LONWORKS™ Host Application Programmer's Guide

Revision 2

ECHELON®
Corporation

078-0016-01B

Document No. 29400

Printed in the United States of America.

Echelon Corporation
4015 Miranda Avenue
Palo Alto, California
94304

Preface

This guide describes how to create LONWORKS™ host applications. Host applications are application programs running on hosts other than NEURON® CHIPS that use the LONTALK™ protocol to communicate with nodes on a LONWORKS network. The availability of host applications makes the LONTALK protocol available to any host processor by using the NEURON CHIP as a communications processor.

Audience

The *Host Application Programmer's Guide* is intended for developers creating host applications for any host. Examples are shown in ANSI C, however, host applications may be written in any language that can implement the LONTALK network interface protocol.

Developers creating host applications using the LONMANAGER™ API do not need to read this guide. The LONMANAGER API implements the network interface protocol and provides a higher level of services to the host application programmer.

Readers of this guide should have C programming experience and be familiar with LONWORKS concepts and LONWORKS application node development. See *Related Manuals* later in the preface for a list of LONWORKS documentation.

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., 3rd edition, Prentice-Hall, Inc., 1991.
- *The C Programming Language*, Brian W. Kernighan and Dennis M. Ritchie, 2nd edition, Prentice-Hall, Inc., 1988.

Content

The Host Application Programmer's Guide has five chapters and four appendices as follows:

- Chapter 1, *Host Application Overview,* provides an introduction to the host application architecture.
- Chapter 2, *Host Application Architecture,* discusses the protocol used by host applications to communicate with a network interface.
- Chapter 3, *Sending and Receiving Messages,* discusses the steps used in sending and receiving LONTALK messages from a host application.
- Chapter 4, *Using a Network Driver,* describes specifications for using a LONWORKS network driver.
- Chapter 5, *Error Conditions,* discusses errors detected by host applications.
- Appendix A, *Sample Host Application,* provides source code for a sample host application.
- Appendix B, *Creating an External Interface File,* describes the procedure for modifying an external interface file to include network variables and message tags used by a host application.

- Appendix C, *Network Interface Messages,* defines the message structures exchanged by a host application and the network interface.
- Appendix D, *Network Interface Commands,* describes the network interface commands specified in a data transfer from a host application to the network interface.

Related Manuals

The following manuals and engineering bulletins are referenced in this guide:

The *LONTALK Protocol* engineering bulletin describes the LONTALK Protocol.

The *How to Use SNVTs in LONWORKS Applications* engineering bulletin describes how standard network variable types (SNVTs) can be used by any application, including host applications, to increase interoperability between LONWORKS nodes.

The *NEURON 3120™ CHIP and NEURON 3150™ CHIP Data Book* Appendix B defines the network management and network diagnostic message formats that can be used by all application nodes, including host application nodes.

The *LONBUILDER™ User's Guide* lists and describes all tasks related to LONWORKS application development using the LONBUILDER Developer's Workbench. Refer to that guide for detailed information on the LONBUILDER user interface.

The *LONMANAGER API Programmer's Guide* and the *LONMANAGER API Programmer's Guide for Windows* describe network management in a LONWORKS network. They outline the components of a LONWORKS network management tool, list the library functions of the LONMANAGER API, and provide examples for building a host application using the LONMANAGER API. In addition to the programmer's guide, there is also a *LONMANAGER API Reference Guide for Windows*, Volumes I and II.

The *NEURON C Programmer's Guide* outlines a recommended general approach to developing a NEURON C application, explains key concepts of programming in NEURON C through the use of code fragments and examples, and provides a complete reference section for NEURON C.

The *Parallel I/O Interface to the NEURON CHIP* engineering bulletin describes hardware and software to interface the NEURON CHIP to a host processor using the parallel I/O port.

The *Custom Node Development* engineering bulletin describes the steps for building an example LONWORKS application node.

The *LONWORKS Installation Overview* engineering bulletin describes LONWORKS network installation and outlines several scenarios that may be used to install LONWORKS networks.

The *NEURON CHIP-based Installation of LONWORKS Networks* engineering bulletin describes network management from NEURON C applications.

The *Serial LONTALK Adapter User's Guide* describes how to use the Serial LONTALK Adapter, a network interface that can be used with any host with a serial interface.

The *LONBUILDER Microprocessor Interface Program (MIP) User's Guide* describes how to create a network interface using the LONBUILDER Microprocessor Interface Program (MIP).

Contents

1 Host Application Overview

This chapter provides an introduction to the host applications. Basic concepts are defined and the host application's intended uses are outlined.

Overview of the Host Application Architecture

Host applications are application programs running on hosts other than NEURON CHIPs that use the LONTALK protocol to communicate with nodes on a LONWORKS network. For PC-based host applications, the LONMANAGER API can be used to greatly reduce the work required for implementing a host application.

Host applications interface with a LONWORKS network via a *network interface*. A network interface uses a NEURON CHIP as a communications processor. The network interface implements layers 1 through 5 of the LONTALK protocol. Layers 6 and 7 of the protocol are implemented by the host application. The network interface may be implemented using a turn-key network interface product such as the Serial LONTALK Adapter (SLTA). A custom network interface may be implemented using the LONBUILDER Microprocessor Interface Program (MIP). See the *Serial LONTALK Adapter User's Guide* and the *MIP User's Guide* for more information.

The host application can use a *network driver* to implement the hardware dependent portion of the network interface protocol. This allows host applications to be independent of the physical interface between the host and the network interface. Figure 1-1 summarizes the host application architecture.

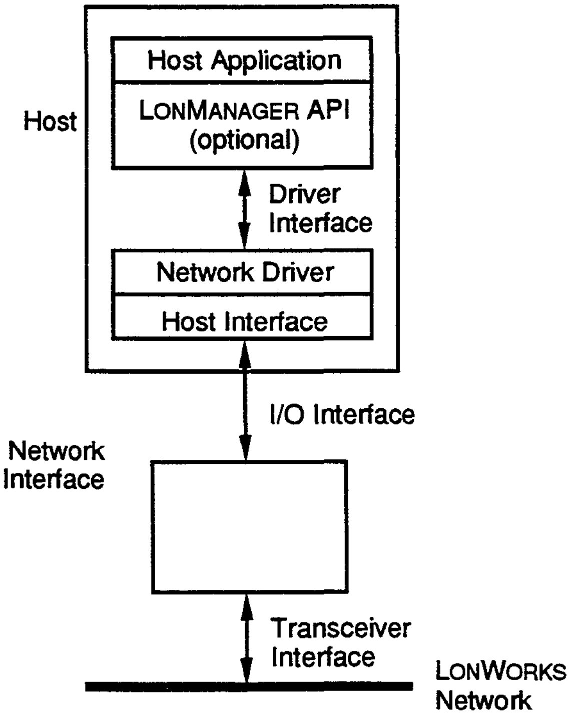


Figure 1-1 Host Application Architecture

Intended Uses of Host Applications

Several types of nodes can be attached to a LONWORKS network. The lowest cost node, based on the NEURON 3120™ CHIP, provides a complete system-on-a-chip, including memory for the application code and data, and protocol firmware. For applications that require more code or data space, the NEURON 3150™ CHIP supports up to 42 Kbytes of off-chip user memory. Nodes using the NEURON CHIP as the applications processor are called *NEURON CHIP-hosted nodes.*

Host applications can be used for nodes that require more processing power, memory, or input/output capability than provided by the NEURON CHIP family. Host applications use the NEURON CHIP as a communications processor. The applications processing occurs on an external host processor. Host applications can also be used to interface an existing application to a LONWORKS network. These nodes are called *host-based nodes.*

Examples

A host application may be implemented on a host microprocessor to expand the input/output capabilities of the node. For example, the host microprocessor family may have special-purpose peripheral chips available. Figure 1-2 illustrates an example host application using a Motorola 68332 processor.

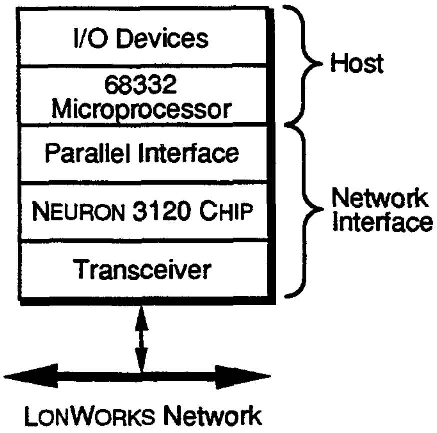


Figure 1-2 Host Application with Motorola 68332 Host

A host application may be implemented on a microcomputer with a standard operating system such as MS-DOS or Unix. In this case, the host will have a wide variety of data storage and user interface hardware available, as well as third-party software and hardware products that can be easily integrated into the LONWORKS network. In general, a host microcomputer may have much greater processing power than a NEURON CHIP for compute-intensive applications. This is illustrated in figure 1-3.

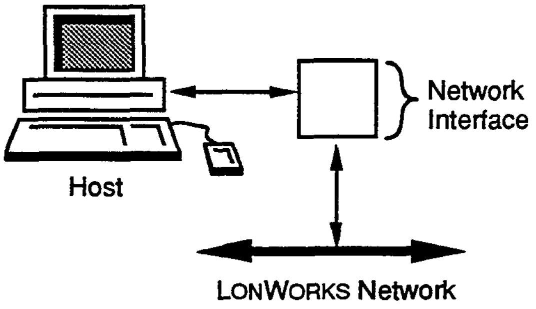


Figure 1-3 Host Application with Microcomputer Host

Host applications may also be used to receive asynchronous updates from more network variable connections than can be received by a NEURON CHIP-hosted application. Any NEURON CHIP-hosted or host application can write to and poll any number of network variables. This is done by sending network variable updates and polls (or fetches) as explicit messages as described in Appendix B of the *NEURON CHIP Data Book.* However, to receive asynchronous updates from a network variable connection, the application node must be bound to the connection. A NEURON CHIP-based application or host application using *network interface selection* (described in Chapter 3) can declare up to 62 network variables. When network variable processing is set to *host selection,* the host can declare up to 4096 network variables. Through the binding process, network variables on multiple nodes are associated with one another so that nodes may receive asynchronous updates from one another.

NOTE: While there is no direct mapping between the number of network variables on a node and the number of connections in which the node may participate, the ability to declare more network variables does make host applications well-suited to large monitoring, data logging, or controller-like applications.

Network Management, Network Control, and Network Monitoring

Host applications can be used for any type of application, but the most typical uses are network management, network control, and network monitoring. A LONWORKS network interface can be used to create nodes that perform any combination of these functions.

Network Management

Network management is the task of installing, maintaining, and configuring the nodes in a network. A network management tool does not participate in the exchange of application messages and network variable messages, and so does not need to be present for the network to operate. Network management tools require a database that allows them to keep track of node and variable addresses on the network, and so they are typically implemented using computers as hosts with the database stored on disk. In that sense, a network management tool is a special case of a host application. Network management applications for complex networks are best implemented using a PC-compatible host and the LONMANAGER Application Programmer's Interface (API) for DOS or Windows, or alternatively with the turnkey LONMANAGER NetMaker tool. These tools include a database management facility that keeps track of network topology and addressing. See the *LONWORKS Installation Overview* engineering bulletin for a description of installation options.

Network Control

A network controller is a central node that coordinates the sense and control processing of a control network. In LONWORKS networks, any node can send and receive messages and network variables to and from any other node on the network, and thus can act as a network controller. The network controller is the source or destination of most of the application messages, and the other nodes communicate only with this central node. LONWORKS networks may also be designed using peer-to-peer communication and control so that a network controller is not required. The system is then invulnerable to failures of any single node.

Network Monitoring

A network monitor is a node that receives application messages or network variable updates from many of the other nodes on the network. Any node in the network may be the destination of LONTALK messages from other nodes, and so may act as a network monitor.

Definitions

Downlink — Data transfers from the host to the network interface.

Host — The host processor with the host application, network driver, and host interface.

Host Application — An application program running on a host other than a NEURON CHIP that uses the LONTALK protocol to communicate with nodes on a LONWORKS network.

Host Interface

The hardware interface between the host processor and the network interface. The host interface is physically connected to the network interface. The host interface is an EIA-232 interface for the Serial LONTALK Adapter (SLTA), and is a parallel or dual-ported RAM interface for network interfaces implemented with the Microprocessor Interface Program (MIP).

Host Node

The host plus the network interface and driver.

Host Processor

The processor that runs the host application. The processor may be a microcontroller, microprocessor, PC, workstation, minicomputer, or mainframe computer.

Microprocessor Interface Program (MIP)

Firmware for the NEURON CHIP that moves the upper layers of the LONTALK protocol off the NEURON CHIP onto a host processor. The MIP implements the NEURON CHIP side of the network interface protocol, and can be used to implement a custom network interface.

Network Driver

The software that interfaces the host application to the host interface hardware. The network driver isolates the host application from the physical interface to the network interface.

Network Driver Protocol

A standard protocol for communications between a host application and a network driver.

Network Interface

A device that provides an interface between a host and a LONWORKS network. The network interface implements the LONTALK network interface protocol. The network interface may be implemented using a turn-key network interface product such as the Echelon Serial LONTALK Adapter (SLTA). A custom network interface may be implemented using the LONBUILDER Microprocessor Interface Program (MIP). LONWORKS network interfaces are also available from third-party manufacturers. See the *Serial LONTALK Adapter User's Guide* and the *MIP User's Guide* for more information.

Network Interface Protocol

A standard protocol for communications between a host and a network interface. There are three variants of the protocol: one for the Serial LONTALK Adapter, one for the MIP/P20 and MIP/P50, and one for the MIP/DPS. The differences are transparent to the host application since they are handled by the network driver.

Network Variable

An object declared on a LONWORKS node that may be connected to multiple nodes on a LONWORKS network. Network variables provide a well-defined interface between LONWORKS nodes.

Serial LONTALK Adapter (SLTA)

A turn-key network interface that communicates with a host using an EIA-232 interface.

SLTA Node

A host node using the SLTA.

Uplink

Data transfers from the network interface to the host.

2

Host Application Architecture

This chapter discusses the protocol used by host applications to communicate with a network interface.

Host Application Architecture

The host application architecture defines standard protocols for communications between a host and a network interface. The architecture has three layers; application, link, and physical. Figure 2-1 illustrates the host application architecture layers.

Application Layer

The application layer, also known as the *LONTALK Network Driver Protocol,* is used by the host application to send and receive LONTALK messages. Chapter 3, *Sending and Receiving Messages*, describes how the host application can send and receive LONTALK messages using the network driver protocol. The network driver protocol is identical for all network interfaces, including the Serial LONTALK Adapter (SLTA) and network interfaces using any version of the Microprocessor Interface Program (MIP).

Link Layer

The link layer, also known as the *LONTALK Network Interface Protocol* is used by the network driver to ensure reliable delivery of packets between the host and the network interface. The link layer is also used by the host to control the network interface. The link layer is different for the SLTA, the MIP/P20 and MIP/50, and the MIP/DPS. The differences in the link layer protocols are managed by the network driver, and are transparent to the host application. The link layer protocol for the SLTA is based on a serial data transfer protocol between the host and network interface. For details of the SLTA link layer protocol, see the *SLTA User's Guide.* The link layer protocol for a MIP/P20 or MIP/P50-based network interface is based on the NEURON CHIP parallel I/O protocol. The link layer protocol for a MIP/DPS-based network interface is based on a dual-ported RAM with semaphores that is mapped into the address space of a NEURON 3150 CHIP and also of the host microprocessor. For details of the link layer protocols used by the MIPs, see the LONBUILDER *Microprocessor Interface Program (MIP) User's Guide.* Appendix D, *Network Interface Commands*, describes the commands that the host can use to control the network interface.

Physical Layer

The physical layer is the physical interface between the host interface and the network interface. The physical layer for the SLTA is an EIA-232 interface as described in the *Serial LONTALK Adapter User's Guide.* The physical layer for a network interface based on the MIP/P20 or MIP/P50 is a parallel interface. The physical layer for a network interface based on the MIP/DPS is a dual-ported RAM with hardware semaphores. This interface should be described in documentation provided with a third-party network interface, and is described in the *Microprocessor Interface Program (MIP) User's Guide* for custom network interfaces.

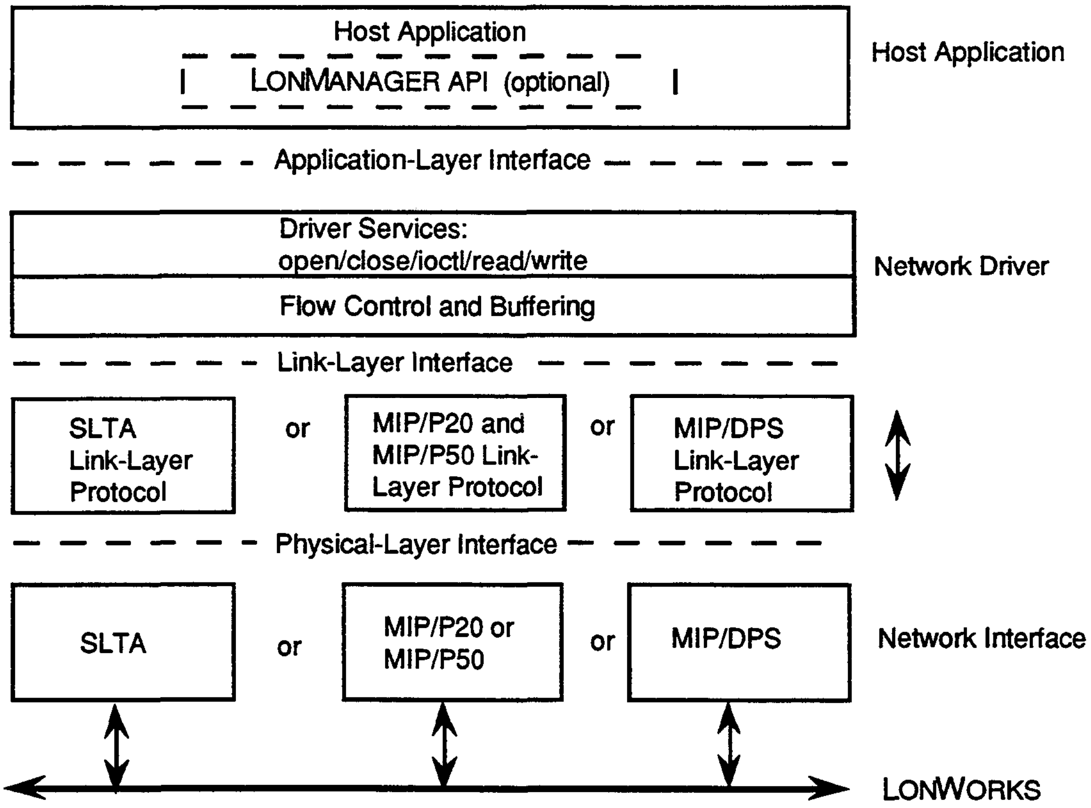


Figure 2-1 Network Interface Protocol Layers

FLUSH State

After the network interface is reset, the NEURON CHIP enters a special FLUSH state. This state causes the network interface to ignore all incoming messages and prevents all outgoing messages, even service pin messages. The FLUSH state is provided to prevent any other network management tool from performing network management functions on the network interface before the host has a chance to configure the network interface. This state must be cancelled with a downlink niFLUSH_CANCEL command from the host before the network interface can participate in any network transactions. After the FLUSH state is cancelled, the network interface is in the NORMAL state.

The network interface sends the niRESET command uplink following any reset. This is the first message received by the host whenever the network interface is reset. The standard network drivers for DOS provide a configuration option for handling the FLUSH state. If the /Z switch was not specified when the driver was loaded (in the CONFIG.SYS file), then the driver will automatically send the network interface the niFLUSH_CANCEL message when the device is opened, and also when it receives an uplink niRESET command. If the /Z switch was specified, then the application is responsible for sending niFLUSH_CANCEL when the device is opened, and when it receives an uplink niRESET.

For the SLTA, another possibility is provided with a jumper option. This jumper specifies that the SLTA not enter the special FLUSH state after reset, so that the host application or the host driver need not send the niFLUSH_CANCEL. See *Configuration Jumpers* in Chapter 2 of the *Serial LONTALK Adapter (SLTA) User's Guide*.

3
Sending and Receiving Messages

This chapter discusses the steps used in sending and receiving LONTALK messages from a host application. Network interface configuration options are also described.

Communicating With Other Nodes

The host application communicates with other nodes by sending and receiving LONTALK messages. These messages may be application, network management, or network diagnostic messages. Application messages may be network variable messages or explicit messages.

The host application sends a LONTALK message by building the message in an application buffer and passing the buffer downlink to the network interface via the network driver. The host application receives LONTALK messages by decoding application buffers received uplink from the network interface via the network driver. The format of the application buffer is defined in this section and is contained in the `ExpAppBuffer` and `ImpAppBuffer` structures in Appendix C.

