LonWorks® Microprocessor Interface Program(MIP) User's Guide

Revision 3

ECHELON®
Corporation

078-0017-01C

Document No. 29500

Printed in the United States of America.

Echelon Corporation
4015 Miranda Avenue
Palo Alto, California
94304

Preface

This document describes how to build a LONWORKS® network interface using the LONWORKS Microprocessor Interface Program (MIP). The MIP is firmware for the Neuron Chip that transforms the Neuron Chip into a communications coprocessor for an attached host processor. The MIP moves the upper layers of the LonTalk® Protocol from the Neuron Chip to the attached host. Network interfaces based on the MIP can be used to create host applications on a variety of host processors including PCs, work-stations, microprocessors, and microcontrollers.

Three MIPs are described in this document. The first two, the MIP/P20 and MIP/P50, use an 11-bit parallel interface to the host processor. The third, the MIP/DPS, uses a dual-ported RAM with hardware semaphores for communicating with the host processor. The MIP/P20 and MIP/P50 are included with the Model 23200 LONWORKS MIP/P20 and MIP/P50 Developer's Kit. The MIP/DPS is included with the Model 23210 LONWORKS MIP/DPS Developer's Kit.

Audience

The *LONWORKS MIP User's Guide* is intended for LONWORKS developers who are developing one of the following:

- a LONWORKS network interface that will be embedded within an application node that includes a host processor; the network interface provides an interface between the host processor and a LONWORKS network
- a generic LONWORKS network interface that will be provided for use by third parties who will integrate the network interface with a host processor to create an application node

Programming examples are shown in ANSI C, however, network drivers and host applications may be written in any language that can implement the LONWORKS network interface protocol. Readers of this guide developing network drivers or host applications 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.

A LonBuilder Developer's Workbench, or a LONWORKS NodeBuilder Development Tool is required to develop a network interface based on any version of the MIP. Once a network interface is developed, these tools are not required to create a host application.

Content

The *LONWORKS MIP User's Guide* has six chapters as follows:

- Chapter 1, *MIP Overview*, provides an introduction to the MIP.
- Chapter 2, *Installing the MIP Software*, describes how to install the MIP software on a PC.
- Chapter 3, *Creating a MIP Image*, describes the process of building the MIP image that will be loaded onto a network interface.
- Chapter 4, *Building a Network Interface*, describes the process of building a network interface with the MIP.
- Chapter 5, *Creating a Network Driver*, describes the process of building a network driver for a host that is to be connected to a MIP-based network interface.
- Chapter 6, *Installing the DOS Network Driver*, describes how a MIP network driver is installed on a DOS host.

Related Manuals

The following manuals and engineering bulletins provide supplemental information to the material in this guide:

- The *LONWORKS Host Application Programmer's Guide* (document no. 29400) 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. Network interfaces based on the MIP provide the communications interface between a host application and a LONWORKS network.

- The *Parallel I/O Interface to the Neuron Chip* engineering bulletin describes the hardware interface used by the MIP/P20 and MIP/P50.
- 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 user interface to the LonBuilder software.
- The *NodeBuilder User's Guide* lists and describes all tasks related to LONWORKS application development using the NodeBuilder Development Tool. Refer to that guide for detailed information on the user interface to the NodeBuilder software.
- The *Neuron C Programmer's Guide* outlines a recommended general approach to developing Neuron C applications, and explains key concepts of programming in Neuron C through the use of code fragments and examples.
- The *Neuron C Reference Guide* provides a complete reference section for Neuron C.
- The *Custom Node Development* engineering bulletin describes the steps for building an example LONWORKS application node.
- The *LonTalk Protocol* engineering bulletin describes the LonTalk Protocol.
- The *Neuron Chip Data Book*, Appendix B, provides a description of the network management and diagnostic message formats that must be handled by host application nodes.
- The *LONWORKS Network Services Architecture Technical Overview* describes how to create host applications that install, maintain, monitor, or control LONWORKS networks.
- The *LONWORKS Component Architecture* document describes how to create host applications for Microsoft Windows '95 and Windows NT.
- The *LonManager DDE Server User's Guide* describes how to create network monitoring and control applications based on the LonManager DDE Server. The LonManager DDE Server greatly simplifies user interface and database application development on Microsoft Windows-based hosts.

Contents

1

MIP Overview

Welcome to Release 1.1 of the LONWORKS MIP/DPS Developer's Kit and Release 2.3 of the LONWORKS MIP/P20 and MIP/P50 Developer's Kit. These kits enable you to create LONWORKS network interfaces. These network interfaces can be used to create host applications that communicate using the LonTalk protocol and run on processors other than the Neuron Chip.

Host applications and network interfaces extend the reach of LONWORKS technology to a variety of hosts including PCs, workstations, embedded microprocessors, and microcontrollers.

This manual describes how to build a network interface using one of the MIPs. See the *LONWORKS Host Application Programmer's Guide* for a description of how to write a host application.

A host application combined with a LONWORKS network interface may be used to:

- Add processing power to a LONWORKS network
- Move an existing product to a LONWORKS network
- Implement a network management, monitoring, or control tool based on the LonManager API

This chapter provides an overview of the MIP products. The architecture of a network interface based on the MIP is described, and the components of the MIP Developer's Kits are described. New features also are described.

LONWORKS Network Interface Architecture

A network interface is a device that provides a communications interface between a host processor and a LONWORKS network. The network interface may be a turn-key device such as the SLTA/2 Serial LonTalk Adapter, the PCLTA PC LonTalk Adapter, or the PCNSS PC Interface Card. A custom network interface may be based on the LTM-10 LonTalk Module, LTS-10 Serial LonTalk Adapter module, the LonManager NSS-10 Network Services Server module, or may contain a Neuron Chip running the Microprocessor Interface Program (MIP), and support circuitry. Network interfaces also include a LONWORKS transceiver to connect the Neuron Chip to the communications medium.

The MIP is firmware that transforms the Neuron Chip into a communications coprocessor for an attached host processor. The MIP moves the upper layers of the LonTalk Protocol from the Neuron Chip to the attached host. Three MIPs are available which support different interfaces to the host. The three MIPs are:

- ***MIP/P20***. Uses an 11-bit parallel interface between the host and network interface Neuron Chip. The host can view the Neuron Chip as an 8-bit I/O port with 3 handshake and control lines, or as two 8-bit memory-mapped locations. The MIP/P20 can be run on a Neuron 3120 Chip or a Neuron 3150 Chip, but is usually used only with the Neuron 3120 Chip.
- ***MIP/P50***. Uses an 11-bit parallel interface between the host and network interface Neuron Chip, with the addition of an optional uplink interrupt generated by a memory write from the Neuron Chip. As with the MIP/P20, the host can view the Neuron Chip as an 8-bit I/O port with 3 handshake and control lines, or as two 8-bit memory-mapped locations. The MIP/P50 provides faster throughput than the MIP/P20, even if the uplink interrupt is not used, but it requires a Neuron 3150, 3120E1, or 3120E2 Chip. Use of the uplink interrupt requires a Neuron 3150 Chip. The MIP/P50 firmware and interrupt decoding hardware is included on the LTM-10 LonTalk Module.
- ***MIP/DPS***. Uses a dual-ported memory interface between the host and network interface Neuron Chip. The dual ported memory must provide hardware semaphores to control access to the shared memory by either the host or network interface. The MIP/DPS provides faster throughput and lower host overhead than either the MIP/P20 or the MIP/P50, but requires a Neuron 3150 Chip and a high-speed dual ported memory chip compatible with the IDT71342 or Cypress 78144. The MIP/DPS is typically used with high-end 32-bit microprocessors, but it may be used with any host processor.

Figure 1.1 illustrates the architecture of a network interface based on the MIP/P20. Figure 1.2 illustrates the architecture of a network interface based on the MIP/P50. Figure 1.3 illustrates the architecture for a MIP/DPS-based network interface.

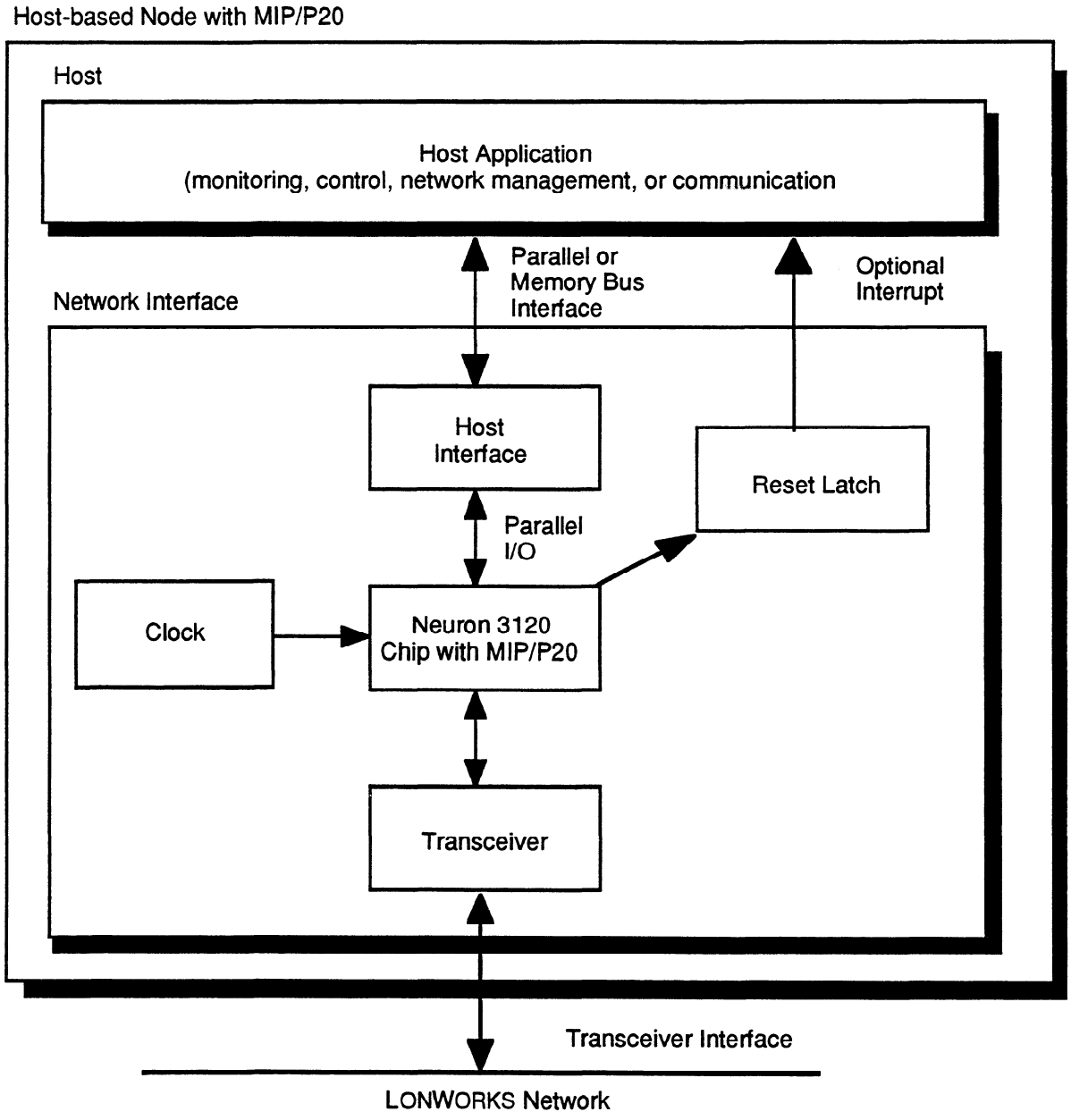


Figure 1.1 MIP/P20-based Network Interface Architecture

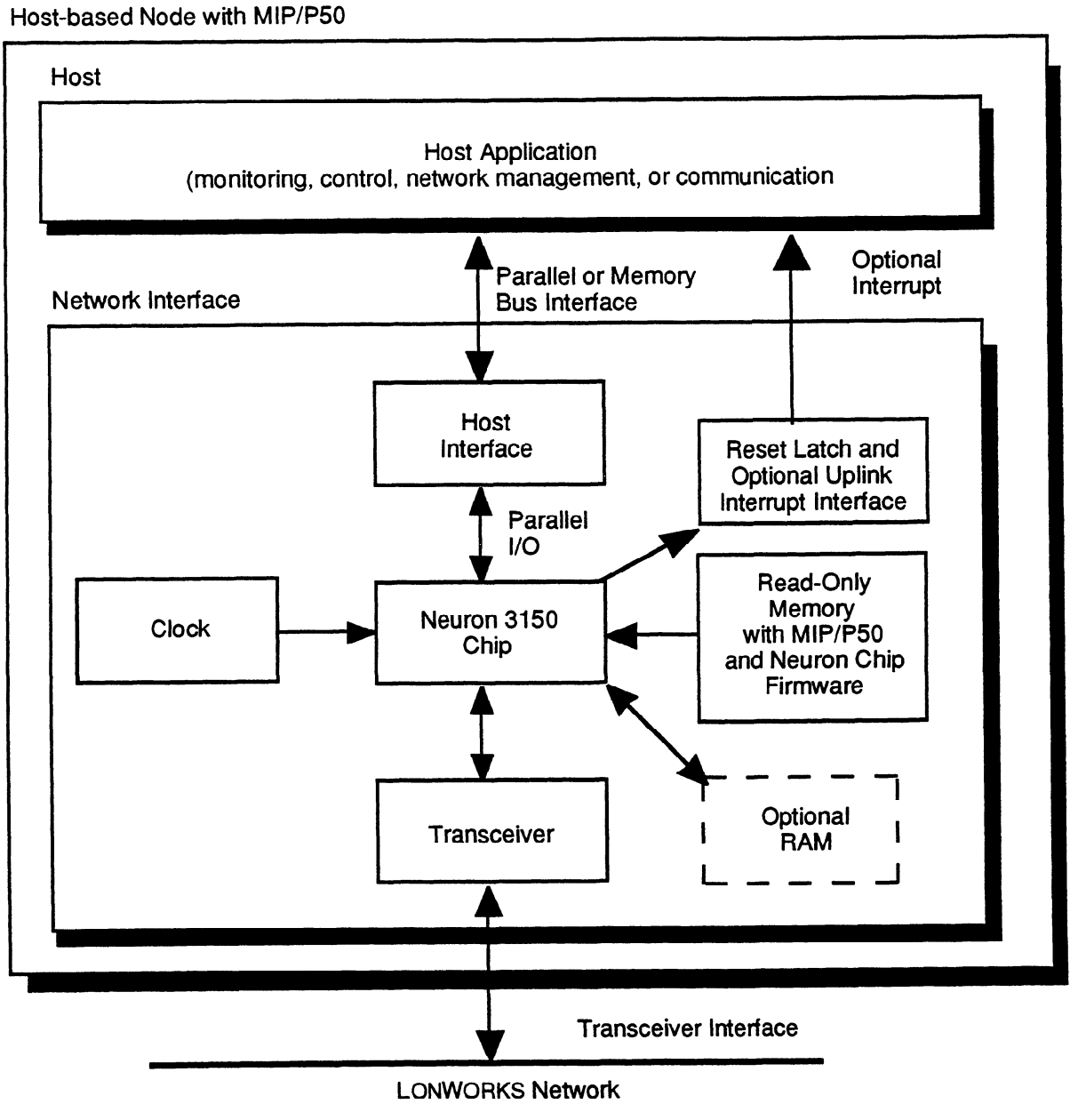


Figure 1.2 MIP/P50-based Network Interface Architecture

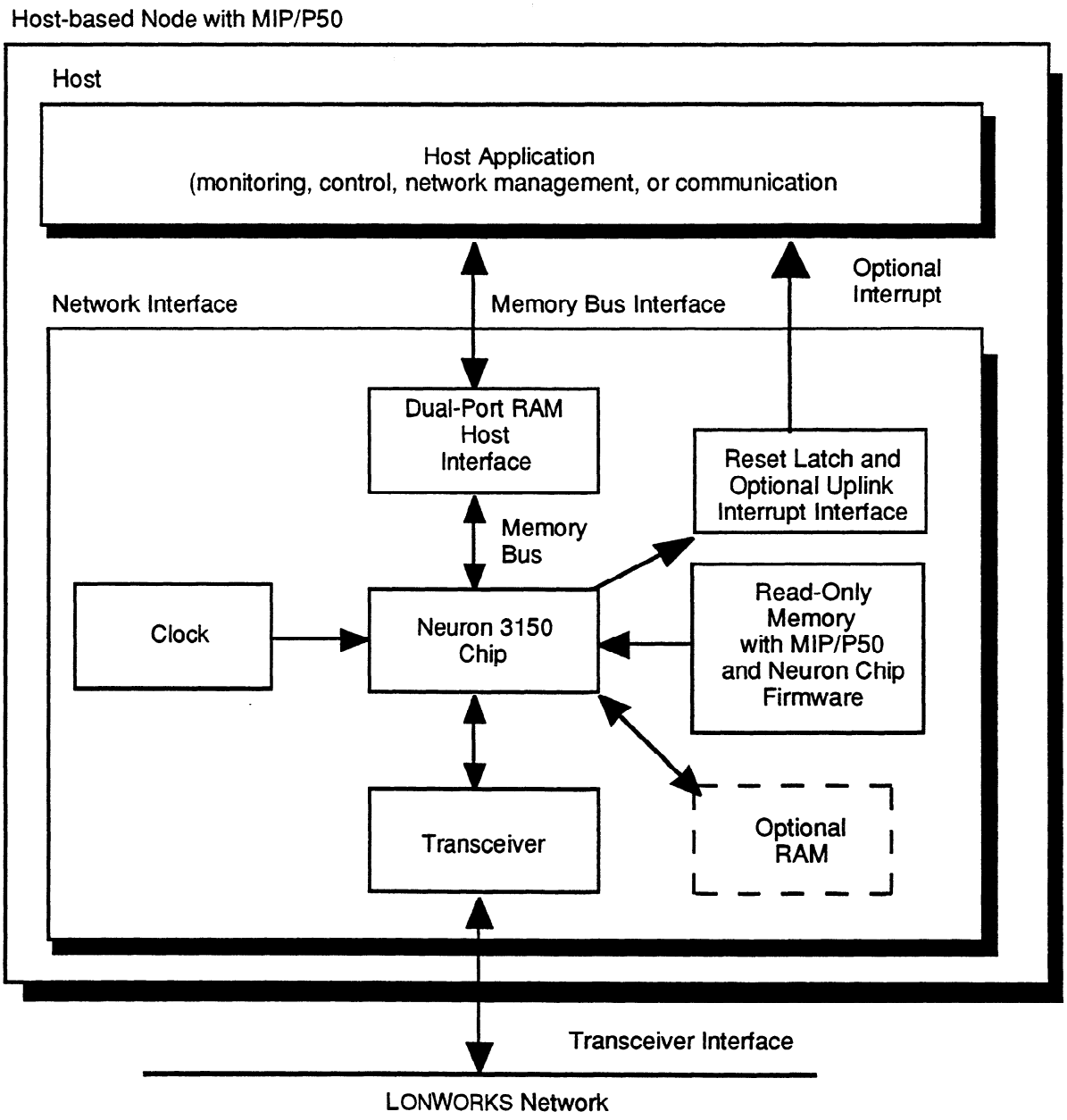


Figure 1.3 MIP/DPS-based Network Interface Architecture

LONWORKS MIP Developer's Kits

Two developer's kits are available for the MIP. The LONWORKS MIP/P20 and MIP/P50 Developer's Kit includes the software required to build a custom network interface based on the MIP/P20 and MIP/P50. The LONWORKS MIP/DPS Developer's Kit provides similar components for the MIP/DPS. The kits include:

