The
Connection Machine
System

NI Systems Programming

Version 7.1
October 1992

Thinking Machines Corporation
Cambridge, Massachusetts

First printing, October 1992

Thinking Machines Corporation
245 First Street
Cambridge, Massachusetts 02142–1264
(617) 234–1000

Contents

Chapter 2 A Generic Network Interface cont'd

Chapter 3 The Data Network 27

Chapter 7 NI Programming Issues cont'd

Appendixes

Appendix D NI Interrupts cont'd

Indexes

List of Figures

About This Manual

Objectives of This Manual

This manual describes in detail the design, features, and proper use of the Network Interface (NI) chip of the Connection Machine CM-5 system, at a level sufficient for low-level CM-5 coders to make full use of the NI's features.

This manual describes the NI from a system programmer's point of view. It discusses user and supervisor features of the NI, and provides enough detail to allow a knowledgeable CM-5 programmer to write code that manipulates the NI. The appendixes of this document include C code examples and references to on-line sources of sample NI code.

Intended Audience

This manual is intended for use by knowledgeable CM-5 programmers. While it contains some overview information, this document is a reference manual, not a tutorial. This manual should be used in conjunction with other programming guides and with assistance from Thinking Machines Corporation representatives.

For examples of NI programs written and compiled in C code, refer to the existing *Programming the NI* manual.

Revision Information

This manual is new as of Version 7.1. It is based on the existing *Programming the NI* manual, but includes additional supervisor information that was excluded from *Programming the NI*.

Organization of This Manual

Chapter 1 The Network Interface Chip
An overview of the NI chip's purpose in the CM-5 hardware, and a description of the important features of the chip.

Chapter 2 A Generic Network Interface
A description of common features found in most of the NI network interfaces.

Chapter 3 The Data Network
The registers and features of the three Data Network interfaces.

Chapter 4 The Control Network
The registers and features of the three Control Network interfaces (broadcast, combine, and global).

Chapter 5 NI Interrupts
A description of the various interrupt classes of the NI, and of the mechanisms used to detect and signal NI interrupts.

Chapter 6 Other NI Interfaces and Features
A description of NI registers and features not covered by the preceding chapters.

Chapter 7 NI Programming Issues
A summary of important programming and performance considerations that you should keep in mind while writing code that manipulates the NI.

Appendix A NI Memory Map
A two-sided memory and register map, showing the arrangement of the NI's registers and register subfields.

Appendix B NI Registers, Fields, and Constants
A summary of the registers and fields of the NI chip and of the programming constants that can be used to locate them.

Appendix C Predefined Low-level Constants
A list of all low-level programming constants defined by the files `cmsys/ni_constants.h` and `cmsys/ni_defines.h`, with the symbols grouped by register and field.

Appendix D NI Interrupts
A description of each of the possible NI interrupts, including what they indicate and how to recover from them.

Appendix E NI Programming Examples
A set of simple C code examples of routines that read and write NI registers and perform other useful functions.

Appendix F CMNA Header Files
Describes the content and relationship between the various header files that define the CM Network Accessor interface.

Related Documents

These documents are part of the Connection Machine documentation set:

- *Connection Machine CM-5 Technical Summary*, January 1992
- *Programming the NI*, March 1992

Notation Conventions

The table below displays the notation conventions observed in this manual.

Convention	Meaning
`bold typewriter`	UNIX and CM System Software commands, command options, and filenames, when they appear embedded in text. Also, syntax statements and programming language elements, such as keywords, operators, and function names, when they appear embedded in text.
italics	Argument names and placeholders in function and command formats.
`typewriter`	Code examples and code fragments.
`%` **`bold typewriter`** `regular typewriter`	In interactive examples, user input is shown in **`bold typewriter`** and system output is shown in `regular typewriter` font.

Customer Support

Thinking Machines Customer Support encourages customers to report errors in Connection Machine operation and to suggest improvements in our products.

When reporting an error, please provide as much information as possible to help us identify and correct the problem. A code example that failed to execute, a session transcript, the record of a backtrace, or other such information can greatly reduce the time it takes Thinking Machines to respond to the report.