The network driver translates the application-layer header to a link-layer header, and manages buffer allocation as described in the *Serial LONTALK Adapter User's Guide* and the *LONBUILDER Microprocessor Interface Program User's Guide*. Application buffers exchanged by the host application and the network driver contain one or more of the following fields:

- **Network Interface Command**. The network interface command specifies the type and size of the application buffer. The network interface command is contained in the `NI_Hdr` structure defined in Appendix C, and in the file `NI_MGMT.H` supplied with the sample host application. Network interface commands are defined in Appendix D. This field is always present, and is the only field specified for local network interface commands, such as the reset command, `niRESET`. Local network interface commands are network management or network diagnostic commands that are sent from the host to the network interface.
- **Message Header**. The message header describes the type of LONTALK message contained in the data field. The message header is contained in the `MsgHdr` union in Appendix C. This field is included if the application buffer is a data transfer or a completion event. The format of this field depends on the type of transfer and is defined by one of the following structures defined in Appendix C:

`NetVarHdr`	Network variable update or completion code when *network interface selection* is enabled as described under *Network Variable Processing Option* later in this chapter.
`ExpMsgHdr`	All other data transfers and completion codes.

- **Network Address**. The network address specifies the destination address for downlink explicitly addressed application buffers, or the source address for uplink application buffers. The network address is contained in the `Explicit_Addr` union in Appendix C. This field is included if the application buffer is a data transfer or a completion event and *explicit addressing on* is enabled as described under *Explicit Addressing On* later in this chapter. The format of this field depends on the type of transfer, and is defined by one of the following structures defined in Appendix C:

`SendAddrDtl` Outgoing explicit messages or network variable updates.

`RcvAddrDtl` Incoming explicit messages or unsolicited network variable updates.

`RespAddrDtl` Incoming responses or network variable updates solicited by a poll.

- **Data**. The data field defines the data to be transferred. The data field is contained in the `MsgData` union in Appendix C. This field is included if the application buffer is a data transfer or a completion event. The format of this field depends on the type of transfer, and is defined by one of the following structures defined in Appendix C:

`UnprocessedNV` Network variable update or completion event when *host selection* is enabled as described under *Network Variable Processing Option* later in this chapter. This field addresses the network variable using the network variable selector; the host application translates the network variable index to and from a network variable selector. Completion events include only the network variable selector contained in the first two bytes.

`ProcessedNV` Network variable update or completion event when *network interface selection* is enabled as described under *Network Variable Processing Option* later in this chapter. This field addresses the network variable using the network variable index; the network interface translates the network variable index to and from a network variable selector. Completion events include only the network variable index contained in the first byte.

`ExplicitMsg` Explicit message or completion code. Completion events include only the message code contained in the first byte. Explicit message formats are defined in Appendix B of the *NEURON CHIP Data Book*.

When working with application buffers, note the following:

- The structure for the application buffer is different depending on whether *explicit addressing on* or *explicit addressing off* is selected as described under *Explicit Addressing Option* later in this chapter. When *explicit addressing off* is selected, the application buffer does not include the 11 byte explicit address field. When *explicit addressing on* is selected, an additional 11 bytes are included to accommodate the explicit address.
- The length field in the application buffer header describes the length of the message only, not the message plus the explicit addressing field.

- All downlink LONTALK messages that are not local network management or network diagnostic messages will eventually result in an uplink completion event message. The completion event message can be used to determine if an acknowledged message is received by all addresses. It is the responsibility of the host application to process these events appropriately.
- The command type of completion event messages is `niCOMM+niRESPONSE`. The `cmpl_code` field of the application buffer should be checked for pass/fail status. This field is zero for incoming LONTALK messages.
- Application buffers must be large enough to hold the largest network variable, explicit message, or response used by the application. Typically, the largest network management message is 17 bytes.

Network Interface Configuration Options

The types of messages passed between the host and the network interface are determined by configuration options specified for the network interface. Defaults for these options specify the type of network variable processing performed by the network interface, the size of the network variable configuration table, use of explicit addressing, and the amount of buffering within the network interface. These options are selected when the network interface is built. If you are building a network interface based on the MIP, specify these options as described in the *Microprocessor Interface Program (MIP) User's Guide*. The settings for these options for the SLTA are described in the *Serial LONTALK Adapter User's Guide*. The settings of these options for third-party network interfaces should be specified in the third-party network interface documentation.

Network Variable Processing Option

There are two values for the network variable processing option: *host selection* and *network interface selection*. These values determine whether the host processor or the network interface perform network variable selection. Network variable selection is one of the three steps a node performs when a network variable update occurs. These three steps are:

1. *Target address decoding*. This step verifies that a network variable update is addressed to the target node and is always performed by the network interface.
2. *Network variable selection*. This step determines which network variable on the node is to be updated. This step is performed by the network interface if *network interface selection* is specified; it is performed by the host application if *host selection* is specified.
3. *Network variable modification*. This step modifies the selected network variable and is always performed by the host application.

When *network interface selection* is specified, the host can declare up to 62 network variables. When *host selection* is specified, the host can declare up to 4096 network variables. To use *host selection*, the host application should process the *Update Net Variable Config* and *Query Net Variable Config* network management commands as described under *Receiving Messages* later in this chapter. If the host itself is the network manager and will not be receiving network variable binding messages from other nodes, this need not be done. *Network interface selection* is easier since the network interface handles all network variable selection. Also, *network interface selection* provides non-volatile storage of network variable configuration. *Host selection* supports more connections, and the host application must provide network variable configuration storage.

The SLTA uses *host selection*. Network interfaces used with the LONMANAGER API must use *host selection*.

Network Variable Configuration Table Size Option

When *network interface selection* is specified, this option defines the size of the network variable configuration table on the network interface. The size may be any value from 0 to 62 entries. This option is not used when *host selection* is specified.

Explicit Addressing Option

This option determines whether space is set aside in the application buffer for explicit addressing information. Specifying *explicit addressing on* adds an 11 byte explicit address field to every application buffer. The host application can use this field to specify an explicit address for any message, bypassing the address table in the network interface. This allows the host application to send LONTALK messages to an unlimited number of nodes. When *explicit addressing off* is specified, the host application can only send messages to the addresses stored in the network interface address table, which has a maximum of 15 entries; this form of addressing is called *implicit addressing*.

When *explicit addressing on* is specified, the host application may still send implicitly addressed messages by clearing the `addr_mode` bit in the message header. Responses to incoming requests must be sent with implicit addressing, since the destination address of the response is implicitly taken from the source address of the request. If the host application has network variables which have been bound by some other network management tool, then these should also be sent with implicit addressing, since the network management tool will have created address table entries in the network interface for the destinations. Otherwise, network variable updates and polls may be sent with explicit addressing.

Specifying *explicit addressing on* can also be used to get the source address of a message received from the network interface. Every LONTALK packet has a source and destination address. The destination address ensures that the packet is delivered to the correct node(s). The source address is used for generating the acknowledgement or response and also for assisting learning routers in learning the network topology. Destination nodes can also use the source address to determine which member of a group sent a network variable update. See *Monitoring Network Variables* in Chapter 3 of the *NEURON C Programmer's Guide* for more information.

The SLTA specifies *explicit addressing on*. Network interfaces used with the LONMANAGER API must specify *explicit addressing on*.

Buffer Options

Network interfaces have two types of buffers, *application buffers* and *network buffers*. The *application buffer* is used between the host and the network interface, and internally within the network interface between the application and network CPUs on the NEURON CHIP. The network buffer is used between the network and media access control (MAC) CPUs on the NEURON CHIP within the network interface. Figure 3-1 illustrates where the application and network buffers are used.

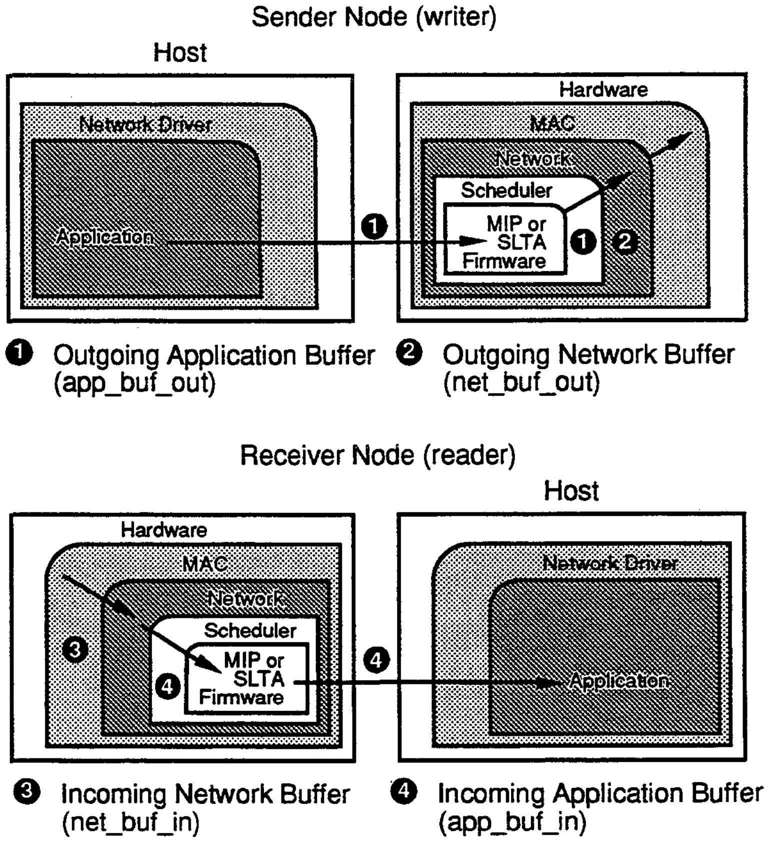


Figure 3-1 Application and Network Buffers

The number and sizes of buffers required are dependent on the host application. For example, a host application which sends large explicit messages will need large output buffers to hold the messages. An application that receives bursts of messages, such as many acknowledgements to a network variable update sent to a group, will need many input buffers.

The buffer configuration for any network interface, including the SLTA, can be changed at any time by sending network management write memory messages to it, either from the host (local network management) or over the network from some other network management tool. See the *NEURON CHIP Data Book*, section A.1, for details of the data structures within the NEURON CHIP that control the partitioning of RAM for buffers. Although the NEURON 3150 CHIP used in a network interface has 2,048 bytes of on-chip RAM, do not attempt to configure a network interface to use more than its available buffer memory, as it will most likely crash or behave erratically, since the remaining on-chip RAM is used by the system image and SLTA or MIP firmware.

For network interfaces based on the MIP, the default allocation of RAM to buffers is controlled by *pragmas* in the NEURON C source file, and the hardware properties used when the MIP image was created. See the *Microprocessor Interface Program (MIP) User's Guide* for details. If you have purchased a network interface from a third party, consult your vendor for details on the default buffer allocation specified when that device was manufactured.

Sending Messages

Host applications send LONTALK messages using the following steps:

1 Build the message within the application buffer data structures. See Appendix C of this document for the definition of the application buffer data structures.

2 Write the application buffer to the network driver.

3 Repeat step 2 if the driver returns the `LDV_NO_BUFF_AVAIL` or `LDV_DEVICE_BUSY` error codes.

4 If necessary, process (read) any response messages from the network interface. These appear as `niRESPONSE+niCOMM` commands.

5 Process the completion event messages from the network interface.

Downlink buffers (from host to network interface) can contain either outgoing messages or outgoing responses to a previous incoming request. Uplink buffers (from network interface to host) can contain incoming messages, incoming responses to a previous request, or completion events. Completion events are generated whenever an outgoing message has completed processing, and indicates whether the message succeeded or failed, indicated by the value of the `cmpl_code` field in the buffer. Buffers that are not completion events have zero in this field. For request/response service, the completion event occurs after all the responses have arrived.

For network management messages delivered to the network interface (command `niNETMGMT`), there are no completion events returned. Responses, however, are returned as usual.

The tag fields of an outgoing message, its completion event, and any responses are all the same. This allows the host application to correlate the responses and completion events with the original message. Similarly, a response generated by the host must use the same tag as that in the incoming request message or network variable poll. To further aid in correlating completion events with the original message, the first two bytes of the data field are included. This contains either the network variable selector (*host selection* enabled network variables), the network variable index (*network interface selection* enabled network variables), or the message code (explicit messaging).

The LONTALK protocol supports two paths for special purpose transceivers, and the `path` field in an incoming message or response indicates how it was received. By default, the alternate path is used for the last two transmission attempts when using Acknowledged, Request/Response, or Unacknowledged/Repeated service. For an outgoing message, the host may override this selection by setting the `path_spec` bit. In this case, the message is delivered on the channel specified by the `path` bit.

The `trnarnd` bit of an outgoing response to a network variable poll must be the same as the `trnarnd` bit of the incoming request when using *network interface selection.* This allows any node to correctly poll its own output network variables.

For outgoing network variables, the message is delivered with priority service only if the `priority` bit in the message is set. Even if *network interface selection* is enabled, the `priority` bit in the network variable configuration table is ignored. Outgoing messages with the `priority` bit set must be delivered to the priority queue, and if the priority bit is clear, they must be delivered to the non-priority queue. The host application should read the `priority` bit from the network variable configuration table so that it can use priority or non-priority service as appropriate for delivering the network variable.

See the sample host application in Appendix A for an example of sending network variable updates from a host application. In the example, the `NV_update()` function updates an output network variable by calling `ni_send_msg_wait()` to send out the update message and wait for the completion event. Similarly, the `NV_poll()` function polls an input network variable by calling `ni_send_msg_wait()` to send the poll request, and then wait for the response(s) and the completion event.

Receiving Messages

When a network interface receives an application message it is passed uplink to the host in a link-layer buffer. The application message may be a network variable update, response to a poll, poll request, or an explicit message.

When a network interface receives a network management or network diagnostic message, it is processed entirely by the NEURON CHIP within the network interface with the following exceptions (these messages are passed uplink as explicit messages):

- *Set Node Mode* (online and offline only)
- *Wink*
- *Update Net Variable Config* (only if *host selection* enabled)
- *Query Net Variable Config* (only if *host selection* enabled)

- *Query SNVT*
- *Network Variable Fetch*

See Appendix B of the *NEURON CHIP Data Book* for a description of these commands.

Messages passed to the host appear as application buffer data structures with the `niCOMM+niINCOMING` command or `niCOMM+niRESPONSE` value.

The form of network variable update messages depends on whether *host selection* or *network interface selection* is enabled.

When network variable selection is performed by the host application (*host selection*), the host application must maintain the network variable configuration table. Depending on the availability of host memory, this table may be as large as the maximum number of network variables on a node, which is 4096 entries. The network variable configuration table is updated with the *Update Net Variable Config* network management command, which is passed to the host application by the network interface and must be processed by the host application. The network variable configuration table is queried with the *Query Net Variable Config* network management command, which is also passed to the host application by the network interface and must also be processed by the host application. See the sample host application in Appendix A for an example of handling these incoming network management messages in a host application (routines `handle_update_nv_cnfg()` and `handle_query_nv_cnfg()`).

Network variable updates and polls are passed to the host application as explicit messages using the `UnprocessedNV` structure defined in Appendix C (`msg_type=0`). The host application determines the network variable to be updated or polled by decoding the network variable selector. See the sample host application in Appendix A for an example of receiving network variable updates and polls in a host application (routine `handle_netvar_msg()`).

In a typical host application which is not itself a network installation tool, the network manager initially configures the network variable configuration table when the network image is downloaded to the node. If the host application manages its own configuration, then it must initialize its own network variable configuration table, whether it is on the network interface or on the host.

A host application that may be queried by a network manager to retrieve its program information must have an initialized network variable configuration table, so that the direction, priority, and authentication attributes of each network variable may be determined.

The following table summarizes the key differences in network variable messages when *host selection* or *network interface selection* is enabled.

Option	*Network Interface Selection*	*Host Selection*
Target address decoding	Network Interface	Network Interface
Network variable selection	Network Interface	Host Application
Network variable configuration table	Network Interface	Host Application
Maximum size of network variable configuration table	62 entries	4096 entries

Format of network variable application buffers	`ProcessedNV`
	`(msg_type=1)`
		`UnprocessedNV`
		`(msg_type=0)`

See Appendix B of the *NEURON CHIP Data Book* for a definition of the network variable message structures, network variable configuration table contents, and the network variable configuration network management commands.

See the sample host application in Appendix A for an example of receiving network variable messages in a host application. In the example, the main loop alternately calls `ni_receive_msg()` to receive a message from the network interface and `kbhit()` to receive input from the keyboard. If a message has been received from the network, the function `process_msg()` determines whether this is an explicit message or a network variable message. If it is a network variable message, the function `handle_netvar_msg()` either updates the variable, or sends a poll response.

Local Control of the Network Interface

A group of network interface commands is responsible for local control of the network interface. In addition to resetting the network interface, these commands can place the network interface in a number of `FLUSH` states, and in the `OFFLINE` or `ONLINE` state. Refer to the *NEURON C Programmer's Guide* for a detailed description of these states. Refer to Appendix D for a description of the network interface commands.

Local Network Management/Diagnostics With the Network Interface

All LONTALK network management and network diagnostics commands can be addressed to the network interface directly from the host. These commands take the form of downlink LONTALK messages and do not actually appear on the LONWORKS network. Instead, they are considered addressed to the network interface itself. The request-response mechanism can be used as though communicating with a remote node. These messages are defined in Appendix A of the *NEURON CHIP Data Book*. Completion event messages are not returned. The network interface command for this process is the `niNETMGMT` + `niTQ` or `niNETMGMT` + `niNTQ` command.

However, if the host application wishes to send network management messages to some other node on the network (not the network interface itself), these are not considered to be network management messages from the point of view of the network interface, and should therefore be sent using the `niCOMM` + `niTQ` or `niCOMM` + `niNTQ` commands. Responses and completion events are returned in the same way as for any application message.

When building the application buffer portion of a locally addressed network management message, the entire network address field must still be present if *explicit addressing on* is enabled. Explicit addressing is always on for the SLTA. The contents of the network address field are ignored for locally addressed network management messages.

Local network management commands can be used by the host application to do self-installation of the host node. For example, the *Update Address* network management message can be used to update the network interface's address table. See the *NEURON CHIP-based Installation of LONWORKS Networks* engineering bulletin for more information on self-installation.

See the sample host application in Appendix A for an example of sending a local network management command. In the example, the `main()` function calls `install_prog_ID()` which creates a local network management message to write the application program ID into the EEPROM memory of the network interface, and then calls `ni_send_msg_wait()` to send it.

Binding to a Host Node

Host-based nodes can be bound to network variable and explicit message tag connections using procedures similar to NEURON CHIP-hosted nodes. Connections can be created using self-installation as described in the previous section, or connections can be created by a network management tool sending network management messages to host-based and NEURON CHIP-hosted nodes. To perform binding, a network management tool requires a description of the network variables on the nodes to be connected. The network management tool can get these descriptions for a host-based node using one of the following methods:

- **SNVT Import.** A network management tool can import the network variable descriptions over the network using the *Query SNVT* network management command described in Appendix B of the *NEURON CHIP Data Book*, even if the network variables are not all of standard types. To support this, the host application must maintain the SNVT structure defined in Appendix A of the *NEURON CHIP Data Book* and return its contents in response to the *Query SNVT* message. See the sample host application routine `handle_query_SNVT()` for an example of handling this message. Also, the network variable configuration table of the host application must be initialized with the appropriate direction, priority, and authentication attributes for each network variable. The SNVT structure contains the following:

  SNVT Header — Defines the length of the SNVT structure, the number of network variables and bindable message tags, and the format of the SNVT structure.

SNVT Descriptor Table	Defines each network variable for every network variable declared in the host application.
Node Self-Documentation	A null-terminated text string containing an optional description of the node.
Extension Records	Optional fields describing update rate estimates, the network variable name, and a text description of the network variable.

- **External Interface File Import**. An external interface file can be used with any type of LONWORKS node. The external interface file provides a text description of the node, its network variables, and its message tags. Network management tools can import external interface files to get all the information needed to create network variable and message tag connections.

  Follow these steps to create and import an external interface file:

  1 If you are using an SLTA, copy the external interface file provided with the SLTA as described in the *Serial LONTALK Adapter User's Guide.*

  or

  If you are using a third-party network interface, copy the external interface file provided with the network interface.

  or

  If you have built your own network interface, create an external interface file as described in *Exporting a Network Interface* in the *Microprocessor Interface Program (MIP) User's Guide.*

  2 Modify the external interface file. Add the network variable and message tag entries as described in Appendix B.

  3 Import this new external interface file to the network management tool. If you are importing the host node into a development network, import the external interface file as described under *Importing External Interface Files* in Chapter 7 of the *LONBUILDER User's Guide.* If you are using the LONMANAGER API for DOS or Windows, use the `ldb_import_xif()` function.

4
Using a Network Driver

This chapter describes specifications for using a LONWORKS network driver. Network interfaces for DOS should come with a DOS network driver. If you are using the Serial LONTALK Adapter, refer to the *Serial LONTALK Adapter User's Guide* for instructions on how to install the SLTA driver. If you are using a different network interface, see their specific documentation on how to install the associated driver. Network driver services and functions are described.

The Network Driver

The network driver provides a device-independent interface between the host application and the network interface.

The LONTALK network driver protocol defines four functions that should be provided by every network driver. These functions are `ldv_open()`, `ldv_close()`, `ldv_read()`, and `ldv_write()`. The `ldv_open()` function initializes the network driver and network interface. The `ldv_close()` function deallocates any resources assigned by the `ldv_open()` function. The `ldv_read()` and `ldv_write()` functions transfer application buffers uplink from the network interface and downlink to the network interface. The syntax for these functions may be operating system dependent. For example, the DOS network driver function calls are described under *DOS Network Driver Services* later in this chapter.

