is lost.

Individual period measurements may be skipped if the sum of two consecutive periods is less than 104µs (10MHz input clock), regardless of the timer/counter clock setting. The minimum value scales with the input clock.

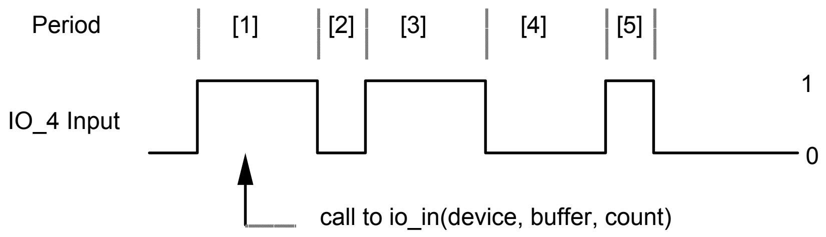

Figure 8.2 io_in() Function Call

If the **IO_4** input pin has been at a constant level for longer than the overflow period before the call to **io_in()** is made, the first value stored in the buffer is not the maximum value, but rather the value for the next period.

Usage

unsigned int *count***;**

unsigned long *input-buffer***[***buffer-size***];**

count = **io_in(***io-object-name***,** *input-buffer***,** *count***);**

Example

```
IO_4 input edgelog clock(7) io_time_stream;

// The next object allows direct reading
// of time_stream level.
IO_4 input bit io_time_stream_level;

unsigned int edges;
unsigned long in_buffer[20];
unsigned long pre_load = 0x4000;

when (reset)
{
      io_edgelog_preload(pre_load);
}

when (io_changes(io_time_stream_level) to 1)
{
      int i;
      // Retrieve edge log
      edges = io_in(io_time_stream, in_buffer, 20);
      // Correct for preload offset
      for (i = 0; i < edges; i++)
          in_buffer[i] += pre_load;
      // Process data
      ...
}
```

Frequency Output — TIMER/COUNTER I/O OBJECT

The **frequency** I/O object type produces a repeating square wave output signal whose period is a function of **output_value** and the selected clock value:

period (ns) = (output_value+n) * 4000 * 2^(clock)/ input_clock (MHz)

where, in the above formula, n = 1 for clock 0, and n = 0 otherwise.

For frequency output, the data type of **output_value** for **io_out()** is an **unsigned long**. An **output_value** of 0 forces the output signal to a low state (unless the **invert** keyword is used in the declaration; see below).

Syntax

pin [**output**] **frequency** [**invert**] [**clock (***const-expr***)**] *io-object-name*
[=*initial-output-level*]**;**

pin	Specifies either pin **IO_0** (using the multiplexed timer/counter) or **IO_1** (using the dedicated timer/counter).
invert	This keyword inverts the output for an output value of 0. The default output for 0 is low.
clock (*const-expr***)**	Specifies a clock in the range 0 to 7, where 0 is the fastest clock and 7 is the slowest clock. The default clock for **frequency** output is **clock (0)**. The **io_set_clock( )** function can be used to change the clock at run-time. The clock values are as follows for an input clock of 10MHz (the values scale inversely proportional to the input clock):

Clock	***Period Range***
0 (default)	800ns to 26.21ms in steps of 400 ns (1-65535)
1	0 to 52.42ms in steps of 800 ns (0-65535)
2	0 to 104.86ms in steps of 1.6 µs (0-65535)
3	0 to 209.71ms in steps of 3.2 µs (0-65535)
4	0 to 419.42ms in steps of 6.4 µs (0-65535)
5	0 to 838.85ms in steps of 12.8 µs (0-65535)
6	0 to 1.677sec in steps of 25.6 µs (0-65535)
7	0 to 3.355sec in steps of 51.2 µs (0-65535)

io-object-name	A user-specified name for the I/O object, in the ANSI C format for variable identifiers.
initial-output-level	A constant expression, in ANSI C format for initializers, used to set the state of the output pin of the I/O object at initialization. The initial state is limited to 0 or 1. The default is 0.

Usage

unsigned long *output-value***;**

io_out(*io-object-name*, *output-value***);**

Example

```
IO_1 output frequency clock(3) io_alarm;
...
when (...)
{
      io_out(io_alarm, 100); // outputs 3.125kHz signal at clock(3)
}

when (...)
{
      io_out(io_alarm, 50);  // outputs 6.25kHz signal at clock(3)
}

when (...)
{
      io_out(io_alarm, 0);      // output signal is stopped
}
```

I^{2}C Input/Output — SERIAL I/O OBJECT

The I^{2}C I/O object type is used to interface to the Philips Semiconductor's Inter-Integrated Circuit (I^{2}C) bus. See the patent notice on the inside front cover of this manual before using this I/O object. Also see the Neurowire, SCI, and SPI I/O objects for alternate forms of serial I/O. The **i2c** I/O object can be declared with pin **IO_8** as the serial clock line (SCL), and pin **IO_9** as the serial data line (SDA), or it can be declared with pin **IO_0** as the serial clock line, and pin **IO_1** as the serial data line. The Neuron Chip or Smart Transceiver acts as a master only. Two external pull-ups are required, and the interface is connected directly to the I/O pins. These I/O pins are operated as open-drain devices in order to support the interface.

An **i2c** I/O object declared on pin **IO_8** can be declared with the **use_stop_condition** option keyword. This option allows for combined format data transfers. For example, you can address and write to a device with one or more **io_out()** calls with stop set to FALSE, followed by a call to **io_out()** with stop set to TRUE to finish the transfer with the STOP condition.

For all transfers an I^{2}C device address argument is required. This byte must be the right-justified 7-bit I^{2}C device address. Up to 255 bytes of data may be transferred at a time. The address is emitted onto the bus at the start of any transfer, just following the I^{2}C bus start condition. A count argument is also required; this controls how many data bytes are to be written or read.

For I^{2}C input/output, **io_in()** and **io_out()** return a 0 or 1 value reflecting the fail (0) or pass (1) status of the transfer. A failed status indicates that the addressed device did not acknowledge positively on the bus, or that the SCL was low at the start of the transfer.

For more information on this protocol and the devices that it supports, see any documentation on Philips Semiconductors Microcontroller Products, under I^{2}C bus descriptions. This I/O object implementation was modified for Neuron C Version 2.1. To use the previous implementation (in case of modification to existing applications where the previous implementation is required for memory considerations) use the #pragma codegen use_i2c_version_1 compiler directive. See Chapter 2, *Compiler Directives* for more information on this pragma.

Syntax

pin **i2c** [**use_stop_condition**] *io-object-name*;

pin	Specify pin **IO_0** or **IO_8**. I^2C requires pins **IO_0** and **IO_1**, or **IO_8** and **IO_9**.
use_stop_condition	Optionally specifies that data transfers should be repeated until a stop condition is reached. A stop condition is defined as a change in the state of the data line (SDA), from LOW to HIGH, while the clock line (SCL) is HIGH. This option keyword can only be used with an **i2c** I/O object declared on pin **IO_8**.
io-object-name	A user-specified name for the I/O object, in the ANSI C format for variable identifiers.

Usage

boolean *return-value***;**
unsigned int *data-buffer***[***buffer-size***];**
unsigned int *dev-address*, *count***;**

// i2c I/O object without use of stop condition
return-value **= io_in(***io-object-name*, *data-buffer*, *dev-address*, *count***);**
return-value **= io_out(***io-object-name*, *data-buffer*, *dev-address*, *count***);**

// i2c I/O object with use of stop condition
return-value **= io_in(***io-object-name*, *data-buffer*, *dev-address*, *count*, *stop***);**
return-value **= io_out(***io-object-name*, *data-buffer*, *dev-address*, *count*, *stop***);**

Example

```
#define AD_ADDR    0x48   // address of the A/D converter
IO_8 i2c io_i2c_bus;
unsigned int adbuff[5], unsigned int ad_cntrl;
boolean retval;
. . .
when (...)
{
    // Read the A/D converter.
    // First, write a control word byte.
    ad_cntrl = 0x04;
    retval = io_out(io_i2c_bus, &ad_cntrl, AD_ADDR, 1);

    // Next, perform a 5-byte read of the A/D converter.
    retval = io_in(io_i2c_bus, adbuff, AD_ADDR, 5);
}
```

Infrared Input

The **infrared** I/O object type is used to capture a data stream generated by a class of infrared remote control devices. This class of devices generates a stream of ones and zeros by modulating an infrared emitter for an on and off cycle, each cycle representing either a one or a zero. The period of this on/off cycle determines the data bit value, a longer cycle implies a one, a shorter cycle implies a zero. See also the **edgelog** input object for an alternate method to decode infrared inputs.

Typically, an infrared signal consists of an infrared source modulated at a carrier frequency between 38kHz and 42kHz. An infrared receiver/demodulator is used external to the Neuron Chip or Smart Transceiver to produce a digital sequence with the carrier removed. Upon execution of the **io_in()** function for the infrared I/O object, the Neuron Chip or Smart Transceiver measures the cycle times and stores the data bits into a buffer passed to the **io_in()** function.

A timer/counter is used to make the series of cycle time measurements. The resolution of these measurements is in units of the clock period:

period (ns) = measured_value * 2000 * 2^(clock) / input_clock (MHz)

For infrared input, the **io_in()** function requires, in addition to the *io_object_name*, four arguments: a pointer to a data buffer in which the series of data bits are stored; a *bit_count* argument, which is the expected number of data bits to be received and stored; a *max_period* argument limiting the range of the timer/counter measurement process; and a *threshold* argument, representing the half way point, in timer/counter count clocks, between a zero data period and a one data period.

The value returned by the **io_in()** function is the actual number of bits read. If less than the expected number of bits (controlled by *bit_count***)** appear at the input pin, the **io_in()** function waits for the *max_period* period before returning. If the expected number of bits, or more, appear at the input pin, the **io_in()** function waits for silence at the input pin before returning. Silence is defined as a lack of input cycles for the *max_period* period. If input cycles persist, the function returns after 256 input cycles occur. This data may be retrieved using the **tst_bit()** function.

The *max_period* argument is an **unsigned long**, and is passed as the negative (two's complement) of the required value. The threshold argument is passed as the *max_period* value plus the required threshold value. See the example below. The **edgelog** input object type can be used to read inputs from infrared devices that do not conform to the assumptions of the infrared input object type.

Syntax

pin [**input**] **infrared** [**mux** | **ded**] [**invert**] [**clock (***const-expr***)**]
*io-object-name***;**

pin	An I/O pin. Infrared input can specify pins **IO_4** through **IO_7**.

mux | ded	Specifies whether the I/O object is assigned to the multiplexed or dedicated timer/counter. This field only applies, and must be used, when pin **IO_4** is the input pin. The **mux** keyword assigns the I/O object to the multiplexed timer/counter. The **ded** keyword assigns the I/O object to the dedicated timer/counter. The multiplexed timer/counter is always used on pins **IO_5** through **IO_7**.
invert	Causes the measurement of the cycle period to be between positive input edges rather than the default, which is between negative input edges.
clock (*const-expr***)**	Specifies a clock in the range 0 to 7, where 0 is the fastest clock and 7 is the slowest clock. The default clock for infrared input is clock 6. The io_set_clock( ) function can be used to change the clock. The clock values for a Neuron input clock of 10 MHz are shown in the table below. The values in the table may be adjusted for different input clocks by scaling them inversely proportional to the change in input clock (for example, for a 20MHz clock, divide all values in the table by 2, and for a 5MHz clock, multiply all values in the table by 2).

Clock	*Range and Resolution of infrared I/O object*
0	0 to 13.11ms in steps of 200 ns (0-65535)
1	0 to 26.21ms in steps of 400 ns
2	0 to 52.42ms in steps of 800 ns
3	0 to 104.86ms in steps of 1.6 µs
4	0 to 209.71ms in steps of 3.2 µs
5	0 to 419.42ms in steps of 6.4 µs
6 (default)	0 to 838.85ms in steps of 12.8 µs
7	0 to 1.677s in steps of 25.6 µs

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

Usage

unsigned int *bit-count***;**
unsigned int *input-buffer***[***buffer-size***];**
unsigned long *max-period***,** *threshold***;**

count = **io_in**(*io-object-name*, *input-buffer*, *bit-count*, *max-period*, *threshold*);

Example

This example works with the NEC µPD1913 encoder chip. This encoder produces a start bit cycle before the actual data stream. During the start bit cycle, the input signal is driven low. This start condition is typical of infrared encoders as it allows a receiver/demodulator's AGC circuit time to adjust. It also gives the Neuron Chip or Smart Transceiver some time to catch this condition from the scheduler, and enter the **io_in()** function. After the start cycle, 32 bits of encoded data appear.

The start cycle is 13ms. The zero cycle is 1.12ms, and the one cycle is 2.24ms. The input clock is 10MHz, and the timer/counter clock is clock (7). This yields a 25.6µs timer/counter clock resolution.

The *max-period* parameter is set to cause an overflow at 110% of the start cycle (the timer/counter will count *up* from this value):

$$65{,}536 - ((1.10 * 13.0e{-3}) / 25.6e{-6})$$
or 64,977.