- **Microprocessor Interface Program (MIP).** The MIP is firmware for the Neuron Chip that extends the Neuron Chip firmware. The MIP is delivered as an object library that is linked with a Neuron C MIP application program to create the MIP image.
- **Sample MIP Application.** The MIP application is a short Neuron C program that specifies the configuration of the MIP node and invokes the MIP function.
- **Sample Network Driver.** An example DOS device driver for the MIP. The network driver provides a device independent interface to the host application. This allows host applications to work transparently with different network interfaces, regardless of their physical interface. For example, a host application can be moved from a PCLTA PC LonTalk Adapter to an Echelon SLTA/2 Serial LonTalk Adapter to a custom network interface based on any of the MIPs without being modified. The sample driver is provided as ANSI C and assembly source code that can be modified for different host interfaces to the network interface.
- **Example Host Application.** A sample host application that polls, writes, and monitors network variables, and responds appropriately to network management messages. The example is provided as ANSI C source code that can be used as a starting point for creating custom host applications.

This documentation describes Release 2.3 of the LONWORKS MIP/P20 and MIP/P50 Developer's Kit, and Release 1.1 of the LONWORKS MIP/DPS Developer's Kit.

New Features

Release 2.3 of the MIP/P20 and MIP/P50 and Release 1.1 of the MIP/DPS include the following new features:

- ***Custom Uplink Interrupt Interface.*** The MIP/P50 now supports a custom uplink interrupt request function to support custom interrupt request hardware.
- ***Easier to use host application.*** The host application example has been enhanced. The network interface (NI) layer now uses a call-back architecture to invoke user-supplied routines to handle all types of incoming messages. This architecture is compatible with the architecture of the host application example provided with the LonManager NSS for Windows and the LonManager NSS-10 Network Services Server Developer's Kits. The host application also transparently handles retries of incoming request messages.

2

Installing the MIP Software

This chapter explains how to install the MIP software on a PC, and describes the software files installed during the installation process. Upgrades from previous releases are also described.

Software Installation Instructions

Follow these steps to install the MIP software on your PC. LonBuilder 3.0 or NodeBuilder 1.5 (or later) is required.

1 Place the installation disk in a floppy disk drive. Drive A: is used in the following steps; substitute your drive letter if you are using a drive other than A:. The installation disk is labeled *LONWORKS MIP/P20 and MIP/P50 Software* for the MIP/P20 and MIP/P50; the installation disk is labeled *LONWORKS MIP/DPS Software* for the MIP/DPS.

2 Start the automatic installation procedure by entering:

```
a:install ↵
```

Enter the drive letter of your floppy drive in place of the "a:" if you are using another drive.

3 After a moment of disk activity, the product name and version number will be displayed, along with the following message:

```
Press [Esc] to quit, any other key to continue . . .
```

The installation can be aborted at any time by pressing the [Esc] key. To continue with the installation, press any other key.

4 The next screen provides some basic instructions concerning installation. Press any key (other than [Esc]) to continue.

5 The next screen allowsyou to select whether you are installing the software into a LonBuilder software installation, or into a NodeBuilder software installation. Press ↵ when the proper choice is highlighted.

6 A list of available hard disk drives will be displayed. Use the arrow keys to select the drive. Press ↵ when the proper drive is highlighted.

7 A prompt for an installation directory will be displayed. Enter the name of a directory. The default directory is \LB or \LONWORKS. Press ↵ when the proper path is shown.

8 A number of compressed files will now be copied from the floppy to your hard disk, where they will be expanded. When all the files have been placed on your hard disk, you will be returned to DOS.

9 A network driver as described in Chapters 5 and 6 must loaded on your host processor before you can run a host application. If your host processor is a PC running DOS, the network driver must be specified in your config.sys file before you can run a host application on the PC. See Chapters 5 and 6 for more information.

Software Contents

The software for the MIP developer's kits is supplied on installation diskettes which contain the following:

Microprocessor Interface Program (MIP). The MIPs are contained in library files. For NodeBuilder installations, these files are installed in the LONWORKS IMAGES directory (the default directory is \LONWORKS\IMAGES). For LonBuilder installations, these files are installed in the version 3, 4, and 6 IMAGES directory (the default directories are \LB\IMAGES\VER3, \LB\IMAGES\VER4, and \LB\IMAGES\VER6). The filenames for the libraries are:

File Name	*MIP/Px0*	*MIP/DPS*	*Description*
mip_pio.lib	•		Parallel I/O MIP library. Includes the MIP/P20 and MIP/P50.
mip_dps.lib		•	MIP/DPS library. This library file may only be used with VER4 and VER6 Neuron Chip firmware
miput16.lib	•	•	Library functions used by the MIP libraries.

Sample MIP Application. The sample MIP applications are contained in Neuron C source files. For NodeBuilder installations, these files are installed in the NB\EXA\MIP directory (the default directory is \LONWORKS\NB\EXA\MIP). For LonBuilder installations, the files are installed in the EXAMPLES\MIP directory (the default directory is \LB\EXAMPLES\MIP). The filenames for the MIP applications are:

File Name	*MIP/Px0*	*MIP/DPS*	*Description*
mip_ap20.nc	•		MIP application for the MIP/P20
mip_ap50.nc	•		MIP application for the MIP/P50
mip_dps.nc		•	MIP application for the MIP/DPS

Sample Network Driver Source. The example DOS device drivers for the MIP are provided as ANSI C and assembly source files. For NodeBuilder installations, the sample network driver for the MIP/P20 and MIP/P50 is installed in the NB\EXA\MIP directory (the default directory is \LONWORKS\NB\EXA\MIP). For LonBuilder installations, the sample network driver for the MIP/P20 and MIP/P50 is installed in the EXAMPLES\MIP directory (the default directory is \LB\EXAMPLES\MIP).

For NodeBuilder installations, the sample network driver for the MIP/DPS is installed in the NB\EXA\MIP_DPS directory (the default directory is \LONWORKS\NB\EXA\MIP). For LonBuilder installations, the sample network driver for the MIP/DPS is installed in the EXAMPLES\MIP_DPS directory (the default directory is \LB\EXAMPLES\MIP_DPS). The sample network driver ANSI C source files are:

File Name	MIP/Px0	MIP/DPS	Description
dps_frst.c		•	Device driver header. Must be the first file linked.
dps_difc.c		•	DOS to network driver interface functions.
dps_mip.c		•	Low-level interface functions for the dual-ported RAM.
dps_last.c		•	Used to determine the size of the network driver. Must be the last file linked.
mip_frst.c	•		Device driver header. Must be the first file linked.
mip_difc.c	•		DOS to network driver interface functions.
mip_last.c	•		Used to determine the size of the device driver. Must be the last file linked.
mip_pio.c	•		Low-level interface functions to the MIP/P20 or MIP/P50 via the user implemented parallel port. Stub operations are provided that are replaced with code for the actual hardware.
mip_exec.c	•		Medium and high-level interface functions for the driver. Includes the read(), write(), open(), close(), and ioctl() functions.
mdv_time.c	•		Contains functions which support the usage of the PC/AT's counter # 0 hardware for timeouts required by various parts of the driver.

The ANSI C header files are:

File Name	MIP/Px0	MIP/DPS	Description
dpr_defs.h		•	Hardware and buffer definitions for the MIP/DPS.
dpr_prto.h		•	Control interface structure definitions and function prototypes for the MIP/DPS.
mip_typs.h	•		Control interface structure definitions and function prototypes type declarations for the MIP/P20 and MIP/P50 network driver C source files.
mdv_time.h	•		Structure and constant definitions, and function prototypes for the services associated with the file MDV_TIME.C.
mip_drvr.h	•	•	Network driver structure, command, and return code definitions.

The assembly source files are:

File Name	MIP/Px0	MIP/DPS	Description
segdata.asm	•	•	Segment data constants. This file is the same for both kits.
tchain.asm	•		Timer interrupt chaining code for the MIP/P20 and MIP/P50 network driver.

The build files are:

File Name			Description
makefile	•	•	Borland C make file. Type `make` at the DOS command prompt to execute the build instructions in this file.
pmip.cfg	•	•	Borland C compiler options for `makefile`. The last line of this file contains the Borland C include file path. If the Borland C include file are not contained in the directory `C:\BORLANDC\INCLUDE`, then this file must be changed.

The driver has been compiled with Borland C++, version 3.1. Modifications may be required for other compilers such as Microsoft C. Full descriptions of the contents of each of these files are included as comments within the files.

Example host application. The example host application is provided as ANSI C source files. These files are installed in the directory `EXAMPLES\HA`, or `\NB\EXA\HA`. For a complete description of this example host application, see the *LONWORKS Host Application Programmer's Guide.*

File Name	Description
applmsg.c	Contains the function definitions for handling network management and network variable messages.
applcmds.c	Contains the code for the user commands to the application.
ha.c	The main line code for this example.
hauif.c	Contains the functions for a primitive command-line user interface.
ioctl.c	Contains the functions required to establish communication with a DOS device driver. This file is required only when using the Microsoft C compiler (the Borland C standard library includes this function).
ldvintfc.c	Contains the lowest-level interface to the DOS network driver.
ni_callb.c	Contains functions to handle callbacks from the network driver.
ni_msg.c	Network interface initialization and LonTalk message send and receive functions. Generic functions that can be used by any host application.
applmsg.h	Contains prototype declarations for the functions defined in `applmsg.c`.
ha_comn.h	Contains common declarations used by all the files in the application.
hauif.h	Contains prototype declarations for the functions defined in `hauif.c`.
ldvintfc.h	Contains prototype declarations for the functions defined in `ldvintfc.c`.
ni_callb.h	Contains prototype declarations for the functions defined in `ni_callb.c`.
ni_msg.h	Defines network interface message structures.

ni_mgmt.h	Defines the subset of network management functions used by the sample host application.
makefile	Borland C make file. Type `make` at the DOS command prompt to execute the build instructions in this file.
msoft.mak	Makefile for Microsoft C compiler. Type `make /f msoft.mak` at the DOS command prompt to execute the build instructions in this file.
ha_v3.xif	Host-based node external interface file.
ha_test.nc	Neuron C source code for a LONWORKS device that may be bound to a host processor running the sample host application.
display.h	Display driver functions for the Gizmo 2 and Gizmo 3. This file is included by the Neuron C file `ha_test.nc`.
read.me	A text file that contains updates to the documentation for the example host application that have occurred since it was printed.

Read-Me Files. The files `..\MIP\README.TXT` and `..\MIP_DPS\README.TXT` are text files with updates to the *MIP User's Guide* that have occurred since it was printed. The file `..\HA\READ.ME` is a text file with updates to the *Host Application Programmer's Guide* that have occurred since it was printed. These example directories are the same as described above for the network driver source, and the sample host application.

Upgrading From Previous Releases

The following procedure describes the upgrade process for users of previous releases of the MIP.

1. Install the MIP software as described under *Software Installation Instructions* earlier in this chapter. The new MIP software will overwrite any previous releases.
2. The MIP image in existing network interfaces does not have to be upgraded unless you want to take advantage of the performance improvements of this release. To upgrade the MIP image in existing network interfaces, rebuild the MIP image as described in Chapter 3, and install this new image on your network interfaces.
3. Existing network interface hardware designs do not have to be modified.
4. Existing network drivers do not have to be modified, unless you want to take advantage of the performance improvements in the new network driver example.
5. Existing host applications do not have to be modified, the network driver protocol for this release is fully compatible with previous releases.

3

Creating a MIP Image

This chapter describes the process of building the MIP image that will be loaded onto a network interface.

Writing the MIP Application

The *MIP image* is the firmware loaded on a network interface that enables the network interface to act as a LonTalk communications processor. The MIP image is made up of the following components:

- ***Neuron Chip firmware***. The MIP image includes the standard Neuron Chip firmware that handles layers 1 - 5 and portions of layer 6 of the LonTalk protocol and controls the execution of the MIP firmware.
- ***MIP firmware***. The MIP firmware includes functions that extend the Neuron Chip firmware by hooking into the layer 5 and 6 processing functions of the Neuron Chip firmware to enable portions of layers 6 and all of layer 7 to be moved to the host processor.
- ***MIP application***. The MIP application is a Neuron C source program that specifies configuration options for the MIP and Neuron Chip firmware. The MIP application should only setup the hardware and call the MIP firmware. For the MIP/P50, the MIP application must also define an uplink interrupt request function. No other user code should be included. Once the MIP firmware is started, it never returns to the MIP application. The MIP application is different for the parallel MIPs (the MIP/P20 and MIP/P50) and the dual-ported MIP (the MIP/DPS). All three versions are described in the following sections.

The MIP image may be built to initially start-up in the configured or unconfigured state as described under the *Target Network Types and Firmware State Selection* section of the *LonBuilder User's Guide*, or in the Help information for the NodeBuilder Device Template Window Export tab (see the *NodeBuilder User's Guide*). Network interfaces that are shipped without host applications should be built to come up unconfigured to prevent network address conflicts when the network interface is first installed in a network.

Example MIP Application for the MIP/P20

The following is an example of a MIP application for the MIP/P20. This example is included with the LonBuilder MIP/P20 and MIP/P50 Developer's Kit in the `mip_ap20.nc` file. This file is installed in the examples directory (the default directory is `\LB\EXAMPLES\MIP` or `\LONWORKS\NB\EXA\MIP`).

```
#pragma micro_interface
#pragma idempotent_duplicate_off

// Select explicit addressing on or off with this pragma.
#pragma explicit_addressing_on

// Select NEURON CHIP (on) or host (off) NV selection.
#pragma netvar_processing_off

// When NEURON CHIP NV selection is specified (on), the
// set_netvar_count and num_addr_table_entries pragmas are
// used to control the amount of memory the NEURON CHIP uses
// for the network variable configuration and address tables.
#pragma set_netvar_count                13
#pragma num_addr_table_entries          14

#pragma receive_trans_count             4
#pragma set_id_string                   "MIP"
```

```
// Modify the following pragmas to set buffer sizes and counts.
#pragma app_buf_out_count                2
#pragma app_buf_out_priority_count       0
#pragma net_buf_out_priority_count       0
#pragma app_buf_in_count                 2
#pragma app_buf_out_size                 66
#pragma app_buf_in_size                  66
#pragma net_buf_out_size                 66
#pragma net_buf_in_size                  66

// Modify this declaration to suit your hardware interface.
IO_0 parallel slave p_bus;

extern void mip_p20_interface(unsigned long throttle);

when (reset) {
        // The argument <throttle> will yield about 375
        // microseconds of delay per count. A value of
        // one yields no throttle.
        mip_p20_interface(1);
}
```

Example MIP Application for the MIP/P50

The following is an example of a MIP application for the MIP/P50. This example is included with the LonBuilder MIP/P20 and MIP/P50 Developer's Kit in the `mip_ap50.nc` file. This file is installed in the examples directory (the default directory is `\LB\EXAMPLES\MIP` or `\LONWORKS\NB\EXA\MIP`).

```
#pragma micro_interface
#pragma idempotent_duplicate_off

// Select explicit addressing on or off with this pragma.
#pragma explicit_addressing_on

// Select NEURON CHIP (on) or host (off) NV selection.
#pragma netvar_processing_off

#pragma set_id_string                    "MIP"

// Modify the following pragmas to set buffer sizes and counts.
#pragma app_buf_out_count                3
#pragma app_buf_out_priority_count       3
#pragma app_buf_in_count                 7
#pragma app_buf_out_size                 66
#pragma app_buf_in_size                  66
#pragma net_buf_out_size                 66
#pragma net_buf_in_size                  66

// Modify this declaration to suit your hardware interface.
IO_0 parallel slave p_bus;

// The argument <irqp> is a pointer to the Interrupt Callback
function.
//
// The argument <throttle> will yield about 350 microseconds
// of delay per count. A value of one yields no throttle.
```

```
extern void mip_p50_interface(void (*irqp)(void),
    unsigned long throttle);

// Define the interrupt callback function. This function should
// restrict its activities to I/O oriented actions.
// In this example the interrupt consists of a write of 0x01
// to location 0xC000. If no interrupt callback functionality is
// required there needs to be at least a dummy function provided.
void irq_callback(void) {
    *(unsigned short *)(0xC000) = 0x01;
}

when (reset) {
    mip_p50_interface(irq_callback, 1UL);
}
```

Example MIP Application for the MIP/DPS

Following is an example of a MIP application for the MIP/DPS. This example is included with the MIP/DPS Developer's Kit in the `mip_dps.nc` file. This file is installed in the examples directory (the default directory is `\LB\EXAMPLES\MIP_DPS` or `\LONWORKS\NB\EXA\MIP_DPS`).

```
#pragma warnings_off
#pragma micro_interface
#pragma idempotent_duplicate_off

// Select explicit addressing on or off with this pragma.
#pragma explicit_addressing_on

// Select NEURON CHIP (on) or host (off) NV selection.
#pragma netvar_processing_off

// Set RAM test to OFF
#pragma ram_test_off

// Places buffers in DP ram.
#pragma all_bufs_offchip

#pragma set_netvar_count                30
#pragma num_addr_table_entries          15
#pragma receive_trans_count             16
#pragma set_id_string                   "DP_MIP"

#pragma app_buf_out_count               23
#pragma app_buf_out_priority_count      3
#pragma app_buf_in_count                23
#pragma app_buf_out_size                66
#pragma app_buf_in_size                 66
#pragma net_buf_out_size                66
#pragma net_buf_in_size                 66

// This allows pin IO_10 to drive an interrupt signal to the host,
// if needed.
IO_10 output bit irqp10 = 1;
// This allows pin IO_0 to be initialized as a message indicator.
IO_0 output oneshot invert clock (7) status_led = 1;
//
```

```
// Protoype for the MIP interface function. The <if_flush> argument
// will cause the MIP application to enter the INIT FLUSH state
// following reset. The <if_oba_ir> argument causes the host to be
// interrupted whenever a new output buffer is posted to the interface
// The <sema_base_page> argument is the 8 bit page address of the
// semaphore area.
//
extern void dpram_mip_interface(unsigned if_flush, unsigned
if_oba_ir,
 unsigned short sema_base_page);

when (reset) {
    // In this example the semaphore area is mapped to base 0x8000.
    dpram_mip_interface(TRUE, TRUE, 0x80);
}
```

Specifying MIP Pragmas

The pragmas specified in the MIP application select the network interface configuration options defined in the *LONWORKS Host Application Programmer's Guide*. The sample MIP applications contain default values for these pragmas. Edit the MIP application source file to change them.