Standard Network Driver Services

The functions and services defined by the LONTALK network driver protocol are:

```
typdef int LNI;
```

```
LNI handle = ldv_open(char *device_name);
```

Initialize the network interface and return a handle for accessing the network interface. If the network interface is held in a reset state after power-up, cancel the reset state.

Initialization includes cancelling the network interface *Flush* state. After a network interface is reset, the network interface enters the *Flush* state. While in the *Flush* state, the network interface ignores all incoming messages and will not send any outgoing messages, even service pin messages. The *Flush* state is provided to prevent a network management tool from performing network management functions on the network interface before the host has configured the network interface. This state is cancelled with the `niFLUSH_CANCEL` command from the host. After the *Flush* state is cancelled, the network interface is in the *Normal* state.

The network interface sends a `niRESET` command uplink following any reset. This will be the first message received by the host whenever the network interface is reset.

```
LDVCode error = ldv_read(LNI handle, void *msg_p, unsigned length);
```

Read an application buffer from the network interface. The `msg_p` argument is a pointer to an application buffer. Application buffers are defined in Chapter 3. If a buffer is not available, return the `LDV_NO_MSG_AVAIL` error code.

```
LDVCode error = ldv_write(LNI handle, void *msg_p, unsigned length);
```

Write an application buffer to the network interface. The msg_p argument is a pointer to an application buffer. Application buffers are defined in Chapter 3. If a buffer is not available, return the LDV_NO_BUFF_AVAIL error code.

```
LDVCode error = ldv_close(LNI handle);
```

Free any resources assigned to the network interface identified by handle, and free the handle. Optionally hold the network interface in a reset condition.

DOS Network Driver Services

A standard LONWORKS network driver interface has been defined for MS-DOS. This standard allows DOS applications, such as a host application incorporating the LONMANAGER API for DOS or Windows, to transparently interface with different network interfaces. The host application can initially be debugged with the network driver for the LONBUILDER Development Station. Later, that same host application program can be used with a network driver for a Serial LONTALK Adapter or with the network driver for a custom network interface based on the MIP.

Installation of the network driver for the SLTA is described in the *Serial LONTALK Adapter User's Guide*. Installation of a network driver for a third-party network interface should be provided by the manufacturer of the network interface. Development and installation of the network driver for a custom network interface based on the MIP is described in the *Microprocessor Interface Program (MIP) User's Guide*.

The network driver and installation instructions for the LONBUILDER development station are included in the LONLINK™ Sampler diskette included with LONBUILDER.

Following are the services provided by a standard DOS network driver:

OPEN To open the network interface use the DOS device/file open function with the device name LON(n). In an ANSI C application the open() function can be used with the access bits O_RDWR and O_BINARY set. This function returns a handle value which is then used in subsequent device operations:

```
int handle = open(char *
        "LON1", int O_RDWR|O_BINARY);
```

The open command invokes the autobaud sequence for the SLTA driver if autobaud is enabled on the driver and the SLTA. The autobaud sequence causes the SLTA to synchronize with the host baud rate.

IOCTL

Once the network driver is opened, it is ready to act as the connection between the LONWORKS network and the host application. Messages are passed between the host application and the network interface via write or read DOS function calls. There are other paths to these services within the network driver if the user chooses to use them. The DOS `ioctl()` function is used to fill in a data structure passed to it with function pointers to three functions within the driver:

```
ldv_read_direct()
ldv_write_direct()
ldv_register()
```

The ANSI C usage is:

```
int ioctl(int handle, int function, far struct
            adapter_info_s *argdx, int size);
```

The `ioctl()` function is provided as part of the Borland C standard runtime library. Microsoft C does not provide this function, and so the source code for `ioctl()` is provided with the sample host application for use with the Microsoft compiler.

The value of `function` is 2, for 'read from device'. The usage of the pointers returned in the `adapter_info_s` structure is described in the next section.

```
struct adapter_info_s {

  unsigned char ioctl_stat;     //returned status

  LDVCode (far *read_fn) (void * msg_p, unsigned length);
                          //pointer to ldv_read_direct()

  LDVCode (far *write_fn) (void * msg_p, unsigned length);
                          //pointer to ldv_write_direct()

  void (far *register_fn) (int handle,
        void (interrupt far *callback)(void));
                          //pointer to ldv_register()
};
```

The error codes returned by the driver direct functions are defined in Chapter 5.

WRITE

The standard DOS write function can be used by the host application to transfer either network interface local commands or LONTALK messages to the network interface from the application via the driver. The ANSI C usage is:

```
int length = write(int handle, void far *msg_p,
                   unsigned length);
```

The msg_p argument is a pointer to an application buffer. Application buffers are discussed in Chapter 3. The first thing in an application buffer will always be the network interface command; i.e., the NI_Hdr structure defined in Appendix C. The length argument is the number of bytes in the application buffer. The return value is the number of bytes written, or -1 if an error occurred. If there are no output buffers available in the network driver, the return value is zero. I/O blocking does not occur, that is, the function will not wait for an output buffer if none is available.

READ

The standard DOS read function can be used by the host application to transfer either responses to network interface local commands or LONTALK messages from the network interface to the host application via the driver. The ANSI C usage is:

```
int read(int handle, void far *msg_p, unsigned
         length);
```

The msg_p argument is a pointer to an application buffer. Application buffers are discussed in Chapter 3. The first thing in an application buffer will always be the network interface command; i.e., the NI_Hdr structure defined in Appendix C. The length argument is the number of bytes in the application buffer. The return value is the number of bytes read, or zero if no input buffers were ready, or -1 if an error occurred. I/O blocking does not occur, that is, the function will not wait for a full input buffer.

CLOSE

To close the network interface, use the DOS device/file close function with the handle returned by the open function:

```
int error = close(int handle);
```

This function call closes the network interface, and the network interface is reset. The device's callback function pointer (see below) is cleared and the callback is disabled.

Driver Direct Functions

There are three driver direct functions available to the user. The ioctl() function call returns pointers to these functions once the network interface has been opened. These functions are required by the LONMANAGER API. The driver direct functions are preferable to DOS read and write calls because error conditions can be handled better. Error conditions within the DOS functions will result in a DOS error prompt unless a DOS critical error handler is installed. The three functions are:

1 `LDVCode error = ldv_write_direct(void far *msg_p, unsigned length);`

This function is similar to the DOS write function, except that it is a direct call to the network driver rather than a call via DOS. Note that a handle is not required since this is a direct function call. The `msg_p` and `length` arguments are the same as those described for the DOS `WRITE` function. This function will return the error code `LDV_NO_BUFF_AVAIL` if no output buffers are available, whereas the DOS write function will return zero.

2 `LDVCode error = ldv_read_direct(void far *msg_p, unsigned length);`

This function provides the read complement of `ldv_write_direct()`. The `msg_p` and `length` arguments are the same as those described for the DOS `READ` function. While the DOS read function returns a zero value if no input messages are available to be read, this function returns the error code `LDV_NO_MSG_AVAIL`.

3
```
void ldv_register(int handle, void (interrupt far
    *callback)(void));
```

This function registers the handle value and the callback function pointer within the driver. The callback function is a function within the user application called by the driver whenever a network driver input buffer is filled by the network interface under interrupt control. The handle value will be included in the call to the callback function in the CPU's AX register. The user's callback function may then flag this event or call the `ldv_read_direct()` function.

NOTE: *This callback function occurs under interrupt control, so the callback function must not do much more than the single read from the device. It should not make any DOS calls, or call any runtime library routines that make DOS calls, since DOS is not reentrant. It should also be careful when accessing static data that may simultaneously be accessed by the main application. The callback function must also behave like an interrupt function by loading its DS register with the application's data segment since the DS register will be set to the driver's data segment. An IRET instruction should terminate this function. This is handled automatically by the compiler when the* `interrupt` *keyword is used in the declaration of the callback routine.*

5
Error Conditions

This chapter describes error conditions that may be encountered by host applications. Errors detected by the network interface and network driver also are described.

Errors Detected By the Host Application

There are several basic types of error detection in a host application. Host applications interact with a network driver, and detect errors based on return codes from I/O calls to the network driver. These errors are typically due to problems accessing the network interface, for example, incorrect jumper assignments, incorrect driver option settings, or hardware problems in the network interface. The network driver error responses are operating system dependent; this section describes the error responses from a DOS network driver.

The second type of error is due to transmission errors on the network, or temporary buffer congestion problems in either the sending or receiving nodes. These are normally handled transparently by the network protocol, which retries the transactions up to the configured retry count. The application is unaware of any retries if the transaction is ultimately successful.

The third type of error occurs when a network transaction fails to complete successfully, and the host application is informed by a failure completion code. This is most often due to the destination nodes not being accessible across the network because the message was misaddressed, the nodes were powered down, the network was disconnected or extremely busy or noisy.

For host applications that perform network management operations on other nodes on the network, a fourth type of error can occur when the receiving node rejects the network management request. It does this by returning a corresponding error code in the response.

The function `ni_handle_error()` in the sample host application handles all types of errors after a call has been made to send a message and wait for the completion event. The arguments to this function are the function return value from the driver, the completion code for the transaction, and the returned message code if the message was a network management message.

Driver Not Installed

If the network driver is not installed, or if the device name in the `open()` function call does not match the device name used to install the driver, DOS returns a `No such file or directory` error when `open()` is called. The network driver is installed with a `device` statement in the DOS `CONFIG.SYS` file as described in Chapter 4. The device name is also specified in the same `device` statement, and defaults to `LON1` for the SLTA, MIP, and LONBUILDER network drivers.

Wrong Driver Invoked

If a driver is opened that is not a LONWORKS network driver, DOS may return an `Invalid argument` error. If this error is not returned, the error symptoms will be the same as if the network interface is not installed.

Network Interface Not Installed

If a network driver is installed, but the network interface is not installed, the host application will not receive any responses to messages. The same error will occur if the network interface is installed, but is configured incorrectly, e.g. the wrong baud rate is selected on an SLTA. The first message sent by a host application should be a niRESET command to the network interface. If installed and correctly configured, the network interface will respond with an uplink niRESET message upon completion of the reset. By waiting for this message with an appropriate timeout (e.g., 2 seconds), the host application can determine if a network interface is installed and operating correctly. See the ni_init() function in Appendix A for an example.

Power Lost to Network Interface After Start-Up

Network drivers that use polling will generally detect a power loss while transferring an application buffer to or from the network interface, and will return the LDV_DEVICE_ERR error code.

Network drivers that are fully interrupt driven will generally not detect an error during a transfer since all the bytes in a packet may be stored in an interrupt buffer on the host. In this case, loss of power will be detected by lack of response to a network message, which must be detected by a timeout in the host application. Since all network messages result in a completion code from the network interface upon successful completion or failure of the message, lack of a completion code within a long period of time (e.g., 10 seconds) can be used to flag loss of the network interface.

In either case, the host application should close, reopen, and reset the network driver. Doing this will restart the autobaud sequence if the network interface is an SLTA with autobaud enabled. If there is still no response from the reset, the network interface cannot be restarted. See the ni_init() function in Appendix A for an example of closing and reinitializing a network interface.

Destination Node Not Available

If the network driver and network interface are installed and operating correctly, but the destination of an acknowledged message or request message is not available, the driver will return the MSG_FAILS completion code. The MSG_FAILS completion code is returned after all retries have been attempted for a message. This completion code is the equivalent of the msg_fails event in NEURON C.

Network Management Request Failed

If a network management request is received by a node, the response contains a code which indicates whether the request was acted on. Failures are typically due to parameters of the request being out of range, for example, a request to query a non-existent entry in a table. See the *NEURON CHIP Data Book* Appendix B for more details.

Errors Detected By a Network Driver

Downlink Timeouts

Downlink timeouts can occur as a result of the network interface's inability to transmit messages on the network at the desired rate. This condition can be seen as the absence of the niACK from the network interface within a user selected period. This error detection scheme can be built into the host's network driver, and is built into the SLTA network driver.

Host Detection of Hardware Failures

Host detection of hardware failures is performed by various parts of the network driver. A *hardware failure* is a timeout condition on the handshake signal from the network interface. If a message transfer starts but is not completed due to this timeout, a device failure error will be returned to the host application. When this occurs, the host should perform a hardware reset of the network interface.

For network interfaces based on the MIP/P20 or MIP/P50, if the host driver has given up the write token, and the network interface has not returned it for some period, such as 500 ms, a hardware error state may be flagged.

For the SLTA, error conditions are typically a result of lost data bytes due to transmission errors between the SLTA and the host. Lost ALERT bytes are handled by both the SLTA and the driver by virtue of a timeout/retry feature. Lost downlink buffer requests are handled by the driver through a timeout/retry feature. If the command or data portion of any other type of transfer is corrupted, it will be the responsibility of the host software to deal with this condition. Corrupted uplink transfers are flagged with an error status which results in an error code (LDV_DEVICE_ERR) when the driver read service is used. The driver has no method of detecting if a downlink transfer has been corrupted. It is up to the host application to determine that these transactions have failed by implementing some form of a timeout at that level.

Error Codes Returned to the Driver Direct Functions

The driver direct functions for LONWORKS standard network drivers return the following error codes:

LDV_OK	0	No errors detected, operation successful.
LDV_ALREADY_OPEN	2	This network interface was already open.
LDV_NOT_OPEN	3	Access to this network interface denied, it is not open.
LDV_DEVICE_ERR	4	Hardware error detected.

LDV_NO_MSG_AVAIL	6	For read operations, no messages buffered.
LDV_NO_BUFF_AVAIL	7	For write operations, no message buffers available.
LDV_DEVICE_BUSY	8	Try again later. This network interface is being initialized.

Errors Detected By a Network Interface

The network interface ignores command bytes that it does not understand. If the network interface states are violated (these states are not applicable to the MIP/DPS), and the host passes downlink messages without first requesting a buffer, the network interface will ignore the downlink message. No error response mechanism is provided. The network interface states are managed by the network driver.

Other error statistics, such as those normally tallied by all application nodes, can be accessed locally via the *query status* network diagnostic message to the network interface. See the *NEURON CHIP Data Book* Appendix B for a description of the network diagnostic messages. The *query status* diagnostic message may be sent from LONBUILDER by activating the *Target HW/Test* command.

For network interfaces based on the MIP/P20 or MIP/P50, the parallel I/O protocol must be maintained. Confusion over who owns the write token nearly always results in a lock-up of one of the processors (network interface or host) once a non-null data message transfer occurs. The parallel I/O states are manged by the network driver.

A network interface based on the MIP /P20 or MIP/P50 will execute a watchdog timeout reset under the following conditions:

- If the network driver initiates a parallel I/O transfer but does not complete the operation.
- If the network driver stops servicing the network interface when the network interface owns the write token (since it is waiting to transfer the write token back to the host).

Refer to the *Parallel I/O Interface to the NEURON CHIP* engineering bulletin for additional information.

A network interface based on the MIP/DPS will execute a watchdog timeout reset under the following conditions:

- If the network driver holds any of the semaphores for too long.
- If the network driver corrupts the control structure or buffers in shared memory.

Appendix A
Sample Host Application

This appendix contains source code for an example DOS host application that can both send and receive polls and updates to network variables, and whose network variables may be bound by an installation tool that sends it network management messages. This host application does not itself do any network management of other nodes. A PC-based host application that performs network management is best implemented using the LONMANAGER API, rather than the host application framework.

Sample Host Application Overview

The example is included in the host application example directory and consists of five C source files, three C header files, and a LONWORKS node external interface file HA_V2.XIF. The source files are HA.C, APPLMSG.C, HAUIF.C, NI_MSG.C, and IOCTL.C. The HA.C file contains the main program for the example; the APPLMSG.C file contains the functions for handling network management and network variable messages; the HAUIF.C file contains the functions for a primitive command-line user interface, and the NI_MSG.C file contains general purpose functions for calling the network driver. The file IOCTL.C is required only when using the Microsoft C compiler. The header files are HAUIF.H, NI_MSG.H and NI_MGMT.H . The file HAUIF.H contains prototype declarations for the user interface functions; the file NI_MSG.H contains complete data structure declarations for the message buffers that are passed to or received from the network interface driver, as well as prototype declarations for the functions in NI_MSG.C; the file NI_MGMT.H contains data structure and message code declarations for the network management messages and for the application-layer data structures for the host application. The files NI_MSG.H and NI_MGMT.H are reproduced in Appendix C.

These files may be compiled and linked with any ANSI C compiler and linker. The file makefile is a make file for Borland C, and the file msoft.mak is a make file for Microsoft C. To make the example, copy the example source files, header files, and makefile to a working directory. For Borland C, type MAKE. The Borland C bin directory must be in your path. The default path for the Borland C lib directory must be specified in tlink.cfg, as described in the Borland C documentation. For Microsoft C, type NMAKE /F MSOFT.MAK. The Microsoft C, the bin directory must be in your path. The DOS environment variables INCLUDE and LIB must be set up to point to the Microsoft include and library directories respectively as described in the Microsoft documentation.

An executable version of the example is also included in the host application example directory. The file name is HA.EXE.

Sample Host Application Requirements

The example host application is structured in a top down way that may easily be modified. The example was designed for the following requirements, but for specific applications, code will most likely need to be added or removed.

- The application supports any network interface that has a LONWORKS standard network driver for DOS. The network interface is configured with host network variable selection, and with explicit addressing on. This is the way that the SLTA is configured.
- Network variables on the node may be bound by a network management tool such as the LONBUILDER network manager, NetMaker, or a network management tool based on LONMANAGER API. This means that the host application must support incoming network management messages to query and update its network variable configuration table, as well as messages to go on-line and off-line.

Note. Two PCs are required to bind this example host application into a network. One PC runs the network management tool (for example, the LONBUILDER network manager or NetMaker). The other PC runs the host application. The host application must be running when the network image is loaded into the host node because it must process incoming network management messages during the loading process. It is not possible to run a DOS-based network management tool and this example host application in different DOS sessions under Microsoft Windows, because Windows cannot switch contexts fast enough between the two DOS applications. This host application is not able to return responses soon enough to the network management tool to avoid transaction timeouts. However, if both the network management tool and the host application are native Windows applications, multi-tasking may be achieved because the applications can relinquish the CPU when waiting for network events.

- The node may be queried by a network management tool to retrieve self-identification and self-documentation information so that an external interface definition for the node may be created. The LONBUILDER *Query* command uses this capability, as does the LONMANAGER API function `lxt_install_node()` in the case that there is no program record for the node defined in the database. This function is not a requirement for all nodes, since the external interface definition may be supplied separately as an external interface file (`.XIF` extension) to the network management tool.
- The network variables of the node may be polled using the *NV fetch* network management message. The node therefore responds appropriately to a network variable browser, such as the one in LONBUILDER and the Data function in NetMaker.
- The network variables of the node may be updated on receipt of a user command from the keyboard. Output network variables that are updated are propagated over the network to whatever nodes they are bound to. The application also has input network variables that are polled on receipt of a user command from the keyboard. The poll command sends the appropriate request message to the destination address that the network variable is bound to, and handles poll responses from single nodes as well as groups of nodes.
- The input network variables of the node are updated when the node receives the relevant update message from the network. Network variables may also be polled when the node receives a request message over the network, and the application will return the correct poll response. This means that the application has to support network variable selection appropriately.

Sample Host Application Data Structures

The host application does not directly use the C native data types `char`, `int`, and `long` for any of its data declarations that need to be compatible with the NEURON CHIP data or message structures. This is because these types differ depending on the host architecture and the compiler used. Instead, the application uses the `typedef` names `byte`, `word`, and `bits` for unsigned 8-bit, unsigned 16-bit, and bit-field data objects respectively. These `typedefs` are defined in the file `NI_MSG.H`. For bitfields, most C compilers for the 80x86 little-endian architecture assign bits from right to left within a byte, whereas the NEURON C compiler and compilers for other big-endian architectures assign bits from left to right. Note that the Microsoft C compiler always assigns bitfields to an even number of bytes, whereas the Borland C compiler allows bitfields to occupy any number of bytes. In most cases the source code is identical, but in some places the symbol _MSC_VER is tested to determine if the Microsoft compiler is being used.