Given the one and zero data periods, the threshold value is:

$$64{,}977 + (((1.12e{-3} + 2.24e{-3}) / 2) / 25.6e{-6})$$
or 64,977 + 66

This encoder always sends 32 bits, so the count will be 32, and the returned *input-buffer* will be an array of 4 bytes.

```
// This is the demodulated IR input.  Use the non-inverted mode
// to read falling to falling input periods.
IO_4 input infrared ded clock (7) io_ir_data;

// This object allows the application to monitor the input signal
// before entering the io_in(ir_data) function.
IO_4 input bit io_ir_data_level;

unsigned int bits_read;
unsigned int irb[4];
. . .
when (io_changes(io_ir_data_level) to 0)
{
      bits_read = io_in(io_ir_data, irb, 32,
                        64977UL, 64977UL + 66UL);
      if (bits_read == 32) {
          // So far, a valid data message.
          . . .
      }
}
```

Infrared Pattern Output

You can use the **infrared_pattern** I/O object to produce a series of timed repeating square wave output signals. The frequency of the square wave output is controlled by the *clock-expr* setting and by the **unsigned long** *output-frequency* value passed to **io_out()**. Normally, this frequency is the modulation frequency used for infrared transmission.

The pattern of this modulation frequency is controlled by an array of unsigned long timing values, also passed to **io_out()**. The first value in this array controls the length of the first burst of modulation frequency signal output—the output is active for this period. The second value in this array controls the length of an absence of the modulation frequency signal—the output is idle for this period. This pattern is then repeated by subsequent values in the array in order to produce a sequence of frequency output bursts separated by idle periods. This array is similar to the arry generated by the **edgelog** input object.

This I/O model is useful for driving an infrared LED to provide infrared control of devices that support infrared remote control. For example, with a 10MHz Neuron input clock, a clock(1) configuration and an *output-frequency* value of 33 will result in a 37.878kHz (38kHz) modulation signal.

The values in the *timing-table* array control the on and off time of the modulation frequency output. This timing also is a product of the Neuron input clock, and is:

On/off period (µs) = (25.2 * value + 29.4) * *S*

The value of S is determined by the input clock speed, as shown below:

S	**Input Clock Speed**
0.25	40MHz
0.5	20MHz
1	10MHz
1.5259	6.5536MHz
2	5MHz
4	2.5MHz
8	1.25MHz
16	625kHz

The square wave output state is always toggled, between idle (off) and active (on), at the end of the **io_out()** function. Typically, the number of elements in the *timing-table* should always be an odd number, which will result in the output being toggled to idle (turned off) at the end of the **io_out()** function. The last element of the *timing-table* controls the last active period before toggling to idle (off) and returning from the **io_out()** function. If the number of elements in the timing table is even, the output will be toggled on at the end of the **io_out( )** function, which is typically not the desired behavior.

Syntax

pin [**output**] **infrared_pattern** [**invert**]
[**clock(***clock-expr***)**] *io-object-name* [**=** *initial-output-level*] **;**

pin	Specifies a Neuron output pin. The value may be **IO_0** or **IO_1**.
invert	Set this option to specify that the output pin is idle at 1. Otherwise, the output pin is idle at 0.
clock(*const-expr***)**	Specifies a clock in the range 0 to 7, where 0 is the fastest clock and 7 is the slowest clock. The default clock for **infrared_pattern** output is **clock(0)**. You can use the **io_set_clock( )** function to change the clock at run time. The clock values are as follows for a Neuron input clock of 10MHz (the values scale inversely proportional to the input clock):

Clock	***Modulation Period Range***
0 (default)	800ns to 26.21ms in steps of 400 ns (1-65535)
1	0 to 52.42ms in steps of 800 ns (0-65535)
2	0 to 104.86ms in steps of 1.6 µs (0-65535)
3	0 to 209.71ms in steps of 3.2 µs (0-65535)
4	0 to 419.42ms in steps of 6.4 µs (0-65535)
5	0 to 838.85ms in steps of 12.8 µs (0-65535)
6	0 to 1.677sec in steps of 25.6 µs (0-65535)
7	0 to 3.355sec in steps of 51.2 µs (0-65535)

io-object-name	Specifies a name for the I/O object, in the ANSI C format for variable identifiers.
initial-output-level	A constant expression, in ANSI C format for initializers, used to set the state of the output pin of the I/O object at initialization. The initial state is limited to 0 or 1. The default is 0.

Usage

unsigned *count***;**
unsigned long *output-frequency*, *timing-table***[***count***];**

io_out(*io-object-name***,** *output-frequency***,** *timing-table***,** *count***);**
(There is no return value for the function.)

Example

```
IO_0 output infrared_pattern ir_out;

unsigned long timing_table[5]={395,395,783,783,395};
unsigned long ir_freq_out = 62;

when (...)
{
      io_out(ir_out, ir_freq_out, timing_table, 5);
}
```

Leveldetect Input

DIRECT I/O OBJECT

The **leveldetect** I/O object type is used to detect a low level (logical zero) on a single pin. The state of the input is latched in hardware every 50nsec with a 40MHz input clock (the interval scales with Neuron input clock speed), capturing any low level input. This event is represented by a TRUE (1) value returned from the **io_in()** call, and the value is then cleared to 0 when read. However, as long as the input pin level stays at logical zero (0), each **io_in()** call will return a 1 value.

The leveldetect input object is useful for capturing events of short duration that would otherwise be missed by the bit input object. For leveldetect input, the data type of **return_value** for **io_in()** is an **unsigned short**. Add a **#pragma enable_io_pullups** directive to enable the Neuron Chip's or Smart Transceiver's built-in pull-up resistors on pins **IO_4** through **IO_7** (see Chapter 2, *Compiler Directives*, for more details).

Syntax

pin **[input] leveldetect** *io-object-name***;**

pin	An I/O pin. Leveldetect input can specify one of the pins **IO_0** through **IO_7**.
io-object-name	A user-specified name for the I/O object, in the ANSI C format for variable identifiers.

Usage

unsigned int *input-value***;**

input-value **= io_in(***io-object-name***);**

Example

```
IO_6 input leveldetect io_edge_trigger;

when (io_changes(io_edge_trigger) to TRUE)
{
      ... // this task will run at each transition to
          // logical 0 level at pin 6
}
```

Magcard Bitstream Input — SERIAL I/O OBJECT

The **magcard_bitstream** I/O object type provides the ability to read unprocessed serial data streams from most magnetic stripe card readers in real time. This function may be used to read magnetic card data in either direction, forward or reverse, since the data does not need to follow any specific format.

This I/O object can read up to 65535 bits of data, stored in 8192 bytes of data, from a magnetic stripe card reader. The data item unit is a single bit, and the *maxbits* and *count* values indicate the number of bits that can be read, or have been read, respectively. In case of a timeout, the *count* will be less than *maxbits*.

Syntax

IO_8 [**input**] **magcard_bitstream** [**timeout** (*pin-nbr*)] [**clockedge** (+|-)]
[**invert**] *io-object-name*;

IO_8	Specifies pin **IO_8**. The magcard bitstream input requires both pins **IO_8** and **IO_9**. Pin **IO_8** is the negative-going clock, **IO_9** is the serial data input.
timeout(*pin-nbr***)**	Optionally specifies the timeout signal pin, in the range of **IO_0** to **IO_7**. The Neuron Chip or Smart Transceiver checks the logic level at this pin whenever it is waiting for either rising or falling edges of the clock. If a high logic level is sensed on the timeout pin, the transfer is terminated.
clockedge (+|-)	Specifies the polarity of the clock input signal. The default is **clockedge (-)**.
invert	Specifies that the data input signal is inverted. The default is no inversion.
io-object-name	A user-specified name for the I/O object, in the ANSI C format for variable identifiers.

Usage

unsigned long *count*, *maxbits*;
unsigned short *input-buffer*[*buffer-size*];

count = **io_in(***io-object-name*, *input-buffer*, *maxbits***);**

Example

```
IO_8 magcard_bitstream timeout(IO_7) magcard_bits;
const unsigned long maxbits = 64*8;
unsigned long count;
unsigned short input_buffer[64];

when (...)
{
      count = io_in(magcard_bits, input_buffer, maxbits);
}
```

Magcard Input — SERIAL I/O OBJECT

The **magcard** I/O object type is used to transfer synchronous serial data from an ISO 7811 track 2 magnetic-stripe card reader. See the **magtrack1** I/O object for track 1 compatible input, and the **magcard_bitstream** input object for a general-purpose magnetic card input. The magcard input object reads track 2 in the forward direction only (the **magcard_bitstream** input object can read in either direction). The data is presented as a data signal input on pin **IO_9**, and a clock, or data strobe, signal input on pin **IO_8**. The data on pin **IO_9** is clocked on or just following the falling edge of the clock signal on **IO_8**, with the least significant bit first.

Data is recognized as a series of 4-bit characters plus an odd parity bit per character. This process begins when the start sentinel (hex **0B**) is recognized, and continues until the end sentinel (hex **0F**) is recognized. No more than 40 characters, including the two sentinels, will be read. The data is stored as packed BCD digits in the buffer space pointed to by the buffer pointer argument to the **io_in()** function with the parity bit stripped, and includes the start and end sentinel characters. This buffer should be 20 bytes long. The data is stored with the first character in the most significant nibble of the first byte in the buffer.

For magcard input, the **io_in()** function requires a pointer to a data buffer, into which the series of BCD pairs are stored. The **io_in()** function returns a **signed int** that contains the actual number of characters stored.

The parity of each character is checked. The longitudinal redundancy check (LRC) character, which appears just after the end sentinel, is also checked. If either of these tests fail, if more than 40 characters are being clocked in, or if the process aborts due to an input pin event (see below), the **io_in()** function returns the value (-1). The LRC character is not stored.

The magcard object optionally uses one of I/O pins **IO_0** through **IO_7** as a timeout/abort pin. Use of this feature is suggested since the **io_in()** function updates the watchdog timer during clock wait states, and could result in a lockup if the card were to stop moving in the middle of the transfer process. If a high level is detected on the I/O timeout pin, the **io_in()** function aborts. This input can be a one-shot timer counter output, an RC circuit, or a ~Data_valid signal from the card reader.

A Neuron Chip or Smart Transceiver with a 10MHz input clock rate can process a bit rate of up to 8334bps (at a bit density of 75 bits per inch this is a card speed of 111 inches per second). Most magnetic card stripes contain a 15-bit sequence of zero data at the start of the card, allowing time for the application to start the card reading function. At 8334bps, this period is about 1.8ms. If the scheduler latency is greater than the 1.8ms value, for example, due to application processing in another **when** task, the **io_in()** function will miss the front end of the data stream.

Syntax

IO_8 [**input**] **magcard** [**timeout (***pin-nbr***)**] [**clockedge (+|-)**] [**invert**]
*io-object-name***;**

IO_8	Specifies pin **IO_8**. Magcard input requires both pins **IO_8** and **IO_9**. Pin **IO_8** is the negative-going clock, **IO_9** is the serial data input.
timeout(*pin-nbr***)**	Optionally specifies the timeout signal pin, in the range of **IO_0** to **IO_7**. The Neuron Chip or Smart Transceiver checks the logic level at this pin whenever it is waiting for either rising or falling edges of the clock. If a high logic level is sensed on the timeout pin, the transfer is terminated.
clockedge (+|-)	Specifies the polarity of the clock input signal. The default is **clockedge (-)**.
invert	Specifies that the data input signal is inverted. The default is no inversion.
io-object-name	A user-specified name for the I/O object, in the ANSI C format for variable identifiers.

Usage

unsigned int *count***,** *input-buffer***[***buffer-size***];**

count = **io_in(***io-object-name*, *input-buffer***);**

Example

```
// In this example I/O pin IO_7 is connected to a
// ~Data_valid signal which is asserted low as long
// as a valid clock input is being generated by the
// reader device.


IO_8 input magcard timeout(IO_7) io_card_data;

// This next object allows monitoring of
// the ~Data_valid input signal.
IO_7 input bit io_not_data_valid;

int nibbles_read;
unsigned int in_buffer[20];
. . .
when (io_changes(io_not_data_valid) to 0)
{
      nibbles_read = io_in(io_card_data, in_buffer);
}
```

Magtrack1 Input DIRECT I/O OBJECT

The **magtrack1** I/O object type is used to transfer synchronous serial data from an ISO 3554 track 1 magnetic stripe card reader. See the **magcard** input object for track 2 compatible input, and the **magcard_bitstream** input object for a general-purpose magnetic card input. The data is presented as a data signal input on pin **IO_9**, and a clock, or data strobe, signal input on pin **IO_8**. The data on pin **IO_9** is clocked on or just following the falling edge of the signal on **IO_8**, least significant bit first.

Data is recognized in the IATA format—as a series of 6-bit characters plus an odd parity bit per character. This process begins when the start sentinel (hex **05**) is recognized, and continues until the end sentinel (hex **0F**) is recognized. No more than 79 characters, including the two sentinels and the LRC character, will be read. The data is stored in the buffer pointed to by the *input-buffer* pointer argument to the **io_in()** function. The data is stored without the parity bit, and the data includes the start and end sentinel characters.

For **magtrack1** input, the **io_in()** function requires a pointer to a data buffer, into which the series of 6-bit characters are stored. The **io_in()** function returns a **signed int** that contains the actual number of bytes stored.

The parity of each character is checked. The longitudinal redundancy check (LRC) character, which appears just after the end sentinel, is also checked. If either of these tests fail, if more than 79 characters are being clocked in, or if the process aborts due to an input pin event (see below), the **io_in()** function returns the value (-1) as an error indication. The LRC character is not stored.

The **magtrack1** object optionally uses one of I/O pins **IO_0** through **IO_7** as a timeout or abort pin. Use of this feature is suggested since the **io_in()** function will update the watchdog timer during clock wait states, and could result in a lockup if the card were to stop moving in the middle of the transfer process. If a high level is detected on the I/O timeout pin, the **io_in()** function aborts. This input can be a one-shot timer counter output, an RC circuit, or a ~Data_valid signal from the card reader.