`#pragma micro_interface`

This pragma indicates to the compiler that this node is a network interface. This pragma also ensures that the MIP image will start execution even if it is not configured. This pragma must be specified for MIP applications.

```
#pragma idempotent_duplicate_off
#pragma idempotent_duplicate_on
```

These pragmas control the idempotent request retry bit in the application buffer. This bit corresponds to the `addr_mode` bit of the `ExpMsgHdr` structure declared in `ni_msg.h` for incoming request messages only. These pragmas only apply to MIP applications running on version 6 of the Neuron Chip firmware or newer. One of these pragmas must be specified. The `idempotent_duplicate_on` pragma should be specified unless the host application depends on the pre-version 6 firmware behavior of the idempotent request retry bit (i.e., it was always set to 0 prior to version 6 for incoming request messages).

If `idempotent_duplicate_on` is specified, the idempotent request retry bit indicates a duplicate request for incoming request messages. The host application will only receive the duplicate request if the response to the original request contained data. When a duplicate request is received, the host application can return the original response, or can provide a new updated response. See *Idempotent Versus Non-Idempotent Requests* in Chapter 4 of the *Neuron C Programmer's Guide* for more information.

If `idempotent_duplicate_off` is specified, or if the Neuron Chip firmware used in the network interface is a version prior to version 6, the idempotent request retry bit is always 0.

```
#pragma explicit_addressing_on
#pragma explicit_addressing_off
```

These pragmas determine whether space is set aside in the application buffer for explicit addressing information. The value in the sample MIP applications is `explicit_addressing_on`, which adds 11 bytes of overhead per application buffer for the explicit addressing information. Explicit addressing allows a host application to bypass the network interface's address table, allowing the host application to send LonTalk messages to an unlimited number of nodes. See the *LONWORKS Host Application Programmer's Guide* for more information. The default value is `explicit_addressing_on`. The `explicit_addressing_on` pragma must be specified for MIP applications to be used with the LonMaker™ Installation Tool or the LonManager API.

```
#pragma netvar_processing_on
#pragma netvar_processing_off
```

These pragmas specify whether network variable selection is performed by the Neuron Chip firmware (*network interface selection*) or the host application (*host selection*). See the *LONWORKS Host Application Programmer's Guide* for a description of these options. When *network interface selection* (`netvar_processing_on`) is specified, the size of the network variable configuration table must be specified with the `set_netvar_count` pragma. The setting `netvar_processing_off` is the default. MIP applications must specify host selection (`netvar_processing_off`) to be used with the LonMaker Installation Tool or the LonManager API.

```
#pragma set_netvar_count nn
```

This pragma defines the size of the network variable configuration table when the `netvar_processing_on` (*network interface selection*) pragma is specified. The value of *nn* is the number of entries to be reserved for the table, and may be any value from 0 to 62. When the MIP/P20 is used on the Neuron 3120 Chip, the size of the network variable configuration table is limited by the amount of free memory in the Neuron 3120 Chip EEPROM. The number of entries can be traded off against the number of address table entries, but typical values for nodes configured for 1 and 2 domains are illustrated in table 4.1.

```
#pragma num_addr_table_entries nn
```

This pragma specifies the number of address table entries to reserve in the MIP image. See the *Neuron C Programmer's Guide* for more information on the use of this pragma. When the MIP/P20 is used on the Neuron 3120 Chip, the size of the address table is limited by the amount of free memory in the Neuron 3120 Chip EEPROM. The number of entries can be traded off against the number of network variable configuration table entries, but typical values for nodes configured for 1 and 2 domains are illustrated in table 4.1. The default size is 15 entries.

`#pragma one_domain`

This pragma limits the domain table to one entry. The default size is two entries. See the *Neuron C Programmer's Guide* for more information on the use of this pragma. When the MIP/P20 is used on a Neuron 3120 Chip, the number of entries can be traded off against the number of network variable configuration table entries and address table entries as shown in table 4.1.

`#pragma ram_test_off`

This pragma disables the standard RAM test that is performed by the Neuron Chip firmware during power-up initialization. This power-up test should normally be performed for network interfaces based on the MIP/P20 or MIP/P50, but may be disabled for network interfaces based on the MIP/DPS to prevent address conflicts with the host during power-up. Recommended usage is to leave the RAM test enabled for the MIP/P20 and MIP/P50; and disable the RAM test with `ram_test_off` for the MIP/DPS.

`#pragma all_bufs_offchip`

This pragma moves all application and network buffers to off-chip RAM. This pragma should be used with the MIP/DPS to ensure that the buffers are located in the dual-ported RAM. Recommended usage is to leave the buffers in their default locations for the MIP/P20 and MIP/P50; and move the buffers off-chip with `all_bufs_offchip` for the MIP/DPS.

`#pragma set_id_string "`*`string`*`"`

This pragma defines the program ID string contained in the MIP image. This string is sent in service pin messages, and is also sent in response to Query ID network management messages. The ID string should be changed in the MIP application source if the MIP image is only to be used with a single host application. If the MIP image will potentially be used with multiple host applications, the program ID can be left as a generic ID that is replaced by the host application when it starts running. The generic program IDs in the sample MIP applications are "MIP" for the MIP/P20 and MIP/P50; and "DP_MIP" for the MIP/DPS.

When the MIP/P20 is used on the Neuron 3120 Chip, the size of the domain table, address table, and network variable configuration table is limited by the amount of free memory in the Neuron 3120 Chip EEPROM. Typical values for 3120-based nodes configured for 1 and 2 domains are illustrated in table 4.1. This table does not apply to the Neuron 3120E1 and 3120E2 Chips, which have more available EEPROM. When modifying the buffer sizes and counts, see the link map generated by the LonBuilder or NodeBuilder linker to determine the available EEPROM and RAM resources.

Table 4.1 Table Size Trade-Offs for 3120-Based Network Interfaces

Domains	*Address Table Entries*	*NV Configuration Table Entries*
1	8	28
2	15	11

Additional pragmas are used to set Neuron Chip resources such as buffer counts and sizes, and receive transaction counts. The sample MIP applications for the MIP/P20 and MIP/P50 are set up to fit the buffers entirely within the Neuron Chip's on-chip RAM. The sample MIP application for the MIP/DPS is set up to fit the buffers within the default off-chip dual-ported RAM. The default sizes should be increased if large explicit messages will be used. The default counts should be increased if the host applications will cause too many messages to be lost due to unavailable buffers. Increased sizes or counts may require the use of off-chip RAM with the MIP/P50. For a complete descriptions of these pragmas, refer to Chapter 1 of the *Neuron C Programmer's Guide*.

The buffer size settings in the sample MIP applications assume explicit messaging and explicit addressing as defined in table 6.1 in the *Neuron C Programmer's Guide*. They are defined by pragmas and may be configured at run-time by writing to the application image. The *Neuron Chip Data Book* Appendix A describes the locations and encodings of these fields within the application image.

Declaring MIP I/O Objects

When using the MIP/P20 or MIP/P50, a parallel I/O object must be declared with the following statement:

```
IO_0 parallel slave p_bus;
      or
IO_0 parallel slave_b p_bus;
```

Select the first declaration for a slave A interface and the second declaration for a slave B interface. Slave A mode is suitable for parallel port interfaces; slave B mode requires fewer external components to interface to a memory bus. See the *Parallel I/O Interface to the Neuron Chip* engineering bulletin for a description of the operational characteristics of the parallel I/O functions including a description of the two slave modes and management of the token.

When using the MIP/DPS, an I/O object must be declared for the uplink interrupt, even if the I/O pin is not used. An optional I/O object may be declared for a message indicator output. The two I/O objects are declared with the following statements:

```
IO_10 output bit irqp10 = 1;
IO_0 output oneshot invert clock (7) status_led = 1;
```

Calling the MIP Function

When using the MIP/P20, the `mip_p20_interface()` function is called to invoke the MIP firmware. This function call never returns. The function prototype for this function is:

```
extern void mip_p20_interface(unsigned long throttle);
```

The `throttle` parameter specifies the throttle delay in units of approximately 375 µs at an input clock rate of 10 MHz. For example, a throttle parameter of 40 specifies approximately a 15 millisecond throttle with a 10 MHz input clock. The throttle delay scales with the input clock. A throttle of 1 specifies no throttle delay. See *Polled I/O* in Chapter 4 for more information on the throttle delay.

When using the MIP/P50, the `mip_p50_interface()` function is called to invoke the MIP firmware. This function call never returns. The function prototype for this function is:

```
extern void mip_p50_interface(void (*irqp)(void),
      unsigned long throttle);
```

The `irqp` parameter specifies the address of a user-supplied Neuron C function which is called when the MIP wishes to assert an uplink host interrupt.

The `mip_ap50.nc` example MIP application described earlier in this chapter includes an example function to write a 1 (one) to location 0xC000 to request an interrupt. External hardware can decode this write to generate an interrupt signal.

The `throttle` parameter specifies the throttle delay in units of approximately 350 µs at an input clock rate of 10 MHz. A throttle parameter of 1 may be specified when the MIP/P50 uplink interrupt is used. In this case, the host driver will normally keep the write token when the network is in the idle state. If the host has some downlink traffic to send, it can immediately use the token to write it to the network interface. If the network interface has some uplink traffic, it can interrupt the host to indicate that the write token should be passed down. This provides optimum latency in both uplink and downlink directions. If the uplink interrupt is not implemented, the token will need to be continuously passed between the host and the network interface so that each will have an opportunity to pass traffic to the other. This will increase latency because delays to wait for the token may be necessary.

See *Interrupt-Driven I/O* in Chapter 4 for more information on the uplink interrupt. If an uplink interrupt is not required, supply the address of an empty function as the `irqp` parameter.

When using the MIP/DPS, the `dpram_mip_interface` function is called to invoke the MIP firmware. This function call never returns. The function prototype for this function is:

```
extern void dpram_mip_interface(unsigned if_flush,
      unsigned if_oba_ir, unsigned short sema_base_page);
```

The `if_flush` parameter specifies whether the MIP application will enter the `FLUSH` state after every reset. If `if_flush` is `TRUE`, the MIP-based node will not be able to communicate on the network after a reset. The MIP application prevents communications by entering a `FLUSH` state. This state causes the MIP to ignore all incoming messages and prevents all outgoing messages, even service pin messages. This `FLUSH` state is provided to prevent any other network management tools from performing network management functions on the MIP-based node before the host has a chance to perform any of its own network management functions. This state is canceled with the `niFLUSH_CANCEL` command from the host. A network driver for a MIP-based network interface may automatically

enable network communications when the network interface is opened by sending the `niFLUSH_CANCEL` command when the driver is opened and when it receives an uplink message from the MIP application indicating that it has been reset. Alternatively, if the MIP application is built with `if_flush` set to `FALSE`, the network interface will not enter the `FLUSH` state after each reset, and network communications will be immediately enabled.

The `if_oba_ir` parameter specifies whether the MIP application will post an uplink interrupt when it frees an output buffer. If `if_oba_ir` is `FALSE`, the MIP application posts an uplink interrupt whenever it has uplink traffic for the host (incoming message, incoming response, completion event, or network interface command). If `if_oba_ir` is TRUE, the MIP application will, in addition, post an uplink interrupt when it has freed a downlink application buffer. This occurs shortly after the host has read a completion event, which allows the MIP to free the corresponding buffer.

The `sema_base_page` parameter specifies the 8-bit page address of the semaphore area in the dual-ported RAM device. For example, a semaphore base page parameter of 0x80 specifies that the semaphores start at location 0x8000 in memory. See the *Implementing a MIP/DPS Network Driver* in Chapter 5 for more information on the use of semaphores.

Building the MIP Image

The MIP image is built using the LonBuilder or NodeBuilder tools following the same procedure as any other node being built from Neuron C source code. This process is described in Chapters 6 and 7 of the *LonBuilder User's Guide*, and in Chapter 5 of the *NodeBuilder User's Guide*. The node build takes place as it would for any other node except that no binding can occur for network interfaces that have not been integrated with a host application.

Memory properties are defined for a MIP-based node as with any other node. The typical memory properties for each MIP are described in the following table:

Memory Type	*MIP/P20*	*MIP/P50*	*MIP/DPS*
ROM	0 (on a Neuron 3120 Chip)	128 pages typical; 66 pages minimum *	128 pages typical; 68 pages minimum
EEPROM	0	0	0
RAM	0 (on a Neuron 3120 Chip)	0 - 166 (required size must be large enough for the number of buffers declared) *	16 - 166 (required size must be large enough for the number of buffers declared)
I/O	0	1 page if memory-mapped uplink interrupt is used *	1 page (semaphore page; start address must match the `sema_base_page` parameter in the call to `dpram_mip_interface()`)
		* or 0 for a Neuron 3120E1 or 3120E2 Chip	

Once a network interface is integrated with a host application, the host node can be bound to connections as described under *Binding to a Host Node* in Chapter 3 of the *LONWORKS Host Application Programmer's Guide.*

Loading the MIP Image

Once the MIP image has been built, there are four methods to load the image into a network interface:

- ***Emulator download.*** If the network interface is initially prototyped on a LonBuilder Neuron Emulator, the MIP image can be loaded directly into the emulator memory using the LonBuilder Load command. When using the MIP/P20 or MIP/P50, the emulator will continuously watchdog reset if no host is present. To reliably load such a network interface, the host should be present and be exchanging tokens with the network interface. The MIP/DPS cannot be loaded on an emulator since the dual-ported memory cannot be emulated.
- ***Network download.*** If the MIP is to be loaded in read/write memory on the network interface such as the on-chip EEPROM, the MIP image can be loaded over the network using the LonBuilder or NodeBuilder Load command. When using the MIP/P20 or MIP/P50, the network interface will continuously watchdog reset if no host is present. To reliably load such a network interface, the host should be present and be exchanging tokens with the network interface.
- ***PROM export.*** When using any version of the MIP with a Neuron 3150 Chip, the MIP image can be exported to a ROM image using the LonBuilder or NodeBuilder Export command. The ROM image is loaded into a PROM using a PROM programmer.
- ***Neuron 3120 image export.*** When using the MIP/P20 with a Neuron 3120, 3120E1, or 3120E2 Chip, the MIP image must be exported to a Neuron 3120 image using the LonBuilder or NodeBuilder Export command. The Neuron 3120 image is loaded into a Neuron 3120, 3120E1, or 3120E2 Chip using a Neuron 3120 programmer such as the LONWORKS Neuron 3120 Programmer.

The MIP application may be exported to initially start-up in the configured or unconfigured state as described under the *Target Network Types and Firmware State Selection* section of the *LonBuilder User's Guide*, or in the Help information for the NodeBuilder Device Template Window Export tab (see Chapter 5 of the *NodeBuilder User's Guide*). Network interfaces that are shipped without host applications should be built to come up unconfigured to prevent network address conflicts when the network interface is first installed in a network.

Once reset, a MIP-based node will always start up online, and cannot start offline. Because of this, the LonBuilder Load command has the same effect as the LonBuilder Load/Start command, they both load and start the MIP-based node.

If the network interface will be loaded or configured over the network, the host must be able to sense when the network interface has been reset. See the discussion in Chapter 4, *Building a Network Interface*, for information on reset latch circuitry.