The main data structures used by the host application are:

- A network variable configuration table whose structure matches the network variable configuration structure defined for any LONWORKS node. See the *NEURON CHIP Data Book*, section A.4 for a description of the network variable configuration structure. The definition of this type is in the file NI_MGMT.H, and is reproduced here:

```
typedef struct {
    bits    selector_hi : 6;
    bits    direction   : 1;     // use nv_direction
    bits    priority    : 1;
    bits    selector_lo : 8;
    bits    addr_index  : 4;
    bits    auth        : 1;
    bits    service     : 2;     // use ServiceType
    bits    turnaround  : 1;
} nv_struct;

#define NULL_IDX  15             /* unused address table index */
typedef enum { NV_IN = 0, NV_OUT = 1 } nv_direction;
typedef enum { ACKD = 0, UNACKD_RPT = 1,
               UNACKD = 2,    REQUEST = 3 } ServiceType;
```

This host application is compiled with eight network variables, four outputs and four inputs. The network variable configuration table itself is declared in the file APPLMSG.C, as follows:

```
#define NUM_NVS 8
     /* Number of Network Variable table entries on this node */

nv_struct nv_config_table[ NUM_NVS ]  // configuration table
    = {
/*        selhi  dir     prio   sello  addridx   auth   svc   trnarnd */
        { 0x3F, NV_OUT, FALSE, 0xFF, NULL_IDX, FALSE, ACKD, FALSE },
        { 0x3F, NV_OUT, FALSE, 0xFE, NULL_IDX, FALSE, ACKD, FALSE },
        { 0x3F, NV_OUT, FALSE, 0xFD, NULL_IDX, FALSE, ACKD, FALSE },
        { 0x3F, NV_OUT, FALSE, 0xFC, NULL_IDX, FALSE, ACKD, FALSE },
        { 0x3F, NV_IN , FALSE, 0xFB, NULL_IDX, FALSE, ACKD, FALSE },
        { 0x3F, NV_IN , FALSE, 0xFA, NULL_IDX, FALSE, ACKD, FALSE },
        { 0x3F, NV_IN , FALSE, 0xF9, NULL_IDX, FALSE, ACKD, FALSE },
        { 0x3F, NV_IN , FALSE, 0xF8, NULL_IDX, FALSE, ACKD, FALSE }
    };
```

The initializer for the network variable configuration table is strictly speaking not necessary, although it *is* necessary to set the direction, priority, authentication and service type bits appropriately for each network variable *if* the node is to be imported over the network in order to create its external interface definition. When the host application is started up, it searches for a file called NVCONFIG.TBL in the current working directory. If it finds the file, it reads the file into the network variable configuration table (function load_NV_config() in the file APPLMSG.C). When the user types the Exit command to leave the host application, it writes the network variable configuration table to the file NVCONFIG.TBL (function exit_func() in

APPLMSG.C). In this way, any binding information that has been loaded into the network variable configuration table will be saved for the next invocation of the host application.

- Self-identification and self-documentation data structures for the network variables (SNVT information). These data structures match the data structures defined in the *NEURON CHIP Data Book*, section A.5, and are required only for LONWORKS nodes that need to support uploading of their external interface definition by a network management tool. The declarations of the elements of this structure are in the file NI_MGMT.H, and are reproduced in appendix C.

  Either version 0 or version 1 formats may be used for nodes that have less than 255 network variables, but version 1 format is required if there are more than this. Version 1 format also supports a compact representation of network variable arrays. This example host application uses version 0 format.

  For illustrative purpose, the network variables of this application are all of standard types (ASCII string, discrete level, continuous level, and floating point count types), although this is, of course, not required. The SNVT information in this node includes the names of each network variable, a self-identification string for each network variable, and a self-identification string for the node. This information is the same as would be generated by the following set of NEURON C declarations. See the *NEURON C Programmer's Guide* chapter 3, and the *LONBUILDER Release 2.2 Supplement* chapter 4 for more information.

```
#pragma enable_sd_nv_names
#pragma set_node_sd_string "Sample host application program"

network output sd_string( "ASCII string output NV" )
        SNVT_str_asc  NV_string_out;
network output sd_string( "Discrete output NV" )
        SNVT_lev_disc NV_disc_out;
network output sd_string( "Continuous output NV" )
        SNVT_lev_cont NV_cont_out;
network output sd_string( "Floating count output NV" )
        SNVT_count_f  NV_float_out;

network input sd_string( "ASCII string input NV" )
        SNVT_str_asc   NV_string_in;
network input sd_string( "Discrete input NV" )
        SNVT_lev_disc  NV_disc_in;
network input sd_string( "Continuous input NV" )
        SNVT_lev_cont  NV_cont_in;
network input sd_string( "Floating count input NV" )
        SNVT_count_f   NV_float_in;
```

  The declaration of the SNVT information for this node is the const data object SNVT_info in the file APPLMSG.C. The initializer for this object is the data that the NEURON C compiler would produce with the above declarations. Portions of this structure are returned as responses to the network management request *Query SNVT*.

- Network variable storage data structures. These structures do *not* have to correspond to any data structures defined for the NEURON CHIP. Network management tools do not directly access these structures, and so they can be designed as needed for the host application's requirements. Specifically, there is no need to have a network variable fixed table as defined in the *NEURON CHIP Data Book*, section A.4.2. For this example application, the data type `network_variable` is used to store application-level data for each network variable. This `typedef` is in the file `NI_MGMT.H` and is reproduced here:

```
typedef struct {
    int             size;
    nv_direction    direction;
    const           char * name;
    void            ( * print_func )( byte * );
    void            ( * read_func )( byte * );
    byte            data[ MAX_NETVAR_DATA ];
} network_variable;
```

  The `size`, `direction`, and `data` fields are used when the network variable is updated or polled.

  The structure defining all the network variables is in the file `APPLMSG.C`, and is reproduced here.

```
network_variable nv_value_table[ NUM_NVS ] =  {
     { 31,  NV_OUT, string_out_name, print_asc,   read_asc   },
     {  1,  NV_OUT, disc_out_name,   print_disc,  read_disc  },
     {  1,  NV_OUT, cont_out_name,   print_cont,  read_cont  },
     {  4,  NV_OUT, float_out_name,  print_float,read_float},
     { 31,  NV_IN,  string_in_name,  print_asc,   read_asc   },
     {  1,  NV_IN,  disc_in_name,    print_disc,  read_disc  },
     {  1,  NV_IN,  cont_in_name,    print_cont,  read_cont  },
     {  4,  NV_IN,  float_in_name,   print_float,read_float}

};
```

  The routines `print_`*xxx* and `read_`*xxx* (in file `APPLMSG.C`) are simple command-line-oriented user interface routines to display and input values of these types to and from the user. They also perform the necessary transformations between NEURON CHIP data representations, and PC-compatible or user-friendly data representations, for example, byte-swapping of multi-byte numeric objects, decoding and encoding of enumerations, and conversions to and from customary units. The variable names are pointers to the names in the self-identification structure (to save space), and are used in the user interface when displaying menus of network variables. If the node has no self-identification data, ASCII string constants can be used instead.

Sample Host Application Architecture

The host application is divided into four files. `NI_MSG.C` is a layer on top of the network driver that handles outgoing message transactions (`ni_send_msg_wait()`), and incoming application messages (`ni_receive_msg()`). The main program in `HA.C` initializes the network interface and then enters an infinite loop, alternately calling `ni_receive_msg()` to look for incoming network traffic, and `kbhit()`, a DOS function to look for user keystrokes from the keyboard. If a message is received from the network, the function `process_msg()` in the file `APPLMSG.C` is called to handle the message, whether it be an incoming network management, network variable, or explicit application message. This routine is essentially a switch() statement that dispatches a handler depending on the message code of the incoming message. If the incoming message is a network management request or network variable poll, the handler calls ni_send_response() to return the appropriate response.

On the other hand, if a keystroke is detected, the main loop calls the function process_cmd() in the file HAUIF.C to dispatch the user command. The commands are defined in the array command_table in the file HAUIF.C as follows:

```
typedef struct  {
    char    letter;                /* command letter */
    void    ( * func )( void );  /* handler function */
    char    * help_text;
} command_struct;

const static command_struct command_table[ ] = {

{ 'E', exit_func,   "(E)xit this application and return to DOS" },
{ 'N', NV_table,    "(N)etwork Variable configuration table",   },
{ 'P', NV_poll,     "(P)oll input network variable"             },
{ 'T', traffic,     "Incoming network (T)raffic summary"        },
{ 'U', NV_update,   "(U)pdate network variable"                 },
{ 'V', verbose,     "Control (V)erbose modes"                   },
{ '\r', null_cmd,   ""                                          },
{ '\0' }
};
```

The routine process_cmd() in the file APPLMSG.C implements a simple table-driven command dispatcher for these commands. Additional user commands may be easily added to the table. Note that the network driver is checked for incoming traffic only when the application is in the main polling loop. If the application is waiting for user input somewhere else, incoming traffic may back up into the network driver buffers and the network interface buffers causing incoming message transactions to fail.

The command functions are as follows:

E Writes the network variable configuration table to disk, and quits back to DOS.

N A debug command to display the current contents of the network variable configuration table.

P Displays a menu of input network variables and polls the variable that the user selects.

T Displays incoming traffic statistics - the number of application messages, network variable updates, network variable polls, and total bytes of explicit message data received.

U Displays a menu of network variables, and updates the variable that the user selects with the value that the user entered.

V Controls the state of two verbose mode flags. All messages to and from the network interface may be displayed in detail - this is useful for debugging the host application. The default setting of this flag is off. All messages received by the host application may be reported in summary. The default setting of this flag is on.

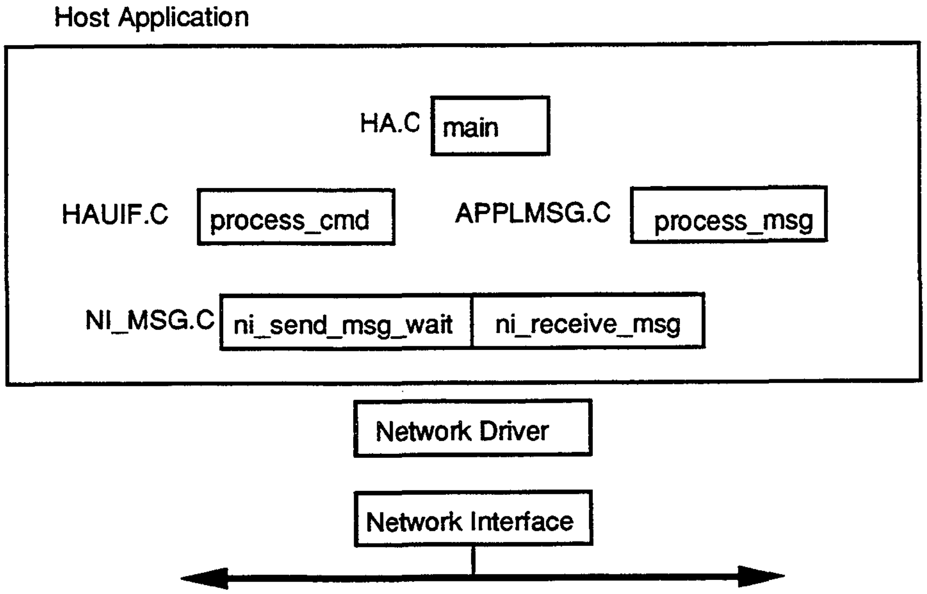


Figure A.1 Top Level Architecture of the Host Application

Network Interface Library

The file NI_MSG.C contains a general-purpose layer on top of the network driver that can be used in any host application. This library handles LONTALK message transactions for the host application, which can therefore be simplified. The main entry points to this library are as follows. All functions return an error code of type NI_Code, defined in the file NI_MSG.H.

ni_init()

```
NI_Code ni_init(char * device_name);
```

This function initializes the network interface. It opens the network driver using a DOS open() call. When the device is opened, the driver automatically sends an niFLUSH_CANCEL command to the network interface, unless the /Z switch was specified when the driver was loaded. The /Z option to LONWORKS network drivers for DOS is provided for those circumstances where the host application must configure the network interface before it is allowed to respond to incoming messages. In that case, the host application must explicitly send the niFLUSH_CANCEL command after every reset. This host application example assumes that the driver automatically sends the niFLUSH_CANCEL when it is opened, and so the application does not need to do so.

The ni_init() function then calls the DOS ioctl() function to get the address of the driver information structure that allows direct calls to the driver. If the driver was successfully opened, it sends a local reset command to the network interface. The DOS device name for LONWORKS network interfaces is LONx, where x, the device unit number, is set by a switch in the DEVICE= command that loads the device driver in the CONFIG.SYS file when the PC is bootstrapped. See the driver documentation for details of how to set the unit number. All drivers default to unit 1, so that the default device name is LON1. This host application example only supports a single network interface.

ni_reset()

```
NI_Code ni_reset(void);
```

This function sends a local reset command to the network interface. It may be used by host applications that perform local network management on the network interface, and need to reset it.

ni_send_msg_wait()

```
NI_Code ni_send_msg_wait(
    ServiceType       service,
    const SendAddrDtl * out_addr,
    const MsgData     * out_data,
    int               out_length,
    boolean           priority,
    boolean           out_auth,
    ComplType       * completion,
    int             * num_responses,
    RespAddrDtl     * in_addr,
    MsgData         * in_data,
    int             * in_length );
```

This function executes a complete outgoing LONTALK message transaction through the network interface. It sends the message, and then waits for any responses and the message completion code before returning. Other messages that may arrive at the node while the transaction is underway are not returned to the caller, but are stored in a heap for later retrieval. This greatly simplifies the logic of a host application, which can defer the processing of these unanticipated incoming messages until later.

Input Parameters

ServiceType service - the LONTALK service type to use for the outgoing message. Network variables may be sent out using ACKD, UNACKD_RPT, or UNACKD service. Network variable polls use REQUEST service.

SendAddrDtl out_addr - the destination address to use for the outgoing message. This data type is a union of six different possibilities, and is defined in the file NI_MSG.H as follows. Also, see the *NEURON CHIP Data Book*, section A.3 for more information on destination addresses.

```
typedef enum {
    UNASSIGNED      = 0,
    SUBNET_NODE     = 1,
    NEURON_ID       = 2,
    BROADCAST       = 3,
    IMPLICIT        = 126,
    LOCAL           = 127
} AddrType;

typedef union {
    SendGroup       gp;
    SendSnode       sn;
    SendBcast       bc;
    SendNrnid       id;
    SendUnassigned ua;
    SendImplicit    im;
} SendAddrDtl;
```

The first byte of a destination address determines the type of addressing used. If the most significant bit of the type is set (types 0x80 - 0xFF), then the address is an explicitly addressed multicast (group) message, and the SendGroup union element is used to define the address. Otherwise, the type may have the following values:

SUBNET_NODE - the address is an explicitly addressed unicast (subnet/node) message, and the SendSnode union element is used to define the address.

NEURON_ID - the address is an explicitly addressed unicast (NEURON ID) message, and the SendNrnid union element is used to define the address.

BROADCAST - the address is an explicitly addressed broadcast (subnet-wide or domain-wide) message, and the SendBcast union element is used to define the address.

Note that for all the above explicit address formats, the address includes the fields retry, rpt_timer, and tx_timer, which must be set according to the requirements of the application. Zero is *not* a suitable default for these fields.

The following two address types are not defined as LONTALK address types, but are used internally within this host application example to address messages to the network interface itself (type LOCAL), and to use addresses defined in the address table of the network interface (type IMPLICIT).

LOCAL - the destination is the network interface. No additional addressing information is required; the message is sent to the network interface using the niNETMGMT command.

IMPLICIT - the destination address is determined implicitly. The SendImplicit union element is used to define the address, which requires specification of the message tag. This is a number between 0 and 14 which is an index into the address table of the network interface. The address table entry in the network interface on-chip EEPROM memory contains the actual destination address. The address table entries are loaded by a network management tool when the node is installed. Implicit addressing is used for bound network variables and message tags, and does not require definition of retry counts and timers since these are implicitly defined in the address table entry.

MsgData out_data - the actual data to be sent in the outgoing message. This data type is a union of three possibilities, defined in the file NI_MSG.H.

```
typedef union {
    UnprocessedNV unv;
    ProcessedNV   pnv;
    ExplicitMsg   exp;
} MsgData;
```

Since this host application is designed to work with network interfaces that are configured with ~~h~~ost selection enab~~led~~, the ProcessedNV union element cannot be used. The UnprocessedNV union element is used for outgoing network variable updates and polls, and the ExplicitMsg union element is used for outgoing explicit application and network management messages.

int out_length - the length of the outgoing message data in bytes (including the code byte for messages, and the selector bytes for network variables).

boolean priority - set to TRUE if the message should be delivered with priority service, FALSE otherwise.

boolean out_auth - set to TRUE if the message should be authenticated, FALSE otherwise. Only acknowledged and request messages may be authenticated.

Output Parameters

All output parameters are returned via pointers passed to the function. If any pointer is NULL, the corresponding parameter is simply not returned.

ComplType *completion - the completion code for the transaction, either MSG_SUCCEEDS or MSG_FAILS. An acknowledged or request/response transaction is successful if all the acknowledgments or responses are received. An implicitly addressed transaction will also succeed if it is sent to an unbound message tag or network variable. A network variable poll sent with network interface selection enabled is successful if at least one response contains network variable data. An unacknowledged or unacknowledged/repeated transaction is successful if all the packets were sent on to the network.

int *num_responses - the number of responses received (for request/response service only). If the message was sent with a unicast or a broadcast address and the transaction succeeded, this will be one. If the message was sent with a multicast address, this will be the number of actual responses received. The first response is returned in the call to ni_send_msg_wait(), and subsequent responses with a call to ni_get_next_response().

RespAddrDtl *in_addr - the address of the first response. See the declaration in NI_MSG.H for details of the fields in this structure. It can be used to determine which node sent the response.

MsgData *in_data - the data in the first response. If this is a response to an explicit message, the union element ExplicitMsg is used. If this is a response to a network variable poll, the union element UnprocessedNV is used.

int *in_length - the length of the incoming response data in bytes (including the code for messages, and the selector for network variables).

Error Codes

NI_OK - returned when the transaction completed normally

NI_DRIVER_ERROR - returned if the network driver reported an error when the driver was called

NI_TIMEOUT - returned if the transaction did not complete within 5 seconds (the constant niWAIT_TIME in NI_MSG.C)

NI_UPLINK_CMD - returned if the network interface sent a local uplink command (typically if it has been reset)

NI_INTERNAL_ERR - something unexpected happened

ni_get_next_response()

```
NI_Code ni_get_next_response(
    RespAddrDtl      * in_addr,
    MsgData          * in_data,
    int              * in_length );
```

The output parameters of this function are the same as the output parameters of ni_send_msg_wait(), which returns the first response for a request/response message. The ni_get_next_response() function is used to return subsequent responses, if any. Note that the heap used to store the subsequent responses is cleared when a new call is made to ni_send_msg_wait(). The error code NI_NO_RESPONSES is returned if the subsequent response heap is empty.

ni_receive_msg()

```
NI_Code ni_receive_msg(
    ServiceType      * service,
    RcvAddrDtl       * in_addr,
    MsgData          * in_data,
    int              * in_length,
    boolean          * in_auth );
```

This function checks to see if an incoming message has been received. If so, it returns NI_OK and the message is passed to the caller using the output parameters of the function call. If there is no message waiting, the function returns NI_TIMEOUT. If an unexpected event is received (for example, a response, a completion event, or a reset from the network interface), then the function returns NI_UPLINK_CMD.

All output parameters are returned via pointers passed to the function. If any pointer is NULL, the corresponding parameter is simply not returned. The output parameters of this function are:

ServiceType *service - the LONTALK service type for the incoming message. Network variable updates are received using ACKD, UNACKD_RPT, or UNACKD service. Network variable polls use REQUEST service. If the incoming message uses REQUEST service, the response should be returned with the function ni_send_response().

RcvAddrDtl *in_addr - the source address in the received message. See the declaration in NI_MSG.H for details of the fields in this structure. It can be used to determine which node sent the message and how it was addressed to the network interface.

MsgData *in_data - the data in the received message. If this is an explicit message, the union element ExplicitMsg is used. If this is a network variable update or poll, the union element UnprocessedNV is used. The most significant bit of the message code is one for a network variable message, and zero for an explicit message.

int *in_length - tthe length of the incoming message data in bytes (including the code byte for messages, and the selector bytes for network variables).

boolean *in_auth - TRUE if the incoming message is authentic, FALSE otherwise. A message is authentic when authenticated service was specified and the sender successfully replied to the challenge from the network interface.

ni_send_response()

```
NI_Code ni_send_response(
    MsgData         * out_data,
    int               out_length );
```

This function is used by the host application to send a response to the last incoming message that specified request service. It is used to respond to incoming network management requests, incoming network variable polls, and other incoming request messages. Responses to requests should always be returned, otherwise the requestor will receive a transaction failure, and the received transaction record in the responder will remain locked. Also, the host application should return responses promptly; if lengthy processing is required to prepare a response, the transaction timer of the sending node and the appropriate receive timer of the host application node should be increased to compensate.

The input parameters of this function are:

MsgData out_data - the actual data to be sent in the outgoing response. Since this host application is designed to work with network interfaces that are configured with host selection enabled, the ProcessedNV union element cannot be used. The UnprocessedNV union element is used for outgoing network variable poll responses, and the ExplicitMsg union element is used for outgoing application and network management responses.

int out_length - the length of the outgoing response data in bytes (including the code byte for explicit message responses, and the selector bytes for network variable poll responses).

ni_send_immediate()

```
NI_Code ni_send_immediate( NI_NoQueueCmd command );
```

This function is used to send an immediate command to the network interface. These commands are defined in Appendix D. Any downlink command other than niCOMM or niNETMGMT may be sent with this function. The host application uses this to inform the network interface that is has received a mode on-line or mode off-line network management message. The output parameter is a command code of type NI_NoQueueCmd (defined in NI_MSG.H).

handle_error()

```
boolean handle_error( NI_Code ni_error,
                      ComplType completion,
                      byte      response_code,
                      const char    * msg_name );
```

This function (in file NI_MSG.C) should be used to check the return status of any call to the ni_xxx() functions. It will print an appropriate message if any error has occurred, and return TRUE. If there was no error, it returns FALSE. The input parameters are:

NI_Code ni_error - an error code returned from a call to one of the ni_xxx() functions

ComplType completion - a completion code returned from a call to ni_send_msg_wait(). If the call being checked was not ni_send_msg_wait(), MSG_SUCCEEDS should be passed in for this parameter.

byte response_code - if the call being checked was a network management message sent with ni_send_msg_wait(), this should be the message code from the response. Otherwise NO_CHECK should be used.

char * msg_name - a text string printed as part of the error message.