A Neuron Chip or Smart Transceiver with a 10MHz input clock rate can process a bit rate of up to 7246bps when the strobe signal has a 33/66 duty cycle (CK_hi = 46µs and CK_lo = 92µs). At a bit density of 210 bits per inch this is a card speed of 34.5 inches per second. Most magnetic card stripes contain a series of zero data at the start of the card, allowing time for the application to start the card reading function.

Syntax

IO_8 [**input**] **magtrack1** [**timeout (***pin-nbr***)**] [**clockedge (+|-)**] [**invert**]
*io-object-name***;**

IO_8	Specifies pin **IO_8**. Magtrack1 input requires both pins **IO_8** and **IO_9**. Pin **IO_8** is the negative-going clock, **IO_9** is the serial data input.
timeout(*pin-nbr***)**	Optionally specifies the timeout signal pin, in the range of **IO_0** to **IO_7**. The Neuron Chip or Smart Transceiver checks the logic level at this pin whenever it is waiting for either rising or falling edges of the clock. If a high logic level is sensed on the timeout pin, the transfer is terminated.
clockedge (+|-)	Specifies the polarity of the clock input signal. The default is **clockedge (-)**.
invert	Specifies that the data input signal is inverted. The default is no inversion.
io-object-name	A user-specified name for the I/O object, in the ANSI C format for variable identifiers.

Usage

unsigned int *count***;**
unsigned int *input-buffer***[***buffer-size***];**

count **= io_in(***io-object-name***,** *input-buffer***);**

Example

```
// In this example I/O pin IO_7 is connected to a
// ~Data_valid signal which is asserted low as long
// as a valid clock input is being generated by the
// reader device.
IO_8 input magtrack1 timeout(IO_7) io_card_data;

// This next object allows monitoring of the
// ~Data_valid input signal.
IO_7 input bit io_not_data_valid;
int chars_read;
unsigned int in_buffer[78];
. . .
when (io_changes(io_not_data_valid) to 0)
{
      chars_read = io_in(io_card_data, in_buffer);
}
```

Muxbus Input/Output PARALLEL I/O OBJECT

The **muxbus** I/O object type uses all eleven I/O pins to form an 8-bit address and bi-directional data bus interface. This I/O object uses pins **IO_0** through **IO_7** for the 8-bit address bus and the 8-bit data bus. Pins **IO_8** through **IO_10** are control signals that are always driven by the Neuron Chip or Smart Transceiver:

Pin	***Function***
IO_0 thru **IO_7**	Address and bi-directional data
IO_8	C_ALS: Address latch strobe, asserted high
IO_9	~C_WS: Write strobe, asserted low
IO_10	~C_RS: Read strobe, asserted low

This I/O object provides the capability to build an 8-bit data bus system with an 8-bit address bus. Typically, an 8-bit D-type latch (such as a 74HC573) is connected to the Neuron I/O pins where pins **IO_0** through **IO_7** are connected to the eight Q inputs. Pin **IO_8** is connected to the Latch Enable input. In this configuration, 8 bits of address are latched on the 8 D output pins of the '573 device.

Pins **IO_9** and **IO_10** are the write and read strobes, normally high.

For **muxbus** output, the **io_out()** function requires an optional 8-bit address argument, and an 8-bit data argument. If the address argument is provided, the Neuron firmware will first set pins **IO_0** through **IO_7** as outputs, then place the address value on these pins, and pulse C_ALS from low to high to low. This latches the address into the address data latch device.

If the address is not provided, this step is skipped. The current value latched in the address latch remains unchanged.

The Neuron firmware then places the data argument value on pins **IO_0** through **IO_7**, and pulses ~C_WS from high to low to high.

For **muxbus** input, the **io_in()** function allows an optional 8-bit address argument only. If this argument is provided, the address is emitted and latched in the same manner as for the **io_out()** function.

Finally, the Neuron firmware sets pins **IO_0** through **IO_7** as inputs. It drops ~C_RS from high to low, inputs the 8 bits of data from pins **IO_0** through **IO_7**, and raises ~C_RS from low to high. The function then returns the 8-bit data value read.

After a read operation, pins **IO_0** to **IO_7** are left in the high impedance state. This could cause excessive power consumption of the 8-bit latch. Using pull-up resistors, or ensuring that the last I/O operation is a write will avoid this situation.

The address argument is optional and can be left off as a performance enhancement where a bus device can be repeatedly read from or written to without changing the bus address. The application must keep track of the current bus address when using this feature. No events are associated with this I/O object.

Syntax

IO_0 muxbus *io-object-name***;**

IO_0	Specifies pin **IO_0**. Muxbus input/output requires all eleven pins and must specify pin **IO_0**.
io-object-name	A user-specified name for the I/O object, in the ANSI C format for variable identifiers.

Usage

unsigned int *data-byte***;**

data-byte = **io_in(***io-object-name*, *address***);**

data-byte = **io_in(***io-object-name***);**

io_out(*io-object-name*, *address*, *data-byte***);**

io_out(*io-object-name*, *data-byte***);**

Example

```
IO_0 muxbus io_local_bus;

when (. . .)
{
      // Write two bytes to addresses 0x20 and 0x21,
      // and wait for the data at 0x20 to contain
      // the 0x80 value.
      io_out(io_local_bus, 0x20, 128);
      io_out(io_local_bus, 0x21, 1);
      if ((io_in(io_local_bus, 0x20) & 0x80) == 0)
      {
          // Continue to read the same address.
          while ((io_in(io_local_bus) & 0x80) == 0);
      }
}
```

Neurowire Input/Output — SERIAL I/O OBJECT

The **neurowire** I/O object type is used to transfer data using a fully synchronous serial data format. Data is shifted in at the same time as data is shifted out. Neurowire I/O is useful for external devices, such as A/D and D/A converters, and display drivers incorporating serial interfaces that conform with National Semiconductor's Microwire™ or Motorola's SPI (Serial Peripheral Interface).

The Neurowire I/O object may be configured in master mode or slave mode. The primary difference between master and slave modes is that the clock signal is an *output* for the master mode, and an *input* for the slave mode.

In Neurowire master mode, one or more of the pins **IO_0** through **IO_7** may be used as a chip select, allowing multiple Neurowire devices to be connected on a 3-wire bus. The clock rate may be specified as 1, 10, or 20kbps at a Neuron Chip or Smart Transceiver input clock rate of 10MHz; these scale proportionally with input clock.

In Neurowire slave mode, one of the **IO_0** through **IO_7** pins may be designated as a timeout pin. A logic one level on the timeout pin causes the Neurowire slave I/O operation to be terminated before the specified number of bits has been transferred. This prevents the Neuron Chip or Smart Transceiver watchdog timer from resetting the chip in the event that fewer than the requested number of bits are transferred by the external clock.

In both master and slave modes, up to 255 bits of data may be transferred at a time. Neurowire I/O suspends application processing until the operation is complete.

For Neurowire input/output, **io_in()** and **io_out()** require a pointer to the data buffer as the **input_value** and **output_value**. Because Neurowire I/O is bidirectional, input and output occur at the same time, and therefore, the calls **io_in()** and **io_out()** are equivalent. Use of either call will initiate a bidirectional transfer. Data is transmitted 8 bits at a time, most significant bit first. The clock edge used to clock the data is specified by the **clockedge** parameter. Data is also then transferred into the same buffer pointed to by **input_value** or **output_value**, most significant bit first, following the clock edge, overwriting the original contents of the buffer. If the number of bits to be transferred is not a factor of eight as defined by *count*, the last byte transferred into the buffer will contain undefined data bit values in the remaining (unfilled) bit locations.

When using multiple serial or Neurowire I/O objects that have differing bit rates, the following directive must be used:

#pragma enable_multiple_baud

This pragma must appear prior to the use of any I/O function such as **io_in()** or **io_out()**.

For examples on the use of the Neurowire input/output object, see the following engineering bulletins: *Driving a Seven Segment Display with the Neuron Chip* (part no. 005-0014-01) and *Analog-to-Digital Conversion with the Neuron Chip* (part no. 005-0019-01).

Syntax

IO_8 neurowire master | slave [**select (***pin-nbr***)**] [**timeout (***pin-nbr***)**]
[**kbaud (***const-expr***)**] [**clockedge (+|-)**] *io-object-name***;**

IO_8	Specifies pin **IO_8**. Neurowire requires pins **IO_8** through **IO_10** and must specify **IO_8**. The **select** pin must be one of **IO_0** through **IO_7**. Pin **IO_8** is the clock, driven by the Neuron Chip or Smart Transceiver (or the external master). Pin **IO_9** is serial data output and **IO_10** is serial data input. Up to 255 bits of data can be transferred at a time.
master	Specifies that the Neuron Chip or Smart Transceiver provides the clock on pin **IO_8**, which is configured as an output pin.
slave	Specifies that the Neuron Chip or Smart Transceiver senses the clock on pin **IO_8**, which is configured as an input pin. The maximum input clock rate is 72kbps, 50/50 duty cycle, with a 40MHz Neuron input clock. This rate scales proportionally to the input clock.

select (*pin-nbr***)**	Specifies the chip select pin for a Neurowire master. ***NOTE****: This is applicable to master mode only.* Before the data transfer, the chip select pin goes low; after the data transfer, the select pin goes high. In addition to this declaration with the **select** keyword, the chip select pin must also be declared with a bit output object, unless there is no chip select pin in use. If no chip select pin is in use, the pin declared as the select pin can also be declared as any of the allowable input objects for that pin (for example, bit input). Not used for a Neurowire slave.
timeout (*pin-nbr***)**	Specifies the optional timeout signal pin for a Neurowire slave, in the range of **IO_0** to **IO_7**. ***NOTE****: This is applicable to slave mode only.* When a timeout signal pin is used, the Neuron firmware checks the logic level at this pin whenever it is waiting for either rising or falling edges of the clock. If a logic level of 1 is sensed, the transfer is terminated. This allows the use of an external timeout signal, or an internally generated timeout signal, such as an inverted oneshot output object, to limit the duration of the transfer. The watchdog timer is updated by this object every falling edge of the clock on pin **IO_8**.
kbaud (*const-expr***)**	Specifies the bit rate for a Neurowire master. The expression *const-expr* can evaluate to 1, 10, or 20. The default is 20kbps with a 10MHz Neuron input clock. The bit rate scales proportionally to the input clock. Not used for a Neurowire slave.
clockedge (+|-)	Specifies the polarity of the clock signal. The default is a rising edge clock, **clockedge (+)**. Specifying **clockedge (-)** causes the data to be clocked at the falling edge of the clock signal.
io-object-name	A user-specified name for the I/O object, in the ANSI C format for variable identifiers.

Usage

unsigned int *count*, *io-buffer***[***buffer-size***];**

io_out(*io-object-name*, *io-buffer*, *count***);**

Example

```
IO_8 neurowire master select(IO_2) io_display;
IO_2 output bit io_display_select = 1;    // active low

unsigned int dd_config = 0x01; // 8 bits=>display config reg
unsigned int dd_data[3];       // 24 bits=>display data reg

when (...)
{
    dd_config = 0x01;
    io_out(io_display, &dd_config, 8);
    dd_data[0] = 0x80;
    dd_data[1] = 0xAB;
    dd_data[2] = 0xCD;
    io_out(io_display, dd_data, 24);
}
```

Nibble Input/Output — DIRECT I/O OBJECT

The **nibble** I/O object type is used to read or control four adjacent pins simultaneously. For nibble input/output, the data type of *return_value* for **io_in()**, and the data type of the output value for **io_out()** is an **unsigned short**. Add a **#pragma enable_io_pullups** directive to enable the Neuron Chip's or Smart Transceiver's built-in pull-up resistors on pins **IO_4** through **IO_7** (see Chapter 2, *Compiler Directives*, for more details).

Syntax

pin **input nibble** *io-object-name***;**

pin **output nibble** *io-object-name* [**=** *initial-output-level*]**;**

pin	An I/O pin. Nibble input/output requires four adjacent pins. The pin specification denotes the lowest numbered pin of the set and can be **IO_0** through **IO_4**. The lowest numbered I/O pin is defined as the least significant bit of the nibble data.
io-object-name	A user-specified name for the I/O object, in the ANSI C format for variable identifiers.
initial-output-level	A constant expression, in ANSI C format for initializers, used to set the state of the output pin of the I/O object at initialization. The initial state can be from 0 to 15. The default is 0.

Usage

unsigned int *input-value***;**

unsigned int *output-value***;**

input-value = **io_in(***io-object-name***);**

io_out(*io-object-name*, *output-value***);**

Nibble Input Example

```
IO_0 input nibble io_column_read;
int column;

when (reset)
{
      io_change_init(io_column_read);
}

when (io_changes(io_column_read))
{
      column = input_value;
}
```

Nibble Output Example

```
IO_4 output nibble io_row_write;

when (...)
{
      io_out(io_row_write, 0b1000U);
}
```

Oneshot Output

The **oneshot** I/O object type produces a single output pulse whose duration is a function of the output value and the selected clock value, calculated as follows:

duration (ns) = output_value * 2000 * 2^(clock) / input_clock (MHz);

The **oneshot** I/O object can be retriggered. A call to **io_out()** for a oneshot object will start a new pulse, even if one is currently in progress.

For **oneshot** output, the data type of the output value for **io_out()** is an **unsigned long**. An output value of zero (0) forces the output to a low state.