4

Building a Network Interface

This chapter describes the process of building a network interface. A detailed discussion of host interface considerations is included.

Building Network Interface Hardware

The network interface hardware is similar to any custom node containing a Neuron Chip, a transceiver, and memory. The hardware itself is built the same way as you build a custom node. Refer to the *LONWORKS Custom Node Development* engineering bulletin for additional information. Network interfaces based on the MIP/P20 or MIP/P50 may be based on LONWORKS control modules which conveniently integrate the Neuron Chip, transceiver, memory, and support circuitry. LONWORKS control modules cannot be used with the MIP/P50 when using uplink interrupts, and also cannot be used with the MIP/DPS.

The network interface external interface file (`.XIF` extension) is exported as with any custom node. The network interface external interface file does not contain network variable records. Host application network variables can be manually added to the external interface file. Refer to the *LonBuilder User's Guide* or the *NodeBuilder User's Guide* for information on exporting external interface files. See Appendix B of the *LONWORKS Host Application Programmer's Guide* for a description of how to modify the external interface file to add network variables and message tags.

Network interfaces based on the MIP/P20 or MIP/P50 may be tested on a LonBuilder Neuron Emulator, or the LTM-10 LonTalk Node prior to building custom hardware. Because of the requirement for dual-ported memory, the MIP/DPS cannot be tested on an emulator or an LTM-10 Node and must be run on a custom node.

There are up to four hardware interfaces between a host and network interface. These interfaces are:

- ***Host interface***. A data interface for transferring commands and data between the host and network interface.
- ***Uplink interrupt***. An optional interrupt interface so that the network interface can interrupt the host when an uplink message buffer is available.
- ***Reset latch***. An optional interface so that the host can be informed that a reset has occurred on the network interface. This interface is required for host-based nodes that will be reset, configured, or loaded over the network.
- ***Semaphores***. Hardware semaphores shared by the host and network interface. Used for resource contention for the dual-ported RAM by the MIP/DPS. Not used for the MIP/P20 or MIP/P50.

The following sections describe these four hardware interfaces.

Building the Host Interface

The host interface provides a communications path for transferring commands and data between the host and network interface. For the MIP/P20 and MIP/P50, the interface is an 11-bit parallel I/O interface. For the MIP/DPS, the interface is a dual-ported RAM with semaphores. Figures 1.1 through 1.3 illustrate the components of a network interface based on the MIP/P20, MIP/P50, and MIP/DPS.

The MIP/P20 and MIP/P50 host interface is implemented as described in the *Parallel I/O Interface to the Neuron Chip* engineering bulletin and Chapter 8 of the *Neuron C Reference Guide*. The host must have the ability to execute the token passing algorithm required by the parallel I/O interface. This algorithm is implemented by the sample DOS network driver described in Chapter 5.

The MIP/P20 and MIP/P50 can be used with either of the two parallel I/O modes: slave A and slave B. The slave A mode is used to communicate with a parallel port. In slave A mode, the master (the host) and slave (the Neuron Chip) communicate through eight data lines, plus chip select, read/write, and handshake control lines.

The slave B mode is used to communicate with a memory or I/O bus. It is logically similar in operation to the slave A mode, however, the handshaking process and the data bus control are specifically tailored for use in a bus environment. Multiple slaves may reside on the same bus, as may other non-slave devices such as memory or other I/O devices. Slave B mode requires the host to alternatively select the data register and control register of the network interface in order to monitor the handshake bit. See the *Neuron Chip Data Book* for a comparison of these two modes.

Figure 4.1 illustrates the parallel I/O modes as used by the MIP/P20 and MIP/P50.

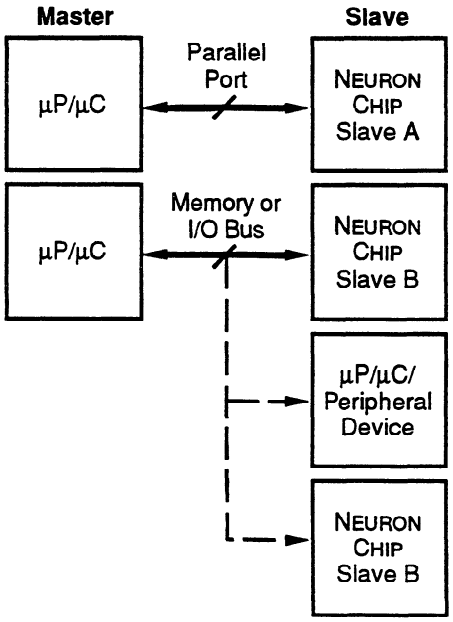


Figure 4.1 MIP/P20 and MIP/P50 Parallel I/O Modes

The host should not read the EOM at the parallel I/O interface level. Refer to the *Parallel I/O Interface to the Neuron Chip* engineering bulletin, which describes the purpose of the EOM step. The EOM is written in order to set up the HANDSHAKE signal for the next read operation.

Figure 4.2 illustrates the use of the slave A mode of the MIP/P20 or MIP/P50 for interfacing the Neuron Chip to a Motorola 68HC11 microcontroller. The 68HC11 is the master and the Neuron Chip is the slave connected to the B and C ports of the 68HC11. The interface circuitry is considerably more involved than the slave B mode interface, which is shown in figure 4.3. See the next section on *Handling Uplink Requests* for other options for generating an uplink interrupt.

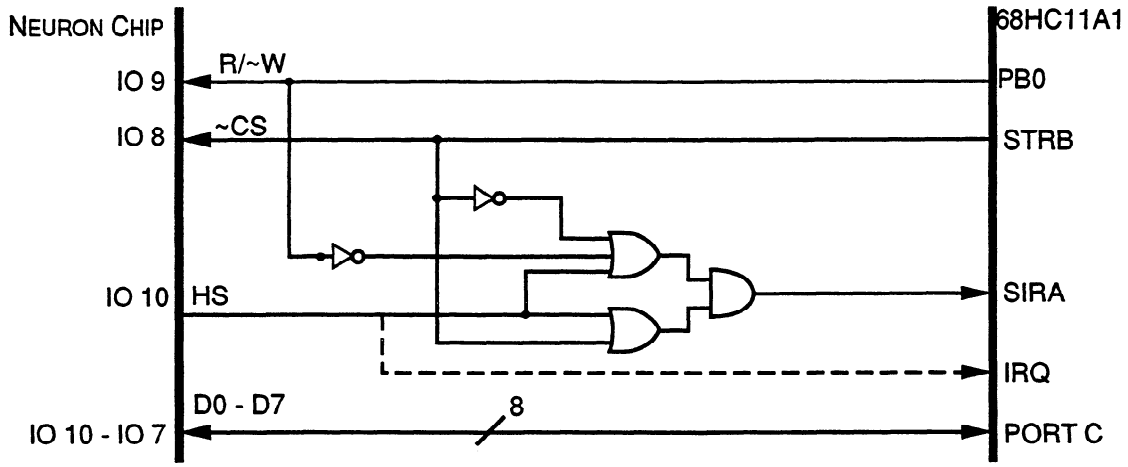


Figure 4.2 Sample Slave A Mode 68HC11 Interface for the MIP/P20 or MIP/P50

Figure 4.3 illustrates the use of the slave B mode of the MIP/P20 or MIP/P50 for interfacing the Neuron Chip to a Motorola 68HC11 microcontroller. The 68HC11 is the master and the Neuron Chip is the slave residing in the 68HC11's address space. No interface circuitry is needed aside from some address decoding logic that allows the 68HC11 to access the Neuron Chip by using specific addresses (one address for the data register and one for the control register).

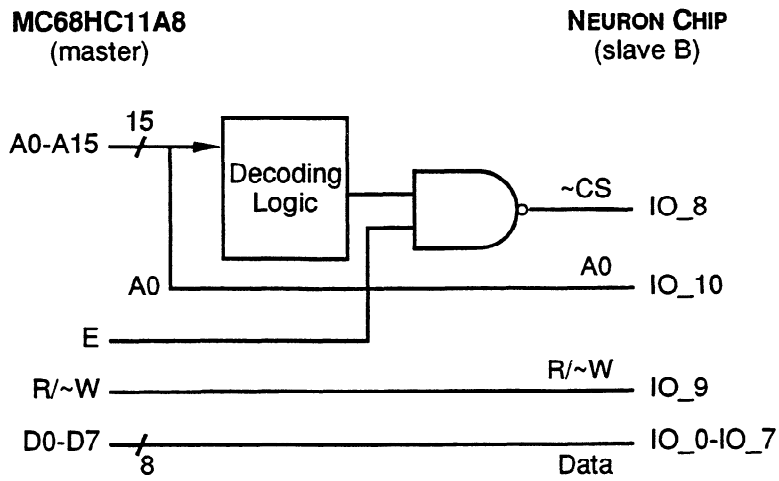


Figure 4.3 Sample Slave B Mode 68HC11 Interface for the MIP/P20 and MIP/P50

If the host to Neuron Chip connection involves any type of external cabling, the data and control signals must be adequately conditioned, in particular the ~CS (chip select) signal. Crosstalk and poor rise times on this signal will interfere with the operation of the parallel I/O protocol.

MIP/DPS Host Interface

For the MIP/DPS, any dual-ported RAM can be used that supports the memory access requirements of the Neuron Chip and implements semaphores as described later in this chapter under *Implementing Semaphores*. Two dual ported RAMs that meet these requirements are the IDT 71342 and the Cypress 78144. The memory must be contiguous and start on a 256-byte page boundary. The starting address must be after the end of the ROM memory used for the MIP image, however the ROM and the dual-ported RAM do not have to be contiguous. The starting address and size of the RAM must be specified as external RAM in the LonBuilder Hardware Properties window or the NodeBuilder device template. Appendix B provides a sample schematic for a network interface based on the MIP/DPS.

Handling Uplink Requests

There are two methods that the host can use to initiate the transfer of a message uplink from the network interface to the host: polled I/O and interrupt-driven I/O. Polled I/O is simpler, but requires that the host periodically check for the availability of an uplink message from the network interface. Interrupt driven I/O is more complex, but provides for asynchronous notification of the host when a message is ready to be transferred uplink. Either method can be used with any of the MIPs, but use of interrupt-driven I/O is more complex with the MIP/P20 (the MIP/P50 and MIP/DPS directly support uplink interrupts).

Once an uplink transfer is started with the MIP/P20 or MIP/P50, each byte of the message may also be transferred using polled I/O or interrupt-driven I/O. In addition, for the MIP/P50 using an uplink interrupt, a DMA (direct memory

access) controller can be used to finish transferring the message. Polled I/O should be used once the transfer has started if the host's interrupt service processing time is long compared to the interrupt period. For example, the interrupt period is 2.4µs for the MIP/P20 or MIP/P50 running at 10 MHz.

The MIP/DPS first writes an uplink message into one of the available uplink buffers in the dual-port RAM and then generates an uplink interrupt. Therefore no polled I/O is required for the MIP/DPS for each byte of an uplink packet.

Polled I/O

For polled I/O, the host network driver software periodically polls the network interface to determine if a message is available for an uplink transfer. The method of polling depends on which MIP is being used, and the interface mode of the MIP:

- ***MIP/P20, slave A mode.*** The handshake pin indicates that the MIP is ready to pass back the parallel interface token. The token may be part of a message packet, or may be a null token.
- ***MIP/P20, slave B mode.*** The handshake bit of the status port indicates that the MIP is ready to pass back the parallel interface token. The token may be part of a message packet, or may be a null token.
- ***MIP/P50, slave A or B mode.*** When the Neuron Chip running the MIP/P50 firmware does not own the write token, it will call the user-supplied uplink interrupt request routine to indicate that a message is available for an uplink transfer. This routine typically writes to a memory-mapped location, and external hardware decodes and latches this memory write. The address of the user-supplied uplink interrupt request function is specified in the call to the MIP function as described under *Calling the MIP Function* in Chapter 3. Alternatively, the handshake pin in slave A mode or the handshake bit in slave B mode can be used as with the MIP/P20. However, the handshake signal is asserted once per byte transferred. The memory-mapped I/O port is written once per uplink transfer request.
- ***MIP/DPS.*** The MIP/DPS firmware changes a state variable in the control interface structure in shared memory to indicate that a message is available for an uplink transfer. The control interface structure is described under *Control Interface Structure* in Chapter 5.

Interrupt-Driven I/O

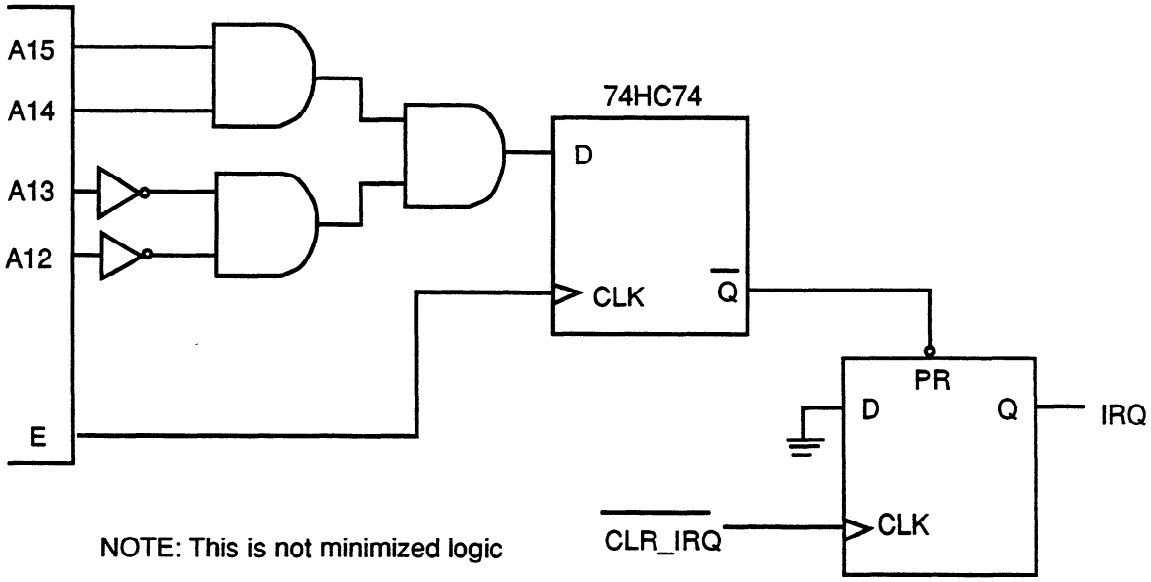
For interrupt-driven I/O, an interrupt signal generated by the network interface informs the host that a message is available for an uplink transfer from the network interface to the host. The source of the interrupt depends on which MIP is being used:

- ***MIP/P20.*** Slave A mode should be used if an uplink interrupt is desired with the MIP/P20 since interrupts are difficult to generate with slave B mode. In slave A mode, the handshake pin interrupts the host every time the MIP is ready to pass back the parallel interface token. The token may be part of a LonTalk message packet from the network, a local network interface command, or a null token. The handshake pin therefore acts as both an uplink interrupt and a null token interrupt. However, the handshake signal is asserted once per byte

transferred. The host must transfer the packet to determine if it is a message or a null token.

- ***MIP/P50***. The MIP/P50 firmware calls the user-supplied uplink interrupt request function to generate an uplink interrupt. This routine typically writes to a memory-mapped location (which requires a Neuron 3150 Chip), and external hardware decodes and latches this memory write. This method is illustrated by the example MIP application in Chapter 3. The address of the user-supplied uplink interrupt request function is specified in the call to the MIP function as described under *Calling the MIP Function* in Chapter 3. The uplink interrupt signals an uplink LonTalk message from the network or a local network interface command. Either slave A or slave B mode may be used. In this model the host normally has the write token. Downlink transfers occur as needed. Uplink transfers occur via two interrupts from the network interface hardware. The first interrupt is a signal that the network interface does not own the token, and needs to send data uplink. The driver then writes downlink, giving up the token. The second interrupt is a signal that the network interface is starting an uplink transfer. In this case the driver reads the uplink data from the network interface. Both interrupts appear on a single interrupt request line since they are generated in the same manner by the network interface hardware. The distinction between them is made by evaluating the token state of the driver. No polling occurs in this model; it is completely interrupt-driven in the uplink transfer case.

  The following figure describes one method of generating the uplink interrupt using the example interrupt request function from Chapter 3. The address that is decoded is 0xC000. The upper four bits only of the address (1100) are decoded, therefore nothing else should be in the memory map from 0xC000 to 0xCFFF. As described in the *LTM-10 User's Guide*, similar hardware is implemented on the LTM-10 module, so that an ~IRQ output is available from the module.



IRQ signals the host system that an uplink message is waiting. The host's interrupt subroutine will acknowledge the interrupt by briefly asserting ~CLR_IRQ.

Figure 4.4 MIP/P50 Uplink Interrupt Generation for Interrupt-driven I/O

After the command byte is received with the uplink interrupt, if the command specifies a message, the length byte is read using polled I/O. Now that the length of the rest of the message is known, the transfer can be completed by polled I/O, interrupt I/O, or a DMA controller. When DMA is used, a separate end-of-transfer interrupt is generated by the DMA controller to signal to the driver that the transfer has been completed. Depending upon the host this driver is being used on, this may or may not be a faster way than using interrupt-driven I/O, as setting up the DMA hardware usually entails a fair amount of overhead. Generally, larger-sized messages will achieve a greater performance improvement than small messages.

Figure 4.5 provides a block diagram of using interrupt I/O and a DMA channel with the MIP/P50.