Application Message Handler

The file APPLMSG.C contains all the functions that deal with incoming messages, including network management messages passed to the application, and incoming network variable updates and polls. The function process_msg() is called by the main dispatch loop whenever it detects that an incoming message has been received with ni_receive_msg(). It dispatches control to different functions depending on the message code of the incoming message.

```
boolean process_msg( ServiceType    service,
                     RcvAddrDtl   * address,
                     MsgData      * in_data,
                     int            in_length,
                     boolean        in_auth ) {

/* handle incoming messages addressed to this node */
/* return TRUE if prompt should be redisplayed */

    last_rcv_addr = * address;
            // save for debug purposes

    switch( in_data->exp.code ) {
            // dispatch on message code
    case NM_update_nv_cnfg:
        return handle_update_nv_cnfg( in_data, service );

    case NM_query_nv_cnfg:
        return handle_query_nv_cnfg( in_data, service );

    case NM_set_node_mode:
        return handle_set_mode( in_data );

    case NM_query_SNVT:
        return handle_query_SNVT( in_data, service );

    case NM_wink:
        if( report_flag )printf( "Received Wink msg\n" );
        printf( "\a" );      // Ding!
        return report_flag;

    case NM_NV_fetch:
        return handle_NV_fetch( in_data, service );

    default:    /* handle all other messages here */

        if( in_data->unv.must_be_one )
               // This is a network variable
             return handle_netvar_msg( in_data, service,
                   in_length, in_auth );

                  // This is an explicit msg
        else return handle_explicit_msg( in_data, service,
                   in_length, in_auth );
    }               // end switch
}
```

handle_update_nv_config()

This function is invoked when a network management tool wishes to bind a network variable. It extracts the network variable index from the incoming message, validates it, and then copies the network variable configuration table entry from the message to the appropriate element of the array `nv_config_table`.

handle_query_nv_config()

This function is invoked when a network management tool wishes to retrieve the binding information for a network variable. It extracts the network variable index from the incoming message, validates it, and then sends a response with a copy of the requested network variable configuration table entry from the array `nv_config_table`.

handle_set_mode()

This function is invoked when a network management tool sends a network management message to the node telling it to go on-line or off-line. It saves the requested mode in the variable `online_flag`, and sends a local message to the network interface to inform it of the change. If the host application is off-line, it will not send out network variable updates and poll message. If it receives a network variable poll, it will respond with no data. If it receives a network variable update, the value will be updated, even if the node is off-line. If the node is on-line, of course, it will behave normally.

handle_query_SNVT()

This function is invoked when a network management tool wishes to import the self-identification and self-documentation information-from the node. It extracts the offset and byte count from the request message, validates them, and sends a response containing the requested data from the `SNVT_info` data structure.

handle_NV_fetch()

This function is invoked when a network management tool wishes to retrieve the value of a network variable by index. It extracts the network variable index from the incoming message, validates it, and then sends a response with a copy of the requested network variable data from the array `nv_value_table`.

handle_netvar_msg()

This function is invoked when an application node updates or polls one of the network variables of this node. It searches through the network variable configuration table for an entry whose selector and direction match the selector and direction in the incoming message. If a match is found, the corresponding entry in the network variable value table is updated (for non-request messages), or returned in a poll response (for request messages). The example host application uses linear searching for simplicity since there are only eight network variables on the node. For large numbers of network variables, a more efficient search algorithm could be used.

handle_explicit_msg()

This function is invoked when the node receives an explicit message which is not one of the defined network management messages. It simply reports the arrival of the message. Note that service pin messages are received as ordinary unacknowledged explicit messages; they are not processed by this application.

Outgoing Network Variable Messages

The file `APPLMSG.C` also contains two functions that handle user commands to update and poll network variables.

NV_update()

This function displays a menu of the defined network variables with their current values, and asks the user to identify which one should be updated. It then reads the new value for the network variable from the keyboard, and updates the value in the array `nv_value_table`. If the specified network variable is an output variable, it then sends a network variable update message to the network interface. It uses the selector, service class, authentication, and priority attributes that are configured for this network variable in the network variable configuration table. It also uses an implicit address, with the message tag obtained from the addr_index field of the network variable configuration table entry. Finally, if this is a turnaround network variable bound to an input on the same node, it updates that input network variable.

NV_poll()

This function displays a menu of the defined input network variables with their current values, and asks the user to identify which one should be polled. It then sends a network variable poll request message to the network interface. It uses the selector, authentication, and priority attributes that are configured for this network variable in the network variable configuration table. It also uses an implicit address, with the message tag obtained from the addr_index field of the network variable configuration table entry. For each valid response that is received, it updates the value in the array nv_value_table. Finally, if this is a turnaround network variable bound to an output on the same node, it uses the value of that output variable to update the value in the array nv_value_table.

Running the Sample Host Application

The sample host application requires a PC to execute. It does not do any network management, and does not bind itself to other nodes. A network manager, such as NetMaker or the LONBUILDER Network Manager, may be used to install and bind the host application to other nodes. See the section *Host Application Requirements* in this chapter for more details.

To run the host application, type

```
HA [-V] [-DLONn]
```

at the DOS prompt. The optional argument -V turns on verbose mode. This mode may also be turned on with the V command to the application's command interpreter. The optional argument -DLON*n* specifies the DOS network driver name for the network interface to use for the application, where *n* is the device number. The default is LON1. The device number is specified in the CONFIG.SYS file when the network driver is loaded.

Once the host application is running, it will respond to network events as well as keyboard commands as described below. A NEURON C test program HA_TEST.NC is provided that may be installed on a NEURON CHIP-hosted node such as a LONBUILDER NEURON Emulator. This test program has network variables that may be bound to the host application node in order to test its functionality. This test program also has four input and four output network variables with the same names and types as the host application's network variables, and it may be bound to the host application node as shown below.

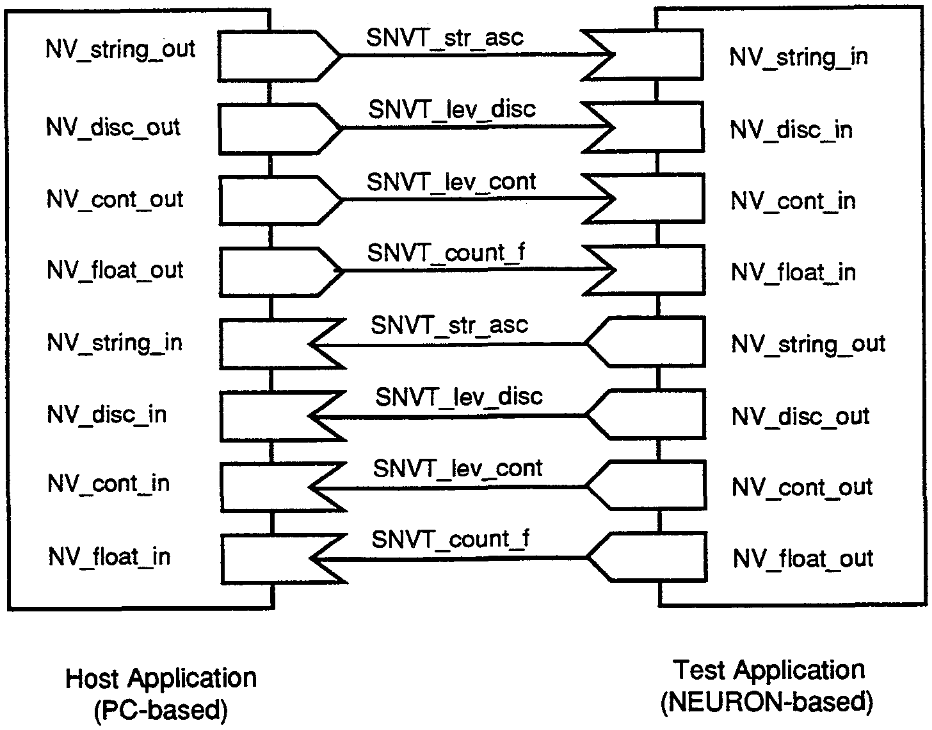


Figure A.2 Network Variables of the Host and Test Applications

LONBUILDER may be used to build this network as described below. The test application uses certain functions from the Extended Arithmetic Library, so this software should have previously been installed before proceeding. See the *Extended Arithmetic Support* engineering bulletin for installation instructions; the *Extended Arithmetic Support* engineering bulletin is included in *LONBUILDER Notes and News*. The Extended Arithmetic Library is available on the LONLINK bulletin board, and is also available on the LONLINK Sampler diskette. Also, the test application uses a LONBUILDER Multifunction I/O Kit (with Gizmo 2), so that this hardware should be installed in the emulator as described in the *LONBUILDER Startup and Hardware Guide*.

You should first create a channel definition in the LONBUILDER hardware database for the network interface, if one does not already exist. If you are using an SLTA as the network interface, the channel type is described on a colored sticker on the outside of the SLTA. Next, make sure that the LONBUILDER network manager, the emulator that you are using for the test application, and the network interface can all communicate with each other, both physically and logically. You may need to define and install LONBUILDER or LONWORKS routers to ensure that this is the case, or alternatively, change the transceivers on the network manager and emulator to the same type as the transceiver on the network interface for the host application. It is preferable for the LONBUILDER protocol analyzer to be able to communicate with the test network for debugging purposes.

Create an application node, target hardware object for the network interface. Its type should be "Custom Node." The hardware properties that you select should reflect the correct NEURON CHIP input clock rate, depending on the clock used in the network interface. The channel that you select should be the correct one also.

Now install the network interface — you will be asked to press the service pin on the network interface. Do not install comunication parameters.

Create an application image of origin "Interface File", with the name HA_V2.XIF. Create a node specification for the host application that specifies HA_V2 as the application image, and the network interface object as the target hardware.

Create a node specification for the test application, and specify the application image name as HA_TEST, and the target hardware as the emulator you are using. This will cause the NEURON C program HA_TEST.NC to be compiled to produce the application image for the emulator. The NEURON C include file DISPLAY.H is required; it contains the display driver for the Gizmo 2 used by the test application.

You can now build and load the emulator and the host application node using the LONBUILDER Project Manager. The host application must be running during the load operation, since it will be required to respond to certain network management messages. You can bind the network variables of the test application and the host application together as in the above diagram.

The test application uses the Gizmo 2 as the user interface device when connected to an emulator or other NEURON C-based node. When the node receives a network variable update from the network, it will display the value using the seven segment display. For the network variable of type SNVT_str_asc, it will display the first four characters if possible to do so using the seven segment display of the Gizmo 2. If there is no reasonable representation in seven segments, a blank will be displayed. For the network variable of type SNVT_count_f, it will display the number using two decimal places and a sign. For the network variable of type SNVT_lev_disc, it will display the value in percent using one decimal place. For the network variable of type SNVT_lev_disc, it will display one of the strings "OFF", "LO", "=Ed", "HI" or "On", corresponding to the ST_OFF, ST_LO, ST_MED, ST_HI and ST_ON values.

You can also use the Gizmo 2 to transmit network variables to the host application. Pressing the right (IO_3) button will cycle the display through the discrete level, continuous level, floating point, and the four first characters of the string output network variables. The quadrature input dial may be used to change the values transmitted to the network, and these will then be displayed by the host application when it receives the network variable updates.

The left (IO_7) button of the Gizmo 2 toggles the node between display of the input and output network variable of a given type. The red LED will be on if the input variable is being displayed, and off if the output variable is being displayed.

Appendix B

Creating an External Interface File

This appendix describes the procedure for modifying an external interface file (.XIF extension) to include network variables and message tags used by a host application.

How to Add Network Variables to the External Interface File

As described under *Binding to a Host Node* in Chapter 3, an external interface file (.XIF extension) can be used by a network management tool to determine the external interface of a node. The external interface file describes the network variables and message tags for a node. External interface files may be generated in several different ways.

The LONBUILDER software will create external interface files for nodes created with the NEURON C compiler. After the node image has been built, invoke the *Export* command in the *App Node / Node Specs* screen, and specify the *Interface File* option. See the *LONBUILDER User's Guide* Chapter 7 for more information. LONBUILDER 2.1 exports an interface file with versions 1 and 2 formats, and LONBUILDER 2.2 exports an interface file with version 3 format. This appendix describes version 2 and version 3 formats. A utility XIF3TO2 is included with LONBUILDER 2.2 to convert version 3 format interface files to version 2.

A variant of this technique may be used to create an external interface file for a host application node. Create a NEURON C program with the appropriate network variable and message tag declarations. There need be no actual code in this program. Compile and build this application using the appropriate hardware properties and channel definitions for the custom node. Then use LONBUILDER to export the external interface file for this node. Afterwards, use a text editor such as the LONBUILDER editor to modify the values in the sixth line appropriately. This technique will work for nodes with 62 network variables or less. Nodes with more than this will require further modifications to the external interface file as described below.

LONBUILDER may also be used to create an external interface file from an existing custom node by querying it over the network. Invoke the *Query* command in the *App Node / Node Specs* screen. If the node was built with network variable names in its SNVT information, those names will automatically be uploaded, otherwise default names will be used for the network variables. These names may be edited if desired. See the *LONBUILDER User's Guide* Chapter 6 for more information.

An external interface file without network variables or message tag definitions is included with the SLTA, and can be generated for network interfaces based on the MIP as described in the *Microprocessor Interface Program (MIP) User's Guide.* This appendix describes how to modify this external interface file to add network variables and message tag definitions to a basic header.

External interface files are used by several LONWORKS tools. All current releases of these tools can read version 2 interface files.

LONBUILDER can read an external interface file for an existing custom node so that its network variables and message tags may be bound. In the *App Node / App Images* screen, create an application image whose origin is *Interface File*, and whose name is the base name of the external interface file which should be in the working directory. Then in the *App Node / Node Specs* screen, create a node specification with that *App Image Name*. Afer that node is built, the network variables and message tags imported from the external interface file will be available to the binder. See the *LONBUILDER User's Guide* Chapter 7 for more information.

External interface files may be read by the function `ldb_import_xif()` in the LONMANAGER API in order to create program records. These program records may then be used to define the network variables of node records in the database. See the *LONMANAGER Reference Guide for Windows* Chapter 2, or the *LONMANAGER API Programmer's Guide for DOS* Appendix C for details.

External interface files may also be read by the LONMANAGER NetProfiler as part of the process of creating an *Application Type*. These *Application Types* form part of the parts catalog that the LONMANAGER NetMaker uses for network installation. See the *LONMANAGER NetProfiler User's Guide* Chapter 5 for details.

External Interface File Format

The external interface file is a text file that describes critical logical (e.g., network variables) and physical (e.g., transceiver) interface characteristics of a node.

Before any modifications, the external interface file for a network interface will look very similar to the following:

```
1: File:  PMIP_78K.XIF generated by APC Revision 1.73, XIF Version 2
2: Copyright (c) 1990, 1992 by Echelon Corporation
3: All Rights Reserved.  Run on Wed Apr 15 10:18:08 1992
4:
5: 4D:49:43:52:4F:5F:50:49
6: 2 15 1 0 0 3 3 3 4 4 4 11 11 11 11 1 0 8
7: 0 4 4 5 3 13 18 1402 0 15 5 3 106
8: *
```

The values in this file header depend upon transceiver type, buffer configuration, and other configuration characteristics of the application. Adding network variables or message tags to an external interface file requires only limited changes to the original contents of the external interface file header. Network variables and message tags are added to an external interface file by adding additional lines of information to the original file. Most of the entries in the header of the file *must not be changed*, except as noted below, otherwise many of the LONWORKS tools will fail to import the file. The other entries are documented here for reference only. Note that blank lines *are* significant. The fourth line must be blank, and there must be no blank lines at the end of the file.

First line: The last character is the version number for this external interface file format. The directions given here apply only to version 2 and version 3 external interface file formats.

Fifth line: The program ID as a sequence of eight hexadecimal values separated by colons.

Sixth line: This consists of eighteen decimal values separated by single spaces, as follows:

1 Number of domains (1 or 2).
2 Number of address table entries (0 to 15).
3 If this node handles incoming explicit messages. (1 for a host application)
4 Number of network variables (0 to 4096). *This value should be modified appropriately.*
5 Number of explicit message tags (0 to 15). *This value should be modified appropriately.*
Note - the encodings of the following ten values are described in the *NEURON CHIP Data Book*, section A.1.
6 Encoded number of network input buffers.
7 Encoded number of network output buffers.
8 Encoded number of priority network output buffers.
9 Encoded number of priority application output buffers.
10 Encoded number of application output buffers.
11 Encoded number of application input buffers.
12 Encoded size of a network input buffer.
13 Encoded size of a network output buffer.
14 Encoded size of an application output buffer.
15 Encoded size of an application input buffer.
16 If this is a host application (1 for a host application). *This value should be modified to 1.*
17 Number of network variables for a host application using network interface selection (0 to 62), 0 otherwise. *This value should be modified if you are using network interface selection.*
18 Number of receive transaction buffers.

Seventh line: This consists of eleven (version 3) or thirteen (version 2) decimal values separated by single spaces, as follows:

1 NEURON CHIP type. 0 for a 3150 and 8 for a 3120)
2 Encoded input clock rate. (1 to 5, corresponding to 625kHz to 10MHz).
3 Encoded comm. port bit rate. (0 to 8, corresponding to 1.25 Mbps to 4.9 kb/s) (version 2 only)
4 Medium type. (version 2 only)
5 Firmware version number.
6 Size of a receive transaction buffer. (13)
7 Size of a transmit transaction buffer. (18)

8 Number of bytes of available on-chip RAM.
9 Number of bytes of available off-chip RAM.
10 Size of a domain table entry. (15)
11 Size of an address table entry. (5)
12 Size of a network variable configuration table entry. (3)
13 Size of network image.

In a version 3 external interface file, the next three lines describe the transceiver properties. In a version 2 external interface file, the next line (only one line) describes the transceiver properties.

Next line: This line consists of a double quote character followed by the node self-identification string, or an asterisk if there is no self ID string.

The next line must be blank.

Network Variables and Message Tags

Each network variable and message tag is represented in an external interface file with an entry that begins with VAR or TAG respectively. Each entry consists of several lines of information; each line consists of one or more data fields. A single space separates the data fields on a line; each line is terminated with a newline. Listed below are several example network variable and message tag declarations and the corresponding external interface file entries. See the *NEURON C Programmer's Guide* for a description of network variable and message tag declarations. Following these examples is a detailed discussion of the external interface file entries for network variables and message tags.

```
network output polled long
      bind_info( offline ackd( nonconfig )
      authenticated( nonconfig )
      priority( nonconfig )
      rate_est( 123 ) max_rate_est( 234 ) ) outvar;

VAR outvar 0 69 76 0
1 1 63 1 0 0 1 0 1 0 1 0 0
*
0 *   1
2 0 0 1 0


network input sync config int invar;

VAR invar 1 0 0 0
0 1 63 0 0 1 0 1 0 1 0 1 1
*
0 *   1
1 0 0 1 0
```

```
typedef struct { int          x;
                 long         y;
                 int          array[5];
                 unsigned     z : 3;
                 unsigned     zz : 5;
                 union {
                   int a; int b;} u;
                } group;
```

```
 network input group ingroup;
```

```
VAR ingroup 2 0 0 0
0 1 63 0 0 1 0 1 0 1 0 0 0
*
0 *   6
1 0 0 1 0
2 0 0 1 0
1 0 0 1 5
3 0 3 0 0
3 3 5 0 0
4 0 1 0 0
```

```
msg_tag bind_info(rate_est(123) max_rate_est(234)) user_tag;
```

```
TAG user_tag 0 69 76 0
0 1 63 1 0 1 0 1 0 1 0 0 0
```

Below is a detailed explanation of the external interface file entries for network variables and message tags. For each line, the data field explanations are listed in the order they appear on the line from left to right.

First Line - Type, Name, and General Information.

Field	Values and Meaning	Comments
Entry Type	"VAR" - network variable "TAG" - message tag	
Name	Message tag or network variable name; character string of up to 16 characters.	
Index	0 to 4095 - network variable 0 to 14 - message tag	Unique number for each type assigned sequentially starting at 0. For arrays, this is the index of the first element

continued...

Rate Estimate	0 - Not applicable; otherwise encoded value representing the average network traffic generated by this network variable or message tag.	See the *NEURON C Programmer's Guide* for an explanation of the encoding scheme
Maximum Rate Estimate	Same as Rate Estimate above.	
Array Size	0 - not an array 1 to 4095 - number of array elements	

Note: Each element of an array is assigned a unique index number. The index number assigned to an entry following that for an array, is assigned an index number equal to the index number of the array plus the number of elements in the array.