Syntax

pin [**output**] **oneshot** [**invert**] [**clock (***const-expr***)**] *io-object-name*
[=*initial-output-level*]**;**

pin	Specifies either pin **IO_0** (using the multiplexed timer/counter) or **IO_1** (using the dedicated timer/counter).
invert	Causes the output to be inverted, producing a signal that is normally high with low pulses. The default is normally low with high pulses.
clock (*const-expr***)**	Specifies a clock in the range 0 to 7, where 0 is the fastest clock and 7 is the slowest clock. The default clock for oneshot output is clock 7. The **io_set_clock()** function can be used to change the clock. The clock values are as follows for a Neuron input clock of 10 MHz:

Clock	*Oneshot Duration*
0	0 to 13.11ms in steps of 200 ns (0-65535)
1	0 to 26.21ms in steps of 400 ns
2	0 to 52.421ms in steps of 800 ns
3	0 to 104.86ms in steps of 1.6 μs
4	0 to 209.71ms in steps of 3.2 μs
5	0 to 419.42ms in steps of 6.4 μs
6	0 to 838.85ms in steps of 12.8 μs
7 (default)	0 to 1.677sec in steps of 25.6 μs

io-object-name	A user-specified name for the I/O object, in the ANSI C format for variable identifiers.
initial-output-level	A constant expression, in ANSI C format for initializers, used to set the state of the output pin of the I/O object at initialization. The initial state may be 0 or 1. The default is 0.

Usage

unsigned long *output-value***;**

io_out(*io-object-name*, *output-value***);**

Example

```
IO_0 output oneshot io_flasher;
unsigned long k = 39062;      // 1 second pulse

mtimer repeating flash_timer;

when (...)
{
      flash_timer = 2000;
      // start timer, flash every 2 secs
}

when (timer_expires(flash_timer))
{
      io_out(io_flasher, k);
      // outputs a 1 sec pulse
}
```

Ontime Input — TIMER/COUNTER I/O OBJECT

The **ontime** I/O object type measures the high or low period of an input signal in units of the clock period:

time_on (ns) = return_value * 2000 * 2^(clock) / input clock (MHz)

For **ontime** input, the data type of the return value for **io_in()** is an **unsigned long**.

The state of the input pin is latched in hardware every 50ns with a 40MHz Neuron input clock (the value scales inversely with clock speed). If no edges occur during the measuring period, an overflow condition occurs. The next **io_in()** after the overflow has occurred will return the out-of-range value of 0xFFFF. The **io_update_occurs** event will not be asserted as TRUE unless the program uses the **io_preserve_input()** function after the **io_select()** when using the multiplexed timer/counter, or in the reset task when using the dedicated timer/counter.

Syntax

pin [**input**] **ontime** [**mux** | **ded**] [**invert**] [**clock** (*const-expr*)]
*io-object-name***;**

pin	An I/O pin. Ontime input can specify one of pins **IO_4** through **IO_7** as the input pin.
mux | ded	Specifies whether the I/O object is assigned to the multiplexed or dedicated timer/counter. This field is used only when pin **IO_4** is used as the input pin.

The **mux** keyword assigns the I/O object to the multiplexed timer/counter. The **ded** keyword assigns the I/O object to the dedicated timer/counter. The multiplexed timer/counter is always used for pins **IO_5** through **IO_7**.

invert Causes the measurement of the low period of the input signal. By default, measurement occurs on the high period of the input signal.

clock (*const-expr***)** Specifies a clock in the range 0 to 7, where 0 is the fastest clock and 7 is the slowest clock. The default clock for **ontime** input is clock 2. The **io_set_clock()** function can be used to change the clock. The clock values are as follows for a Neuron input clock of 10MHz:

Clock	*Input Range and Resolution*
0	0 to 13.11ms in steps of 200 ns (0-65535)
1	0 to 26.21ms in steps of 400 ns
2 (default)	0 to 52.42ms in steps of 800 ns
3	0 to 104.86ms in steps of 1.6 µs
4	0 to 209.71ms in steps of 3.2 µs
5	0 to 419.42ms in steps of 6.4 µs
6	0 to 838.85ms in steps of 12.8 µs
7	0 to 1.677sec in steps of 25.6 µs

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

Usage

unsigned long *input-value*;

input-value = **io_in(***io-object-name***);**

Example

```
IO_4 input ontime ded clock(7) io_gate_time;
unsigned long pulse_duration;

when (io_update_occurs(io_gate_time))
{
      pulse_duration = input_value;
                  // measures up to 1.677 seconds
}
```

Parallel Input/Output

The **parallel** I/O object type uses all eleven I/O pins for an 8 bit parallel interface with handshaking. This interface allows data transfer at rates up to 3.3Mbps. A parallel interface can be used for the following applications:

- To interface a Neuron Chip or Smart Transceiver to an attached microprocessor or to the bus of a computer system. This interface can use the Neuron Chip or Smart Transceiver as a communications chip with an existing processor-based system, provide more application performance, or supply more memory. This type of interface is enhanced with a ShortStack™ Micro Server (with an SCI or SPI interface) or Microprocessor Interface Program (MIP; with a parallel or dual-ported RAM interface). The ShortStack Micro Server and MIP move network variable and application message processing to the attached processor.
- For application-level gateways, two Neuron Chips or Smart Transceivers (or one of each) are connected back to back across the parallel interface, producing two transceiver interfaces to transport data from one system to the other.

This interface is bidirectional, with the direction (read/write) controlled by the device declared as the master. When using this interface, the Neuron Chip or Smart Transceiver can be either a master or a slave. The parallel I/O object provides three different configurations of the parallel I/O interface: master, slave A, and slave B. Master-slave A connections are typically used for parallel port interfaces and for Neuron Chip/Smart Transceiver to Neuron Chip/Smart Transceiver communication. Slave B is typically used for communicating from a microprocessor bus to a Neuron Chip or Smart Transceiver. Multiple slave B devices can be connected to a single bus. The difference between slave A and slave B concerns the use of one of the three control signals (see the following description of the **slave**, **slave_b**, and **master** keywords).

No other I/O objects can be declared on pins **IO_0** through **IO_10** when the parallel I/O object is being used.

Neuron C Resources

In order to use the parallel I/O object of the Neuron Chip or Smart Transceiver, **io_in()** and **io_out()** require a pointer to the **parallel_io_interface** structure :

```
struct parallel_io_interface
{
   unsigned length;             // length of data field
   unsigned data[MAXLENGTH];    // data field
} piofc;
```

The **parallel_io_interface** structure must be declared in the application program, with an appropriate definition of **MAXLENGTH** signifying the largest expected buffer size for any data transfer.

In the case of the **io_out()** function, **length** is the number of bytes to be transferred out and is set by the application program. In the case of **io_in()**, **length** is the number of bytes to be transferred in. If the incoming length is larger than **length**, then the incoming data stream is flushed, and **length** is set to zero. Otherwise, **length** is set to the number of data bytes read. The length field must be set before calling **io_in()** or **io_out()**. The maximum value for the **length** field (and the **MAXLENGTH** value) is 255.

The following functions and events are provided specifically for use with the parallel I/O object:

io_in_ready	This event becomes TRUE whenever a message arrives on the parallel bus that must be read. The application must then call **io_in()** to retrieve the data.
io_out_request()	This function is used to request an **io_out_ready** indication for an I/O object. It is up to the application to buffer the data until the **io_out_ready** event is TRUE. This function acquires the token for the parallel I/O interface.
io_out_ready	This event becomes TRUE whenever the parallel bus is in a state where it can be written to and the **io_out_request()** function was previously invoked. The application must then call the **io_out()** function to write the data to the parallel port. This function relinquishes the token for the parallel I/O interface.

Neuron C applications may also use the parallel bus in a unidirectional manner (i.e., applications don't need to use both the **when(io_in_ready)** or **when(io_out_ready)** clauses if they only need to use one).

See *Parallel I/O Object* in Chapter 2, *Focusing on a Single Device*, of the *Neuron C Programmer's Guide* for additional information. Also see the *Parallel I/O Interface to the Neuron Chip* engineering bulletin (part no. 005-0021-01) for more information.

To prevent contention for the data bus, a virtual write token is passed back and forth between the master device and the slave device (in both slave A and slave B modes). The master device has the write token initially after a reset. The parallel I/O object automatically manages the write token; no specific application code is needed.

Syntax

IO_0 parallel slave | slave_b | master *io-object-name***;**

IO_0 Parallel input/output requires all eleven pins and must specify pin **IO_0**. The pins are used as follows:

Pin	***Master***	***Slave A***	***Slave B***
IO_0 thru IO_7	Data Bus	Data Bus	Data Bus
IO_8	Chip select output	Chip select input	Chip select input
IO_9	RD/~WR output	RD/~WR input	RD/~WR input
IO_10	HANDSHAKE input	HANDSHAKE input	A0 input

slave | slave_b | master

Specifies slave A, slave B, or master mode. For master and slave A modes, **IO_10** is a handshake signal. For slave B mode, **IO_10** becomes an address line input, A0, and the handshake signal appears on the data bus on pin **IO_0** when **A0=1**. When **A0=0**, the data appears on the data bus. This mode is used to allow a Neuron Chip or Smart Transceiver to reside on a microprocessor bus with the data at one address location and the handshake signal at another.

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

Usage

```
struct parallel_io_interface {
        unsigned int length;
        unsigned int data[data-size];
} piofc;
```

io_in(*io-object-name***, &piofc);**

io_request(*io-object-name***);**

io_out(*io-object-name***, &piofc);**

Example

The following example shows how to use the **io_in_ready** and **io_out_ready** events, in conjunction with the **io_out_request()** function, to handle parallel I/O processing. (See also the description of the parallel I/O object in Chapter 2, *Focusing on a Single Device*, of the *Neuron C Programmer's Guide* and the *Parallel I/O Interface to the Neuron Chip* engineering bulletin (part no. 005-0021-01).

```
IO_0 parallel slave s_bus;
#define DATA_SIZE 255
struct parallel_io_interface
{
   unsigned int length;    // length of data field
   unsigned int data [DATA_SIZE];
} piofc;

when (io_in_ready(s_bus))  // ready to input data
{
   piofc.length = DATA_SIZE;  // number of bytes to read
   io_in(s_bus, &piofc);   // get 10 bytes of incoming data
}

when (io_out_ready(s_bus)) // ready to output data
{
   piofc.length = 10;      // number of bytes to write
   io_out(s_bus, &piofc);  // output 10 bytes from buffer
}

when (...)                 // user defined event
{
   io_out_request(s_bus);  // post the write transfer request
}
```

Period Input — TIMER/COUNTER I/O OBJECT

The **period** I/O object type measures the total period, from edge to edge, of an input signal in units of the clock period, calculated as follows:

period (ns) = (*return-value*+n) * 2000 * 2^(clock) / input_clock (MHz)

where, in the above formula, n = 1 for clock(0) and n = 0 otherwise. Also, the value *return-value* is equivalent to the *input-value* shown below in *Usage*.

For *period-input*, the data type of the *return-value* for **io_in()** is an **unsigned long**.

The input is latched every 50ns with a 40MHz Neuron input clock. This value scales inversely with the input clock speed. If no edges occur during the measuring period, an overflow condition occurs. The next **io_in()** after the overflow has occurred will return the out-of-range value of 0xFFFF. The **io_update_occurs** event will not be asserted as TRUE unless the program uses the **io_preserve_input()** function after the **io_select()** when using the multiplexed timer/counter, or in the reset task when using the dedicated timer/counter.

Syntax

pin [**input**] **period** [**mux** | **ded**] [**invert**]
[**clock (***const-expr***)**] *io-object-name***;**

pin	An I/O pin. Period input can specify pins **IO_4** through **IO_7**.
mux | **ded**	Specifies whether the I/O object is assigned to the multiplexed or dedicated timer/counter. This field only applies, and must be used, when pin **IO_4** is the input pin. The **mux** keyword assigns the I/O object to the multiplexed timer/counter. The **ded** keyword assigns the I/O object to the dedicated timer/counter. The multiplexed timer/counter is always used for pins **IO_5** through **IO_7**.
invert	Causes the measurement of time between positive edges and typically has no effect. By default, period input measures the time between negative edges.
clock (*const-expr***)**	Specifies a clock in the range 0 to 7, where 0 is the fastest clock and 7 is the slowest clock. The default clock for period input is clock 0. The **io_set_clock()** function can be used to change the clock. The clock values are as follows for a Neuron input clock of 10 MHz:

Clock	***Range and Resolution of Period I/O object***
0 (default)	0 to 13.11ms in steps of 200 ns (0-65535)
1	0 to 26.21ms in steps of 400 ns
2	0 to 52.42ms in steps of 800 ns
3	0 to 104.86ms in steps of 1.6 µs
4	0 to 209.71ms in steps of 3.2 µs
5	0 to 419.42ms in steps of 6.4 µs
6	0 to 838.85ms in steps of 12.8 µs
7	0 to 1.677s in steps of 25.6 µs

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

Usage

unsigned long *input-value***;**

input-value = **io_in(***io-object-name***);**

Example

```
IO_4 input period mux clock(7) io_switch_4;

when (io_update_occurs(io_switch_4))      // END OF PERIOD
{
      unsigned short timegap; // in tenths of a second

      timegap = (unsigned short)(io_in(io_switch_4) / 3906);
                  // convert to tenths of sec
}
```

Pulsecount Input — TIMER/COUNTER I/O OBJECT

The **pulsecount** I/O object type counts the number of input edges at the input pin over a period of 0.8388608 seconds. For pulsecount input, the data type of the return value for **io_in()** is an **unsigned long**.

The input is latched every 50ns with a 40MHz Neuron input clock. This value scales inversely with the input clock. The value of a pulsecount input object is updated every 0.8388608 seconds and the **io_update_occurs** event becomes TRUE.