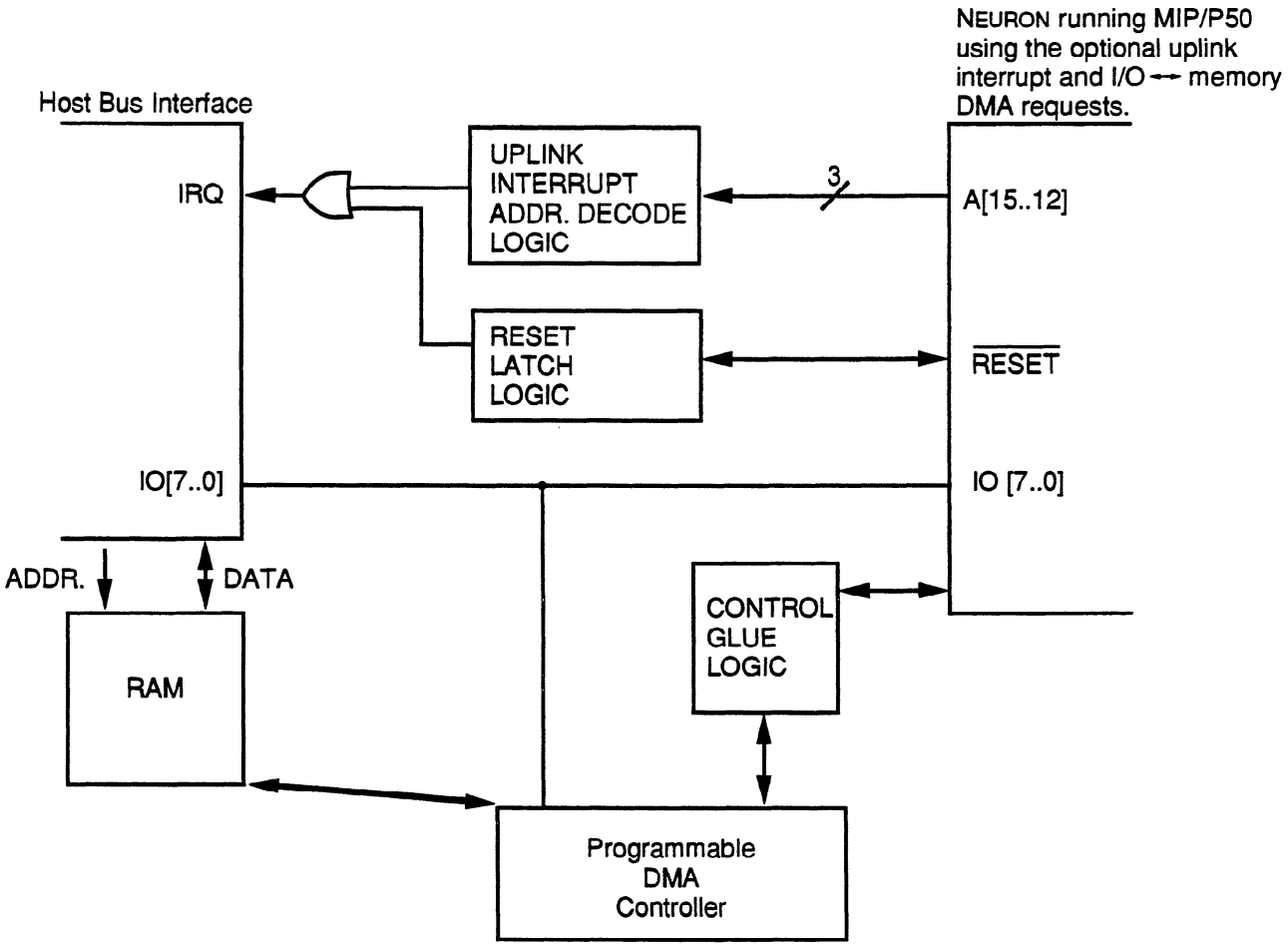


Figure 4.5 MIP/P50 Interrupt I/O with DMA Functional Block Diagram

- ***MIP/DPS***. The MIP/DPS firmware asserts IO_10 low for 10 µs to generate an uplink interrupt. The IO_10 pin can be used directly to generate an interrupt on the host. The IO_10 pin should be latched if a latching interrupt is required by the host hardware. The uplink interrupt signals an uplink LonTalk message from the network or a local network interface command. The example MIP/DPS schematic in Appendix B includes an interrupt latch.

Implementing a Reset Latch

The host must respond to any resets that occur on the network interface. For example, a network interface may be reset during the loading process to start execution with a new network image or application image. If the host ignores the reset, the load may fail and the host application will not operate.

For the MIP/P20 and the MIP/P50, the host must reinitiate the parallel interface token passing protocol after a reset occurs since the master is assumed to have the token after a Neuron Chip reset.

For the MIP/DPS, the host must terminate all writes to the shared memory and release any semaphores since the MIP/DPS firmware initializes the shared memory after a reset and rebuilds the control structures (see *Implementing Semaphores* below). The host must also terminate any reads after a reset since the data may no longer be valid.

Once the MIP/DPS initialization has completed, the `control_iface.out` and `control_iface.out_p` structures will be filled in with valid (non-null) pointers. The host will be informed that the reset initialization has been completed by an uplink interrupt for an uplink `niRESET` command. The `niRESET` command will be posted at `control_iface.command_in`. Once the `niRESET` command has been transferred uplink, the host may proceed with normal operation of the network interface. The MIP/DPS will be in the FLUSH state (no network messages allowed, in or out) until canceled with the `niFLUSH_CANCEL` command from the host.

For the MIP/P20 and MIP/P50, the reset latch can either be polled by the host, or can be used to generate an interrupt to the host. For the MIP/DPS, a reset interrupt should be used to ensure that the host does not write to the dual-ported RAM while it is being initialized by the MIP/DPS firmware. If a reset interrupt is used, the reset signal may be logically OR'd with the uplink interrupt to generate a single host interrupt, as long as the reset latch is also made available to the host so that the host can determine the source of the interrupt. If both the uplink and reset bits are set, then the reset sequence is initiated nonetheless.

A reset latch is implemented by latching the reset output of the network interface Neuron Chip. An example schematic for a reset latch is shown in figure 4.6. The example MIP/DPS schematic in Appendix B also includes a reset latch.

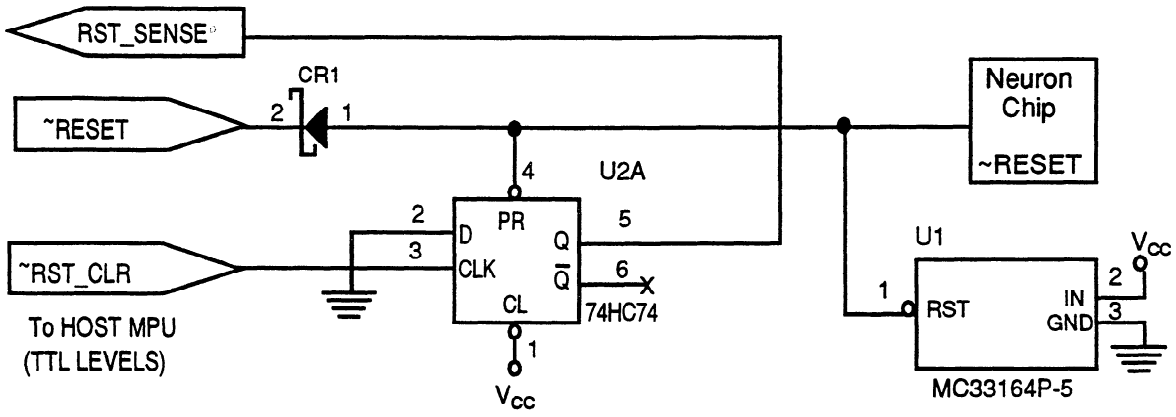


Figure 4.6 Sample Reset Latch Schematic

Implementing Semaphores

The MIP/DPS uses hardware semaphores to control access to the shared memory area by either the network interface processor or the host processor (the MIP/P20 and MIP/P50 do not use semaphores). Only *exact* address conflicts need to be avoided; either processor has free access to the shared memory area provided they do not access the same byte location within a "very small period" of each other. This period is guaranteed by the semaphores and ownership rules concerning certain elements in the shared memory.

There are eight single bit semaphores mapped into eight consecutive addresses. The starting address is passed as a function call argument from the MIP application (MIP_DPS.NC) to the MIP function (that is in the MIP_DPS.LIB library). The MIP function requires that the semaphores start on a 256 byte page boundary.

The semaphores are implemented in hardware within the dual-ported RAM to ensure that only one processor can own a semaphore at a time. To access (own) a semaphore the processor writes a '0' value and then reads back the same location. If the value read is a '0' value then that processor owns the semaphore. If the value read is a '1' then that processor does not own the semaphore and the process of trying to own it may be repeated. The semaphore hardware ensures that if both processors write a '0' value at the same time then only one of the processors will read back a '0' value.

The semaphore must eventually be released by the processor that owns it. This is accomplished by writing a '1' value to that semaphore. When the network interface Neuron Chip gets a semaphore, it will own it for 17.4 µs with a 10 MHz input clock (the time scales with the input clock).

The MIP/DPS will not operate properly unless the semaphores in the dual-ported RAM are initially freed by the host for use by the MIP/DPS. When a network interface is powered up, it is likely that all of the required semaphores will not be free due to the power up state of the semaphores. The MIP/DPS will wait on these semaphores, and possibly reset due to a watchdog timeout, until the semaphores are freed by the host side of the dual-ported RAM.

5

Creating a Network Driver

This chapter describes the process of building a network driver for a host that is to be connected to a MIP-based network interface.

Implementing a Network Driver

The network driver provides a hardware-independent interface between the host application and the network interface. The LonTalk network driver protocol defines a standard calling convention for network driver functions. By using network drivers with consistent calling conventions, host applications can be transparently moved between different network interfaces. For example, using a standard MS-DOS network driver for the MIP allows applications, such as those based on the LonManager API for DOS or Windows, to be debugged using the network driver for the LonBuilder Development Station, then later be used with the network driver for the SLTA/2 Serial LonTalk Adapter, PCLTA PC LonTalk Adapter or a network driver for a custom network interface based on any of the MIPs. You can do all of this without modifying the host application.

Figure 5.1 illustrates how the network driver fits into the host application architecture.

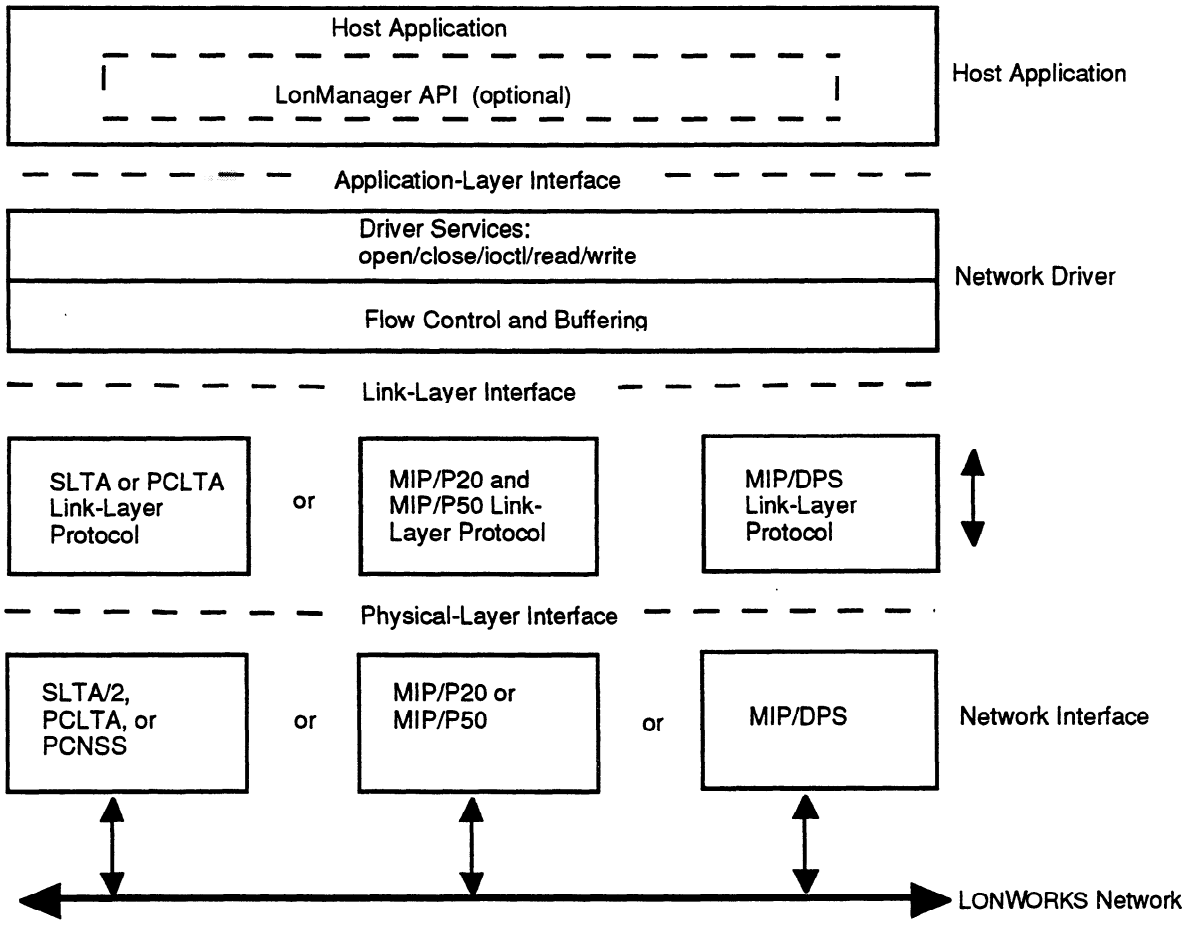


Figure 5.1 Host Application Architecture

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. The DOS network driver function calls are defined in Chapter 4 of the *LONWORKS Host Application Programmer's Guide*.

The network driver protocol defines the interface between the host application and the network driver. LonTalk packets are transferred between the host application and the network driver using application buffers. The network interface protocol defines the interface between the network driver and the network interface. LonTalk packets are transferred between the network driver and the network interface using MIP link-layer buffers. The network driver must translate between application buffers and link-layer buffers. Figure 5.2 illustrates the application and link-layer buffer formats. The contents of the buffers are identical, with the exception of the application and link-layer headers, as well as the EOM byte at the end of the MIP/P20 and MIP/P50 buffers. The network driver does not have to modify the length or command/queue bytes, it only has to modify their ordering. See the *LONWORKS Host Application Programmer's Guide* for a detailed description of the buffer contents and network interface commands.

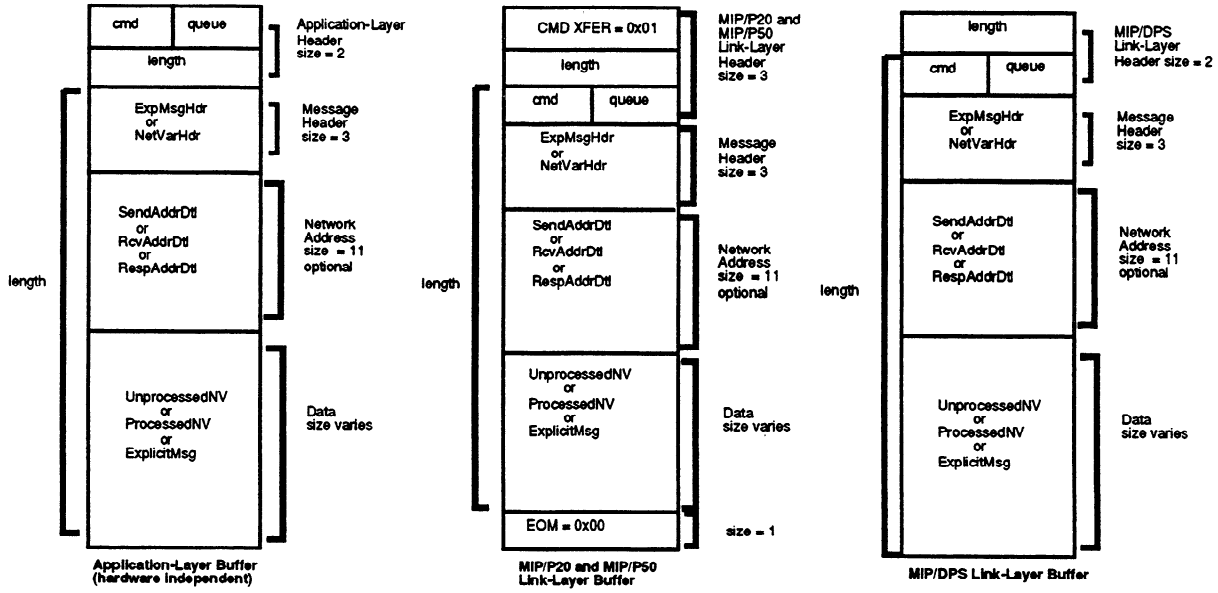


Figure 5.2 Application and Link-Layer Buffer Formats

The network interface protocol for network interfaces based on the MIP/P20 and MIP/P50 is based on the Neuron Chip parallel I/O protocol. This protocol is defined in the *Parallel I/O Interface to the Neuron Chip* engineering bulletin. The network interface protocol for the MIP/DPS is built on a shared memory interface with semaphores.

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

```
typedef int LNI;
LDVCode error = ldv_open(const char *device_name, LNI *pHandle);
```

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 of the *LONWORKS Host Application Programmer's Guide*. If a buffer is not available, return the `LDV_NO_MSG_AVAIL` error code. If a buffer is available, translate the buffer from the MIP link layer buffer format to the application layer buffer format, and return the buffer in the `*msg_p` structure. The uplink buffer transfer is described later in this chapter.

```
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 of the *LONWORKS Host Application Programmer's Guide*. For the MIP/P20 and MIP/P50, if the application buffer is a `niCOMM` or `niNETMGMT` command, first request an output buffer as described under *Downlink Buffer Transfer* for the MIP/P20 and MIP/P50 later in this chapter. If a buffer is not available, return the `LDV_NO_BUFF_AVAIL` error code. If a buffer is available, translate the the application layer buffer format to the MIP link layer buffer format and transfer the link layer buffer to the network interface. The downlink buffer transfer is described later in this chapter.

