

[10] Programming Model

The programming language used to write applications for the Neuron Chip is a derivative of the ANSI C programming language called Neuron C. Neuron C is based on ANSI C, enhanced to support I/O, event processing, message passing, and distributed data objects. Several major differences exist between Neuron C and ANSI C in the data types supported. The numeric data types supported by Neuron C are the following.

char	8 bits	signed or unsigned
short	8 bits	signed or unsigned
int	8 bits	signed or unsigned
long	16 bits	signed or unsigned

Neuron C also supports *typedefs*, *enums*, arrays, pointers, *structs*, and *unions*. Unlike ANSI C, Neuron C does not include a standard run-time library supporting file I/O and other features common to larger target processors, such as floating point arithmetic. However, Neuron C has a special run-time library and language syntax extensions supporting intelligent distributed control applications using Neuron Chips. Neuron C extensions include software timers, network variables, explicit messages, a multi-tasking scheduler, EEPROM variables, and miscellaneous functions. An extended arithmetic function library implementing IEEE754 floating point, 32-bit fixed point, and string operations is also available within Neuron C. For further details on the Neuron C programming language, see the *Neuron C Programmer's Guide*.

1. Timers

An application program can use up to fifteen timers that decrement either every second or every millisecond and optionally repeat. These timers are implemented in software running of the Network CPU, and are independent of the Neuron Chip input clock rate. The expiration of a timer is an event that may cause the execution of a user-written task (see below). This event is called timer expires. The value of a timer variable is an unsigned long with a value of (0-65,535). Note that when operating at a 20-MHz input clock, the value of the milli timer variable is a value between 0 to 32,767. The milli timer variable value is set between 0 to 65,535.

Timers can be set to any value in this range at any time by the application program.

2. Network Variables

The application program can declare a special class of static objects called network variables, which may be of input or output class. Assignment of a value to an output network variable causes propagation of that value to all nodes declaring an input variable that is connected to the output network variable. For example, a node that contains a temperature sending device could declare an output network variable which contains the current temperature sensed by the node. Every time the node measures a new value for the temperature, it updates the output network variable. Another node or nodes needing to know the current temperature, such as a heating control node, can then declare an input network variable for current temperature. Whenever the heating control node wants to use the value of the current temperature, it simply refers to the input network variable which will always contain the last temperature measured by the sensing node. At installation time, the output network variable on the sensing node is connected to the input network variable on the controlling node.

Binding is the process of connecting network variables from different nodes together, and is typically performed by a network management tool. The LonBuilder Developer's Workbench, the LONWORKS Network Service (LNS) Products and the LonMarker™ for Windows all include such a binding capability. The binding is physically implemented by sending network management messages containing the necessary addressing information to the nodes to be bound. Nodes may also update their own binding information for simple networks, without network management tools. Tables containing binding information are in the EEPROM, and hence may be written after the node has been manufactured.

Nodes declaring an input network variable need only refer to that variable to determine the latest value propagated across the network. A node declaring an input network variable may also call the library routine *"poll ()"* to cause the latest value to be propagated to it. The library routine *"is_bound ()"* may be used to check whether a particular network variable is connected to (bound to) a certain other network variable on another node.

Note that declaring network variables within a node's code occurs at compile time. The binding of the network variable outputs from one node to inputs on other nodes occurs at a later time, which may be either before, during, or after the node is installed in a network.

The network variable concept greatly simplifies the programming of complex distributed applications. Network variables provide for very flexible applications of distributed data to be operated on by the nodes in a system. The programmer need not deal with message buffers, node addressing, request/response/retry processing, or other low-level details.

A node running a Neuron C application program may declare up to 62 network variables, including array elements. In most cases, this is not a significant limitation, since a single input network variable can receive data of the same type from an unlimited number of other nodes, and a single output network variable can send data of the same type to an unlimited number of other nodes. Additional network variables can be accessed from Neuron C by explicitly building a network variable update or fetch message as described in section B.3. A network variable may be a Neuron C variable or structure up to 31 bytes in length. Arrays of up to 31 bytes may be embedded in a structure and propagated as a single network variable. If more than 31 bytes of data are needed in a single message, explicit messaging can be used as described in the next section. A network variable may also be an array of elements, each of which may be individually connected to network variables on other nodes. An output network variable on a node may also be connected to an input network variable on the same node (turnaround network variable).