If your site has an applications engineer or a local site coordinator, please contact that person directly for support. Otherwise, please contact Thinking Machines' home office customer support staff:

U.S. Mail:	Thinking Machines Corporation Customer Support 245 First Street Cambridge, Massachusetts 02142–1264
Internet Electronic Mail:	customer–support@think.com
uucp Electronic Mail:	ames!think!customer–support
Telephone:	(617) 234–4000 (617) 876–1111

Chapter 1

The Network Interface Chip

The Network Interface chip, or NI, manages the internal communications networks of the CM-5. This chapter presents an overview of the NI's location and function within the CM-5, as well as a description of the features of the NI that are important to you as a programmer.

1.1 The CM-5 System: Nodes and Networks

The CM-5 contains a large number of processing nodes linked together by two main internal networks, the *Data Network* and the *Control Network*.

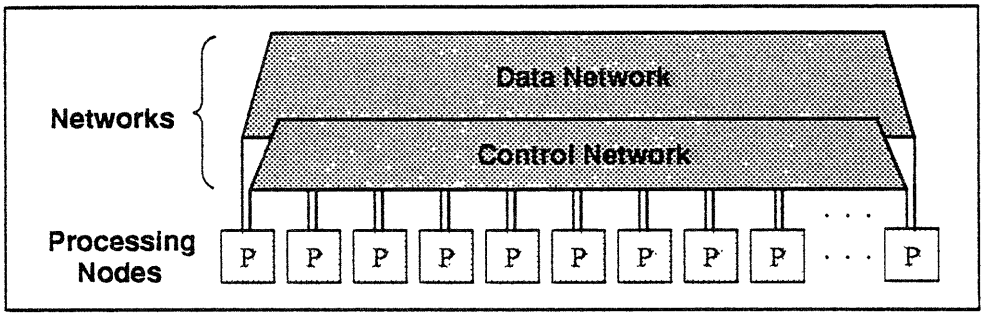


Figure 1. CM-5 processing nodes linked by Data Network and Control Network.

The Data Network is used for high-volume exchange of data between nodes. The Control Network is used to control and synchronize the operations of the nodes.

1.1.1 The CM-5 Networks

The two CM-5 networks are similar to each other in design. Both are scalable high-speed data communications networks. However, the networks are quite different in structure and purpose.

The Data Network

The Data Network is a high-speed, high-bandwidth network designed to handle the simultaneous node-to-node transmission of thousands of messages. The Data Network is composed of two halves, the *left interface* and the *right interface*, both of which are connected to all processing nodes. The left and right interfaces can be used either independently or together as the combined *Data Network*.

Terminology Note: This combination of the left and right halves of the Data Network is sometimes called the "middle" interface by NI programmers.

The Control Network

The Control Network is used for control tasks that require the joint cooperation of all nodes. It provides three separate functions:

- The *broadcast interface* distributes a single numeric value to every node. It consists of two subinterfaces: a *user* broadcast interface and a *supervisor* broadcast interface.
- The *combine interface* receives a single value from each node, combines the values arithmetically or logically, and then distributes the combined result to all nodes.
- The *global interface* handles global synchronization of the nodes. It consists of a number of distinct interfaces for synchronous and asynchronous messaging by user and supervisor (OS) code.

For the Curious: The Diagnostic Network

There is also a third major CM-5 network, the Diagnostic Network, used by the system manager to determine the operational condition of the CM-5 hardware and to diagnose hardware problems. However, because the NI chip is not used to access it, the Diagnostic Network is not discussed further in this manual.

1.1.2 Processing Nodes

Each processing node contains a RISC microprocessor, a memory subsystem, and a Network Interface (NI) chip, linked together in a bus arrangement:

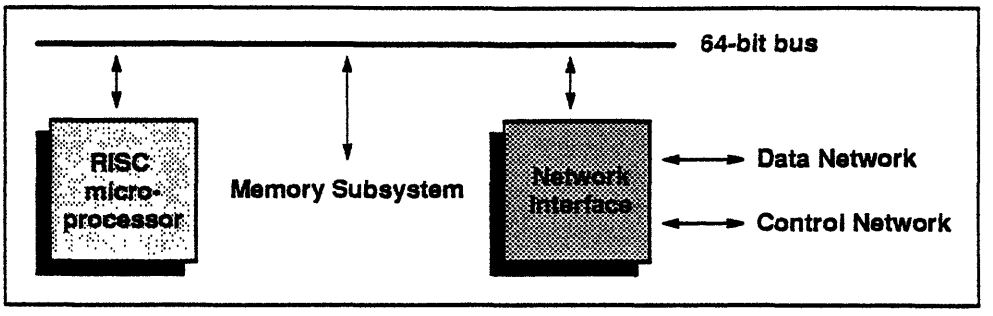


Figure 2. The components of a typical processing node.