```
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.

Example Network Driver

The MIP is delivered with source code for an example network driver for DOS. This example driver for the MIP/P20 and MIP/P50 can be used as a starting point for creating a working network driver. The example source code is modular, and changes for the hardware interface should be localized to the files MIP_PIO.C and MIP_TYPS.H. The example driver for the MIP/DPS is a complete working example for network interface hardware meeting the following specifications:

Dual ported RAM size: 4 Kbytes

Dual ported RAM Neuron Chip address: 0xA000 - 0xAFFF

Dual ported RAM host base address: 0xC000:0x0000 to 0xDE00:0x0000

Control interface structure offset from base address: 0x0FE0

Control interface structure Neuron Chip address: 0xAFE0

See the comments in the source code for the network drivers for an explanation of how the network drivers work. These drivers are templates for LONWORKS standard network drivers and are compatible with the LonMaker Installation Tool and the LonManager APIs for DOS and Windows.

Implementing a MIP/P20 or MIP/P50 Network Driver

All transfers between the network driver and the network interface are either NULL token transfers, data transfers, RESYNC transfers, or ACK_RESYNC transfers as described in the *Parallel I/O Interface to the Neuron Chip* engineering bulletin. Data transfers start with the link-layer header sequence described in figure 5.2. The link layer header consists of a CMD_XFER byte, hex 01, followed by a length byte, followed by a network interface command byte. The data transfer is terminated with an EOM (end of message byte), which may be any value, but is usually 0. The EOM byte is never read, but is necessary to put the handshake line in the correct state to imply passing the write token to the other side of the parallel interface.

The length byte describes the length of the command field plus the length of the data field. This value will always be at least 1.

The network driver receives application buffers from the host application, translates them to link-layer buffers, and passes the link-layer buffers to the network interface using the parallel I/O protocol. If the application buffer is a `niCOMM` or `niNETMGMT` command, the network interface must first request an output buffer before sending the link-layer buffer. The network driver may hold the buffers in an output queue until the network interface is ready to receive them. The network driver takes the network interface through three states to request a buffer and send the link layer buffer. Figure 5.3 illustrates the downlink state transition diagram.

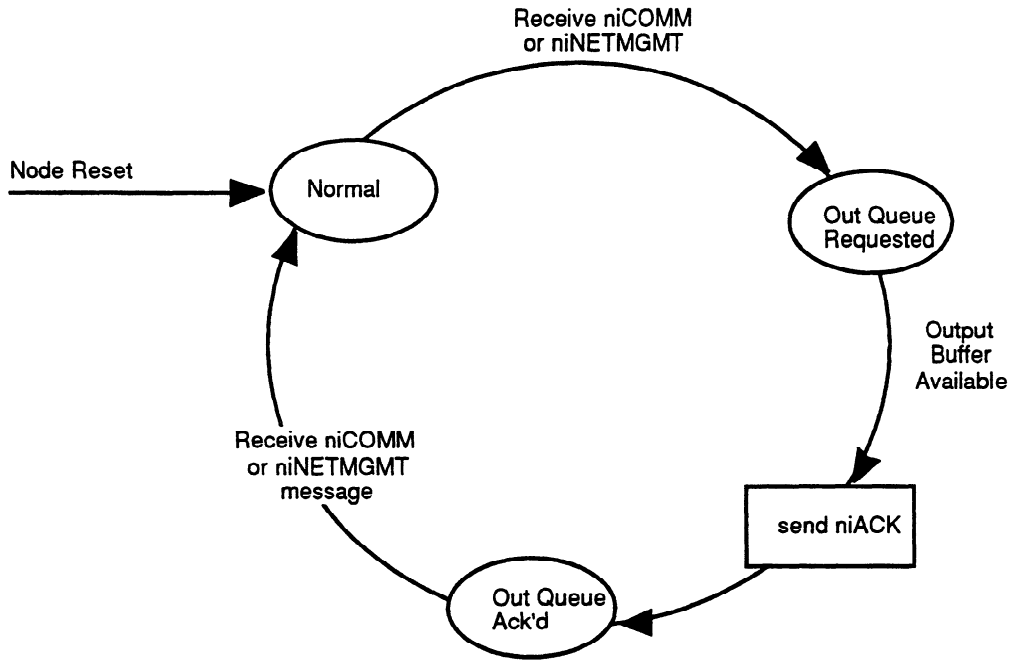


Figure 5.3 Network Interface Downlink State Transition Diagram

Following is the sequence of events for transferring an `niCOMM` or `niNETMGMT` command downlink to the network interface:

1. The network interface is initially in the *Normal* state.

2. The network driver requests an output buffer by sending a link-layer header with a `niCOMM` or `niNETMGMT` command and the appropriate queue value (`niTQ`, `niTQ_P`, `niNTQ`, or `niNTQ_P`). The length byte must be one (1) for this header. [**Note:** The driver sends the complete link_layer buffer once the network interface is in the *Output Queue Ack'd* state.] This puts the network interface in the *Output Queue Requested* state.

3. If an output buffer is not available, the network interface responds with a null token or another uplink transfer, which is not an `niACK`. The network driver cannot send a downlink LonTalk message, as long as the network interface remains in the *Output Queue Requested* state. The network driver can only send downlink transfers which are local commands. Uplink transfers may also occur.

4. When an output buffer is available, the network interface responds with an acknowledgement (command code niACK). The network interface is now in the *Output Queue Ack'd* state. While in this state, the network driver can only transfer downlink LonTalk messages, Uplink Source Quench commands (niPUPXOFF), Uplink Source Resume commands (niPUPXON), or Reset commands (niRESET) since the network interface is waiting for a message in this state. All other network interface commands sent downlink will be ignored, and will return the network interface to the *Normal* state.

5. Upon receiving the niACK acknowledgement, the host transfers the link-layer buffer to the network interface. This returns the network interface to the *Normal* state.

The network driver must preserve the continuity of the *type* of buffer request and the *type* of message sent downlink. For example, if the network driver sends the niCOMM+niTQ_P command requesting a priority output buffer, and follows this with a message transfer with the non-priority niCOMM+niTQ command, the network interface will incorrectly store the message in a priority output buffer, the type originally requested.

The network driver should not send another downlink buffer request just because it did not get an niACK uplink. If there still is no output buffer allocated from the previous buffer request, so the buffer request is still posted in the MIP, this action will be benign - the request simply overwrites the previous one. If the host was owning the token, which it would just before sending a second downlink buffer request, and the MIP has just allocated the output buffer, the MIP will be in the buffer allocated state, even though it has not sent the uplink niACK yet. A subsequent downlink niCOMM or niNETMGMT will be assumed to be a downlink message because the buffer has been allocated. This will be an invalid message since the rest of the message did not exist - it was just a buffer request.

Therefore the network driver should not re-request output buffers. The first request cannot get lost. When the niACK comes uplink, the network driver should send the downlink message. When in the *Output Queue Ack'd* state, the network driver may send down a NI_NoQueueCmd command, but it should not send another niCOMM or niNETMGMT request.

Uplink link-layer buffers may be incoming LonTalk messages, output buffer request acknowledgements, completion events, local commands, or null tokens. The network driver translates the link-layer buffers to application buffer format and stores the buffers in a queue until the host application is ready to read them. The network driver can use uplink flow control to stop uplink transfers when no network driver input buffers are available. When no network driver input buffers are available, the network driver sends the Uplink Source Quench (`niPUPXOFF`) command to the network interface. This prevents the network interface from sending any LonTalk messages uplink. When the network driver senses that network driver input buffers are available, it sends the Uplink Source Resume (`niPUPXON`) command to the network interface in order to resume uplink transfers. Figure 5.4 illustrates the uplink state transition diagram.

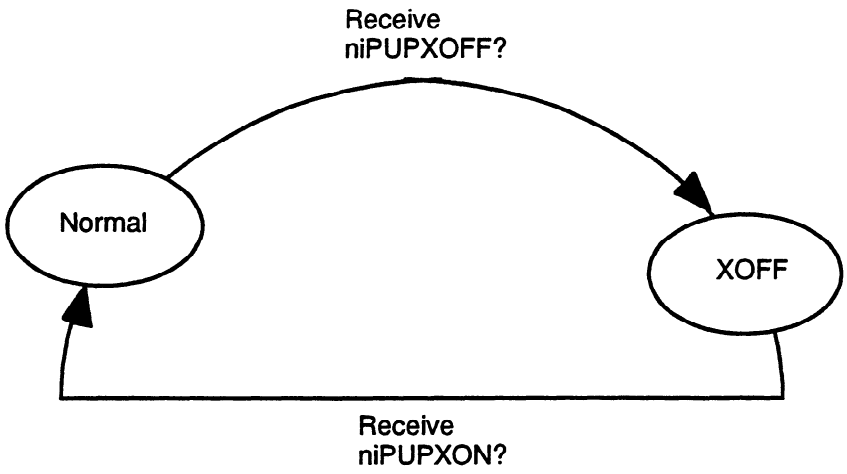


Figure 5.4 Network Interface Uplink State Transition Diagram

Figure 5.5 illustrates the structure of the example network driver for DOS included with the MIP/P20 and MIP/P50.

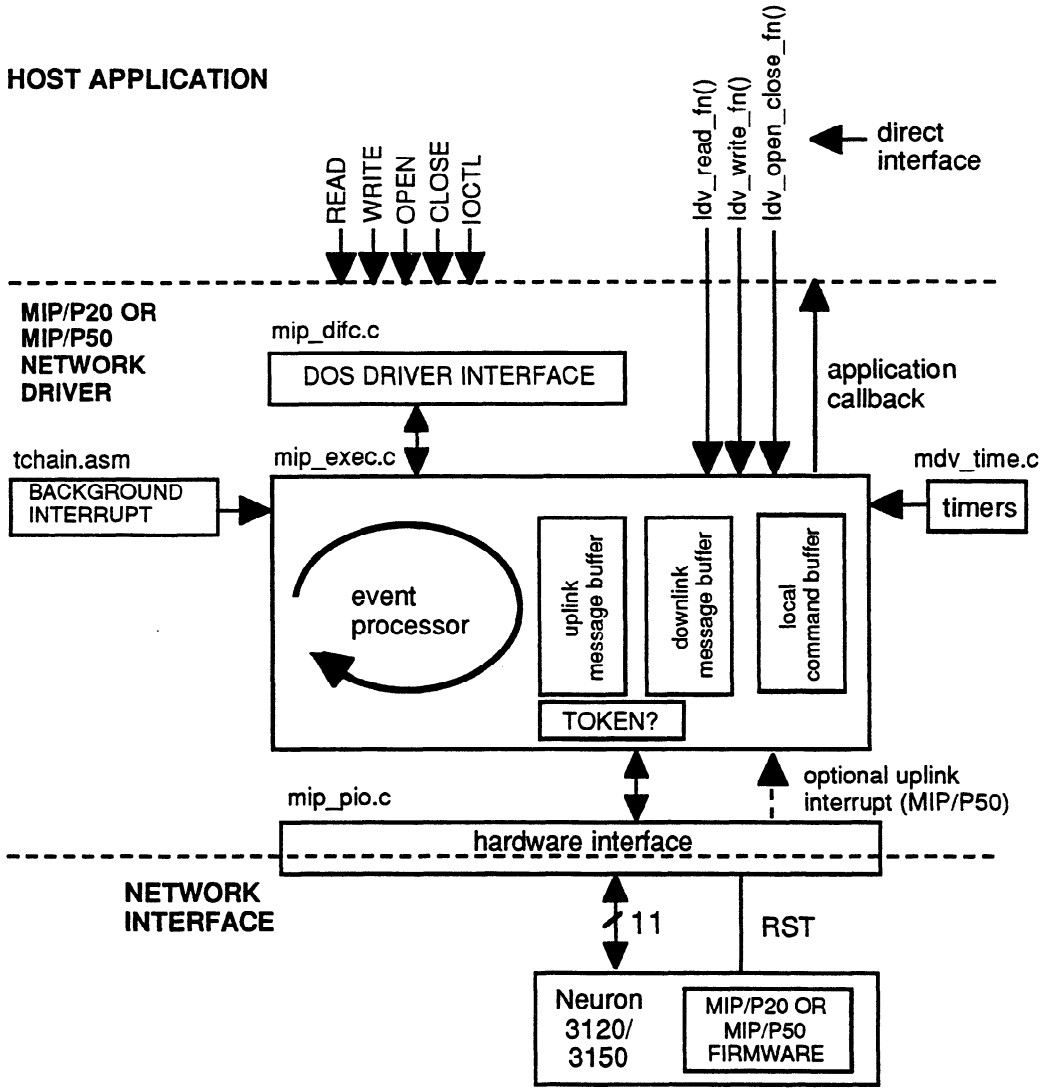


Figure 5.5 MIP/P20 and MIP/P50 Network Driver Block Diagram

The sample DOS network driver supplied with the MIP/P20 and MIP/P50 uses the DOS system timer tick interrupt (vector 0x1C) to perform background processing of the parallel network interface. This interrupt normally occurs every 55 ms. The driver hooks into this interrupt vector and executes driver code whenever the `LON(n)` device is opened. Flags internal to the driver prevent the interrupt code thread from interfering with the normal application foreground execution of functions within the driver.

Within the driver there is a function, `mip_isr()`, which services the parallel interface to the network interface. This function may be executed from within the tick interrupt, or from the beginning of the read function call to the device, or from the end of the write function call to the device. This service involves passing the write token to the network interface, in the form of either a message to the network interface if there is a message buffered, or as a NULL token if there is no message buffered.

Next, unless the `mip_isr()` function has been executed from a write function call to the device, the driver will wait up to four milliseconds for a pending transfer of the token from the Neuron Chip by checking the handshake signal. This period gives the network interface some time to process the data it has read, and to initiate a write operation. If there is a transfer pending, the driver reads the network interface. This yields either a message transfer or a NULL token transfer. The driver now owns the write token, which is the *Normal* state for the driver. Next, if there are buffered input messages, and if this function has been executed from the tick interrupt, the application callback function is executed.

When the `mip_isr()` function is executed from either the read or write case, this service repeats itself as long as there are incoming messages to be processed by the driver.

This provides a code path which can process more than a single input message every 55 milliseconds. The tick interrupt service is limited to prevent the PC from becoming interrupt bound.

The example network driver is fully buffered for both outgoing and incoming messaging. Read and write functions work with circular buffers within the driver. The host interface service handles the other ends of these buffer queues.

The example network driver only supports a single set of output buffers. An elaboration on this design could implement a set of priority output buffers. The write function could determine into which of the two buffer sets to place messages, and the driver service function could service the priority buffers first.

MIP/P20 and MIP/P50 Processing

Figures 5.6 through 5.8 are flow charts illustrating the processing done by the MIP/P20 and MIP/P50 firmware. Figure 5.6 shows the MIP processing when the host owns the token. Transfers downlink from the host to the MIP only occur in this figure. The two boxes labeled *Execute "background" tasks* are expanded in figure 5.7. Figure 5.8 shows the MIP processing when the MIP owns the token. Transfers uplink from the MIP to the host only occur in this figure (with the exception of the uplink `niACK` command shown if figure 5.7).

The MIP firmware performs a synchronization step prior to starting the processing shown in the following figures. The Neuron Chip running the MIP firmware will watchdog reset until the synchronization is complete. After synchronizing, the MIP firmware enters the Flush state. This state affects the lower layers of the network interface protocol only, blocking incoming and outgoing network traffic. It does not affect other MIP processing.

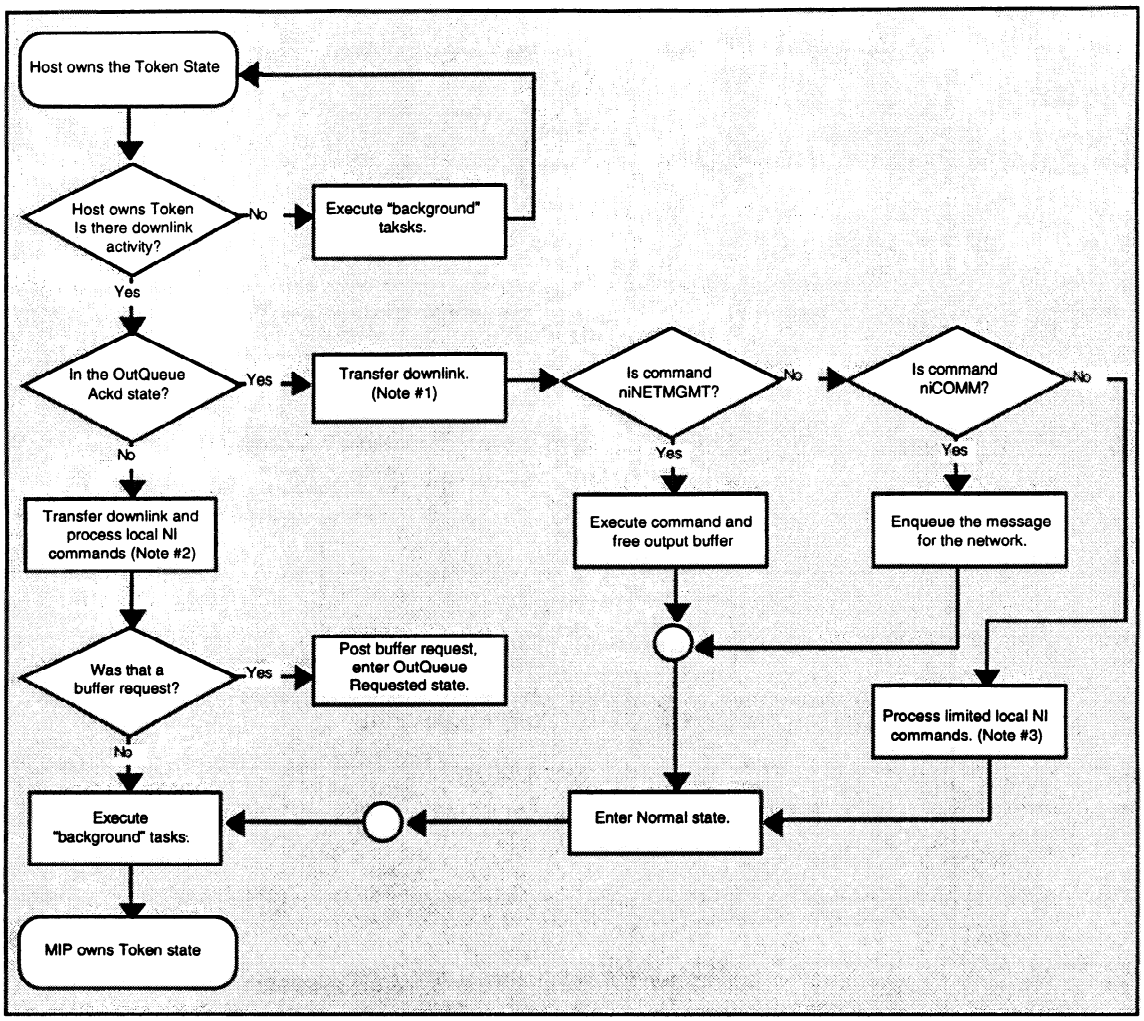


Figure 5.6 Host Owns Token Processing

Note 1. If the command is `niNETMGMT` or `niCOMM`, and the actual downlink message is a buffer request rather than a true message, an incomplete message will be processed instead of treating it as if it were another buffer request.