Nodes may be implemented with all their applications and communications processing running on a Neuron Chip. Such nodes are called *Neuron Chip-hosted nodes* and are programmed using the Neuron C programming language.

Nodes may also be implemented using the Neuron Chip as a communications processor and a second processor as the host for the applications processing. Such nodes are called *host-based nodes* and are programmed using the native programming tools of the host processor. The host may be a microcontroller, microprocessor, PC, workstation, or any other computer. The host communicates with a LONWORKS network via a *LONWORKS network interface*.

A LONWORKS network interface implements layers 1 through 5 of the LonTalk protocol, and moves layers 6 and 7 application support to the host processor. The LonTalk protocol includes the specification of a standard protocol between the host and network interface, which is called the LONWORKS network interface protocol.

Turn-key LONWORKS network interfaces are available, such as Echelon's PCC-10 PC Card Network Adapter, PCLTA PC LonTalk Adapter, and SLTA Serial LonTalk Adapter, to provide a ready-made network interface. A custom LONWORKS network interface can also be built using the LONWORKS Microprocessor Interface Program (MIP). The MIP is firmware for the Neuron Chip, that transforms the Neuron Chip into a high-performance communications processor for an attached host.

When using a LONWORKS network interface, network variables are moved to the host processor. Network variable configuration management may be performed entirely by the host (called *host selection*), or may be split between the host and network interface (called *network interface selection*). When host selection is used, the host application can implement up to 4096 bound network variables, versus the 62 bound network variables for Neuron Chip-hosted nodes. For further details, see the *Host Application Programmer's Guide* and the *MIP User's Guide*.

Network variable updates may be sent with four classes of service:

<i>Acknowledged</i>	Acknowledged service with retries
<i>Unacknowledged</i>	Unacknowledged service
<i>Repeated</i>	Unacknowledged repeated service (message sent multiple times)
<i>Request/Response</i>	Request/response service, used for polling network variables

Network variables may be specified as authenticated, meaning that only authenticated messages are used to transmit their values (see section 6). Network variables may also be specified to have priority, meaning that a priority time slot is used to transmit their values (see section 7). Finally, network variables may be specified as synchronous, in which case all values assigned to the network variable are propagated. Normally when network variable updates are generated faster than they can be propagated, the network variables propagate only their most recent value. Intermediate values may be discarded.

The following built-in events may be checked by the scheduler to allow asynchronous processing of network variables:

<i>nv_update_occurs</i>	A new value has been received for an input network variable
<i>nv_update_fails</i>	Propagation of the value of an output network variable has failed
<i>nv_update_succeeds</i>	Propagation of the value of an output network variable has succeeded
<i>nv_update_completes</i>	Propagation of the value of an output network variable has completed, either successfully or unsuccessfully

Completion status is binary. A network variable update either succeeded or it did not. With acknowledged service, a network variable update succeeds only if acknowledgements are received from all the recipients. With unacknowledged (repeated) service, a network variable update succeeds only if the update message was transmitted onto the network. Additional information about the source of failures can be derived by examining node statistics using the *Query Status* network management message.

If the destination node application wishes to know the source address of an incoming network variable update message, it can refer to the built-in variable `nv_in_addr`. The definition of this variable is in the `Echelon.H` include file and is reproduced here for reference.

```
const struct {
    unsigned domain      : 1;           // domain table reference
    unsigned flex_domain : 1;           // received on flexible domain
    unsigned format      : 6;           // 0 = broadcast, 1 = group,
                                         // 2 = subnet/node, 3 = NEURON ID
                                         // 4 = turnaround

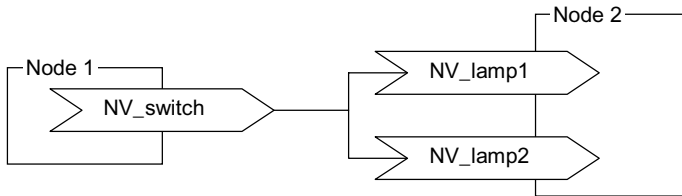
    struct {
        unsigned subnet;
        unsigned      : 1;
        unsigned node  : 7;
    } src_addr;                          // source address
    struct {
        unsigned group;                  // group destination address
    } dest_addr;
} nv_in_addr;
```

3. Network Variable Aliases

When using network variables, connection between network variables is limited due to restrictions on selector numbers (A.4.1). For example, to connect as shown in Figure 3.1 (a), connection cannot be made due to a restriction on the node 2 selector number. In such a case, two output network variables (NV_switch) can be used to solve the problem; however, the application program must be greatly modified.