For the Curious: In the current implementation, the microprocessor is a SPARC chip; it executes both user and operating system (OS) code. The memory subsystem consists of up to 32 Mbytes of DRAM memory, controlled either by a single memory controller or by a set of four vector units.

1.1.3 Partitions and Partition Managers

The processing nodes are grouped by software into *partitions*, with each partition monitored by a *partition manager* (PM). (See Figure 3.) Each partition can be as small as 32 nodes, or as large as the entire machine. The partitioning is controlled by the system administrator, who can create and alter partitions as needed.

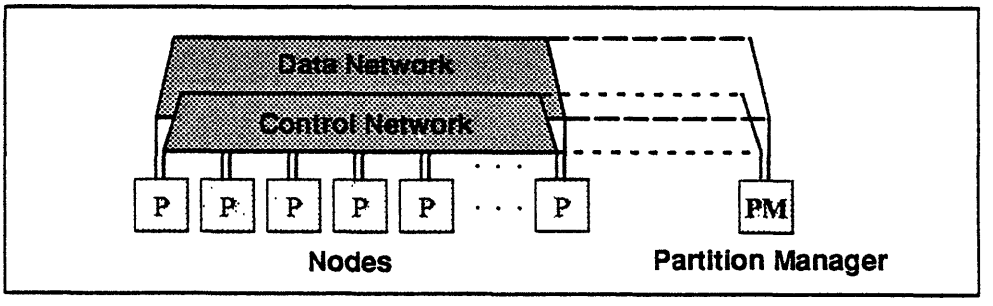


Figure 3. A partition of nodes and its partition manager.

1.1.4 Programming Models

User Programming Model

From a user's point of view, the CM-5 is the single partition of nodes associated with the PM that compiles and executes the user's code. CM-5 user programs compile into two separate sets of code, one for the PM and one for the nodes.

The PM typically controls program flow, and handles all external interactions (communicating with the user by keyboard input and screen output, exchanging files and data with other computing systems over external networks, etc.).

The nodes typically operate in an event-driven loop, waiting for instructions from the PM about which section of code to execute next.

OS Programming Model

From an OS point of view, the CM-5 is a set of partitions, each of which has a number of associated processes that can be swapped in.

The CM-5 OS manages the execution and swapping of processes within partitions, as well as any exchange of data that takes place between partitions (for example, when a user program needs to read or write data from an I/O device).

Under the CMOST operating system shipped with the CM-5, each PM runs a full and complete UNIX-based operating system, while each of the nodes runs a small kernel of OS code that is optimized for computation and communication. It is this kernel of code that provides the event-driven dispatch loop described in the user programming model above.

1.2 The NI Chip

The NI chip serves as an intermediary between the microprocessor and the two CM-5 networks. Each network provides a specific set of *network interfaces*, and the role of the Network Interface chip is to make those interfaces available to the node microprocessor, and thereby to user code and OS code.

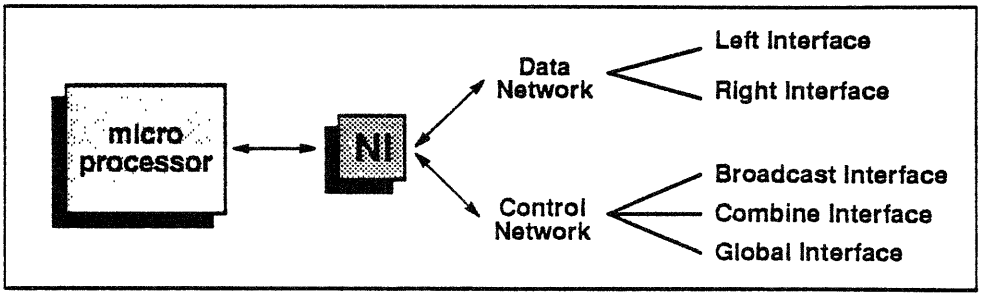


Figure 4. NI provides access to features of the Data Network and Control Network.