Second Line - Connection Information. Where appropriate, the same terminology is used in these descriptions as is used in the NEURON C `bind_info` declaration. For message tags, only the first three fields are significant, but the other fields must be present; their values are ignored.

Field	**Values and Meaning**	**Comments**
Offline	0 - update when online or offline 1 - update only when offline	
Bindable	0 - cannot be bound 1 - can be bound	For binding host nodes, this should always be 1.
Unused	63	
Direction	0 - input 1 - output	
Service Type	0 - ackd 1 - unackd_rpt 2 - unackd	
Service Type - Configurable	0 - nonconfig (no) 1 - config (yes)	Specify config (1) if the service type may be changed by a network management message
Authenticated	0 - nonauthenticated NV 1 - authenticatedNV	
Authenticated - Configurable	0 - nonconfig (no) 1 - config (yes)	Specify config (1) if the authentication field may be changed by a network management message
Priority	0 - nonpriority 1 - priority	
Priority - Configurable	0 - nonpriority (no) 1 - config (yes)	Specify config (1) if the priority may be changed by a network management message

continued...

Polled	For output NVs: 0 - asynchronous updates 1 - update only when polled For input NVs: 0 - not polled by this node 1 - polled by this node	
Synchronized	0 - regular updates 1 - send all values, preserve order	
Config	0 - can be changed by this node 1 - cannot be changed by this node	

Third Line - Network Variable Self-Documenting text. This and all following lines are only included for network variables. If self-documenting text is not supplied, this line consists only of a single asterisk. If supplied, one or more lines of text appear here; each line begins with a double-quote character and ends with a newline. When the lines are concatenated together without the double-quote or newline characters, this forms the self-documentation text. Each line may be up to 60 characters long not including the double-quote or newline.

Fourth Line - Network Variable Type Information. This line follows the last line of the network variable self-documenting text.

Field	Values and Meaning	Comments
SNVT Type	0 - not a standard type 1 to 255 - standard type number	See the SNVT list for the type numbers
Unused	*	
Number of Elements	1 - not a struct or union 1 to 256 - number of elements in a struct	

Fifth and Subsequent Line(s) - Element Description(s). For each element, the characteristics of the data are defined. There is one line for each element; the number of elements is specified in the *Number of Elements* field in the preceding line.

Field	Values and Meaning	Comments
Type	0 - char 1 - int (NEURON C definition) 2 - long (NEURON C definition) 3 - bitfield 4 - union 5 - typeless	If Type is typeless, none of the remaining fields are applicable.
Bitfield Offset	0 if not applicable 0 to 7 otherwise	Applies only when Type is bitfield.
Size	0 - not applicable 1 to 7 - number of bits (bitfield only) 1 to 31 - number of bytes (union only)	Applies only when Type is bitfield or union.
Sign or Signed	0 - not applicable or unsigned 1 - signed	
Array Bound	0 - not an array or not applicable 1 to 31 - number of array elements	

Adding Network Variables and Message Tags to the Network Interface External Interface File

As mentioned above, the external interface file is a text file. The external interface file can be modified using any text editor, including the LONBUILDER editor. Follow the steps listed below to add network variables and message tags to the network interface external interface file.

1 Start with the original network interface external interface file. That file should be similar to that listed below; the file name will be different and some of the numeric values will be different depending upon transceiver type, buffer configuration, and other configuration characteristics. Also, the line number and colon are not part of the external interface file.

```
1: File:  PMIP_78K.XIF generated by APC Revision 1.73, XIF Version 2
2: Copyright (c) 1990, 1992 by Echelon Corporation
3: All Rights Reserved.  Run on Wed Apr 15 10:18:08 1992
4:
5: 4D:49:43:52:4F:5F:50:49
6: 2 15 1 0 0 3 3 3 4 4 4 11 11 11 11 1 0 8
7: 0 4 4 5 3 13 18 1402 0 15 5 3 106
8: *
```

2 Three values in the original file need to be changed. All of these are on the sixth line. The following changes should be made:

a The fourth (4th) number is the number of network variables; this is a value from 0 to 4096. Change this to the number of network variable entries that are to be added.

b The fifth (5th) number is the number of message tags; this is a value from 0 to 15. Change this to the number of message tag entries that are to be added.

c The sixteenth (16th) number should be changed from '1' to '0'. This indicates that this external interface file is for a host application program.

These changes are shown below with a double underscore.

```
File:  PMIP_78K.XIF generated by APC Revision 1.73, XIF Version 2
Copyright (c) 1990, 1992 by Echelon Corporation
All Rights Reserved.  Run on Wed Apr 15 10:18:08 1992

4D:49:43:52:4F:5F:50:49
2 15 1 3 1 3 3 3 4 4 4 11 11 11 11 0  0 8
       = =                         =
0 4 4 5 3 13 18 1402 0 15 5 3 106
*
```

3 Place the cursor at the end of the last line, that is immediately after the * on the last line. Enter a newline, then enter a second newline to create a blank line before the first network variable or message tag entry.

4 Enter the first network variable or message tag entry at the current position in the file. This is illustrated below.

```
File:  PMIP_78K.XIF generated by APC Revision 1.73, XIF Version 2
Copyright (c) 1990, 1992 by Echelon Corporation
All Rights Reserved.  Run on Wed Apr 15 10:18:08 1992

4D:49:43:52:4F:5F:50:49
2 15 1 0 0 3 3 3 4 4 4 11 11 11 11 0 0 8
0 4 4 5 3 13 18 1402 0 15 5 3 106
*

VAR outvar 0 69 76 0
1 1 63 1 0 0 1 0 1 0 1 0 0
*
0 *   1
2 0 0 1 0
```

5 Enter the next network variable or message tag entry immediately after the first entry, i.e., there is no blank line separating the entries. A second entry is shown below.

```
File:  PMIP_78K.XIF generated by APC Revision 1.73, XIF Version 2
Copyright (c) 1990, 1992 by Echelon Corporation
All Rights Reserved.  Run on Wed Apr 15 10:18:08 1992
```

```
4D:49:43:52:4F:5F:50:49
2 15 1 0 0 3 3 3 4 4 4 11 11 11 11 0 0 8
0 4 4 5 3 13 18 1402 0 15 5 3 106
*

VAR outvar 0 69 76 0
1 1 63 1 0 0 1 0 1 0 1 0 0
*
0 *   1
2 0 0 1 0
VAR invar 1 0 0 0
0 1 63 0 0 1 0 1 0 1 0 1 1
*
0 *   1
1 0 0 1 0
```

6. Continue adding entries in the same way until all of the required network variables and message tags have been included.

There is an alternative to the above procedure which may be more appropriate in some circumstances. The alternative is defined below:

1. Create a NEURON C source program containing the NEURON C network variable and message tag declarations corresponding to the entries needed in the network interface external interface file. This file need not have any executable statements in it.
2. Create and build an application node with this NEURON C program.
3. Export the external interface file for this application node.
4. Perform steps 1 through 3 in the first procedure.
5. Use a text editor to *cut and paste* the network variable and message tag entries from the external interface file exported in step 3 of this procedure to the network interface external interface file.

While creating the NEURON C source program and building the application requires more work than simply editing the external interface file, this approach is more reliable because the NEURON C compiler generates the external interface file entries. This is the recommended approach if more than a very small number of simple network variable or message tag entries are required.

NOTE: When using anything other than Standard Network Variable Types (SNVTs), the size of a NEURON C-type (e.g., int) may not be the same as that for the host application. Special conversion processing may be required in the host for connections between a host application and a NEURON CHIP-hosted application, or the typeless data type may need to be used for connections between host applications.

Appendix C

Network Interface Messages

This appendix defines the message structures exchanged by a host application and the network driver. These messages will consist of network variable updates, explicit messages, or local commands to the network interface. Explicit messages may be application, network management, or network diagnostic messages as defined in Appendix B of the *NEURON CHIP Data Book*.

The message structures are defined using ANSI C structure definitions, and also using bit-field diagrams. The ANSI C structure definitions are contained in two ANSI C header files that are included in the host application example directory. These files are `NI_MSG.H` and `NI_MGMT.H`. The `NI_MSG.H` header file defines the network interface message structures and also defines other structures and variables used by the `NI_MSG.C` functions listed in Appendix A. The `NI_MGMT.H` header file defines the subset of network management messages used by the example host application in Appendix A.

The bit-field diagrams can be used by host application developer's using a language other than ANSI C. The bit-field diagrams also illustrate the difference between the application-layer header and the link-layer headers.

NI_MSG.H

```
/*
 *          NI_MSG.H
 *
 * Message definitions for the LONWORKS network driver protocol.
 */

/*
 ****************************************************************************
 * Application buffer structures for sending and receiving messages to and
 * from a network interface.  The 'ExpAppBuffer' and 'ImpAppBuffer'
 * structures define the application buffer structures with and without
 * explicit addressing.  These structures have up to four parts:
 *
 *   Network Interface Command (NI_Hdr)                       (2 bytes)
 *   Message Header (MsgHdr)                                  (3 bytes)
 *   Network Address (ExplicitAddr)                           (11 bytes)
 *   Data (MsgData)                                           (varies)
 *
 * Network Interface Command (NI_Hdr):
 *
 *   The network interface command is always present.  It contains the
 *   network interface command and queue specifier.  This is the only
 *   field required for local network interface commands such as niRESET.
 *
 * Message Header (MsgHdr: union of NetVarHdr and ExpMsgHdr):
 *
 *   This field is present if the buffer is a data transfer or a completion
 *   event.  The message header describes the type of LONTALK message
 *   contained in the data field.
 *
 *   NetVarHdr is used if the message is a network variable message and
 *   network interface selection is enabled.
 *
 *   ExpMsgHdr is used if the message is an explicit message, or a network
 *   variable message and host selection is enabled (this is the default
 *   for the SLTA).
 *
 * Network Address (ExplicitAddr:  SendAddrDtl, RcvAddrDtl, or RespAddrDtl)
 *
 *   This field is present if the message is a data transfer or completion
 *   event, and explicit addressing is enabled.  The network address
 *   specifies the destination address for downlink application buffers,
 *   or the source address for uplink application buffers.  Explicit
 *   addressing is the default for the SLTA.
 *
 *   SendAddrDtl is used for outgoing messages or NV updates.
 *
 *   RcvAddrDtl is used  for incoming messages or unsolicited NV updates.
 *
 *   RespAddrDtl is used for incoming responses or NV updates solicited
 *   by a poll.
 *
 * Data (MsgData: union of UnprocessedNV, ProcessedNV, and ExplicitMsg)
 *
 *   This field is present if the message is a data transfer or completion
 *   event.
 *
 *   If the message is a completion event, then the first two bytes of the
 *   data are included.  This provides the NV index, the NV selector, or the
 *   message code as appropriate.
 *
 *   UnprocessedNV is used if the message is a network variable update, and
 *   host selection is enabled. It consists of a two-byte header followed by
 *   the NV data.
 *
 *   ProcessedNV is used if the message is a network variable update, and
 *   network interface selection is enabled. It consists of a two-byte header
 *   followed by the NV data.
 *
 *   ExplicitMsg is used if the message is an explicit message.  It consists
 *   of a one-byte code field followed by the message data.
 *
 * Note - the fields defined here are for a little-endian (Intel-style)
 * host processor, such as the 80x86 processors used in PC compatibles.
 * Bit fields are allocated right-to-left within a byte.
 * For a big-endian (Motorola-style) host, bit fields are typically
 * allocated left-to-right.  For this type of processor, reverse
 * the bit fields within each byte.  Compare the NEURON C include files
 * ADDRDEFS.H and MSG_ADDR.H, which are defined for the big-endian NEURON
 * CHIP.
 ****************************************************************************
 */

/* Change the following declarations to port to a non-80x86  */

typedef unsigned char byte;     /* 8 bits */
typedef unsigned bits;          /* bit fields */
typedef unsigned int word;      /* 16 bits */

typedef enum { FALSE = 0, TRUE } boolean;

/*
 ****************************************************************************
 * Network Interface Command data structure.  This is the application-layer
 * header used for all messages to and from a LONWORKS network interface.
 ****************************************************************************
 */

/* Literals for the 'cmd.q.queue' nibble of NI_Hdr. */

typedef enum {
    niTQ          =  2,           /* Transaction queue                     */
    niTQ_P        =  3,           /* Priority transaction queue            */
    niNTQ         =  4,           /* Non-transaction queue                 */
    niNTQ_P       =  5,           /* Priority non-transaction queue        */
    niRESPONSE    =  6,           /* Response msg & completion event queue*/
    niINCOMING    =  8            /* Received message queue                */
} NI_Queue;
```

```
/* Literals for the 'cmd.q.q_cmd' nibble of NI_Hdr. */

typedef enum {
    niCOMM           = 1,           /* Data transfer to/from network        */
    niNETMGMT        = 2            /* Local network management/diagnostics */
} NI_QueueCmd;

/* Literals for the 'cmd.noq' byte of NI_Hdr. */

typedef enum {
    niNULL           = 0x00,
    niTIMEOUT        = 0x30,        /* Not used                             */
    niCRC            = 0x40,        /* Not used                             */
    niRESET          = 0x50,
    niFLUSH_COMPLETE = 0x60,        /* Uplink                               */
    niFLUSH_CANCEL   = 0x60,        /* Downlink                             */
    niONLINE         = 0x70,
    niOFFLINE        = 0x80,
    niFLUSH          = 0x90,
    niFLUSH_IGN      = 0xA0,
    niSLEEP          = 0xB0,        /* SLTA only                            */
    niACK            = 0xC0,
    niNACK           = 0xC1,        /* SLTA only                            */
    niSSTATUS        = 0xE0,        /* SLTA only                            */
    niPUPXOFF        = 0xE1,
    niPUPXON         = 0xE2,
    niPTRHROTL       = 0xE4,        /* Not used                             */
    niDRV_CMD        = 0xF0,        /* Not used                             */
} NI_NoQueueCmd;

/*
 * Header for network interface messages.  The header is a union of
 * two command formats: the 'q' format is used for the niCOMM and
 * niNETMGMT commands that require a queue specification; the 'noq'
 * format is used for all other network interface commands.
 * Both formats have a length specification where:
 *
 *      length = header (3) + address field (11 if present) + data field
 *
 * WARNING:  The fields shown in this structure do NOT reflect the actual
 * structure required by the network interface.  Depending on the network
 * interface, the network driver may change the order of the data and add
 * additional fields to change the application-layer header to a link-layer
 * header.  See the description of the link-layer header in Chapter 2 of the
 * Host Application Programmer's Guide.
 */

typedef union {
    struct {
        bits queue  :4;         /* Network interface message queue      */
                                /* Use value of type 'NI_Queue'         */
        bits q_cmd  :4;         /* Network interface command with queue */
                                /* Use value of type 'NI_QueueCmd'      */
        bits length :8;         /* Length of the buffer to follow       */
    } q;                        /* Queue option                         */
    struct {
        byte     cmd;           /* Network interface command w/o queue  */
                                /* Use value of type 'NI_NoQueueCmd'    */
        byte     length;        /* Length of the buffer to follow       */
    } noq;                      /* No queue option                      */
} NI_Hdr;

/*
 ***************************************************************************
 * Message Header structure for sending and receiving explicit
 * messages and network variables which are not processed by the
 * network interface (host selection enabled).
 ***************************************************************************
 */

/* Literals for 'st' fields of ExpMsgHdr and NetVarHdr. */

typedef enum {
   ACKD             = 0,
   UNACKD_RPT       = 1,
   UNACKD           = 2,
   REQUEST          = 3
} ServiceType;

/* Literals for 'cmpl_code' fields of ExpMsgHdr and NetVarHdr. */

typedef enum {
    MSG_NOT_COMPL  = 0,         /* Not a completion event               */
    MSG_SUCCEEDS   = 1,         /* Successful completion event          */
    MSG_FAILS      = 2          /* Failed completion event              */
} ComplType;

/* Explicit message and Unprocessed NV Application Buffer. */

typedef struct {

    bits   tag       :4;       /* Message tag for implicit addressing  */
                               /* Magic cookie for explicit addressing */
    bits   auth      :1;       /* 1 => Authenticated                   */
    bits   st        :2;       /* Service Type - see 'ServiceType'     */
    bits   msg_type  :1;       /* 0 => explicit message                */
                               /*      or unprocessed NV               */
/*---------------------------------------------------------------------*/
    bits   response  :1;       /* 1 => Response, 0 => Other            */
    bits   pool      :1;       /* 0 => Outgoing                        */
    bits   alt_path  :1;       /* 1 => Use path specified in 'path'    */
                               /* 0 => Use default path                */
    bits   addr_mode :1;       /* 1 => Explicit addressing,            */
                               /* 0 => Implicit                        */
                               /* Outgoing buffers only                */
    bits   cmpl_code :2;       /* Completion Code - see 'ComplType'    */
    bits   path      :1;       /* 1 => Use alternate path,             */
                               /* 0 => Use primary path                */
                               /*      (if 'alt_path' is set)          */
    bits   priority  :1;       /* 1 => Priority message                */
/*---------------------------------------------------------------------*/
    byte   length;             /* Length of msg or NV to follow        */
```

```
                                    /* not including any explicit address    */
                                    /* field, includes code byte or          */
                                    /* selector bytes                        */
} ExpMsgHdr;

/*
 **********************************************************************
 * Message Header structure for sending and receiving network variables
 * that are processed by the network interface (network interface
 * selection enabled).
 **********************************************************************
 */

typedef struct {
    bits   tag        :4;       /* Magic cookie for correlating          */
                                /* responses and completion events       */
    bits   rsvd0      :2;
    bits   poll       :1;       /* 1 => Poll, 0 => Other                 */
    bits   msg_type   :1;       /* 1 => Processed network variable       */
/*-----------------------------------------------------------------------*/
    bits   response   :1;       /* 1 => Poll response, 0 => Other        */
    bits   pool       :1;       /* 0 => Outgoing                         */
    bits   trnarnd    :1;       /* 1 => Turnaround Poll, 0 => Other      */
    bits   addr_mode  :1;       /* 1 => Explicit addressing,             */
                                /* 0 => Implicit addressing              */
    bits   cmpl_code  :2;       /* Completion Code - see above           */
    bits   path       :1;       /* 1 => Used alternate path              */
                                /* 0 => Used primary path                */
                                /*      (incoming only)                  */
    bits   priority   :1;       /* 1 => Priority msg (incoming only)     */
/*-----------------------------------------------------------------------*/
    byte   length;              /* Length of network variable to follow */
                                /* not including any explicit address    */
                                /* not including index and rsvd0 byte    */
} NetVarHdr;

/* Union of all message headers. */

typedef union {
    ExpMsgHdr  exp;
    NetVarHdr  pnv;
} MsgHdr;

/*
 **********************************************************************
 * Network Address structures for sending messages with explicit addressing
 * enabled.
 **********************************************************************
 */

/* Literals for 'type' field of destination addresses for outgoing messages. */

typedef enum {
    UNASSIGNED      = 0,
    SUBNET_NODE     = 1,
    NEURON_ID       = 2,
    BROADCAST      = 3,
    IMPLICIT       = 126,    /* not a real destination type */
    LOCAL          = 127,    /* not a real destination type */
} AddrType;

/* Group address structure.  Use for multicast destination addresses. */

typedef struct {
    bits   size      :7;         /* Group size (0 => huge group)         */
    bits   type      :1;         /* 1 => Group                           */

    bits   member    :6;         /* Member ID (0 => huge group)          */
    bits   rsvd0     :1;
    bits   domain    :1;         /* Domain index                         */

    bits   retry     :4;         /* Retry count                          */
    bits   rpt_timer :4;         /* Retry repeat timer                   */

    bits   tx_timer  :4;         /* Transmit timer index                 */
    bits   rsvd1     :4;

    byte       group;            /* Group ID                             */
} SendGroup;

/* Subnet/node ID address.  Use for a unicast destination address. */

typedef struct {
    byte   type;                 /* SUBNET_NODE                          */

    bits   node      :7;         /* Node number                          */
    bits   domain    :1;         /* Domain index                         */

    bits   retry     :4;         /* Retry count                          */
    bits   rpt_timer :4;         /* Retry repeat timer                   */

    bits   tx_timer  :4;         /* Transmit timer index                 */
    bits   rsvd1     :4;

    bits   subnet    :8;         /* Subnet ID                            */
} SendSnode;

/* 48-bit NEURON ID destination address. */

#define NEURON_ID_LEN 6

typedef struct {
    byte   type;                 /* NEURON_ID                            */

    bits   rsvd0      :7;
    bits   domain     :1;        /* Domain index                         */

    bits   retry      :4;        /* Retry count                          */
    bits   rpt_timer  :4;        /* Retry repeat timer                   */

    bits   tx_timer   :4;        /* Transmit timer index                 */
    bits   rsvd2      :4;
    bits   subnet     :8;       /* Subnet ID, 0 => pass all routers      */
    byte   nid[ NEURON_ID_LEN ];  /* NEURON ID                           */
} SendNrnid;

/* Broadcast destination address. */

typedef struct {
    byte   type;                /* BROADCAST                             */

    bits   backlog     :6;      /* Backlog                               */
    bits   rsvd0       :1;
    bits   domain      :1;      /* Domain index                          */

    bits   retry       :4;      /* Retry count                           */
    bits   rpt_timer   :4;      /* Retry repeat timer                    */

    bits   tx_timer    :4;      /* Transmit timer index                  */
    bits   rsvd2       :4;

    bits   subnet      :8;      /* Subnet ID, 0 => domain-wide           */
} SendBcast;

/* Address format to clear an address table entry.          */
/* Sets the first 2 bytes of the address table entry to 0. */

typedef struct {
    byte   type;                /* UNASSIGNED or LOCAL          */
} SendUnassigned;

typedef struct {
   byte    type;                /* IMPLICIT */
   byte    msg_tag;             /* address table entry number */
} SendImplicit;

/* Union of all destination addresses. */

typedef union {
    SendGroup      gp;
    SendSnode      sn;
    SendBcast      bc;
    SendNrnid      id;
    SendUnassigned ua;
    SendImplicit   im;
} SendAddrDtl;

/*
 *****************************************************************************
 * Network Address structures for receiving messages with explicit
 * addressing enabled.
 *****************************************************************************
 */

/* Received subnet/node ID destination address.  Used for unicast messages. */

typedef struct {
    bits       subnet :8;
    bits       node   :7;
    bits              :1;
} RcvSnode;

/* Received 48-bit NEURON ID destination address. */

typedef struct {
    byte   subnet;
    byte   nid[ NEURON_ID_LEN ];
} RcvNrnid;

/* Union of all received destination addresses. */

typedef union {
    byte       gp;                  /* Group ID for multicast destination   */
    RcvSnode   sn;                  /* Subnet/node ID for unicast           */
    RcvNrnid   id;                  /* 48-bit NEURON ID destination address */
    byte       subnet;              /* Subnet ID for broadcast destination  */
                                    /* 0 => domain-wide */
} RcvDestAddr;

/* Source address of received message.  Identifies */
/* network address of node sending the message.    */

typedef struct {
    bits   subnet  :8;
    bits   node    :7;
    bits           :1;
} RcvSrcAddr;

/* Literals for the 'format' field of RcvAddrDtl. */

typedef enum {
    ADDR_RCV_BCAST  = 0,
    ADDR_RCV_GROUP  = 1,
    ADDR_RCV_SNODE  = 2,
    ADDR_RCV_NRNID  = 3
} RcvDstAddrFormat;

/* Address field of incoming message. */

typedef struct {
#ifdef _MSC_VER
    byte    kludge;    /* Microsoft C does not allow odd-length bitfields */
#else
    bits    format      :6;     /* Destination address type             */
                                /* See 'RcvDstAddrFormat'               */
    bits    flex_domain :1;     /* 1 => broadcast to unconfigured node  */
    bits    domain      :1;     /* Domain table index                   */
#endif
    RcvSrcAddr  source;             /* Source address of incoming message   */
    RcvDestAddr dest;               /* Destination address of incoming msg  */
} RcvAddrDtl;

/*
```

```
 ****************************************************************************
 * Network Address structures for receiving responses with explicit
 * addressing enabled.
 ****************************************************************************
 */

/* Source address of response message. */

typedef struct {
    bits   subnet   :8;
    bits   node     :7;
    bits   is_snode :1;        /* 0 => Group response,                  */
                               /* 1 => snode response                   */
} RespSrcAddr;

/* Destination of response to unicast request. */

typedef struct {
    bits   subnet   :8;
    bits   node     :7;
    bits            :1;
} RespSnode;

/* Destination of response to multicast request. */

typedef struct {
    bits   subnet   :8;
    bits   node     :7;
    bits            :1;
    bits   group    :8;
    bits   member   :6;
    bits            :2;
} RespGroup;

/* Union of all response destination addresses. */

typedef union {
    RespSnode  sn;
    RespGroup  gp;
} RespDestAddr;

/* Address field of incoming response. */

typedef struct {
#ifdef _MSC_VER
    byte     kludge;     /* Microsoft C does not allow odd-length bitfields */
#else
    bits                 :6;
    bits     flex_domain :1;    /* 1=> Broadcast to unconfigured node    */
    bits     domain      :1;    /* Domain table index                    */
#endif
    RespSrcAddr  source;            /* Source address of incoming response  */
    RespDestAddr dest;              /* Destination address of incoming resp */
} RespAddrDtl;

/* Explicit address field if explicit addressing is enabled. */
```

```
typedef union {
    RcvAddrDtl  rcv;
    SendAddrDtl snd;
    RespAddrDtl rsp;
} ExplicitAddr;

/*
 *****************************************************************************
 * Data field structures for explicit messages and network variables.
 *****************************************************************************
 */

/*
 * MAX_NETMSG_DATA specifies the maximum size of the data portion of an
 * application buffer.  MAX_NETVAR_DATA specifies the maximum size of the
 * data portion of a network variable update.  The values specified here
 * are the absolute maximums,based on the LONTALK protocol. Actual limits
 * are based on the buffer sizes defined on the attached NEURON CHIP.
 */

#define MAX_NETMSG_DATA 228
#define MAX_NETVAR_DATA 31

/* Data field for network variables (host selection enabled). */

typedef struct {
    bits   NV_selector_hi :6;
    bits   direction      :1;      /* 1 => output NV, 0 => input NV      */
    bits   must_be_one    :1;      /* Must be set to 1 for NV            */
    bits   NV_selector_lo :8;
    byte       data[ MAX_NETVAR_DATA ]; /* Network variable data             */
} UnprocessedNV;

/* Data field for network variables (network interface selection enabled).  */

typedef struct {
    byte       index;                  /* Index into NV configuration table  */
    byte       rsvd0;
    byte       data[ MAX_NETVAR_DATA ]; /* Network variable data             */
} ProcessedNV;

/* Data field for explicit messages. */

typedef struct {
    byte       code;                   /* Message code                       */
    byte       data[ MAX_NETMSG_DATA ]; /* Message data                      */
} ExplicitMsg;

/* Union of all data fields. */

typedef union {
    UnprocessedNV unv;
    ProcessedNV   pnv;
    ExplicitMsg   exp;
} MsgData;
```

```
/*
 ****************************************************************************
 * Message buffer types.
 ****************************************************************************
 */

/* Application buffer when using explicit addressing. */

typedef struct {
    NI_Hdr       ni_hdr;            /* Network interface header             */
    MsgHdr       msg_hdr;           /* Message header                       */
    ExplicitAddr addr;              /* Network address                      */
    MsgData      data;              /* Message data                         */
} ExpAppBuffer;

/* Application buffer when not using explicit addressing. */

typedef struct {
    NI_Hdr       ni_hdr;            /* Network interface header             */
    MsgHdr       msg_hdr;           /* Message header                       */
    MsgData      data;              /* Message data                         */
} ImpAppBuffer;

/*
 ****************************************************************************
 * Network driver error codes.
 ****************************************************************************
 */

typedef enum {
    LDV_OK = 0,
    LDV_NOT_FOUND,
    LDV_ALREADY_OPEN,
    LDV_NOT_OPEN,
    LDV_DEVICE_ERR,
    LDV_INVALID_DEVICE_ID,
    LDV_NO_MSG_AVAIL,
    LDV_NO_BUFF_AVAIL,
    LDV_DEVICE_BUSY
} LDVCode;

/*
 ****************************************************************************
 * Network interface error codes.
 ****************************************************************************
 */

typedef enum {
    NI_OK = 0,
    NI_NO_DEVICE,
    NI_DRIVER_NOT_OPEN,
    NI_DRIVER_NOT_INIT,
    NI_DRIVER_NOT_RESET,
    NI_DRIVER_ERROR,
    NI_NO_RESPONSES,
    NI_RESET_FAILS,
    NI_TIMEOUT,
    NI_UPLINK_CMD,
    NI_INTERNAL_ERR,
} NI_Code;

/***************************** Externals *****************************/

NI_Code ni_init(char * device_name);    /* Initialize network interface   */

NI_Code ni_reset(void);                 /* Reset network interface        */

    /* Assumes only one network interface is open */

    /* Assume network interface is configured with explicit addressing ON,
       and network variable processing OFF */

NI_Code ni_send_msg_wait(
    ServiceType       service,          /* ACKD, UNACKD_RPT, UNACKD, REQUEST */
    const SendAddrDtl * out_addr,       /* address of outgoing message */
    const MsgData     * out_data,       /* data of outgoing message */
    int               out_length,       /* length of outgoing message */
    boolean           priority,         /* outgoing message priority */
    boolean           out_auth,         /* outgoing message authenticated */
    ComplType       * completion,       /* MSG_SUCCEEDS or MSG_FAILS */
    int             * num_responses,    /* number of received responses */
    RespAddrDtl     * in_addr,          /* address of first response */
    MsgData         * in_data,          /* data of first response */
    int             * in_length         /* length of first response */
);

NI_Code ni_get_next_response(         /* get subsequent responses here */
    RespAddrDtl     * in_addr,
    MsgData         * in_data,
    int             * in_length
);

NI_Code ni_receive_msg(
    ServiceType     * service,        /* ACKD, UNACKD_RPT, UNACKD, REQUEST */
    RcvAddrDtl      * in_addr,        /* address of incoming msg */
    MsgData         * in_data,        /* data of incoming msg */
    int             * in_length,      /* length of incoming msg */
    boolean         * in_auth         /* if incoming was authenticated */
);

NI_Code ni_send_response(         /* send response to last received request */
    MsgData       * out_data,     /* data for outgoing response */
    int             out_length    /* length of outgoing response */
);

NI_Code ni_send_immediate( NI_NoQueueCmd command );
            // send an immediate (no queue) command to network interface */

extern ExpAppBuffer  msg_out;          /* Outgoing message buffer      */
extern ExpAppBuffer  msg_in;           /* Incoming message buffer      */
extern void ni_msg_hdr_init( int msg_size, ServiceType service,
    boolean priority, boolean local, boolean auth, boolean implicit,
    byte msg_tag );

void ni_error_display(const char * s, NI_Code ni_error);
                                     /* Display a network interface error*/

void ni_ldv_error_display(const char * s, LDVCode ldv_error);
                                     /* Display a network driver error   */

void ni_msg_display( ExpAppBuffer *msg_ptr );

NI_Code ni_put_msg( void );                /* prototypes for low level calls */
NI_Code ni_get_msg( boolean wait );
```