If no edges occur during the measuring period, an overflow condition occurs. The next **io_in()** after the overflow has occurred will return the out-of-range value of 0xFFFF. The **io_update_occurs** event will not be asserted as TRUE unless the program uses the **io_preserve_input()** function after the **io_select()** when using the multiplexed timer/counter, or in the reset task when using the dedicated timer/counter.

Syntax

pin **input pulsecount** [**mux** | **ded**] [**invert**] *io-object-name***;**

pin	An I/O pin. Pulsecount input can specify pins **IO_4** through **IO_7**.
mux | ded	Specifies whether the I/O object is assigned to the multiplexed or dedicated timer/counter. This field is used only when pin **IO_4** is used as the input pin. The **mux** keyword assigns the I/O object to the multiplexed timer/counter. The **ded** keyword assigns the I/O object to the dedicated timer/counter. The multiplexed timer/counter is always used for pins **IO_5** through **IO_7**.

invert	Causes positive edges to be counted. Typically this has no effect since the number of positive edges equals the number of negative edges. By default, pulsecount input counts the number of negative input edges.
io-object-name	A user-specified name for the I/O object, in the ANSI C format for variable identifiers.

Usage

unsigned long *input-value***;**

input-value = **io_in(***io-object-name***);**

Example

```
IO_7 input pulsecount io_total_ticks;
unsigned long k;

when (io_update_occurs(io_total_ticks))
{
      k = input_value;
                  // for up to 65535 ticks per 0.839 seconds
}
```

Pulsecount Output — TIMER/COUNTER I/O OBJECT

The **pulsecount** I/O object type produces a sequence of pulses whose period is a function of the clock period, calculated as follows:

period (ns) = 256 * 2000 * 2^(clock) / input_clock (MHz)

The **output_value** determines the number of pulses output. When this I/O object is used, the **io_out()** function call does not return until all pulses have been produced. This process ties up the application processor for the duration of the pulsecount.

For pulsecount output, the data type of the output value for **io_out()** is an **unsigned long**. An output value of 0 forces the output signal to its normal state.

Syntax

pin **output pulsecount** [**invert**] [**clock (***const-expr***)**] *io-object-name* [=*initial-output-level*]**;**

pin	Specifies either pin **IO_0** (using the multiplexed timer/counter) or **IO_1** (using the dedicated timer/counter).
invert	Causes the signal to be inverted, normally high with low pulses. By default, the signal is normally low with high pulses.

clock (*const*-expr**)** Specifies a clock in the range 1 to 7, where 1 is the fastest clock and 7 is the slowest clock. The default clock for pulsecount output is clock 7. The **io_set_clock()** function can be used to change the clock. (Specifying clock 0 for **io_set_clock()** results in an unspecified number of counts, since this is not a valid clock for pulsecount output.)

The periods of the pulses for a Neuron input clock of 10MHz are as follows:

Clock	***Pulse Period***
1	102.4 µs
2	204.8 µs
3	409.6 µs
4	819.2 µs
5	1.638 ms
6	3.277 ms
7 (default)	6.554 ms

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

initial-output-level A constant expression, in ANSI C format for initializers, used to set the state of the output pin of the I/O object at initialization. The initial state may be 0 or 1. The default is 0.

Usage

unsigned long *output-value***;**

io_out(*io-object-name*, *output-value***);**

Example

```
IO_1 output pulsecount io_train_out;

when (...)
{
      io_out(io_train_out, 100);
      // will produce 100 pulses on pin 1
}     // each pulse of period 6.554 milliseconds
```

Pulsewidth Output

The **pulsewidth** I/O object type produces a repeating waveform whose duty cycle is a function of *output-value* and whose period is a function of the clock period, calculated as follows:

pulsewidth (ns) = *output-value* * 2000 * 2^(clock) / input_clock (MHz)

total_period (ns) = 256 * 2000 * 2^(clock) / input_clock (MHz)

For 8-bit pulsewidth output, the data type of *output-value* for **io_out()** is an **unsigned short**. An *output-value* of 0 results in a 0% duty cycle. A value of 255 (the maximum value allowed) results in a 100% duty cycle. The duty cycle of the pulse train is (*output-value*/256), except when *output-value* is 255; in that case, the duty cycle is 100%.

For 16-bit pulsewidth output, the data type of *output-value* for **io_out()** is an **unsigned long**. An *output-value* of 0 results in a 0% duty cycle. A value of 65535 (the maximum value allowed) results in a 99.998% duty cycle. The duty cycle of the pulse train is (*output-value*/65536).

WARNING: Use of an *output-value* of 1 in combination with **clock(0)** should be avoided.

Syntax

pin [**output**] **pulsewidth** [**short** | **long**] [**invert**]
[**clock (***const-expr***)**] *io-object-name* [=*initial-output-level*]**;**

pin	Specifies either pin **IO_0** (using the multiplexed timer/counter) or **IO_1** (using the dedicated timer/counter).
short | long	Resolution of the data value: **short** specifies 8-bit pulsewidth output, **long** specifies 16-bit. If neither of these options is specified, the **pulsewidth** I/O object defaults to the 8-bit (short) mode.
invert	Causes the output signal to be inverted, normally high for a 0% duty cycle. By default, the output signal is normally low for a 0% duty cycle.

clock (*const-expr***)** Specifies a clock in the range 0 to 7, where 0 is the fastest clock and 7 is the slowest clock. The default clock for pulsewidth output is clock 0. The **io_set_clock()** function can be used to change the clock. The clock values are as follows for an input clock of 10 MHz:

8-bit Pulsewidth Output

Clock	***Control Range***
0 (default)	19.53kHz in steps of 200 ns (2-255)
1	9.77kHz in steps of 400 ns (1-255)
2	4.88kHz in steps of 800 ns (1-255)
3	2.44kHz in steps of 1.6 µs (1-255)
4	1.22kHz in steps of 3.2 µs (1-255)
5	610.3Hz in steps of 6.4 µs (1-255)
6	305.1Hz in steps of 12.8 µs (1-255)
7	152.6Hz in steps of 25.6 µs (1-255)

16-bit Pulsewidth Output

Clock	***Control Range***
0 (default)	76.29Hz in steps of 200 ns (2-65535)
1	38.16Hz in steps of 400 ns (1-65535)
2	19.06Hz in steps of 800 ns (1-65535)
3	9.53Hz in steps of 1.6 µs (1-65535)
4	4.77Hz in steps of 3.2 µs (1-65535)
5	2.38Hz in steps of 6.4 µs (1-65535)
6	1.19Hz in steps of 12.8 µs (1-65535)
7	0.60Hz in steps of 25.6 µs (1-65535)

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

initial-output-level A constant expression, in ANSI C format for initializers, used to set the state of the output pin of the I/O object at initialization. The initial state is limited to 0 or 1. The default is 0.

Usage

unsigned int *output-value***;** // for 8-bit output

unsigned long *output-value***;** // for 16-bit output

io_out(*io-object-name***,** *output-value***);**

Example

```
IO_1 output pulsewidth clock(7) io_lamp_led;

mtimer repeating tick_timer;
unsigned short brightness;

when (...)
{
    tick_timer = 10;          // start clock for fading
    brightness = 255;   // start brightness for fading
}

when (timer_expires(tick_timer))
{
    brightness -= 1;
    io_out(io_lamp_led, brightness);
    if (brightness == 0)
        tick_timer = 0; // turn off the timer
}
```

Quadrature Input — TIMER/COUNTER I/O OBJECT

The **quadrature** I/O object type is used to read a shaft or positional encoder input on two adjacent pins. A **signed long** value is returned from **io_in()**, based on the change since the last input. The input is sampled every 50 ns with a 40MHz Neuron input clock. This value scales inversely with the input clock speed. Add a **#pragma enable_io_pullups** directive to enable the Neuron Chip's or Smart Transceiver's built-in pull-up resistors. For more information on quadrature input, see *Neuron Chip Quadrature Input Function Interface* engineering bulletin.

Syntax

pin [**input**] **quadrature** *io-object-name***;**

pin	An I/O pin. Quadrature input requires two adjacent pins. The pin specification denotes the lower-numbered pin of the pair. The pin can be **IO_4** (which uses the dedicated timer/counter) or **IO_6** (which uses the multiplexed timer/counter). Figure 8.3 illustrates the use of the two signal inputs A and B. Both edges of input A are counted. Input B indicates whether input A is moving in a positive or a negative direction.
io-object-name	A user-specified name for the I/O object, in the ANSI C format for variable identifiers.

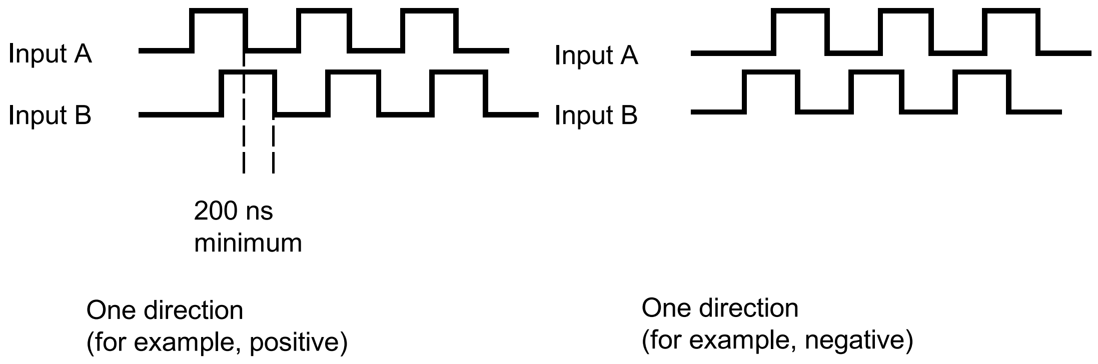
Input A
Input B
200 ns minimum
One direction (for example, positive)
Input A
Input B
One direction (for example, negative)

Figure 8.3 Quadrature Input

Usage

long *input-value***;**

input-value **= io_in(***io-object-name***);**

Example

```
IO_4 input quadrature io_dial;

long dial_angle = 0;

when (io_update_occurs(io_dial))
{
    dial_angle += input_value;
                  // integrate angle in software
}
```

SCI (UART)

You can use the hardware **Serial Communications Interface** (**SCI**) I/O object in applications that you develop for Smart Transceivers or Neuron Chips with integrated UART hardware such as the PL 3120 or PL 3150 Smart Transceiver. SCI is an asynchronous serial communication interface that is compatible with EIA-232 serial interfaces with the exception of voltage levels. External driver hardware may be used to adjust those voltage levels. The SCI I/O object uses the UART hardware and interrupt capability in designated Neuron Chips and Smart Transceivers. You cannot use hardware SCI and hardware SPI I/O in the same application.

The hardware SCI I/O object does not include any form of hardware flow control such as CTS/RTS flow control. If your application requires flow control, you must implement some form of handshaking in your application.

You can enable and disable SCI interrupts. For example, you can turn off interrupts when going offline or to assure that other time-critical application functions are not disturbed by SCI interrupts. The SCI interrupt signal is used by the firmware driver for the SCI I/O object. It is not directly accessible by the application program.

The SCI interrupt is enabled by default. The following function disables I/O interrupts:

void io_idis(void);

The following function enables I/O interrupts:

void io_iena(void);

You must specify the Neuron input clock rate in the Neuron C application when using hardware SCI I/O. To specify the clock rate, use the following new compiler directive:

#pragma specify_io_clock *clock-rate*

The *clock-rate* value must be one of the following quoted strings: “10 MHz”, “6.5536 MHz”, “5 MHz”, or “2.5 MHz”.

Syntax

IO_8 sci [**baud (***const-expr***)**] [**twostopbits**] *io-object-name***;**

baud (*const-expr***)**	Optionally specifies 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. This clause is optional in the declaration, but, if omitted, the **io_set_baud( )** function, below, must be used. These bit rates are accurate for devices running at Neuron input clock rates that are a multiple of 2.5 MHz. Devices using the 6.5536 MHz clock rate will be inaccurate (off by more than 3%) at baud rates of 38400 and higher since the bit rate divisor has been optimized for input clocks that are a multiple of 2.5 MHz.
twostopbits	Set this option to use two stop bits. By default, there is one stop bit.
io-object-name	Specifies a name for the I/O object, in the ANSI C format for variable identifiers.

You can call **io_set_baud(***io-obj-name***,** *rate***)** to change the bit rate for the SCI interface. The specified *rate* must be one of the enumeration values listed in the table above.

Usage

unsigned short *buffer-size***;**
unsigned short *buffer***[***buffer-size***];**

unsigned short io_in_request(*io-object-name***,** *buffer***,** *buffer-size***);**
unsigned short io_out_request(*io-object-name***,** *buffer***,** *buffer-size***);**

unsigned short io_in_ready(*io-object-name***);**
unsigned short io_out_ready(*io-object-name***);**
unsigned short sci_get_error(*io-object-name***);**
void sci_abort(*io-object-name***);**

The SCI I/O object uses pins **IO_8** for RX data (in) and **IO_10** for TX data (out).

The **io_in()** and **io_out()** functions are not available with the hardware SCI model. For input, call **io_in_request(***io-object-name***, void ****buf***, unsigned** *len***)** to set up and initiate an input operation. For output, call **io_out_request(***io-object-name***, void ****buf***, unsigned** *len***)** to set up and initiate an output operation. A call to either **io_in_request()** or **io_out_request()** will clear any previous SCI error code – see **sci_get_error()** below.

You can use the **io_in_ready(***io-object-name***)** and **io_out_ready(***io-object-name***)** event functions to test the state of the SCI interface. You can use these events to determine when the transmission is complete. The **io_out_ready** event returns TRUE when output is complete. The **io_in_ready** event returns the number of bytes read in as an **unsigned short**, so when this value matches the *len* parameter from the call to **io_in_request()** the input operation is complete.