When the microprocessor directs the NI to send a message via a particular network interface, the NI handles the dispatching of the message, and collects any replies from the networks. The NI uses send and receive FIFOs to hold outgoing messages until they can be sent, and to hold incoming messages until the microprocessor reads them.

The NI chip is register-based; its network functions are controlled entirely by reading and writing NI registers. Access to these registers is provided by memory-mapping — the NI registers are mapped into the microprocessor's memory address space. From a programmer's point of view, therefore, the NI appears as a region of memory with some unique properties.

The microprocessor can either examine the registers of the NI chip to see whether a message has arrived, or it can instruct the NI to signal an interrupt when a message arrives. Interrupts can also be "broadcast" from one NI chip to all other NIs in a partition.

Control of the NI is therefore based on register operations, interrupts, and (in extreme cases) NI Resets, which are described later in this chapter.

1.3 The NI Registers

The NI registers occupy a virtual memory region 512 Kbytes long. However, the NI registers are actually mapped into microprocessor memory twice, as two separate virtual memory areas: the *user area* and the *supervisor area.* (See Figure 5.)

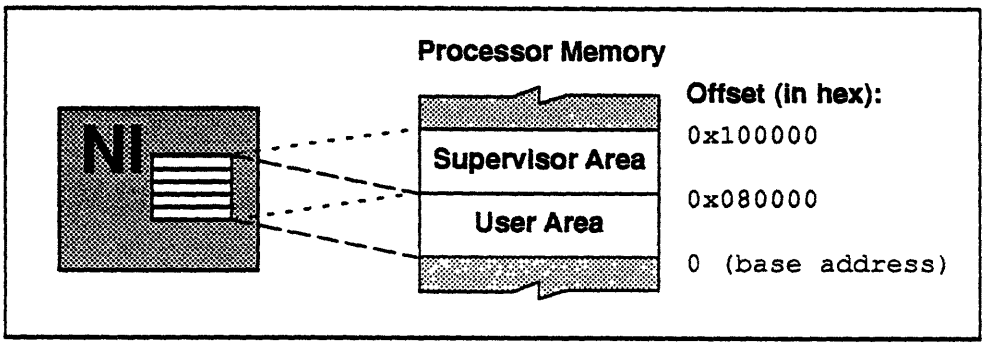


Figure 5. The NI registers are mapped into user and supervisor memory areas.

The user area occupies 512K bytes of virtual memory, starting at the base address of the NI memory region (see Section 1.3.4). The supervisor area occupies the 512K bytes immediately following the user area.

The user and supervisor areas contain the same registers at the same offsets, but the hardware mapping is designed so that the NI registers for supervisor features are only accessible from the supervisor area. Any attempt to access supervisor registers from the user area signals a Bus Error. (A programmer sees this as a segmentation violation.) Thus, when this manual speaks of "the supervisor" performing an operation, or of an NI feature that is "restricted to the supervisor," this simply means that only programs with access to the NI supervisor area can perform the described operation or use the described feature.

In general, it is the responsibility of the operating system to make sure that user programs don't have access to the NI supervisor area. Typically, this is done by using virtual address mapping to place the supervisor area in a memory region to which user programs don't have access.

Note: Some locations in the NI memory region don't correspond to registers. The effect of reading or writing these locations is not defined, but is never of practical use to programmers. Typically, a Bus Error (see Section 1.4) is signaled.

1.3.1 NI Register Types

There are three basic types of NI registers:

FIFO Registers — These "registers" are actually the entry and exit points of send and receive FIFOs associated with the CM-5 networks. Writing a value to a FIFO register pushes that value into the *send FIFO* of the corresponding network. Likewise, reading the value of a FIFO register pops a value from the *receive FIFO* of the network.

Status Registers — These registers are composed of one-bit flags and multi-bit fields, which indicate the state of the NI and its message FIFOs. For example, most networks have two important status flags, **`send_ok`** and **`rec_ok`**, which indicate the current status of messages being sent or received.

Control Registers — These are status registers containing flags that not only report the state of the NI, but also allow you to control it. Altering the value of a control register's flags has a corresponding effect on the state of the NI. For example, each of the Control interfaces has one or more *abstain* flags that control whether or not the NI participates in the transactions of the network.