To avoid such restrictions, at network variable connection, instead of modifying application programs, use network variable aliases to increase network variables with the same meaning. (example: Figure 3.1 (b))

(a) Example where connection is restricted



(b) Example where alias is used

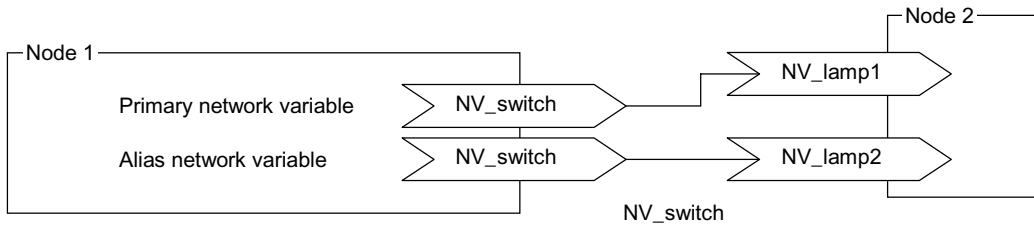


Figure 3.1 Alias Schematic

An alias is an abstract variable for networks controlled by the Neuron Chip firmware.

An alias corresponds to a non-alias network variable called a primary network variable. The alias inherits the attributes of the primary network variable such as data value, direction, and type. The primary network variable value is shared with the corresponding alias.

A network variable update message received by the primary input network variable or by any alias updates the primary network variable value and generates an update event for the primary network variable. A poll request for the primary output network variable or its alias generates a response based on the primary network variable value.

When the application program updates the primary output network variable, the Neuron Chip firmware sends the network variable update message for the related primary network variable and alias on the send node. When the application program starts polling of the primary input network variable, the Neuron Chip firmware sends the network variable poll request for the related primary network variable and alias on the polling node.

Note: ● When using aliases, the number of the alias table entries in the target node applications program must be specified in advance.

```
"#pragma num_alias_table_entries <number>"
```

For details of alias tables, see Chapter A.4.

- Some network management tools such as LonBuilder network manager do not support aliases.

4. Explicit Messages

For most applications, network variables allow the most compact and simple possible application software implementation, often resulting in the smallest attainable code size. However, for applications in which data objects larger than 31 bytes need to be transmitted, where request/response service is desired, or when the network variable model is not suitable, then the application can use explicit messaging.

The application program can construct messages containing up to 228 bytes of data, addressed to other nodes or groups of nodes via implicit address connections called message tags. For further details, see the LONWORKS Engineering Bulletin *File Transfer*. Alternately, messages may be explicitly addressed to other nodes using subnet/node, group, broadcast, or Neuron ID addressing. The messages may be sent under one of four classes of service:

<i>Acknowledged</i>	Acknowledged service with retry
<i>Unacknowledged</i>	Unacknowledged service
<i>Repeated</i>	Unacknowledged repeated service (message sent multiple times)
<i>Request/Response</i>	Request/response service

Request/response service allows the node receiving a message to respond with data in its response messages, as distinct from an acknowledgment which contains no application-level data. When a node receives a request message, it prepares a response to the incoming message and returns it to the original sender using the `resp_send()` call. If the response contains no data aside from the message code, then the network processor handles retries in such a way that the message is only delivered to the destination application once. If the response contains one or more bytes of application data, then the message is delivered to the destination application on each retry. Messages may be specified as authenticated (see section 6). Messages may also be specified to have priority, meaning that they use a priority time slot on the channel if the node has a priority slot allocated to it. In any case, priority messages are always sent by the node before non-priority messages (see section 7). The application program may explicitly assign the destination addresses, or it may use the default address associated with the message tag.