You can use the **sci_get_error(***io-object-name***)** function to test for SCI errors. Calling the **sci_get_error()** function will clear the SCI error code after it is returned. This function returns a cumulative OR of the following bits that reflect data errors:

0x04	Framing error
0x08	Noise detected
0x10	Receive overrun detected

You can use the **sci_abort(***io-object-name***)** function to terminate any reception in progress. After an abort, the **io_in_ready()** function returns the number of characters read up to the abort.

Example

```
#pragma specify_io_clock "10 MHz"
IO_8 sci twostopbits baud(SCI_2400) iosci;
unsigned short buf[20];

when (...)
{
      io_set_baud(iosci, SCI_38400); // Optional baud change
      io_out_request(iosci, buf, 20);
}

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 ...
      }
}
```

Serial Input/Output

The **serial** I/O object type is used to transfer data using an asynchronous serial data format, as in EIA-232 (formerly RS-232) and Serial Communications Interface (SCI) communications. External driver circuitry is required to adjust the signal voltage levels to be compatible to the EIA-232 standard. The format for the transfer is: one start bit, followed by eight data bits (least significant bit first), followed by one stop bit. The input serial I/O object will wait for the start of the data frame to be received for up to the time it would take to receive 20 characters before returning a zero. Input is terminated when either the total count in bytes is received, or the amount of time it would take to receive 20 characters has passed with no data received. The input serial I/O object will stop receiving data on invalid stop bit. At 2400bps, the input timeout is 83ms.

Unlike the SCI and SPI I/O models, which are only available with certain Neuron Chip models, the serial input/output model does not require special hardware and is available with all Neuron Chip models.

Both serial input and output models are purely software I/O objects, with no hardware support other than the physical I/O pins. The serial stream is read in and transmitted out using CPU timing. See the **sci** I/O object for an equivalent I/O object that uses UART hardware on certain Smart Transceivers and Neuron Chips. The following issues should be considered when using the **serial** I/O object:

- The **io_out()** function is a blocking function, so the function won't return until the entire data set is transmitted.
- Serial input can only work successfully if the application is responsive enough to capture the start bit of the first byte received. Usually the best way to succeed with the serial input model is to employ bi-directional handshaking using two additional I/O pins, so that the sender can coordinate the transfer with the Neuron C application. If this is not possible, the serial input can be monitored with a **when(io_changes(***io-object-name***))** statement, however, you must ensure that the **io_in()** function is called less than 25% into the start bit. For example, the start bit is approximately 4.2ms at 2400bps. For reliable reception of a 240bps start bit, the **io_in()** function must be called within 1ms of the beginning of the start bit. The minimum scheduler latency is approximately 0.24ms with a 40MHz input clock, and will typically be longer depending on the number and type of **when** clauses in the application. See *I/O Timing Issues* in the Neuron Chip Smart Transceiver databooks for a description of the scheduler-related I/O timing.

When using multiple serial I/O devices that have differing bit rates, the following directive must be used:

```
#pragma enable_multiple_baud
```

This pragma must appear prior to the use of any I/O function, *e.g.* **io_in()**, **io_out()**.

For serial input/output, **io_in()** and **io_out()** require a pointer to the data buffer as the **input_value** and **output_value**. The **io_in()** function returns an **unsigned short int** that contains the count of the actual number of bytes received. See the *EIA-232C Serial Interfacing with the Neuron Chip* engineering bulletin (part no. 005-0008-01) for more information.

The serial input model provides only one bit of buffering and a maximum speed of 4800bps. For higher bit rates, use a Smart Transceiver or Neuron Chip with integrated UART hardware, such as the PL 3120 or PL 3150 Smart Transceiver. Alternatively, for bit rates up to 115.2kbps, and 16 bytes of buffering, consider using the PSG-20 or PSG/3 programmable serial gateway devices. See the *LTS-20 LonTalk Serial Adapter and PSG-20 User's Guide* for more details.

Syntax

pin **input serial** [**baud (***const-expr***)**] *io-object-name***;**

pin **output serial** [**baud (***const-expr***)**] *io-object-name***;**

pin	An I/O pin. Serial input requires one pin and must specify **IO_8**. Serial output also requires one pin and must specify **IO_10**.
baud (*const-expr***)**	Specifies the bit rate. The expression *const-expr* can be 600, 1200, 2400, or 4800. The default is 2400bps with a 10MHz input clock. The baud rate scales proportionally to the Neuron input clock.
io-object-name	A user-specified name for the I/O object, in the ANSI C format for variable identifiers.

Usage

unsigned int *count***,** *input-buffer***[***buffer-size***],** *output-buffer***[***buffer-size***];**

count **= io_in(***io-object-name***,** *input-buffer***,** *count***);**
io_out(*io-object-name***,** *output-buffer***,** *count***);**

Serial Input Example

```
IO_8 input serial io_keyboard;
char in_buffer[20];
unsigned int num_chars;

when (...)
{
      num_chars = io_in(io_keyboard, in_buffer, 20);
}
```

Serial Output Example

```
IO_10 output serial io_crt_screen;
char out_buffer[20];

when (...)
{
      io_out(io_crt_screen, out_buffer, 20);
}
```

SPI

SERIAL I/O OBJECT

You can use the hardware Synchronous Peripheral Interface (SPI) I/O object in applications that you develop for Smart Transceivers or Neuron Chips with integrated SPI hardware such as the PL 3120 or PL 3150 Smart Transceiver. SPI is a full-duplex synchronous serial communication interface initially advanced by Motorola, but now available on a wide variety of devices. The **spi** I/O object uses the SPI hardware and the I/O interrupt capability in designated Neuron Chips and Smart Transceivers. You cannot use hardware SCI and hardware SPI I/O in the same application.

The hardware SPI I/O object does not include any form of hardware flow control such as CTS/RTS flow control. If your application requires flow control, you must implement some form of handshaking in your application.

You can enable and disable SPI interrupts. For example, you can turn off interrupts when going offline, or to assure that other time-critical application functions are not disturbed by background interrupts. The SCI interrupt signal is used by the firmware driver for the SCI I/O object. It is not directly accessible by the application program.

The SPI interrupt is enabled by default. The following function disables I/O interrupts:

void io_idis(void);

The following function enables I/O interrupts:

void io_iena(void);

Syntax

IO_8 spi master|slave [**select(IO_7)**] [**clock(***const-expr***)**] [**invert**]
[**clockedge(+|-)**] [**neurowire**] *io-object-name***;**

master|slave	Determines whether the hardware is in master or slave mode, which affects the meaning of the other parameters as described below.

select(IO_7*)*	Set this option to have pin **IO_7** used as a slave select signal in slave mode or as a master detection contention signal. In slave mode, this option is only used when there are multiple slaves connected to a master. In master mode, this option is only used when there are multiple masters that may collide.
clock(*const-expr***)**	The clock selection can be an integer from 0 to 7, and selects a clock divisor for the SPI interface. This clock divisor and the input clock control the serial bit rate of the SPI interface. Clock selection applies only to master mode. At 10Mhz, the minimum serial bit rate is 19531 bps and the maximum rate is 156250 bps.
invert	Set this option to specify that the clock is idle at 1. Otherwise, the clock is idle at 0.
clockedge(+|-)	Set this option to **+** to specify that data is valid on the rising edge of the clock. Set this option to – to specify that data is valid on the falling edge of the clock. By default, data is valid on the rising edge of the clock.
neurowire	Set this option to select Neurowire compatible mode, where the MOSI and MISO pins do not change direction based on any slave select. The default is SPI mode.
io-object-name	Specifies a name for the I/O object, in the ANSI C format for variable identifiers.

You can call the **io_set_clock( )** function as shown below to change the clock divisor and clock edge at run-time. You cannot change the master/slave or Neurowire/SPI modes at run-time.

io_set_clock(*io-object-name*, **clockedge(***clock-code***));**

io_set_clock(*io-object-name*, **clockedge(***clock-code***), invert);**

The *clock-code* value can either be a single plus character ("**+**") or a single minus character ("–"), as described under the **clockedge** parameter in the table above.

Usage

unsigned short *buffer-size***;**
unsigned short *buffer***[***buffer-size***];**

unsigned short io_in(*io-object-name, buffer*, *buffer-size***);**
unsigned short io_out(*io-object-name, buffer*, *buffer-size***);**

unsigned short io_in_ready(*io-object-name***);**
unsigned short io_out_ready(*io-object-name***);**
unsigned short sci_get_error(*io-object-name***);**
void sci_abort(*io-object-name***);**

The SPI I/O object uses pins **IO_8** as Clock output, **IO_9** as Data input, and **IO_10** as Data output in **master** mode. In **slave** mode **IO_8** is Clock input, **IO_9** is Data output, and **IO_10** is Data input. Using **neurowire** mode, **IO_9** is Data output and **IO_10** is Data input.

You can use the **io_in()** and **io_out()** functions to read and write a hardware SPI interface. This interface is very similar to the **neurowire** I/O object. The **io_in()** and **io_out()** calls are functionally equivalent, since SPI input and output occur simultaneously. Since the SPI interface is full duplex, the same buffer is used for both transmission and reception of data, with data transferred serially out of and into the single data buffer at the same time. You can use either **io_in()** or **io_out()** to initiate an I/O operation:

io_in(*io-object-name***, void *** *buf***, unsigned** *len***)**

io_out(*io-object-name***, void *** *buf***, unsigned** *len***)**

The **io_in()** and **io_out()** functions are non-blocking; they just initiate the data transfer. You can use the **io_in_ready(***io-object-name***)** and **io_out_ready(***io-object-name***)** event functions to test the state of the SPI interface. These functions are used to determine when the transmission is complete. The **io_out_ready** event returns TRUE when output is complete. The **io_in_ready** event returns the number of bytes read in as an **unsigned short**, so when this value matches the *len* parameter from the call to **io_in_request()** the input operation is complete.

You can use the **spi_get_error(***io-object-name***)** function to test for SPI errors. Calling **io_in()** or **io_out()** will clear any previous SPI error code. The **spi_get_error()** function also will clear any SPI error code after returning it. This function returns a cumulative OR of the following bits that reflect data errors:

0x10	Mode fault occurred
0x20	Receive overrun detected

You can use the **spi_abort(***io-object-name***)** function to terminate any operation in progress. After an abort, the **io_in_ready()** function returns the number of characters read up to the abort.

Example

```
IO_8 spi master clock(4) iospi;
unsigned short buf[20];

when (...)
{
      io_out(iospi, buf, 20);
}

when (io_out_ready(iospi))
{
      unsigned short spi_error;
      spi_error = spi_get_error(iospi));
      if (spi_error) {
            // Process SPI error
      } else {
            // Process end of SPI transmission ...
      }
}
```

Totalcount Input — TIMER/COUNTER I/O OBJECT

The **totalcount** I/O object type counts the number of input edges at the input pin since the last **io_in()** operation, or since initialization. For totalcount input, the data type of **return_value** for **io_in()** is an **unsigned long**.

The minimum duration for a high or low input signal for this I/O object is 50ns with a 40MHz Neuron input clock. This value scales inversely with the input clock speed.

Syntax

pin [**input**] **totalcount** [**mux** | **ded**] [**invert**] *io-object-name***;**

pin	An I/O pin. Totalcount input can specify pins **IO_4** through **IO_7**.
mux | **ded**	Specifies whether the I/O object is assigned to the multiplexed or dedicated timer/counter. This field is used only when pin **IO_4** is used as the input pin. The **mux** keyword assigns the I/O object to the multiplexed timer/counter. The **ded** keyword assigns the I/O object to the dedicated timer/counter. The multiplexed timer/counter is always used for pins **IO_5** through **IO_7**.
invert	Causes positive edges to be counted. By default, totalcount input counts the number of negative input edges.
io-object-name	A user-specified name for the I/O object, in the ANSI C format for variable identifiers.

Usage

unsigned long *input-value***;**

input-value = **io_in(***io-object-name***);**

Example

```
IO_4 input totalcount ded io_event_count;
unsigned long total_num_events = 0;
mtimer repeating t;

when (timer_expires(t))
{
      total_num_events += io_in(io_event_count);
         // this sums up all events since initialization-time
}
```

Touch Input/Output   DIRECT I/O OBJECT

The **touch** I/O object type is used to interface to the 1-WIRE protocol developed by Dallas Semiconductor Corporation to communicate with Touch Memories and similar devices. The touch I/O object will only operate within the timing specifications set forth by Dallas Semiconductor Corporation for the 1-WIRE protocol at Neuron input clock rates of 10MHz or 5MHz. This interface supports bi-directional data transfers across a signal and ground wire pair. An external pull-up is required, and the interface is connected directly to the designated I/O pin. This I/O pin is operated as an open-drain device in order to support the interface.

Up to 255 bytes of data may be transferred at a time.

For more information on this protocol and the devices that it supports, see the publication *Book of DS19xx Touch Memory Standards*, Dallas Semiconductor Corporation, Edition 2.0 or later.

Syntax

pin **touch** [**output_pin(***pin***)**] [**timing(***t-low***,** *t-rdi***,** *t-wrd***)**] *io-object-name***;**

pin	An I/O pin. Touch I/O can specify one of the pins **IO_0** through **IO_7**. Multiple Touch I/O objects can be declared. If you do not explicitly declare a separate output pin with the **output_pin( )** parameter, this pin specifies both the input and output pin. Otherwise, it specifies only the input pin.
output_pin(*pin***)**	Optionally specifies the output pin. If not specified, the output pin is the same as the input pin.

timing(...)