Important: Some registers are less than 32 bits long, even though they occupy a 32-bit memory location. When such a register is read, the value of the unused bits is undefined. However, when writing to the register, the unused bits should be written with either the same value that was last read from them, or with zeros. The effect of failing to follow this restriction is not defined, but in some cases serious failures can result. (In at least one case, failing to zero out the unused bits causes your partition of nodes to crash. See Section 7.3.1.)

1.3.2 NI Register and Field Names

In this manual, the names of NI registers and register fields are given in the form:

`ni_`*interface_purpose*

The *interface* part of the name identifies the network interface, and is typically one of the following abbreviations:

`dr`	Data Network (left and right)	**`bc`**	broadcast interface
`ldr`	left interface	**`com`**	combine interface
`rdr`	right interface	**`global`**	global interface

The *purpose* describes the purpose of the register or field. Some common examples are:

send	Register used to send a network message.
recv	Register used to receieve a message.
send_ok	Flag indicating that a message was sent successfully.
rec_ok	Flag indicating that a message has been received.

For conciseness, this manual sometimes refers to a register or field by its *purpose* alone. However, this is done only when the intended reference is unambiguous.

The appendixes of this manual include a memory map and a series of lists that exactly specify each register's location and the position and length of any subfields it may have.

1.3.3 NI Register and Field Programming Constants

There are a number of predefined programming constants that you can use to refer to NI registers and fields in your code.

These constants are defined in such a way that they can be used for both user and supervisor code; the names of the register and field constants are the same for both the user and supervisor areas, and are typically based on the names of the registers and fields themselves.

To get access to these predefined constants, include the header file **cmna.h**:

```
#include <cm/cmna.h>
```

Note: Assembly-language coders may wish to load a more specific file of constants. See the discussion of the CMNA header files in Appendix F.

Finding the Constant You Need

Appendix B of this manual lists the names of the NI registers, fields, and flags, and gives the corresponding constants to use in accessing them. Appendix C provides a complete list of the available low-level register and field constants. The types of predefined constants are described below.

Register Constants

The constants for registers specify the actual address of the register, and there is one such constant for each NI register. To get the name of the constant that corresponds to a register, uppercase the name of the register, and add the suffix "**_A**". For example, the constant for the register **ni_dr_status** is **NI_DR_STATUS_A**.

Note for C Programmers: The register constants are unsigned pointer values. To use them in C code, you must cast them to type **(unsigned *)**:

```
unsigned *ni_dr_status = ((unsigned *) NI_DR_STATUS);
```

If you don't perform this casting step, the C compiler by default treats the constants as integers, causing warnings about "illegal pointer operations."

Field Constants

The constants for NI fields provide the starting bit position and length of each field. However, since a number of NI registers have some basic fields and flags in common, the name of the appropriate constant isn't always directly derivable from the name of the field or flag in question. In many cases, you can obtain the constant name by uppercasing the field or flag name, and adding the suffix "**_P**" for the starting bit position, or "**_L**" for the field length.

For example, the **ni_dr_status** register has a field named **ni_dr_rec_tag**. This field has two corresponding constants, **NI_DR_REC_TAG_P** and **NI_DR_REC_TAG_L**, that give, respectively, the position and length of the field.

However, there is also a flag called **ni_send_ok** in the same register. Since most of the networks have a **send_ok** flag, there is a single pair of constants, named **NI_SEND_OK_P** and **NI_SEND_OK_L**, which apply to all the networks.

NI Base Address Constant

There is also a predefined constant that you can use to refer to the base address of the NI memory region (either user or supervisor) that you are using:

NI_BASE — Base address of NI memory region (user or supervisor).

The value of this constant depends on the environment in which you compile your code.

1.3.4 For the Curious: The NI Base Address — Physical and Virtual

The *physical* base address of the entire NI region (user and supervisor areas) is fixed at a value determined for each node by hardware (essentially by two input pins on the NI chip that are permanently set either high or low for each circuit board). The actual physical address chosen by this method is the same for each node throughout the CM-5 hardware.

The *virtual* base address of the user and supervisor areas depends on the way the operating system sets up the virtual memory map. The operating system is free to map the two areas of the NI memory region to virtual memory location, so long as the user and supervisor areas each remain contiguous and user programs are prevented from accessing the supervisor area.