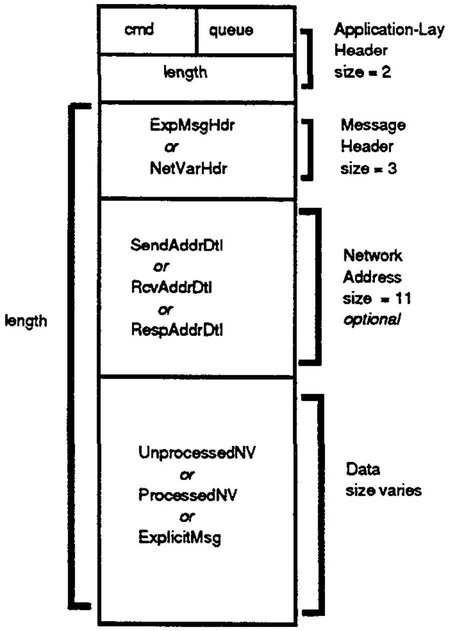
NI_MGMT.H

```
/*
 *           NI_MGMT.H
 *
 * Network management codes and messages for an application node.
 * This file contains a subset of the network management structures
 * The types have been changed where necessary because of the difference
 * in representations between Neuron and DOS objects.
 */

#define ID_STR_LEN 8            /* program ID length */

#define NULL_IDX  15            /* unused address table index */

typedef struct {
    unsigned    selector_hi : 6;
    unsigned    direction   : 1;
    unsigned    priority    : 1;
    unsigned    selector_lo : 8;
    unsigned    addr_index  : 4;
    unsigned    auth        : 1;
    unsigned    service     : 2;
    unsigned    turnaround  : 1;
} nv_struct;

// Note - Microsoft C will make nv_struct 4 bytes long because it does
// not allow odd-length bit-fields.  It is really 3 bytes long.
// Be careful when using sizeof() any structure that includes an nv_struct.

#define NM_update_nv_cnfg        0x6B
#define NM_update_nv_cnfg_fail   0x0B
#define NM_update_nv_cnfg_succ   0x2B

typedef struct {
    byte        code;
    nv_struct   nv_cnfg;
} NM_query_nv_cnfg_response;

#define NM_query_nv_cnfg        0x68
#define NM_query_nv_cnfg_fail   0x08
#define NM_query_nv_cnfg_succ   0x28

typedef enum {
    APPL_OFFLINE   = 0,          /* Soft offline state                  */
    APPL_ONLINE,
    APPL_RESET,
    CHANGE_STATE
} nm_node_mode;

typedef enum {
    APPL_UNCNFG    = 2,
    NO_APPL_UNCNFG = 3,
    CNFG_ONLINE    = 4,
    CNFG_OFFLINE   = 6,          /* Hard offline state                  */
    SOFT_OFFLINE   = 0xC
} nm_node_state;

typedef struct {
    byte        code;
    byte        mode;               /* Interpret with 'nm_node_mode'        */
    byte        node_state;         /* Optional field if mode==CHANGE_STATE */
                                    /* Interpret with 'nm_node_state'       */
} NM_set_node_mode_request;

#define NM_set_node_mode 0x6C

typedef enum {
    ABSOLUTE           = 0,
    READ_ONLY_RELATIVE = 1,
    CONFIG_RELATIVE    = 2,
} nm_mem_mode;

typedef enum {
    NO_ACTION      = 0,
    BOTH_CS_RECALC = 1,
    CNFG_CS_RECALC = 4,
    ONLY_RESET     = 8,
    BOTH_CS_RECALC_RESET = 9,
    CNFG_CS_RECALC_RESET = 12,
} nm_mem_form;

typedef struct {
    byte    code;
    byte    mode;
    byte    offset_hi;
    byte    offset_lo;
    byte    count;
    byte    form;                     // followed by the data
} NM_write_memory_request;

#define NM_write_memory 0x6E

typedef struct {
    byte    length_hi;
    byte    length_lo;
    byte    num_netvars;
    byte    version;          // version 0 format
    byte    mtag_count;
} snvt_struct_v0;

typedef struct {
    byte    length_hi;
    byte    length_lo;
    byte    num_netvars_lo;
    byte    version;           // version 1 format
    byte    num_netvars_hi;
    byte    mtag_count;
} snvt_struct_v1;

// Partial list of SNVT type index values
```

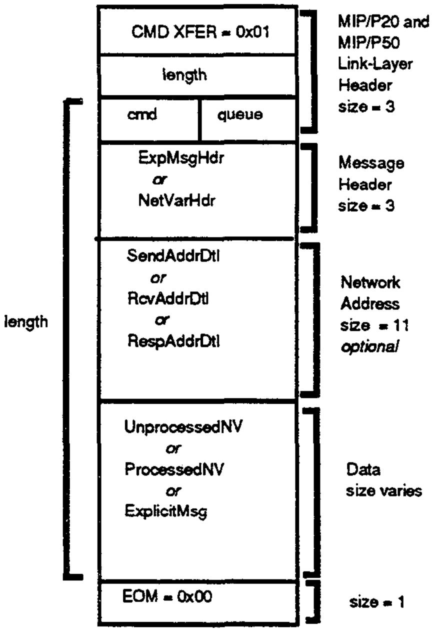
```
typedef enum {
    SNVT_str_asc  = 36,
    SNVT_lev_cont = 21,
    SNVT_lev_disc = 22,
    SNVT_count_f  = 51,
} SNVT_t;

typedef struct {
    unsigned    nv_config_class     :1;
    unsigned    nv_auth_config      :1;
    unsigned    nv_priority_config  :1;
    unsigned    nv_service_type_config :1;
    unsigned    nv_offline          :1;
    unsigned    nv_polled           :1;
    unsigned    nv_sync             :1;
    unsigned    ext_rec             :1;
    bits        snvt_type_index     :8;        // use enum SNVT_t
} snvt_desc_struct;

typedef struct {
#ifdef   _MSC_VER
    byte        mask;          // Microsoft C does not allow odd-length
#else                          // bit fields
    unsigned    unused :3;
    unsigned    nc     :1;     // array count
    unsigned    sd     :1;     // self-documentation
    unsigned    nm     :1;     // network variable name
    unsigned    re     :1;     // rate estimate
    unsigned    mre    :1;     // max rate estimate
#endif
} snvt_ext_rec_mask;

// Network management message codes

#define NM_query_SNVT           0x72
#define NM_query_SNVT_fail      0x12
#define NM_query_SNVT_succ      0x32

typedef struct {
    byte    code;
    word    offset;     // big-endian 16-bits
    byte    count;
} NM_query_SNVT_request;

#define NM_wink 0x70

#define NM_NV_fetch         0x73
#define NM_NV_fetch_fail    0x13
#define NM_NV_fetch_succ    0x33

// Application-specific structure used by host application to store network variables

typedef enum { NV_IN = 0, NV_OUT = 1 } nv_direction;

typedef struct {                          // structure to define NVs
```

```
    int             size;                // number of bytes
    nv_direction    direction;           // input or output
    const           char * name;         // name of variable
    void            ( * print_func )( byte * ); // routine to print value
    void            ( * read_func )( byte * );  // routine to read value
    byte            data[ MAX_NETVAR_DATA ];    // actual storage for value
} network_variable;
```

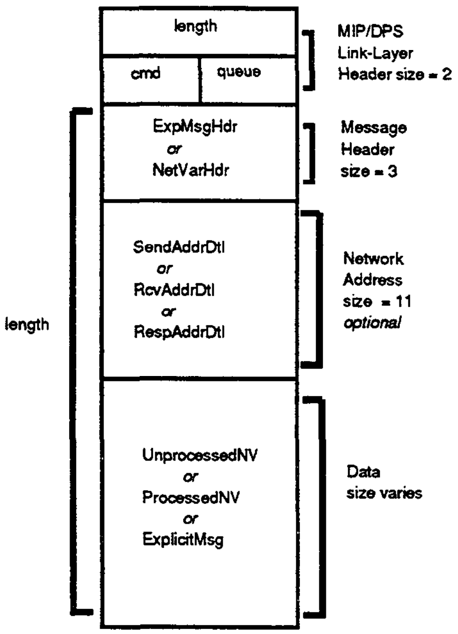
Bit Field Diagrams



Application-Layer Buffer (hardware independent)



MIP/P20 and MIP/P50 Link-Layer Buffer



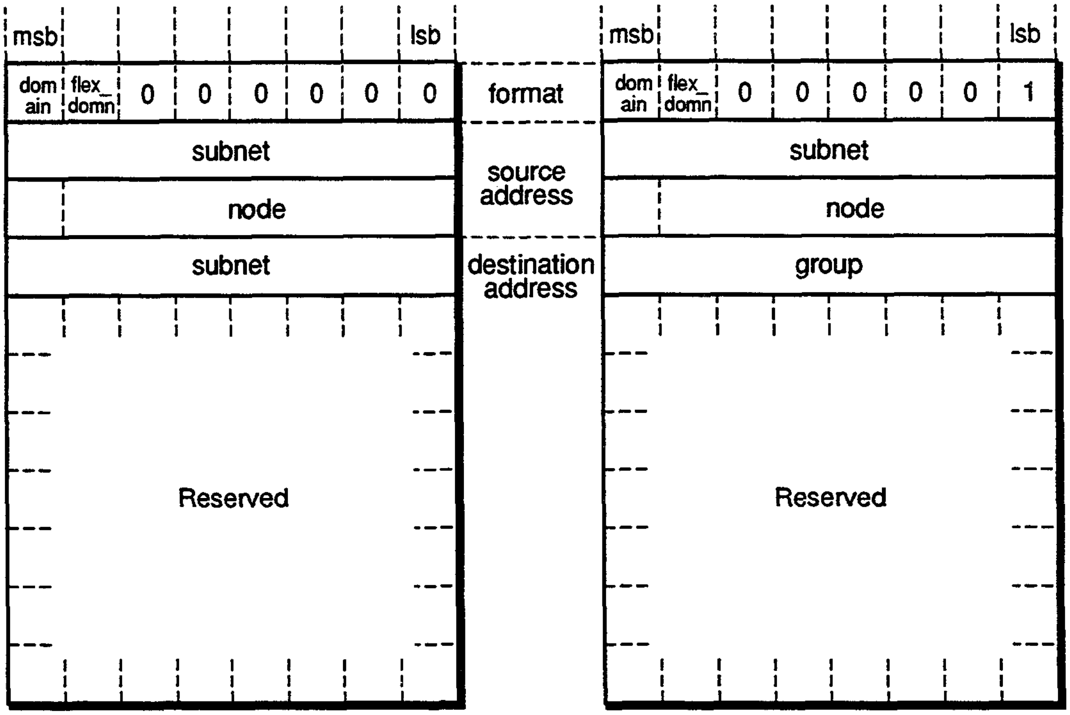
MIP/DPS Link-Layer Buffer

<table>
<tr><td>msb</td><td></td><td></td><td></td><td></td><td></td><td></td><td>lsb</td></tr>
<tr><td>0</td><td colspan="2">st</td><td>auth</td><td colspan="4">tag</td></tr>
<tr><td>priority</td><td>path</td><td colspan="2">cmpl_code</td><td>addr mode</td><td>atl_ path</td><td>pool</td><td>resp</td></tr>
<tr><td colspan="8">length</td></tr>
</table>