Messages are exchanged between nodes by calling run-time library support routines:

<code>msg_alloc()</code>	Allocate a message buffer.
<code>msg_alloc_priority()</code>	Allocate a priority message buffer.
<code>msg_cancel()</code>	Cancel a message being built for sending.
<code>msg_free()</code>	Free a message buffer.
<code>msg_receive()</code>	Receive a message at this node.
<code>msg_send()</code>	Send a message to another node.
<code>resp_alloc()</code>	Allocate a buffer for a response to a request message.
<code>resp_cancel()</code>	Cancel a response being built.
<code>resp_free()</code>	Free a response buffer.
<code>resp_receive()</code>	Receive a response to a request message.
<code>resp_send()</code>	Send a response to a request message.

The following built-in events may be checked by the scheduler to allow asynchronous transmission and reception of messages:

<code>msg_arrives</code>	A message has arrived at this node.
<code>msg_completes</code>	An outgoing message has been handled, either successfully or unsuccessfully.
<code>msg_succeeds</code>	An outgoing message has been successfully sent.
<code>msg_fails</code>	Some acknowledgements have not been received for an outgoing message.
<code>resp_arrives</code>	A response to a request has been received.

The following built-in objects are defined for use by application program:

<code>msg_in</code>	Currently received message
<code>msg_out</code>	Message to be sent
<code>resp_in</code>	Currently received response to a previous request message
<code>resp_out</code>	Response to be sent to a previous request

The declarations of the structures for these objects are built into the Neuron C compiler. They are reproduced here for reference.

```

typedef enum {
    ACKD = 0,                // acknowledged
    UNACKD_RPT = 1,         // unacknowledged repeated
    UNACKD = 2,             // unacknowledged
    REQUEST = 3             // request/response
} service_type;

struct {
    int      code;           // message code
    int      len;           // length of message data
    int      data[ ];       // message data
    boolean  authenticated;  // TRUE if message was
                            // successfully
                            // authenticated
    service_type service;   // service type
    msg_in_addr addr;       // optional field if explicit
                            // addressing used
    boolean  duplicate;     // 1=> msg is a duplicate
    int      rcvtx;         // receive transaction index
} msg_in;

struct {
    boolean  priority_on;   // TRUE if message is to be sent
                            // with priority
    msg_tag  tag;           // destination message tag
    int      code;           // message code
    int      data[];        // message data
    boolean  authenticated; // TRUE if message is to be
                            // authenticated
    service_type service;   // service type
    msg_out_addr addr;      // optional field if explicit
                            // addressing used
} msg_out;

struct {
    int      code;           // response message code
    int      len;           // length of response message data
    int      data[];        // message data
    resp_in_addr addr;      // optional field if explicit
                            // addressing used
} resp_in;

struct {
    int      code;           // response message code
    int      data[];        // response message data
} resp_out;

```

If an explicit address is used to address an outgoing message, the source node application creates an addr data structure in the `msg_out` buffer. If the destination node application wishes to know the source address of an incoming message, it can refer to the addr data structure in the `msg_in` buffer. If a requesting node wishes to know the source address of an incoming response, it can refer to the addr data structure in the `resp_in` buffer. These definitions are in the `MSG_ADDR.H` include file, and are reproduced here for reference. For definitions of the data types `addr_type`, `group_struct`, `snode_struct`, `nrnid_struct`, and `bcast_struct`, see Section A.3.

```

typedef union {
    addr_type    no_address;        // unbound
    group_struct group;            // group
    snode_struct snode;            // subnet/node
    nrnid_struct nrnid;            // NEURON ID
    bcast_struct bcast;            // broadcast
} msg_out_addr;
typedef struct {
    unsigned    domain      : 1;    // domain table reference
    unsigned    flex_domain : 1;    // received on flexible domain
    unsigned    format      : 6;    // 0 = broadcast, 1 = group,
                                     // 2 = subnet/node, 3 = NEURON ID

    struct {
        unsigned subnet;
        unsigned   : 1;
        unsigned node : 7;
    } src_addr;                    // source address
    union {
        unsigned bcast_subnet; // broadcast destination address
        unsigned group;        // group destination address
        struct {
            unsigned subnet;
            unsigned   : 1;
            unsigned node: 7;
        } snode;                // subnet/node destination address
        struct {
            unsigned subnet;
            unsigned nid[ NEURON_ID_LEN ];
        } nrnid;                // NEURON ID destination address
    } dest_addr;                 // destination address
} msg_in_addr;