Note 2. If the command is `niNETMGMT` or `niCOMM`, and the actual downlink message is a true message with a transfer length greater than 1 rather than a buffer request, then this transfer must be flushed and the entire downlink transfer is lost.

Note 3. The only local commands that can be executed in this state are Uplink Source Quench commands (`niPUPXON`), Uplink Source Resume commands (`niPUPXOFF`), and Reset commands (`niRESET`).

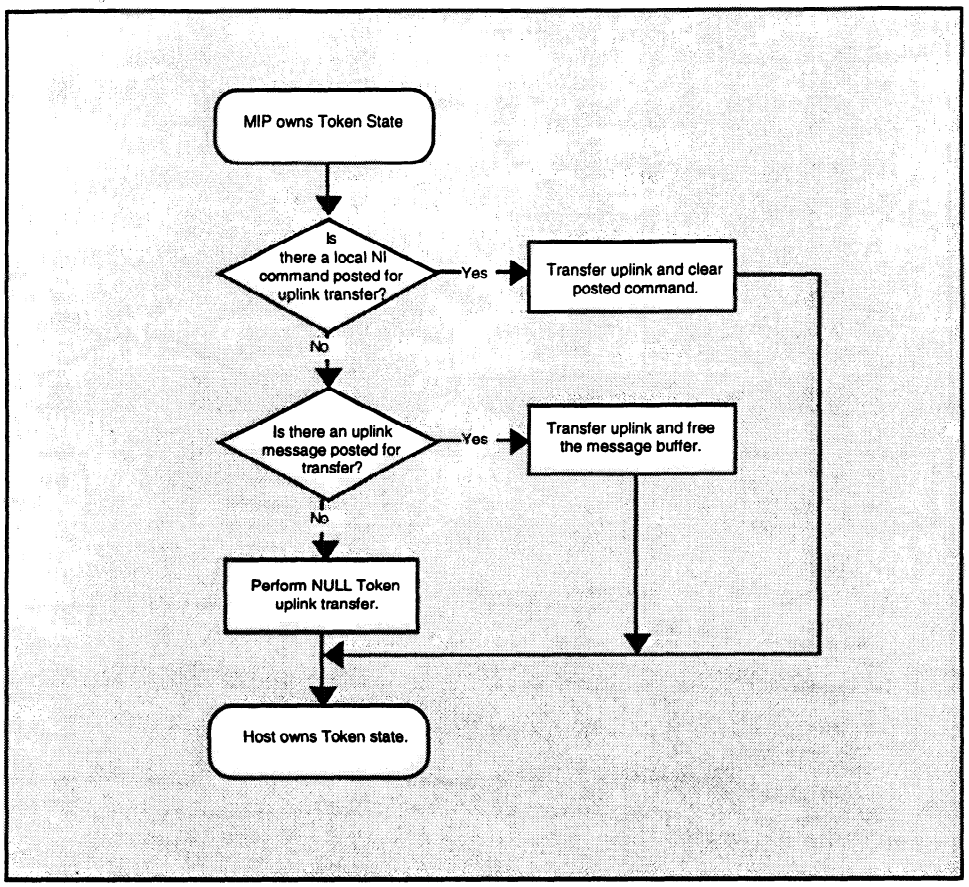


Figure 5.8 MIP Owns Token Processing

Implementing a MIP/DPS Network Driver

A network driver for the MIP/DPS must follow conventions for use of the dual-ported RAM shared by the host and network interface. A control interface structure in the shared memory controls access to uplink and downlink buffers used for transferring messages between the host and network interface. Access to the control interface structure is controlled by hardware semaphores. The following sections define the control interface structure, resource control using the semaphores, and the scenarios for downlink and uplink transfers.

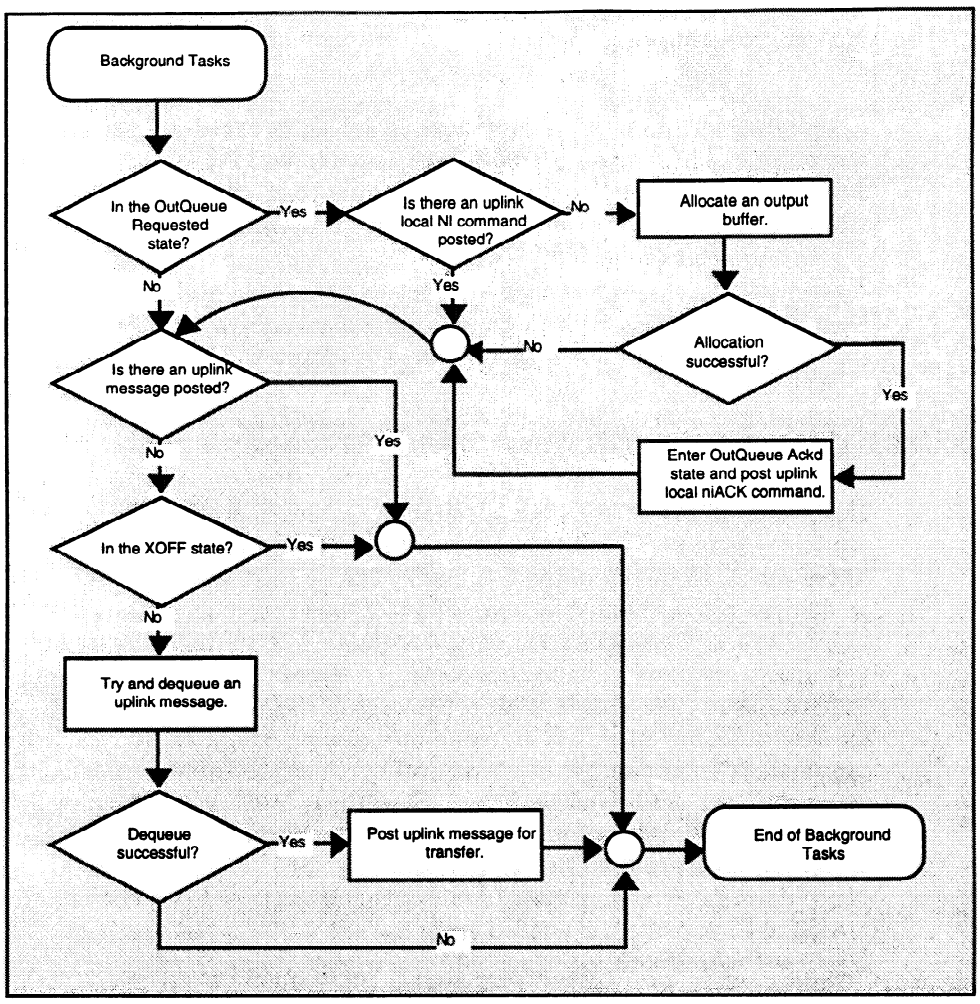


Figure 5.7 MIP Background Processing

Control Interface Structure

A 32-byte control interface structure is stored in the dual-ported RAM. The structure of this control interface structure is defined in the `control_iface` declaration in Appendix B. The base address of the control interface structure starts 32 bytes from the end of off-chip dual ported memory.

Addresses stored in the control interface structure are Neuron Chip addresses. Host access to these addresses will typically require offsetting of these values for the proper location within the host's own memory map.

Resource Control and the Semaphores

The design of the MIP/DPS minimizes the duration of any spin loops that the host processor may be forced to execute while waiting for a semaphore. Referring to Appendix A, elements of the `control_iface.out` and `control_iface.outp` structures are controlled by semaphore 0, and the `control_iface.in` structure by semaphore 1. The `control_iface.command_out` byte is controlled by semaphore 2, and the `control_iface.command_in` byte is controlled by semaphore 3. The buffers themselves reside within the rest of the dual-ported RAM. Address conflicts for the buffering space are avoided due to ownership rules. Modification contentions for the buffer state control bytes are avoided due to network interface/host modification rules. Downlink buffers are allocated from the downlink buffer pool, one at a time, by the network interface Neuron Chip. The buffer pointers are then posted into the `control_iface.out` or `control_iface.out_p` structure. Once posted they are owned by the host until they are filled in and the associated state byte is modified by the host. While owned by the host the network interface will not access them. Uplink buffers are de-queued by the network interface and posted for consumption by the host. Once posted they are no longer owned by the network interface until the host posts them back for freeing by the network interface. Buffers are only read or written to by the side who currently owns them.

The state bytes are the only elements of either structure controlled by the semaphore. Read/write rules control access to the buffer pointers and their corresponding buffers in memory. This assures that the host and network interface will not be accessing the pointers at the same time.

Likewise, the `control_iface.command_out` element is controlled by semaphore 2, and the `control_iface.command_in` element is controlled by semaphore 3.

Two downlink buffers of each type, priority and non-priority, are typically posted to the host, provided they can be allocated by the network interface from the internal pool. Along with these two buffer pointers for each type (named A and B) are two buffer state control bytes, and two A/B selectors, one of which is used by the host, and the other which is used by the network interface.

The values for the output buffer state bytes are:

0 Not available for use by the host.

1 Available to the host for writing.

2 Filled by host.

Address contention for the buffer pointers is avoided by access rules. The host will not read or write to the pointer if its state is '0' or '2'. The network interface will not read or write to the pointer if its state is '1'.

Given that there are two downlink buffers available it becomes important that these buffers are processed by the network interface in the same order as they were filled in by the host. This order is maintained by two single byte A/B selectors which are used to determine the next buffer to be used or checked. Since there are only two buffer pointers (per type, priority and non-priority) there are only two values for the selectors. The host will use the selector in shared RAM rather than a private variable. This assures that the variable is reset if the network interface is ever reset. Semaphore control is not required for the A/B selectors since they are only accessed by their owner: the host or the network interface.

When a downlink message needs to be sent, the host will read one of these pointers, fill in the buffer, and change the corresponding state byte value from '1' to '2'. The network interface determines the message's transaction queue type, transaction or non-transaction, by examining the command/queue byte which is stored in the message header.

The process of filling in an output buffer by a host is as follows: Determine if this is a priority or non-priority message, and use the corresponding structure elements in the control interface. Next, read the A/B selector value to determine which of the two buffer pointers to access. If the corresponding state byte is '1', read the buffer pointer and fill in that buffer with the message. Toggle the A/B selector and change the corresponding state byte to '2'.

If the state byte is '0' the network interface has not yet posted a new downlink buffer. If the state byte is '2' the network interface has not yet processed this message. In either case the host must back off and try this process again later, typically after a response or completion event is passed uplink.

Uplink Buffer Transfer

The mechanism for controlling the uplink buffers is similar to the methods described for the downlink buffers, except in reverse. Only one pair of uplink buffer pointers exist. These buffer pointers have their corresponding state bytes, and host/network interface A/B selectors.

The values for the input buffer state bytes are:

0 Not available for use by the host.

1 Available to the host for reading.

2 Read by host.

Address contention for the buffer pointers is avoided by access rules: The host will not read or write to the pointer if its state is '0' or '2'. The network interface will not read or write to the pointer if its state is '1'.

The process of reading these uplink buffers by the host is as follows: Read the A/B selector value to determine which of the two buffer pointers to access. If the state byte for that buffer pointer is '1' then read the message, toggle the A/B selector, and change the state byte from '1' to '2'. When the network interface sees the state byte is '2' it will free this buffer and change the state to '0'. When the network interface de-queues an uplink buffer it will post the buffer pointer and change the state to '1'.

Write contention for both uplink and downlink buffer state bytes is controlled by rules concerning write access to these state bytes. The network interface only writes (changes) the state byte when it is '0' or '2'. The host only writes the state byte when it is '1'.

The MIP/DPS does not use the `niACK`, `niNACK`, `niPUPXOFF`, or `niPUPXON` uplink commands since they are not required for the dual ported memory interface.

Local Command Processing

A path exists for supporting single byte commands, which do not use application buffers, in both directions between the host and the network interface. These commands are defined by the enumeration `NI_NoQueueCmd` in Appendix C of the *LONWORKS Host Application Programmer's Guide*. Host to network interface commands are passed by first checking the `control_iface.command_out` element. If it is zero a new command may be posted. Likewise, if the `control_iface.command_in` element is non-zero this location should be read, processed, and zeroed.

Figure 5.9 illustrates the structure of the example network driver for DOS included with the MIP/DPS.

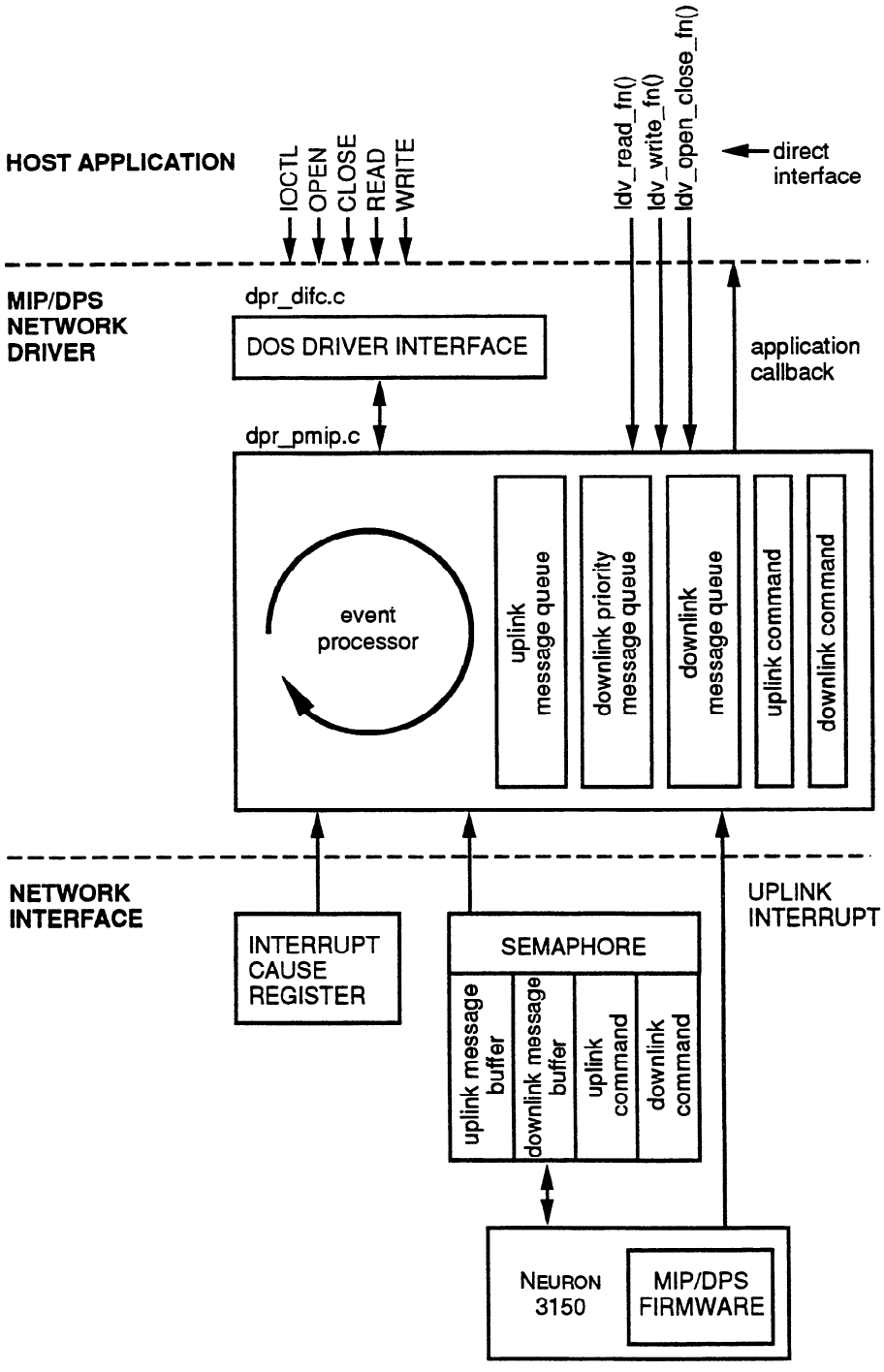


Figure 5.9 MIP/DPS Network Driver Block Diagram

6

Installing the DOS Network Driver

This chapter describes how a MIP network driver is installed on a MS-DOS or PC-DOS host.