ExpMsgHdr
(explicit messages
or unprocessed NVs)

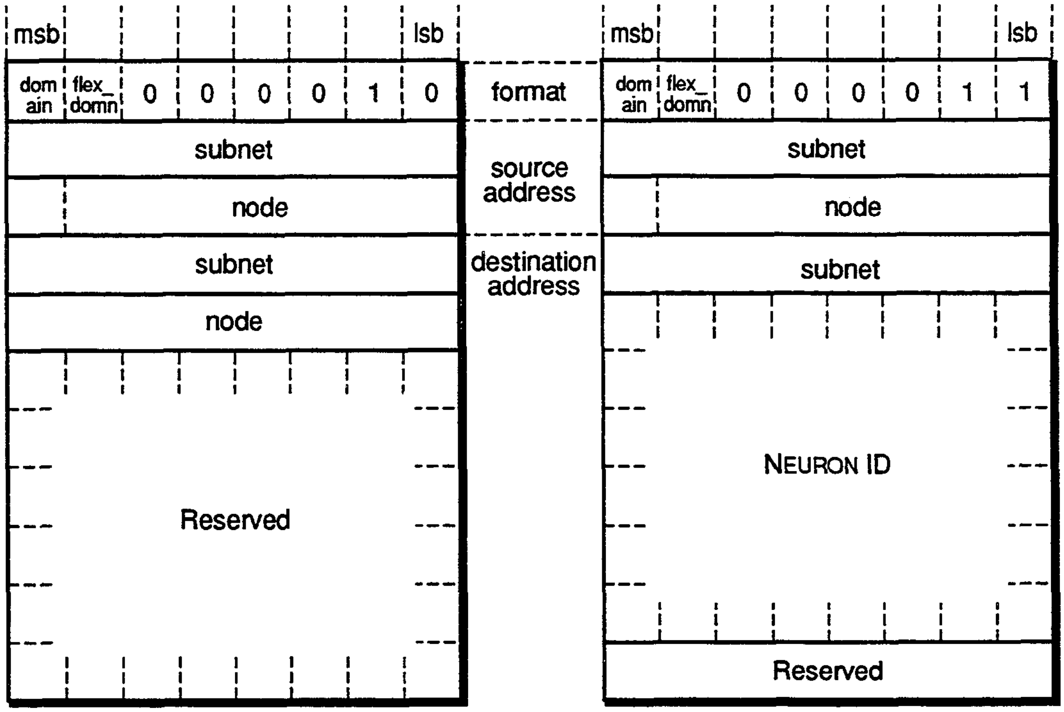
<table>
<tr><td>msb</td><td></td><td></td><td></td><td></td><td></td><td></td><td>lsb</td></tr>
<tr><td>1</td><td>poll</td><td colspan="2">rsvd0</td><td colspan="4">tag</td></tr>
<tr><td>priority</td><td>path</td><td colspan="2">cmpl_code</td><td>addr mode</td><td>trn arnd</td><td>pool</td><td>resp</td></tr>
<tr><td colspan="8">length</td></tr>
</table>

NetVarHdr
(processed NVs)



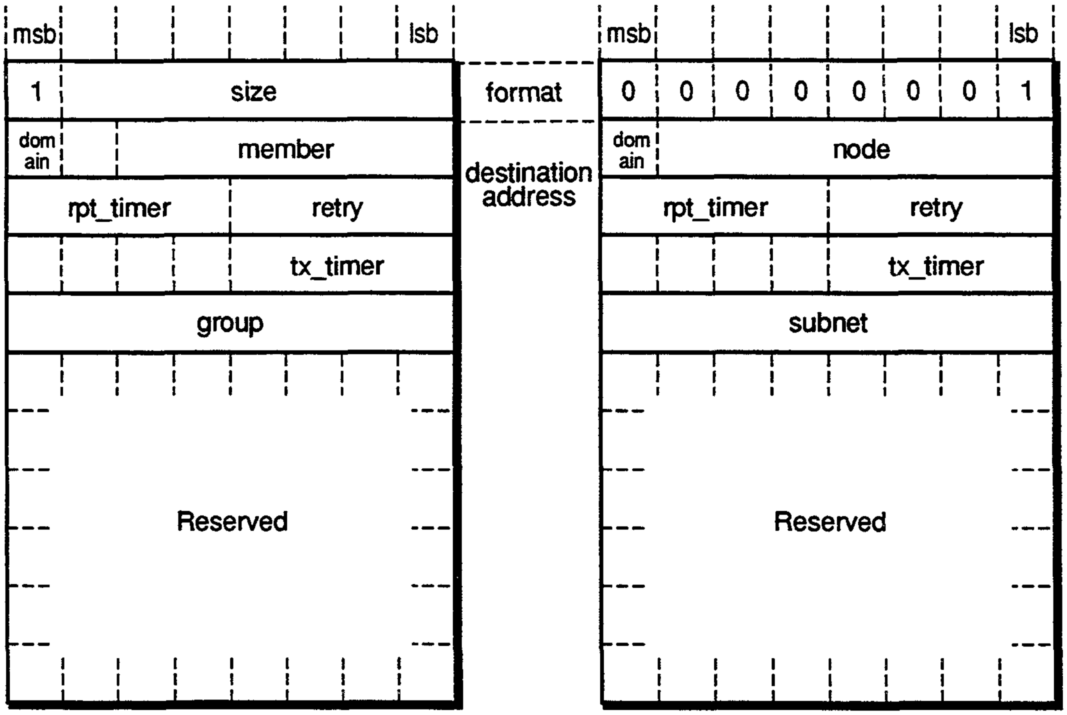
RcvAddrDtl
received address
for broadcast addressing

RcvAddrDtl
received address
for group addressing

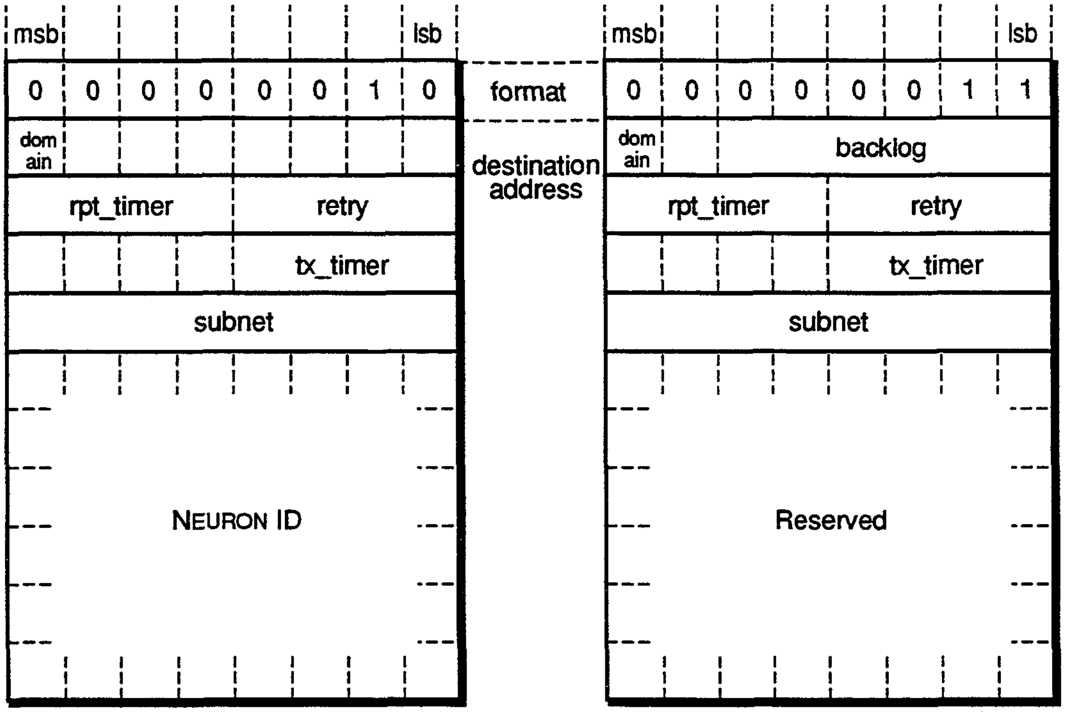


RcvAddrDtl
received address
for subnet/node addressing

RcvAddrDtl
received address
for NEURON ID addressing

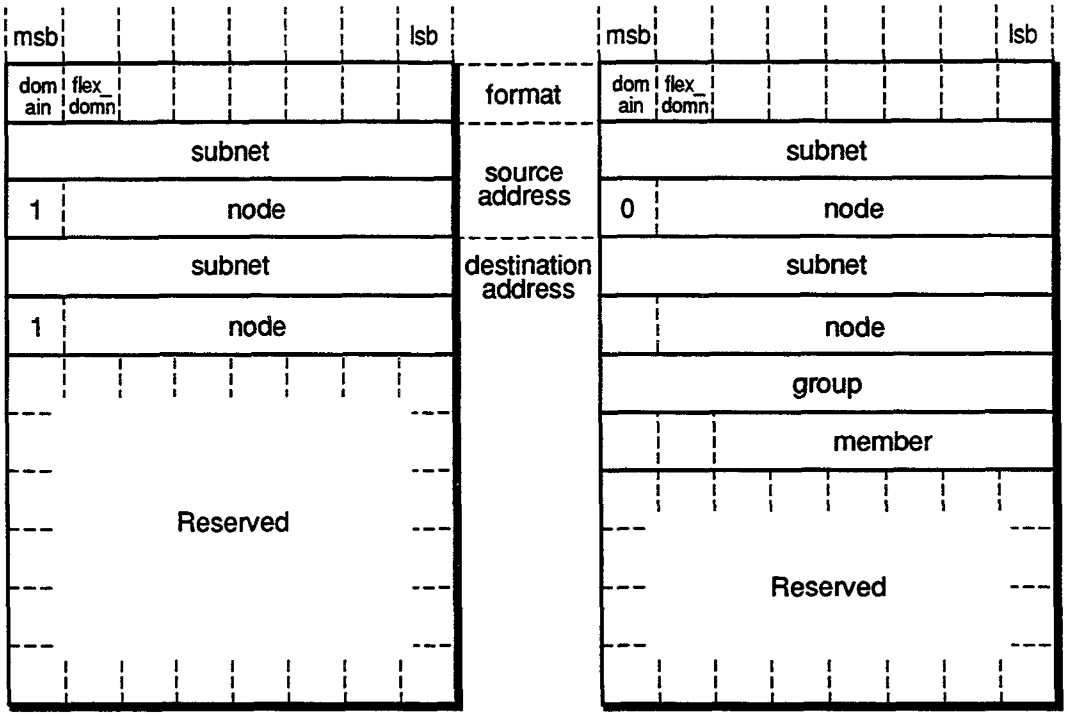


SendGroup **SendSnode**



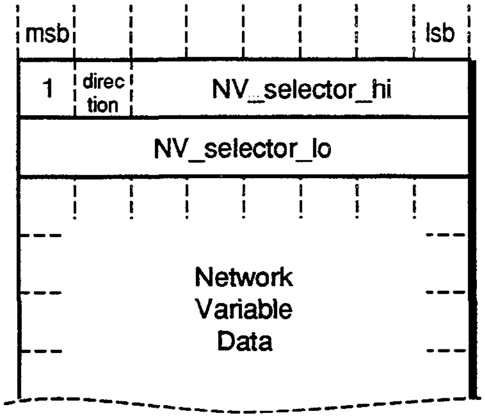
SendNrnid

SendBcast

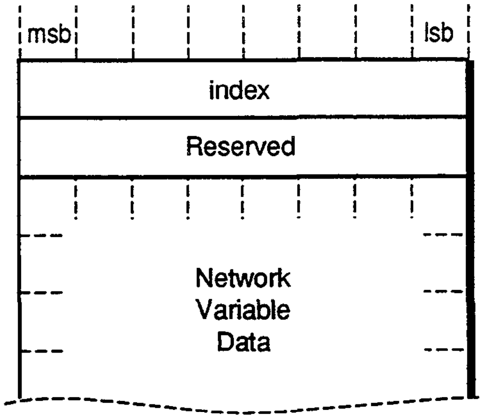


RespAddrDtl
response address
for subnet/node addressing

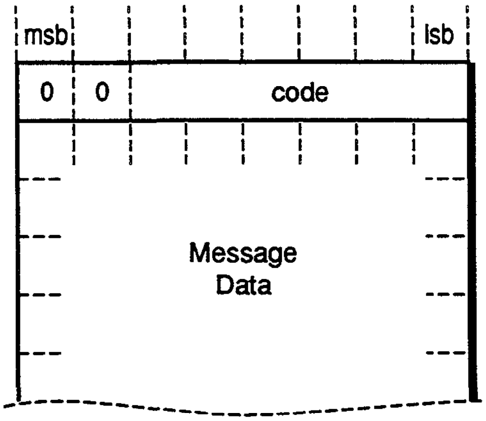
RespAddrDtl
response address
for group addressing



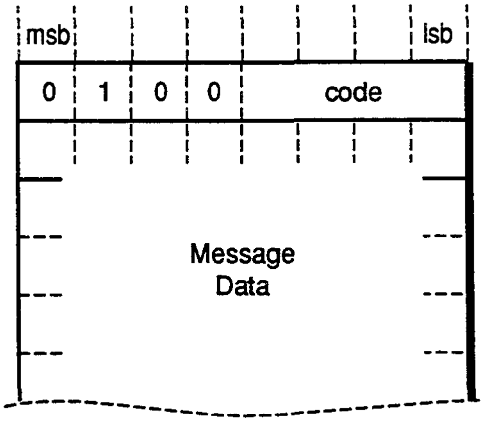
UnprocessedNV



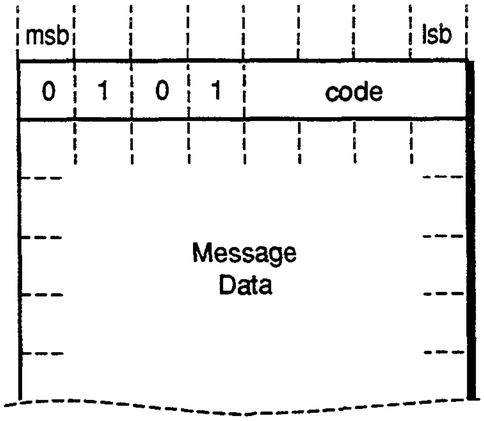
ProcessedNV



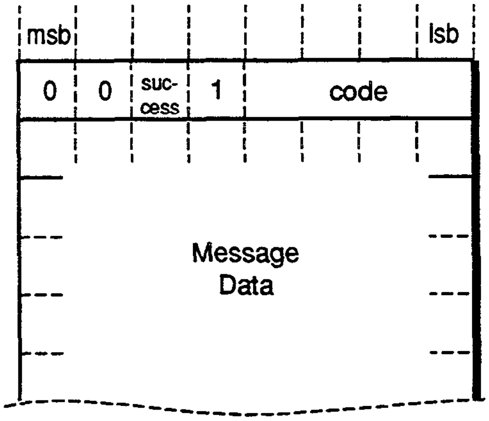
ExplicitMsg
(application messages
and responses)



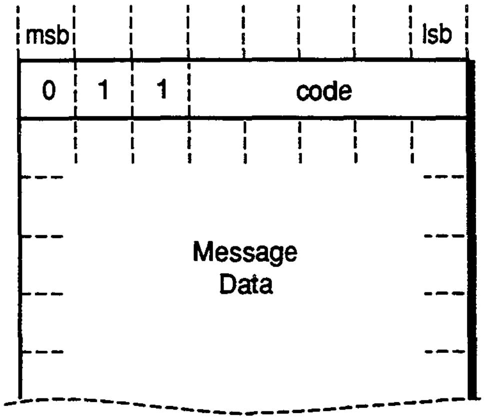
ExplicitMsg
(foreign messages
and responses)



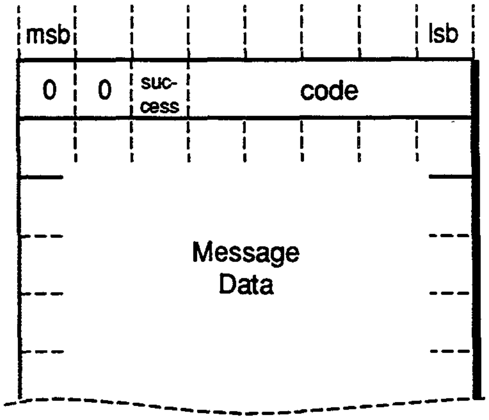
ExplicitMsg
(network diagnostic messages)



ExplicitMsg
(network diagnostic responses)



ExplicitMsg
(network management messages)



ExplicitMsg
(network management responses)

Appendix D

Network Interface Commands

This appendix defines the network interface commands specified in a data transfer from a host application to the network interface. All data transfers (non-NULL transfers) to the network interface include a length byte and a command byte.

Network Interface Commands

code	value	up/down link
niCOMM	0x10	u+d

The niCOMM command is ORed with the buffer queue code, and is used to both request an output buffer and to send an output buffer. It is also used by the network interface to pass completion events to the host. Buffer queue codes are defined in the next section.

code	value	up/down link
niNETMGMT	0x20	d

Network Management Command. This code is ORed with one of the buffer queue codes for performing local network management or network diagnostic commands on the network interface itself. Typical buffer queue codes used would be `niTQ` for request/response commands or `niNTQ` for unacked commands. Buffer queue codes are defined in the next section.

code	value	up/down link
niRESET	0x50	u+d

This code is sent uplink whenever the network interface has executed a hardware or software reset. When this code is sent downlink, the network interface resets immediately.

code	value	up/down link
niFLUSH_CANCEL	0x60	d

This command cancels any flush operations posted within the network interface.

code	value	up/down link
niFLUSH_COMPLETE	0x60	u

This command informs the host that the FLUSH operation is complete.

code	value	up/down link
niONLINE	0x70	d

This command places the network interface in the online state.

code	value	up/down link
niOFFLINE	0x80	d

This command places the network interface in the offline state.

code	value	up/down link
niFLUSH	0x90	d

This command places the network interface in the flush state.

code	value	up/down link
niFLUSH_IGN	0xA0	d

This command places the network interface in the flush, ignore comm. state.

code	value	up/down link
niACK	0xC0	u

This command is sent uplink following a buffer queue request when a buffer becomes available. This command is not available with the MIP/DPS.

code	value	up/down link
niNACK	0xC1	u

This command is sent uplink immediately following a buffer queue request if there are no buffers available (SLTA only).

code	value	up/down link
niPUPXOFF	0xE1	d

Uplink source quench command. This command is not available with the MIP/DPS.

code	value	up/down link
niPUPXON	0xE2	d

Uplink source resume command. This command is not available with the MIP/DPS.

code	value	up/down link
niSLEEP	0xB0	d

This command forces the SLTA, MIP/P50, or MIP/DPS to enter the sleep state. For the SLTA, the EIA-232 transceivers are disabled, the UART clock is shut down, and the NEURON CHIP enters the sleep state. Network activity will not wake up the node. Only depressing the service pin or sending a downlink transaction to the SLTA will awaken the NEURON CHIP. The provided driver takes care of waking up the SLTA by sending it a break character frame. The SLTA then acknowledges that it has awakened by sending an `niACK` command uplink. For the MIP/P50 and MIP/DPS, the NEURON CHIP enters the sleep state. Network activity and depressing the service pin will awaken the NEURON CHIP. Downlink transactions will not wake up the node. The `niSLEEP` command is not available with the MIP/P20.

code	value	up/down link
niSSTATUS	0xE0	u+d

This command, when sent downlink, will cause the SLTA to respond with a `niSSTATUS` command plus one byte of data. This byte of data contains the four most significant bits of the config switches on the SLTA (CFG0-3) in the four most significant bits of the data byte, plus the battery test bit value in the least significant bit of the data byte. The battery test bit is low when the battery voltage is below the acceptable voltage for continued operation. Not available with the MIP.

Buffer Queue Values

For the `niCOMM` and `niNETMGMT` commands there is a buffer queue code encoded into the lower four bits. This code defines which buffer queue within the NEURON CHIP will be used for downlink messages.

code	value	up/down link
niTQ	0x02	d

Use the transaction queue. The transaction queue is for all transaction oriented outgoing messages and for outgoing network variable updates. Messages are sent in the same order as they are queued. Because only one such transaction may be outstanding at a time, another message will not be sent from this queue until all acknowledgements for the previous message are received.

code	value	up/down link
niTQ_P	0x03	d

This is the priority version of the transaction queue.

code	value	up/down link
niNTQ	0x04	d

Use the non-transaction queue. The non-transaction queue is for any response to a message or response to a network variable poll or for non-transaction oriented messages (unacknowledged or repeated service). These messages have priority over transaction oriented messages.

code	value	up/down link
niNTQ_P	0x05	d

This is the priority version of the non-transaction queue.

code	value	up/down link
niRESPONSE	0x06	u

Response message queue. This queue contains any received response message addressed to this node. When using the request-response message service the responses appear in this queue. Completion events are also passed back to the host in this queue.

code	value	up/down link
niINCOMING	0x08	u

Received message queue. This queue contains any received message which was addressed to this node and whose code was an application or foreign type. This queue also contains all network variable updates and polls. If host selection is enabled, then network variable configuration network management messages also appear in this queue.

Index