```

```

typedef struct {
    unsigned domain      : 1;      // domain table reference
    unsigned flex_domain : 1;      // received on flexible domain
    struct {
        unsigned subnet;
        unsigned is_snode : 1; // 0 = group response,
                               // 1 = subnet/node response
        unsigned node : 7;
    } src_addr;                    // source address
    union {
        struct {
            unsigned subnet;
            unsigned      : 1;
            unsigned node : 7;
        } snode;                  // subnet/node destination address
        struct {
            unsigned subnet;
            unsigned      : 1;
            unsigned node : 7;
            unsigned group;
            unsigned      : 2;
            unsigned member : 6;
        } group;                  // group destination address
    } dest_addr;                  // destination address
} resp_in_addr;

```

Preemption Mode

Use of the buffer allocation functions (`msg_alloc ()` and `msg_alloc_priority ()`) is optional. If they are not used and if buffers are not available when message construction commences, the node enters preemption mode. This suspends the application program except for completion events (e.g., `msg_complets`), incoming message events, and network variable array event processing. If no buffer becomes available within a configurable number of seconds known as the *maximum free buffer wait time*, the node resets. This is known as a *preemption mode timeout*. See Section A.6 for a description of the configuration data structure, where this is defined. Preemption mode can also be entered when synchronous network variables are updated or when the `flush_wait ()` function is used.

The use of the `preempt_safe` keyword before a *when* clause causes the evaluation of the *when* clause, even if the node is in preemption mode. See the *Neuron C Programmer's Guide* for more information.

5. Scheduler

The scheduler executes user-written tasks in response to events or conditions specified in *when* clauses by the application program. When a specified event occurs or condition become true, the associated task body is executed. The user has the capability of specifying one or more *when* clauses as having priority, and the scheduler checks all priority *when* clauses in order of their appearance in the Neuron C program, followed by one non-priority *when* clause. Each of the remaining non-priority *when* clauses is checked during successive scheduler cycles, at one clause per cycle. The task scheduler can handle up to eighty *when* clauses, depending on their type.

Events fall into five classes:

System-Wide Events

<i>reset</i>	Node has been reset
<i>offline</i>	Node has been set off-line
<i>online</i>	Node has been set on-line
<i>flush_completes</i>	Node has complicated preparations to enter sleep mode
<i>wink</i>	Node has received <i>wink</i> network management message with <i>wink</i> subcommand

Input/Output Events

<i>io_changes</i>	Value read from an I/O object has changed since last reading. Changes may be unconditional, or changed to a specified value, or by a specified amount.
<i>io_update_occurs</i>	Value read from a timer/counter input object has been updated.
<i>io_in_ready</i>	Parallel I/O object is ready to receive data from external CPU.
<i>io_out_ready</i>	Parallel I/O object is ready to transmit data to external CPU.

Timer Events

<i>timer_expires</i>	Software timer value has decremented to zero.
----------------------	---

Message and Network Variable Events

The following events associated with the transmission of network variables and messages are discussed above in Sections 2 and 3:

nv_update_occurs, nv_update_fails, nv_update_succeeds,
nv_update_completes msg_arrives, msg_completes, msg_succeeds, msg_fails, resp_arrives