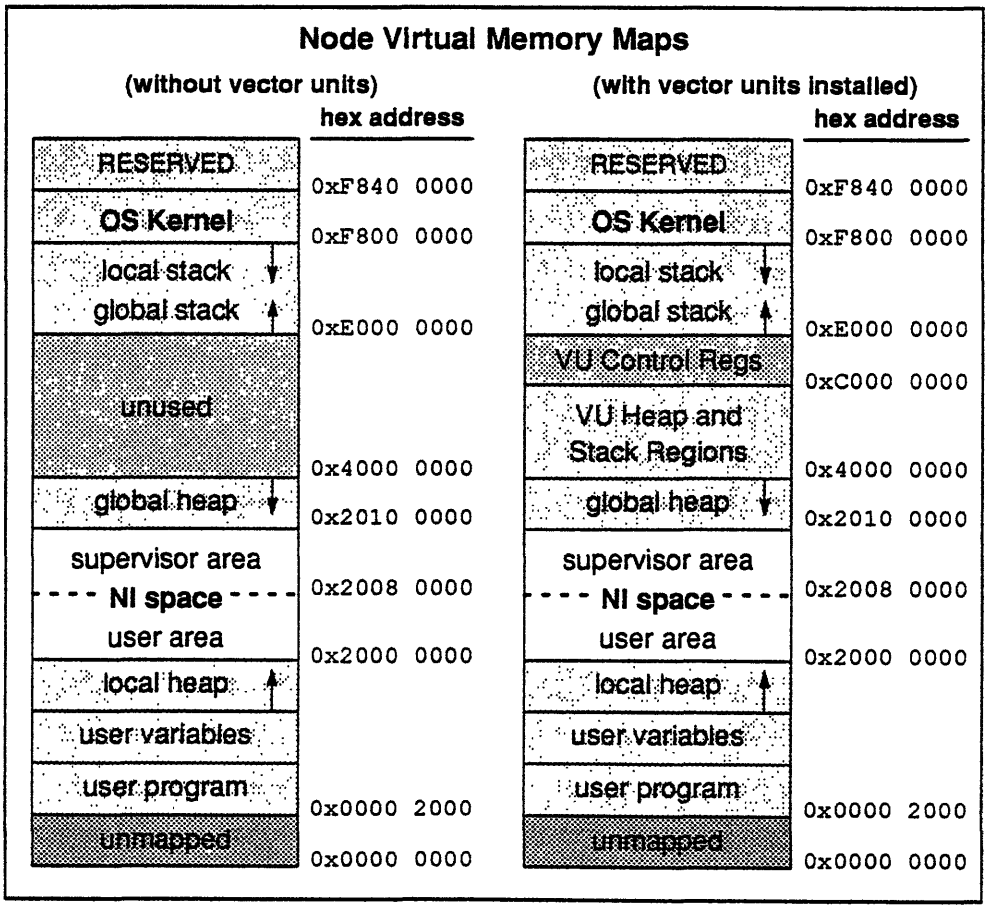


Figure 6. Sample virtual memory maps showing location of NI memory region.

The CMOST operating system distributed with the CM-5 maps the two NI regions into a contiguous 1024 Kbyte block, as described in the preceding section. Figure 6 shows two possible CMOST virtual memory maps, one without the vector units, and one with the vector units installed.

1.4 Interrupts

In addition to using registers to control the NI, you can also instruct the NI to signal an interrupt to the microprocessor under certain conditions, such as the arrival of a network message via a specific interface. These kinds of interrupts can be used to trigger calls to routines of your program (for example, handlers that automate the receipt of network messages). The NI also signals interrupts for fatal sofware/hardware errors and other important events.

The NI can signal five different classes of interrupt: Red, Orange, Yellow, Green, and Bus Errors. Red interrupts tend to be the most severe and Green interrupts the least severe.

The five interrupt classes can be briefly summarized as follows:

- **Red interrupts** indicate a hardware failure, or message checksum error.
- **Orange interrupts** indicate events that the operating system must handle.
- **Yellow interrupts** are triggered by fatal errors in user or OS software.
- **Green interrupts** are triggered by important non-fatal events that user or OS software may want to handle specially.
- **Bus Errors** indicate address errors in user or OS software that prevent a bus transaction from being completed.

The five types of interrupts, along with the registers used for enabling and controlling them, are described in more detail in Chapter 5.