Installing the Sample Network Driver

If you are implementing a network driver for DOS and you have added your hardware dependent code to the example driver, the driver can be installed in the same manner as any DOS driver. Add a line to your `config.sys` file which includes the driver name and specify driver options. For the MIP/P20 and MIP/P50, the default name is `ldvpmip.sys`; for the MIP/DPS, the default name is `ldvdpmip.sys`. An example `config.sys` entry for the MIP/P20 or MIP/P50 is:

```
device=c:\lonworks\bin\ldvpmip.sys /o10 /i10
```

An example `config.sys` entry for the MIP/DPS network driver (loaded as `LON2`) is:

```
device=c:\lonworks\bin\ldvdpmip.sys /i10 /d2 /mC900
/sC800
```

In the following descriptions, **nn** represents decimal digits, and **hh** represents hex digits. The command line option switches recognized by the example network driver are:

/Inn Sets the number of input buffers the driver will use to buffer incoming LonTalk messages to *nn*. You must have at least two buffers. A maximum of 90 buffers is allowed. The default input buffer count is 8.

/Dn Sets the device ID to *n*, where n is 1 to 9. The device name then becomes `LONn`. The default device name is `LON1`. This option can be used to load multiple network drivers for multiple network interfaces. The device name must be unique, meaning, for instance, that there cannot be two `LON1` drivers.

/Z Disables automatic cancellation of the `FLUSH` state. If this flag is not specified, the driver automatically cancels the MIP `FLUSH` state. If the flag is specified, the host application must cancel the `FLUSH` state explicitly using the `niFLUSH_CANCEL` command. For more information, see the discussion on the network interface `FLUSH` state in Chapter 5.

The following additional options are available only in the `ldvpmip.sys` driver for the MIP/P20 and MIP/P50:

/Onn Sets the number of output buffers the driver will use to buffer LONTALK messages between the network interface and the host application to *nn*. You must have at least two buffers. Each buffer, input or output, requires 258 bytes of PC memory. The maximum number of output buffers is 90. The default output buffer count is 8.

/Pn Sets the port number for the driver to use to *n*. The default port is 1. Interpretation of the port number depends on the host interface implementation.

The following additional options are available only in the `ldvdpmip.sys` driver for the MIP/DPS:

/Mhhhh Sets the RAM address paragraph number in the first megabyte of host address space (a paragraph is 16 bytes). The default is 0xCD00.

/Shhhh Defines the semaphore address paragraph number in the first megabyte of host address space (a paragraph is 16 bytes). The default is 0xCC00.

The example network driver assumes that the reset latch described in Chapter 4 is implemented on the network interface. Once the driver is installed at boot time by DOS, the reset input to the Neuron Chip is held in the reset state until the network driver is opened. All driver states and input and output buffers are cleared by the reset. When the network driver is closed, the reset input to the Neuron Chip is returned to the reset state.

Appendix A

MIP/DPS Control Structures

This appendix includes ANSI C declarations for the MIP/DPS control structures. Buffer pointers are 16-bit values stored with the high byte followed by the low byte.

MipPtr

This structure typedef is used in the following definitions to define the 16-bit pointers to Neuron Chip memory addresses. 16-bit quantities are not used because the underlying hardware architecture on which the the MIP/DPS driver is ported to may be either little or big endian.

The member `p_state` is set by the network interface to the value 1 to indicate to the MIP/DPS network driver that it may read the information out of the buffer for uplink buffers, or write information into the buffer for downlink buffers. The network interface sets the `p_state` to 0 for all buffers that are not available to be accessed by the network driver. When the MIP/DPS network driver is done reading from or writing to the buffer, it sets `p_state` to the value 2, indicating to the network interface that the buffer can be reused for another message.

```
typedef unsigned char byte;

// This typedef is used for all MIP pointers.
// Note that it is an ODD length.
typedef struct {
      byte p_state;  // state byte always controlled by a semaphore
      byte p_hi;
      byte p_lo;
} MipPtr;
```

mipci_outbuffs_s

The `mipci_outbuffs_s` structure is used to implement the two downlink message buffer queues, for non-priority and priority messages. The downlink message queues are both controlled by semaphore 0.

```
// This structure is used for host->mip messaging, priority and
// non-priority, and is controlled by semaphore 0.

typedef struct mipci_outbuffs_s {
      // This group of pointers point to output buffers:
      MipPtr out_a;           // pointer for buffer A
      MipPtr out_b;           // pointer for buffer B
      // These are the selectors for accessing 'A' or 'B'
      // 0 => a
      // 1 => b
      byte host_o_absel;      // Used by Host.
      byte mip_o_absel;       // Used by MIP.
};
```

mipci_inbuffs_s

The `mipci_inbuffs_s` structure is used to implement the uplink message buffer queue, which is controlled by semaphore 1.

```
// This structure is used for mip->host messaging
// and is controlled by semaphore 1.
```

```
typedef struct mipci_inbuffs_s {
      // This group of pointers point to input buffers:
      // uplink, response and completion events
      MipPtr in_a;            // pointer for buffer A
      MipPtr in_b;            // pointer for buffer B
      // These are the selectors for accessing 'A' or 'B'
      // 0 => a
      // 1 => b
      byte host_i_absel;      // Used by Host.
      byte mip_i_absel;       // Used by MIP.
};
```

control_iface_s

The control_iface_s structure defines the last 32 bytes of the dual-ported RAM. Upon a reset, the MIP/DPS code waits for all eight semaphores and then initializes this area.

```
// This structure represents the layout of the memory at the last
// 32 byte page of the dual ported RAM.

struct control_iface_s {
// State bytes controlled by semaphore 0
      mipci_outbuffs_s out_p;       // Priority downlink niTQ_P, niNTQ_P
      mipci_outbuffs_s out;         // Non-priority downlink niTQ, niNTQ

// State bytes controlled by semaphore 1
      mipci_inbuffs_s in;           // Uplink niRESPONSE, niINCOMING

// Used to pass local commands to the MIP
// Controlled by semaphore 2.
      byte command_out;             // Downlink NI_NoQueueCmd

// Used to pass local commands to the HOST
// Controlled by semaphore 3.
      byte command_in;              // Uplink NI_NoQueueCmd

      byte pad1[6];                 // Forces the size to 32.
} control_iface;
```

Appendix B

MIP/DPS Example Schematic

This appendix is an example schematic for a network interface based on the MIP/DPS. Figure B-1 is a top-level schematic for the network interface. Figures B-2 and B-3 provide the detailed schematics for the Neuron Subsystem and Interrupt Logic blocks shown in the top-level schematic. The Host Interface and Transceiver blocks are implementation dependent, and are not included with this example.

The example implements the memory map shown in Table B-1.

Memory Map

The example implements the memory map shown in table B.1.

Table B.1 Neuron Chip Memory Map

Start Address (hex)	*End Address (hex)*	*Usage*
0000	7FFF	PROM
8000	8007	Dual-Port RAM Semaphores 0 - 7
8008	9FFF	Do not use (maps to semaphore bits)
A000	BFFF	Dual-Port RAM (8KBytes)
E000	FFFF	Neuron Chip Internal

Example Notes

The `mipci_outbuffs_s` structure is used to implement the two downlink message buffer queues, for non-priority and priority messages. The downlink message queues are both controlled by semaphore 0. Following are additional notes concerning the example:

- The `HCE_RAM-` and `HCE_Sem-` signals shown in figure B.1 are mutually exclusive, so they must be enabled by different addresses on the host.
- The dual ported RAM shown in the example is a CY7B144 8KByte DPRAM. A CY7B138 or compatible part can be substituted for a 4KByte DPRAM.
- An AS family part is required to generate the `WE-` signal shown in figure B.2.
- Diode D1 in figure B.3 resets the Neuron Chip when the host resets (`HReset-` asserted), and prevents the Neuron Chip from resetting the host (`Reset-` asserted).

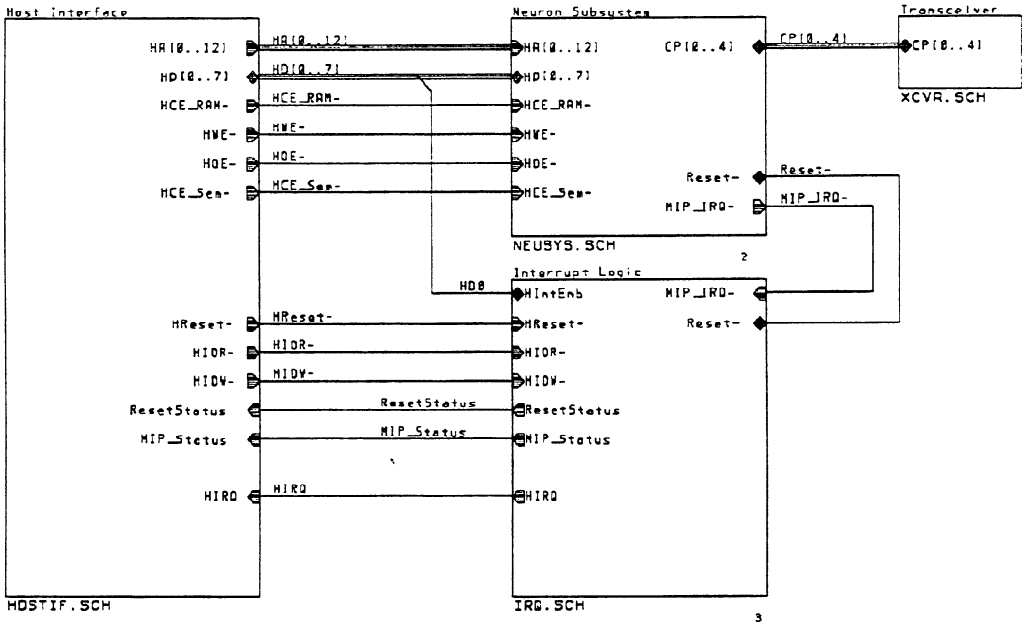


Figure B-1 MIP/DPS Example Top-Level Schematic

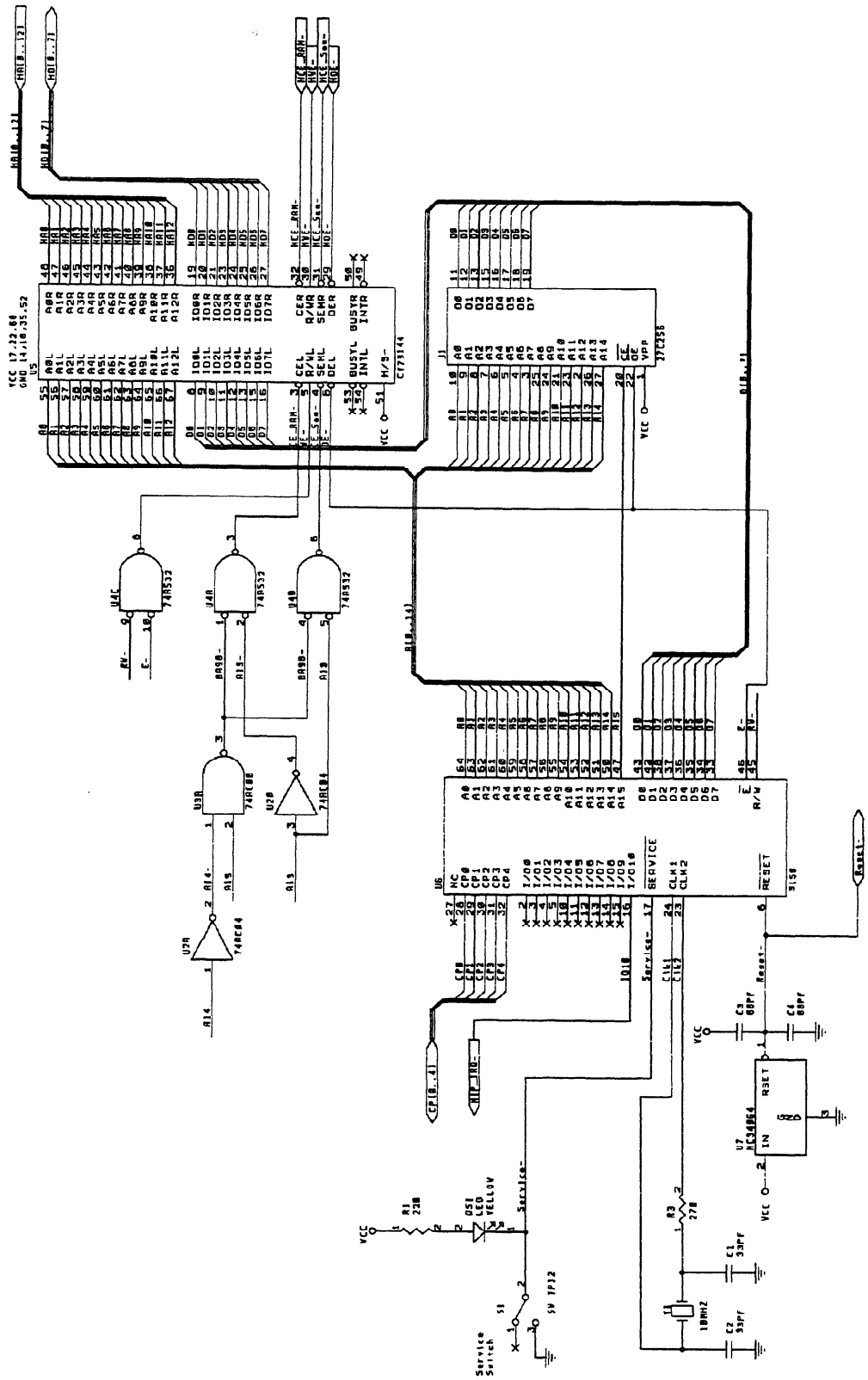


Figure B.2 Neuron Subsystem (Example Circuit Only)

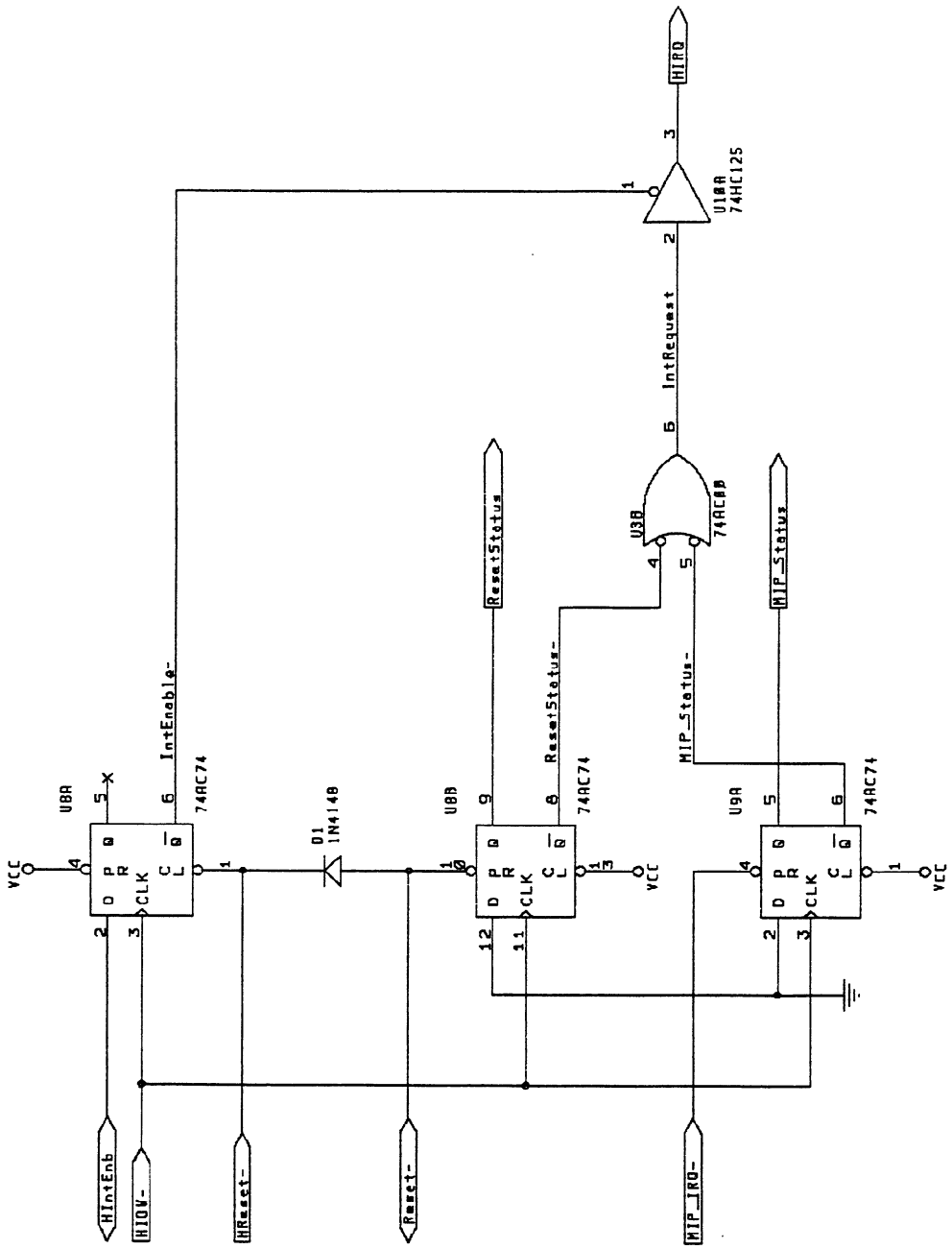


Figure B.3 Interrupt Logic (Example Circuit Only)