User-Specified Events

<*boolean expression*> User-specified expression, evaluated as TRUE or FALSE

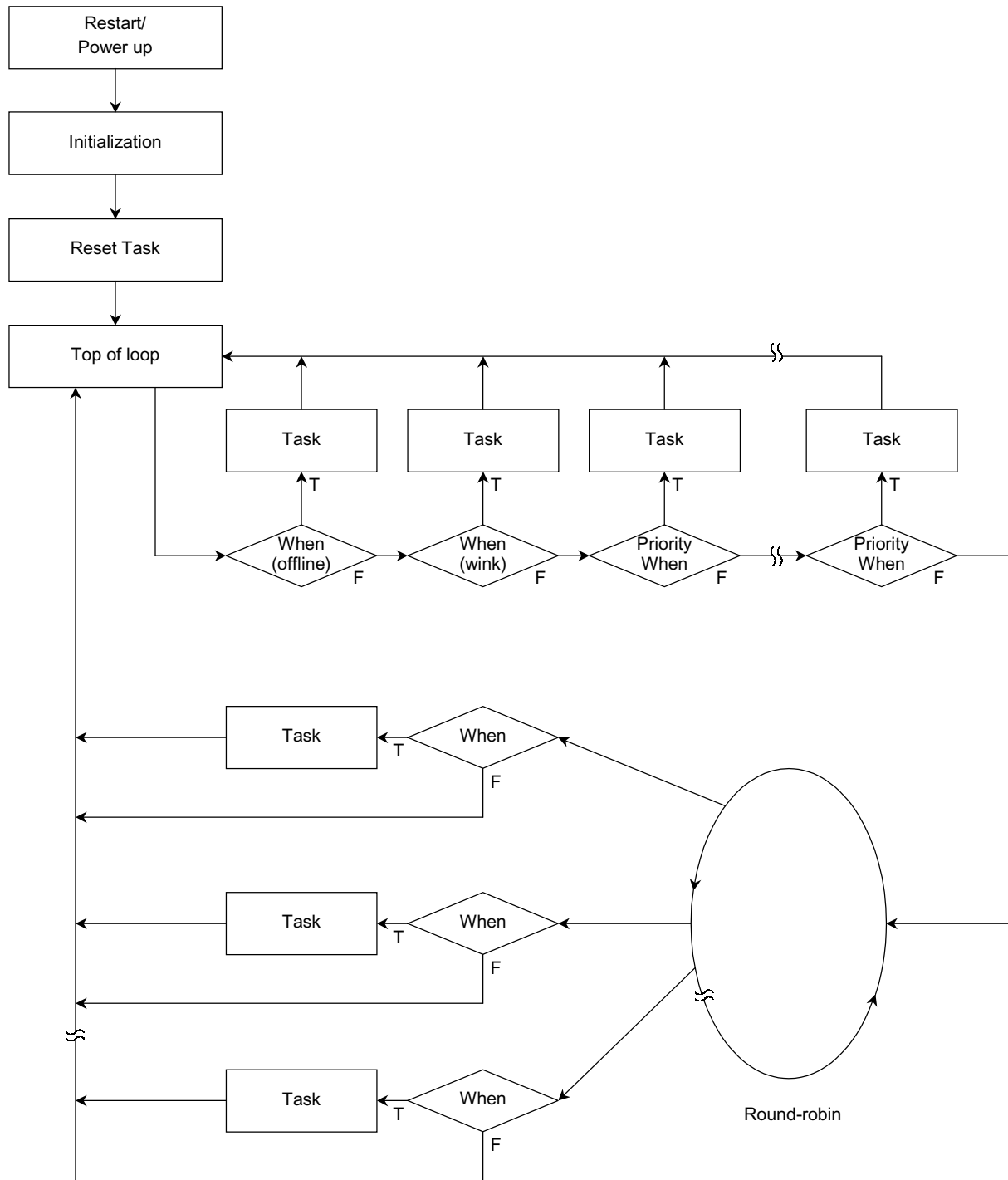


Figure 5.1 Neuron Chip Firmware Scheduling of Nonpriority and Priority When Clauses

6. Additional Functions

The following miscellaneous functions are available from the application program. These functions are built into the Neuron C Compiler, are part of the Neuron Chip system image, or are linked into the application image from a system library.

When a function is used, whether the function is in the system image (within firmware) or linked to the application image (application area is used) depends on the Neuron Chip type.

The table below lists the relations between functions and Neuron Chips.

○: Function is included in the system image

×: Function is linked to the application image

N/A: Function cannot be used

● Execution Control

Function	TMPN3150	TMPN3120B	TMPN3120 E1	TMPN3120 E3	TMPN3120 FE3/A20/FE5
delay ()	○	○	○	○	○
flush ()	○	○	○	○	○
flush_cancel ()	○	○	○	○	○
flush_wait ()	○	×	×	○	○
get_tick_count ()	○	○	○	○	○
go_offline ()	○	○	○	○	○
post_events ()	○	○	○	○	○
power_up ()	○	×	×	○	○
preemption_mode ()	×	×	×	○	○
scaled_delay ()	○	○	○	○	○
sleep ()	○	○	○	○	○
timers_off ()	○	○	○	○	○
watchdog_update ()	○	○	○	○	○