In this manual, the names of interrupts are given as abbreviations based on the names of the register fields used to detect and clear them. For example, the Green interrupt triggered by the arrival of a broadcast message is `bc rec ok`.

1.5 NI Reset

Under certain conditions, the NI chip is completely reset. Among other things, this causes a number of its registers to be set to known states. The causes and effects of an NI Reset are described in Section 6.10.

Chapter 2

A Generic Network Interface

Each network interface of the Data and Control Networks has a corresponding register interface — a set of NI registers that are used to send and receive messages through the network. Many of these register interfaces have a number of features in common. This chapter presents a "generic" network interface that describes these features. With one exception (the global interface), all network interfaces conform to the model described here. Individual variations for each network interface are described in following chapters.

Important: The interface presented in this chapter is an abstract description that is built upon in later chapters. There is no actual "generic network interface" for the NI chip — merely a set of similar but independent network interfaces.

2.1 Network Interface Registers

For each *interface* that follows the generic model, the following NI registers are used to communicate with that interface:

`ni_`*interface*`_send_first`	Used to send the first value of a message.
`ni_`*interface*`_send`	Used to send the rest of the message.
`ni_`*interface*`_recv`	Used to receive a message.
`ni_`*interface*`_status`	Status register.
`ni_`*interface*`_control`	Control register.
`ni_`*interface*`_private`	Supervisor control register.

The purpose and use of each of these registers and subfields is described in the sections below. Figure 7 contains a memory map showing the relative locations of these registers in the user and supervisor memory areas.

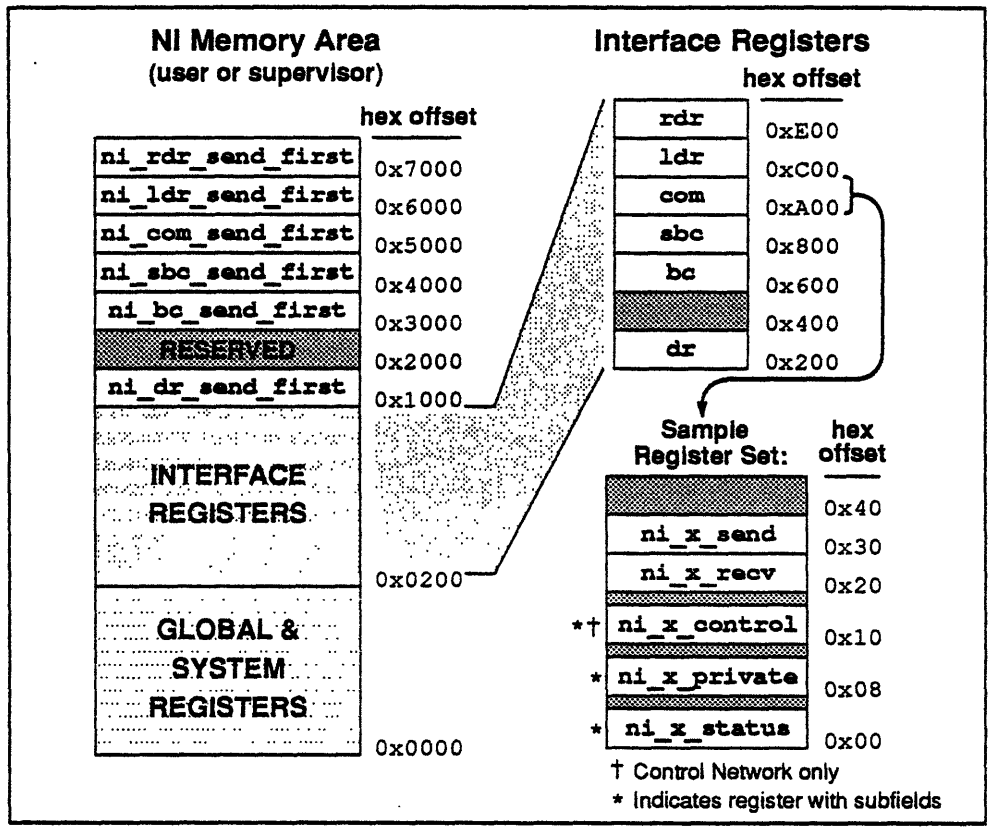


Figure 7. NI registers associated with each interface.