Optionally specifies three timing parameters. There are three time periods associated with each bit time slot. All values here apply to a 10MHz Neuron input clock and will *double* for a 5MHz input clock. Since these timing controls affect the low-level single bit function used by both read and write operations they are required for both 1-wire read and write operations. A value of 0 for a timing control is the same as 256.

You can optionally specify the following three timing parameters when you declare the **touch** I/O object:

t-low The length of t_{LOW}. This is the interval where the Neuron firmware asserts a low on the 1-wire bus signaling the start of the bit slot. This argument has a minimum value of 7.2µs (*t_low* = 1) from the start of t_{LOW}. The incremental resolution of *t_low* is 3µs, so the control range is 4.2 + n * 3 (in µs) where n is 1 to 255, and a *t-low* value of 0 is equivalent to n=256.

t-rdi The length of t_{RDI}. This is the interval where the Neuron firmware asserts either a low or a high on the 1-wire bus, depending on the output data bit polarity. For read operations this data polarity is always high. This argument has a minimum value of 7.8µs (*t-rdi* = 1) from the start of t_{RDI}. The incremental resolution of *t-rdi* is 3µs, so the control range is 4.8 + n * 3 (in µs) where n is 1 to 255, and a *t-rdi* value of 0 is equivalent to n=256.

t-wrd Start of t_{WRD} (end of t_{RDI}). This is the point where the Neuron firmware samples the 1-wire bus for the input data bit, and occurs for both read and write operations. This argument has a minimum value of 15µs (*t-wrd* = 1) from the start of t_{WRD}. The incremental resolution of *t-wrd* is 3µs, so the control range is 12 + n * 3 (in µs) where n is 1 to 255, and a *t-wrd* value of 0 is equivalent to n=256.

At the end of *t-wrd*, the Neuron firmware releases the 1-wire bus.

io-object-name

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

Usage

unsigned int *count*;

unsigned int *touch-buffer*[*buffer-size*];

io_out(*io-object-name*, *touch-buffer*, *count***);**
io_in(*io-object-name*, *touch-buffer*, *count***);**

The *touch-buffer* can be any type or structure. The address of the buffer is passed to the **io_out( )** and **io_in( )** functions.

There are several additional support functions for the Touch I/O object. They are the following:

int touch_reset(*io-object-name***);**

The **touch_reset( )** function asserts the reset pulse and returns a 1 value if a presence pulse was detected, or a 0 value if no presence pulse was detected, or a -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 500µs. Next, the Neuron firmware releases the 1-WIRE bus and waits for the 1-WIRE bus to return to the high state. This period is limited to 275µs, after which the **touch_reset( )** function will return a -1 value with the assumption that the 1-WIRE bus is stuck low. There also is a minimum value for this period, it must be >4.8µs @10MHz, or 9.6µs @5MHz.

Once the 1-WIRE bus has appeared to go high, the Neuron firmware waits for the presence pulse for a period up to 80µs. If a low input level is not sensed within this period the function returns a 0 value. Once a presence pulse is detected the Neuron firmware then waits for the end of the presence pulse by waiting for a high level on the bus. This period is limited to 250µs, after which the function will again return a -1 if the period elapses with the input level still low. Otherwise, once the input level is high again the function returns with a 1 value.

The **touch_reset( )** function does not return until the end of the presence pulse has been detected.

unsigned touch_byte(*io-object-name*, **unsigned** *write-data***);**

The **touch_byte( )** function sequentially writes and reads eight bits of data on the 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 will contain 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 will normally contain those same bits.

This function allows combined read and write operations within a single eight-bit boundary. For example, a 2-bit write may be followed by a 6-bit read. This can be accomplished with a single call to **touch_byte()** with a *write-data* argument of **0b***NN***111111** where (*NN*) represents the 2 bits of write data and (**111111**) is used to perform the 6-bit read.

unsigned touch_bit(*io-object-name***, unsigned** *write-data***);**

The **touch_bit()** function writes and reads a single bit of data on the 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 will contain 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 will normally contain that same bit value, and can be ignored. This function provides access to the same internal process that **touch_byte()** calls.

int touch_first(*io-object-name***, search_data *** *sd***);**

int touch_next(*io-object-name***, search_data *** *sd***);**

These functions execute the ROM Search algorithm as described in *Book of DS19xx Touch Memory Standards*, Dallas Semiconductor, Edition 2.0. 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.

```
typedef struct search_data_s {
      int  search_done;
      int  last_discrepancy;
      unsigned rom_data[8];
} search_data;
```

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. Each call to **touch_first()** or **touch_next()** will take 41ms to execute at 10MHz (63ms at 5MHz) when a device is being read.

unsigned crc8(unsigned *crc***, unsigned** *new-data***);**

This function performs the Dallas 1-WIRE CRC-8 function on the *crc* and *new-data* arguments, and returns the new 8-bit CRC value. You must include **<stdlib.h>** to use this function.

unsigned long crc16(unsigned long *crc***, unsigned** *new-data***);**

This function performs the Dallas 1-WIRE CRC-16 function on the *crc* and *new-data* arguments, and returns the new 16-bit CRC value. You must include **<stdlib.h>** to use this function.

Example

```
// In this example a leveldetect input is used on the 1-WIRE
// interface in order to detect the 'presence' signal when a
// Touch Memory device appears on the bus.

#include <stdlib.h>

#define DS_READ_ROM 0x33
unsigned int id_data[8];
IO_3 input leveldetect io_twire_pres;
IO_3 touch io_twire;
. . .


when (io_in(io_twire_pres) == 1)
{
    unsigned int i, crc_data;

    // Reset the device using touch_reset().
    // Skip if there is no device sensed.
    if (touch_reset(io_twire)) {
        // Send a single READ_ROM command byte:
        id_data[0] = DS_READ_ROM;
        io_out(io_twire, id_data, 1);
        // Read the 8 byte I.D.:
        io_in(io_twire, id_data, 8);
        // check the crc of the I.D.:
        crc_data = 0;
        for (i=0; i<7; i++)
            crc_data = crc8(crc_data, id_data[i]);
        if (crc_data == id_data[7]) {
            // Valid crc: process I.D. data here.
        }
    }
    // Clear leveldetect input.
    (void)io_in(io_twire_pres);
}
```

Detailed timing specifications for these operations exist and can be found in the Neuron Chip and Smart Transceiver databooks.

Triac Output

The **triac** I/O object type is used to control the delay of an output pulse signal with respect to an input trigger signal. For control of AC circuits using a triac I/O object, the sync input is typically a zero-crossing signal, and the pulse output is the triac trigger signal. The output pulse is 25µs wide, normally low. The pulsewidth is independent of the Neuron input clock.

Execution of this I/O object type is synchronized with the sync pin input and may not return for up to 10ms. (The application program could thus be delayed for as long as 10ms.) Because of this synchronization, the frequency of the sync pin input (and the frequency of the AC circuit being controlled) is limited to the 50-60Hz range.

When using the pulse output configuration, an output value of 65535 (the overrange value) assures that no output pulse is generated. This is the equivalent of an OFF state. When using the level output configuration, there always will be some amount of output signal; use an output value that is about 95% of the half-cycle period to approximate the OFF state.

Syntax

pin [**output**] **triac** [**pulse** | **level**] **sync (***pin-nbr***)** [**invert**]
[**clock (***const-expr***)**] [**clockedge (+)** | **(-)** | **(+-)**] *io-object-name***;**

pin	An I/O pin. Triac output can specify pins **IO_0** or **IO_1**. If **IO_0** is specified, the sync pin can be **IO_4** through **IO_7**. If **IO_1** is specified, the sync pin must be **IO_4**.
sync (*pin-nbr***)**	Specifies the sync pin, which is the input trigger signal.
invert	Causes the output signal to be inverted, normally high. The default output signal is normally low.
clock (*const-expr***)**	Specifies a clock in the range 0 to 7, where 0 is the fastest clock and 7 is the slowest clock. The default (and recommended) clock for triac output is clock 7. The **io_set_clock( )** function can be used to change the clock.

Other clock values are as follows for a Neuron input clock of 10MHz:

Clock	***Pulse Delay***
0	0 to 13.11ms in steps of 200 ns (0-65535)
1	0 to 26.21ms in steps of 400 ns
2	0 to 52.421ms in steps of 800 ns
3	0 to 104.86ms in steps of 1.6 μs
4	0 to 209.71ms in steps of 3.2 μs
5	0 to 419.42ms in steps of 6.4 μs
6	0 to 838.85ms in steps of 12.8 μs
7 (default)	0 to 1.677sec in steps of 25.6 μs

clockedge (+) | (-) | (+-)

(+) Causes the sync input to be positive-edge sensitive.

(-) This is the default, and it causes the sync input to be negative-edge sensitive.

(+-) Causes the sync input to be both positive- and negative-edge sensitive (valid on all Neuron 3120xx Chips, all models of Neuron 3150 Chips except minor model 0, and all Smart Transceivers). Can be used with pulse mode only.

NOTE: The **clockedge (+-)** option does not work with minor model 0 of Neuron 3150 Chips. When using a Neuron 3150 Chip, a 3150 Smart Transceiver, or a LonBuilder emulator, the compiler inserts code in the application that checks for the availability of this feature. This code logs an error if the chip does not support the feature.

io-object-name

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

[**pulse** | **level**]

Specifies whether the output signal produces a 25μs pulse at the delay point, or a level, which stays on from the delay point until the next sync input edge.

When using the pulse output configuration the output pulse is generated by an internal clock with a constant period of 25.6μs (independent of the Neuron input clock). Since the input sync edge is asynchronous relative to the internal clock there is a jitter associated with the pulse output relative to the input sync edge. This jitter will span a period of 25.6μs.

Usage

unsigned long *output-value***;**

io_out(*io-object-name***,** *output-value***);**

Example 1

```
IO_0 output triac sync (IO_5) io_dimmer_trigger;

when (...)
{
      io_out(io_dimmer_trigger, 325);
                        // delay pulse by 8.3 ms
}

when (...)
{
      io_out(io_dimmer_trigger, 650);
                        // delay pulse by 16.6 ms
}

when (...)
{
      io_out(io_dimmer_trigger, 0); // full on
}
```

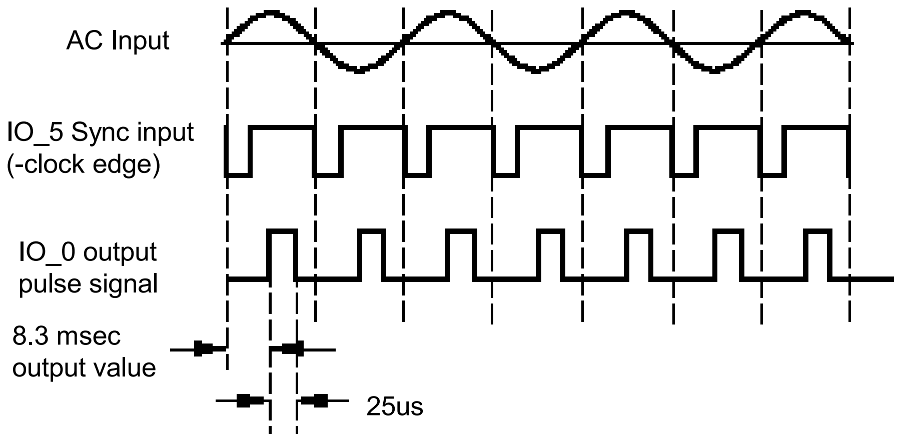

Figure 8.4 Triac Output, Example 1

Example 2

```
IO_1 output triac sync (IO_4) clockedge (+-) io_dimmer_2;
...
io_out(io_dimmer_2,325);
```

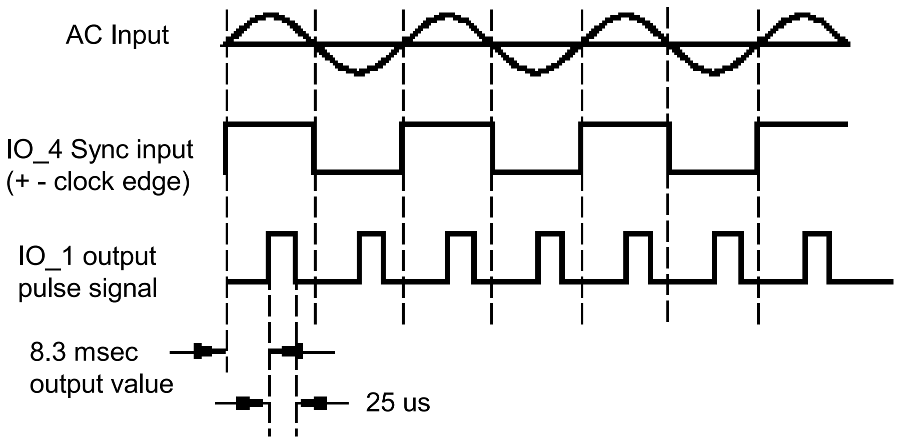

Figure 8.5 Triac Output, Example 2
(except for model 0 Neuron 3150 Chips)

Triggeredcount Output   TIMER/COUNTER I/O OBJECT

The **triggeredcount** I/O object type is used to control an output pin to the active state and keep it active until *output-value* negative edges are counted at the input sync pin. After *output-value* edges have counted off, the output pin returns to the low state.

For triggeredcount output, the data type of *output-value* for **io_out()** is an **unsigned long**. An *output-value* of 0 forces the output signal to an inactive state.