● Network Configuration

Function	TMPN3150	TMPN3120B	TMPN3120 E1	TMPN3120 E3	TMPN3120 FE3/A20/FE5
access_address ()	○	×	×	○	○
access_alias ()	○	N/A	×	○	○
access_domain ()	○	×	×	○	○
access_nv ()	○	×	×	○	○
addr_table_index ()	○	○	○	○	○
application_restart ()	○	○	○	○	○
go_unconfigured ()	○	×	×	○	○
node_reset ()	○	○	○	○	○
nv_table_index ()	○	○	○	○	○
offline_confirm ()	○	○	○	○	○
update_address ()	○	×	×	○	○
update_alias ()	○	N/A	×	○	○
update_clone_domain ()	○	×	×	○	○
update_config_date ()	○	×	×	○	○
update_domain ()	○	×	×	○	○
update_nv ()	○	×	×	○	○

● Inter Math

Function	TMPN3150	TMPN3120B	TMPN3120 E1	TMPN3120 E3	TMPN3120 FE3/A20/FE5
abs ()	○	○	○	○	○
bcd2bin ()	○	×	×	○	○
bin2bcd ()	○	×	×	○	○
high_byte ()	○	○	○	○	○
low_byte ()	○	○	○	○	○
make_long ()	○	○	○	○	○
max ()	○	○	○	○	○
min ()	○	○	○	○	○
muldiv ()	○	×	×	○	○
muldiv24 ()	×	×	×	○	○
muldiv24s ()	×	×	×	○	○
muldivs ()	○	×	×	○	○
random ()	○	○	○	○	○
reverse ()	○	×	×	○	○
rotate_long_left ()	×	×	×	×	○
rotate_long_right ()	×	×	×	×	○
rotate_short_left ()	×	×	×	×	○
rotate_short_right ()	×	×	×	×	○
s32_abs ()	×	×	×	○	○
s32_add ()	×	×	×	○	○
s32_cmp ()	×	×	×	○	○

Function	TMPN3150	TMPN3120B	TMPN3120 E1	TMPN3120 E3	TMPN3120 FE3/A20/FE5
s32_dec ()	x	x	x	○	○
s32_div ()	x	x	x	○	○
s32_div2 ()	x	x	x	○	○
s32_eq ()	x	x	x	○	○
s32_from_ascii ()	x	x	x	x	x
s32_from_slong ()	x	x	x	○	○
s32_from_ulong ()	x	x	x	x	x
s32_ge ()	x	x	x	x	x
s32_gt ()	x	x	x	x	x
s32_inc ()	x	x	x	x	x
s32_le ()	x	x	x	x	x
s32_lt ()	x	x	x	x	x
s32_max ()	x	x	x	x	x
s32_min ()	x	x	x	x	x
s32_mul ()	x	x	x	x	x
s32_mul2 ()	x	x	x	x	x
s32_ne ()	x	x	x	x	x
s32_neg ()	x	x	x	x	x
s32_rand ()	x	x	x	x	x
s32_rem ()	x	x	x	x	x
s32_sign ()	x	x	x	x	x
s32_sub ()	x	x	x	x	x
s32_to_ascii ()	x	x	x	○	○
s32_to_slong ()	x	x	x	○	○
s32_to_ulong ()	x	x	x	○	○
swap_bytes ()	○	○	○	○	○

● Floating Point Math

Function	TMPN3150	TMPN3120B	TMPN3120 E1	TMPN3120 E3	TMPN3120 FE3/A20/FE5
fl_abs ()	×	×	×	○	○
fl_add ()	×	N/A	×	○	○
fl_ceil ()	×	N/A	×	○	○
fl_cmp ()	×	×	×	○	○
fl_div ()	×	N/A	×	○	○
fl_div2 ()	×	×	×	×	×
fl_eq ()	×	×	×	○	○
fl_floor ()	×	N/A	×	○	○
fl_from_ascii ()	×	N/A	N/A	×	×
fl_from_s32 ()	×	N/A	×	×	×
fl_from_slong ()	×	×	×	○	○
fl_from_ulong ()	×	×	×	○	○
fl_ge ()	×	×	×	○	○
fl_gt ()	×	×	×	○	○
fl_le ()	×	×	×	○	○
fl_lt ()	×	×	×	○	○
fl_max ()	×	×	×	○	○
fl_min ()	×	×	×	○	○
fl_mul ()	×	×	×	○	○
fl_mul2 ()	×	×	×	×	×
fl_ne ()	×	×	×	○	○
fl_neg ()	×	×	×	○	○
fl_rand ()	×	×	×	○	○
fl_round ()	×	N/A	×	○	○
fl_sign ()	×	×	×	○	○
fl_sqrt ()	×	N/A	N/A	○	○
fl_sub ()	×	N/A	×	○	○
fl_to_ascii ()	×	N/A	N/A	×	×
fl_to_ascii_fmt ()	×	N/A	N/A	×	×
fl_to_s32 ()	×	×	×	×	×
fl_to_slong ()	×	×	×	○	○
fl_to_ulong ()	×	×	×	○	○
fl_trunc ()	×	×	×	○	○