Syntax

pin [**output**] **triggeredcount sync (***pin-nbr***)**
[**invert**] *io-object-name* [=*initial-output-level*] **;**

pin	An I/O pin. Triggeredcount output can specify pins **IO_0** or **IO_1**. If **IO_0** is specified, the multiplexed timer/counter is used and the sync pin can be **IO_4** through **IO_7**. If **IO_1** is specified, the dedicated timer/counter is used and the sync pin must be **IO_4**.
sync (*pin-nbr***)**	Specifies the sync pin, which is the counting input signal with low pulses.

invert	Causes the output signal to be inverted, normally high. By default, the output signal is normally low with high pulses.
io-object-name	A user-specified name for the I/O object, in the ANSI C format for variable identifiers.
initial-output-level	A constant expression, in ANSI C format for initializers, used to set the state of the output pin of the I/O object at initialization. The initial state may be 0 or 1. The default initial state is 0.

In Figure 8.6, an **io_out()** function call is executed with a count argument of 11. After 11 negative edges at the input pin, the output goes low. The delay from the last input edge to the output falling edge is 50ns or less at a Neuron input clock of 40MHz.

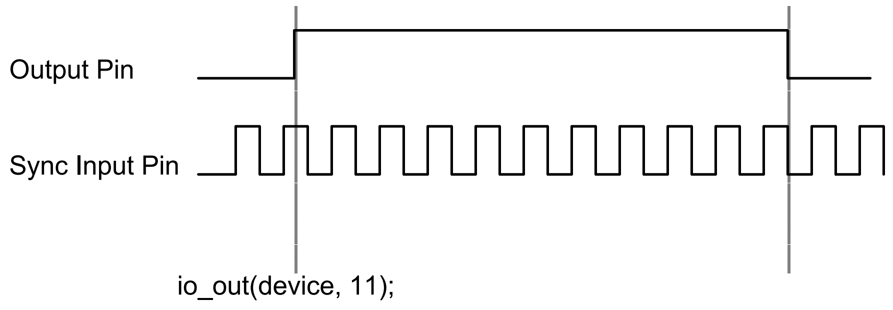

Figure 8.6 Triggeredcount Output Object

Usage

unsigned long *output-value*;

io_out(*io-object-name*, *output-value***)**;

Example

```
IO_0 output triggeredcount sync (IO_4) io_cascader;

when (...)
{
      io_out(io_cascader, 10);
            // 1 big output pulse for 10 input pulses
}
```

Wiegand Input

The **wiegand** I/O object type is used to transfer data from a Wiegand format data stream source. This format encodes data as a series of pulses on two signal lines: one which is designated as the zero data bit signal; and another which is designated as a one data bit signal. Data pulses appear exclusively of each other and are typically spaced approximately 1ms apart. Specifications for the duration of the pulse are typically between 50 to 100µs, but they can be as short as 50ns with a 40MHz input clock. The inter-bit period (the period between bit pulses) is shown in the table below:

Parameter	***Min***	***Max***	***Typical***
Pulse Width	200ns	880ms	100µs
Inter-Bit Time	150µs	(none)	900µs

Wiegand data is asynchronous. The **io_in()** function must be executing before the second bit arrives, otherwise the first bit data is lost since it then becomes impossible to determine the order of a zero and one event sequence.

Data is read MSB first, that is, the first data bit read will be stored in the most significant bit location of the first byte of the array when eight bits are read into that byte. If the number of bits transferred is not a multiple of eight, as defined by *count*, the last byte transferred into the array will contain the remaining bits right justified within the byte.

For Wiegand input, one of the **IO_0** through **IO_7** pins may optionally be designated as a timeout pin. A logic one level on the timeout pin causes the Wiegand input operation to be terminated before the specified number of bits has been transferred. The Neuron Chip or Smart Transceiver updates the watchdog timer while waiting for the next zero or one data bit to arrive. This timeout input can be a one-shot timer counter output, an RC circuit, or a ~Data_valid signal from the reader device.

The **return_value**, which is an **unsigned short**, for the **io_in()** function for this object, indicates the number of bits stored into the array. Whenever the **io_in()** function for this object is called it will immediately return if there is currently no activity on the indicated I/O pins. Otherwise, the function will continue to process input data until either *count* bits are stored, or until the timeout event occurs. When the timeout event occurs the number of bits read and stored is returned.

The **io_in()** function is blocking, and may take more than one second to process the card information, depending upon the speed at which the card travels through the reader. Because this function ties up the application processor, it takes care of updating the watchdog timer.

Syntax

pin [**input**] **wiegand** [**timeout(***pin-nbr***)**] *io-object-name***;**

pin	An I/O pin. Wiegand input requires two adjacent pins. The DATA 0 pin is the pin specified, and the DATA 1 pin is the following pin. The pin specification denotes the lower-numbered pin of the pair and can be **IO_0** through **IO_6**.
timeout (*pin*-nbr**)**	Optionally specifies the timeout signal pin, in the range of **IO_0** to **IO_7**. The Neuron firmware checks the logic level at this pin whenever it is waiting for a pulse at either the DATA 0 or DATA 1 pins. If a logic level 1 is sensed, the transfer is terminated.
io-object-name	A user-specified name for the I/O object, in the ANSI C format for variable identifiers.

Usage

unsigned int *count, input-buffer***[***buffer-size***]**, *bit-count***;**

count **= io_in(***object-name, input-buffer, bit-count***);**

Example

```
// This application is written so that the
// Wiegand input is being polled for a majority
// of the time, breaking out and returning to the
// scheduler only periodically.  This makes the
// probability of capturing the first bits of
// the input much higher since the bits arrive
// asynchronously.  Timeout is from a hardware oneshot.
unsigned int wieg_array[4], breaker, nbits;
IO_2 input wiegand timeout (IO_0) io_card_data;
IO_0 output oneshot invert clock (7) io_pintimer = 1;
when(TRUE)
{
    for (breaker=200; breaker; breaker--) {
        io_out(io_pintimer, 19500UL);
        // Store 26 bits into wieg_array
        nbits = io_in(io_card_data, wieg_array, 26);
        if (nbits) {
          . . . // Process data just read
        }
    }
}
```

Appendix A

Syntax Summary

This appendix provides a summary of Neuron C Version 2.1 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 with headers 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 may appear in any order. Floating-point types (**double** and **float**) are not supported in Neuron C.

type-keyword :

char
double (This keyword is reserved for future implementations)
float (This keyword is reserved for future implementations)
int
long
quad (This keyword is 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*]   (configuration NVs (CPNVs))
    cp_family [*cp-info*] [*range-mod*]   (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** clause list, followed by a task. A task is a compound statement (like an ANSI C function body).

task-declaration :

> *when-clause-list 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*

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 may evaluate to either TRUE or FALSE. This 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 may also create custom events by using any parenthesized expression as an event, including one or more function calls, etc.

when-event :
  (reset)
  predefined-event
  parenthesized-expr

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**

I/O Object Declarations

An I/O object declaration is similar to an ANSI C variable declaration. It may 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 | triggeredcount
    wiegand

I/O Options

Most I/O options only apply to a few specific object types. The detailed reference documentation in Chapter 8, *I/O Objects* will explain each option that applies for that I/O object.

io-option :

baud (*constant-expr* **)**
clock (*constant-expr* **)**
clockedge (*clock-edge* **)**
ded
invert
kbaud (*constant-expr* **)**
long
master
mux
numbits (*constant-expr* **)**
select (*io-object-pin-name* **)**
short
single_tc
slave
slave_b
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 may only have at most one property list, there may be any number of device property list declarations throughout a program, and the lists will be 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 <u>previously declared</u> network variable configuration property or the name of a previously declared CP family. The instantiation may 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 :
    <u>one of</u>   **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*
> *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**

builtin-variables :

  <u>one of</u> **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 :

<u>one of</u> **msg_in** | **msg_out** | **resp_in** | **resp_out**

builtin-functions :

<u>one of</u>
abs | **addr_table_index**
bcd2bin | **bin2bcd**
eeprom_memcpy
fblock_director
get_fblock_count | **get_nv_count**
get_tick_count
high_byte
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**
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**

msg-call-kwd :

<u>one of</u>
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
```

Appendix B

Reserved Words

This chapter lists all Neuron C Version 2.1 reserved words, including the standard reserved words 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. The code “(c)” indicates a keyword from the ANSI C language. Code “(1)” indicates keywords from Neuron C Version 1 and “(2)” indicates a keyword introduced in Neuron C Version 2. Code "(2.1)" indicates a keyword introduced in Neuron C Version 2.1.

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)

ACKD (e)

COMM_IGNORE (e)

FALSE (e)

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_DIR_IN (e)

IO_DIR_OUT (e)

PULLUPS_ON (e)

REQUEST (e)

TIMERS_OFF (e)

TRUE (e)

UNACKD (e)

UNACKD_RPT (e)

abs (f)

ackd (1)

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)

by (1)

byte (1)

case (c)

changeable_type (2)

char (c)

charge_pump_enable (f)

clock (1)

clockedge (1)

config (1)

config_prop (2)

const (c)

continue (c)

cp (2)

cp_family (2)

cp_info (2)

cp_modifiable_value_file (v)

cp_modifiable_value_file_len (v)

cp_readonly_value_file (v)

cp_readonly_value_file_len (v)

cp_template_file (v)

cp_template_file_len (v)

ded (1)

default (c)

delay (1)

device_properties (2)

device_specific (2)

director (2)

do (c)

double (c)

dualslope (1)

edgedivide (1)

edgelog (1)

eeprom (1)

eeprom_memcpy (1)

else (c)

enum (c)

expand_array_info (2)

extern (c)

external_name (2)

external_resource_name (2)

far (1)

fastaccess (1)

fb_properties (2)

fblock (2)

fblock_director (f)

fblock_index_map (v)

float (c)

flush_completes (w)

for (c)

frequency (1)

get_fblock_count (f)

get_nv_count (f)

global (2)

global_index (p)

goto (c)

high_byte (f)

i2c (1)

if (c)

implementation_specific (2)

implements (2)

infrared (1)

infrared_pattern (2.1)

input (1)

input_is_new (v)

input_value (v)

int (c)

invert (1)

io_change_init (f)

io_changes (w)

io_direction (et,t)

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_update_occurs (w)

is_bound (f)

kbaud (1)

level (1)

leveldetect (1)

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)

mux (1)

muxbus (1)

network (1)

neurowire (1)

nibble (1)

nonauth (1)

nonauthenticated (1)

nonbind (1)

nonconfig (1)

nonpriority (1)

numbits (1)

nv_array_index (v)

nv_in_addr (v)

nv_in_addr_t (st,t)

nv_in_index (v)

nv_len (p)

nv_properties (2)

nv_table_index (f)

nv_update_completes (w)

nv_update_fails (w)

nv_update_occurs (w)

nv_update_succeeds (w)

object_disabled (2)

offchip (1)

offline (w,2)

onchip (1)

oneshot (1)

online (w)

ontime (1)

output (1)

output_pin (2.1)

parallel (1)

period (1)

poll (f)

polled (1)

preempt_safe (1)

priority (1)

propagate (f)

pulse (1)

pulsecount (1)

pulsewidth (1)

quad (1)

quadrature (1)

ram (1)

random (f)

range_mod_string (2)

rate_est (1)

register (c)

repeating (1)

reset (w)

reset_required (2)

resp_alloc (f)

resp_arrives (w)

resp_cancel (f)

resp_free (f)

resp_in (v)

resp_out (v)

resp_receive (f)

resp_send (f)

return (c)

reverse (f)	**spi** (2.1)	**touch_first** (f)
scaled_delay (f)	**spi_abort** (2.1)	**touch_next** (f)
sci (2.1)	**spi_get_error** (2.1)	**touch_reset** (f)
sci_abort (2.1)	**static** (c)	**triac** (1)
sci_get_error (2.1)	**stimer** (1)	**triggeredcount** (1)
sd_string (1)	**struct** (c)	**twostopbits** (2.1)
search_data (t)	**swap_bytes** (f)	**typedef** (c)
search_data_s (st)	**switch** (c)	**unackd** (1)
select (1)	**sync** (1)	**unackd_rpt** (1)
serial (1)	**synchronized** (1)	**uninit** (1)
service_type (et,t)	**system** (1)	**union** (c)
short (c)	**timeout** (1)	**unsigned** (c)
signed (c)	**timer_expires** (w)	**use_stop_condition** (2.1)
single_tc (2.1)	**timing** (2.1)	**void** (c)
sizeof (c)	**to** (1)	**volatile** (c)
slave (1)	**totalcount** (1)	**when** (1)
slave_b (1)	**touch** (1)	**while** (c)
sleep (f)	**touch_bit** (f)	**wiegand** (1)
sleep_flags (et,t)	**touch_byte** (f)	**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

_byte_input

_byte_output

_dualslope_input

_dualslope_start

_edgelog_input

_flush_completes

_frequency_output

_i2c_read

_i2c_write

_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_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

_memcpy

_memcpy16

_memcpy8

_memset

_msg_arrives

_msg_cancel

_msg_auth_get

_msg_auth_set

_msg_code_arrives

_msg_code_get

_msg_code_set
_msg_completes
_memset16
_memset8
_msg_addr_blockget
_msg_addr_blockset
_msg_addr_get
_msg_addr_set
_msg_alloc
_msg_alloc_priority
_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
_wiegand_input
_wink

alt_i2c_read	**edgelog_input_single**	**ext_touch_read**
alt_i2c_write	**edgelog_setup_single**	**ext_touch_reset**
burst_sequence_output	**ext_touch_bit**	**ext_touch_write**
cp_modifiable_value_file_len_fake	**ext_touch_byte**	**i2c_read_opt**
cp_readonly_value_file_len_fake	**ext_touch_first**	**i2c_write_opt**
cp_template_file_len_fake	**ext_touch_next**	**magx_input**

Index