● **Strings**

Function	TMPN3150	TMPN3120B	TMPN3120 E1	TMPN3120 E3	TMPN3120 FE3/A20/FE5
strcat ()	x	x	x	○	○
strchr ()	x	x	x	○	○
strcmp ()	x	x	x	○	○
strcpy ()	x	x	x	○	○
strlen ()	x	x	x	○	○
strncat ()	x	x	x	○	○
strncmp ()	x	x	x	○	○
strncpy ()	x	x	x	○	○
strrchr ()	x	x	x	○	○

● **Utilities**

Function	TMPN3150	TMPN3120B	TMPN3120 E1	TMPN3120 E3	TMPN3120 FE3/A20/FE5
ansi_memcpy ()	x	x	x	○	○
ansi_memset ()	x	x	x	○	○
clear_status ()	x	x	x	○	○
clr_bit ()	x	x	x	○	○
crc8 ()	x	x	x	○	○
crc16 ()	x	x	x	○	○
eeeprom_memcpy ()	○	○	○	○	○
error_log ()	x	x	x	○	○
memcpy ()	x	x	x	○	○
memchr ()	x	x	x	○	○
memcmp ()	x	x	x	○	○
memcpy ()	○	x	x	○	○
memset ()	○	x	x	○	○
refresh_memory ()	○	○	○	○	○
retrieve_status ()	○	x	x	○	○
retrieve_xcvr_status ()	x	x	x	○	○
set_bit ()	x	x	x	x	○
set_eeeprom_lock ()	○	○	○	○	○
tst_bit ()	x	x	x	x	○

● Input/Output

Function	TMPN3150	TMPN3120B	TMPN3120 E1	TMPN3120 E3	TMPN3120 FE3/A20/FE5
io_change_init ()	○	○	○	○	○
io_edgelog_preload ()	○	×	×	○	○
io_in () differs depending on the I/O object used					
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 ()	○	○	○	○	○
io_in_request ()	○	×	×	○	○
io_out () differs depending on the I/O object used					
Bitshift output	○	○	×	○	○
Neurowire I/O slave mode	○	×	×	×	×
Neurowire I/O with invert option	×	×	×	×	×
Serial output	○	○	×	○	○
Touch I/O	×	×	×	×	×
Others	○	○	○	○	○
io_out_ready ()	○	○	○	○	○
io_out_request ()	○	○	○	○	○
io_preserve_input ()	○	×	×	○	○
io_select ()	○	○	○	○	○
io_set_clock ()	○	○	○	○	○
io_set_direction ()	○	○	○	○	○

7. Built-In Variables

<i>activate_service_led</i>	Service LED state control
<i>config_data</i>	Read-only copy of the node's configuration data
<i>input_value</i>	Data read by last explicit <code>io_in()</code> call or by last input/output event (e.g., <code>io_changes</code>)
<i>input_is_new</i>	True if last input from a timer/counter object read an update value
<i>msg_tag_index</i>	Message tag index for the most recent explicit message response or completion event received
<i>nv_array_index</i>	Index of element in network variable array with updated value
<i>nv_in_addr</i>	Source address of the last network variable update
<i>nv_in_index</i>	Network variable index for the most recent network variable update
<i>read_only_data</i>	Copy of node's read only data structure
<i>read_only_data_2</i>	Copy of node's read only data structure extension

8. TMPN3120xx Chip Firmware Extensions

On the TMPN3120xx Chip, all application code is placed in on-chip EEPROM. System library functions specified in Section 6 are linked with the application and are also placed in on-chip EEPROM. In addition, when any of the following Neuron C features are used, object code is brought in from a system library and placed in on-chip EEPROM.

- Explicitly addressed messages
- Structure assignment (length ≥ 256 bytes)
- Use of a signed bit field