2.2 Network Messages

A network message is a sequence of word-length (32-bit) values. Its content, format, and length limit depend on the network. Each message is accompanied by a small amount of *auxiliary information* (such as the length of the message, a tag field, etc.). The format of this auxiliary data is also network-dependent.

Sending a message involves writing its sequence of values to the send FIFO register of a network interface. As the message is written, the individual values are collected in the send FIFO. When the entire message has been written to the FIFO, the NI begins trying to send the message through the network. Similarly, receiving a message involves reading its values from the receive FIFO register of the network interface.

When a message arrives from one of the networks, the NI accumulates the message in the corresponding receive FIFO. When the entire message has been received, the NI sets a status flag, indicating a message is available. Your program can then read the individual words of the message from the receive FIFO.

The send and receive FIFOs have a length limit (typically 5 words in the current implementation). Longer messages must be divided into packets at the sending node and combined at the receiving node. If you attempt to send a message that is longer than the total length of the FIFO (that is, a message that couldn't possibly fit, even if the FIFO was empty) a Bus Error is signaled.

2.2.1 Performance Note — Using Doubleword Operations

You can use doubleword (64-bit) operations to read and write FIFO registers. A doubleword read or write has exactly the same effect as the corresponding pair of single-word (32-bit) reads or writes, but the doubleword operation is usually more efficient. (See Section 7.2.2.) From here on, where this manual refers to a "value" of a message, you should understand this as referring to either a single- or doubleword value. Any network-specific restrictions that prevent the use of doubleword operations are noted in the descriptions of the networks themselves.

2.3 Sending a Message

For each network interface, there is a single send FIFO, but two FIFO registers are used to access it in the process of sending a message:

ni_*interface*_send_first	Used for first value of a message.
ni_*interface*_send	Used for the rest of the message.

Important: There is a specific protocol to follow in sending a message:

- The first value of a message must be written to the **send_first** FIFO register. This tells the NI that a message is being composed, and also specifies the message's auxiliary information (see Section 2.3.2 below).
- The remaining values (if any) must be written to the **send** FIFO register.

If this protocol is not followed, a Bus Error is signaled, and the message currently being composed is discarded.

2.3.1 Message Discarding

A message being written to the send FIFO register of a network interface can be discarded for any of a number of reasons:

- The send FIFO may be temporarily full.
- The supervisor may have disabled message sending for that interface.
- The message may not have been written according to protocol.

Whatever the reason, when a message is discarded, it is *completely* discarded. Any previously written values for that message are removed from the send FIFO, and a new message can be started by writing a value to the **send_first** register. It is as though you never began writing the discarded message in the first place. (Writing additional values to the **send** register after a message has been discarded is legal, but has no effect.)

Performance Note: You can use message discarding to your advantage and thereby make your code more efficient. Rather than check the **send_ok** flag after writing each word of a message to the send FIFO, you can simply check the flag once, after the entire message has been written. (For more information, see Section 7.2.3.)

2.3.2 Auxiliary Information

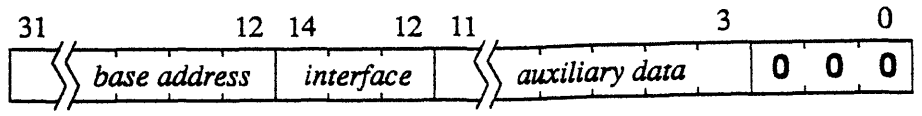
The auxiliary information of a message typically includes the length of the message in words, as well as network-specific data such as a message tag. This auxiliary information is transmitted implicitly when you write the first value of a message to the **send_first** register.

The **send_first** register for each network interface is actually mapped onto a block of memory locations. Writing a value to any one of these locations has the effect of writing that value to the **send_first** register, but the actual memory location that you use implicitly supplies the auxiliary information of the message. (The low-order bits of the address actually contain the auxiliary data itself.)

Another way of saying this is that the length of a message, among other things, determines the **send_first** address you must use to send it.

2.3.3 Calculating ni_*interface*_send_first Addresses

The **send_first** address for a network message is a 32-bit value of the form:



where *interface* is the interface number (an integer from 0 to 7 representing the interface being used), *auxiliary data* is the auxiliary information of the message, and *base address* is the base address of the NI memory area (user or supervisor).

The interface numbering is as follows:

1 — Data Network (left and right)
6 — left Data Network interface
7 — right Data Network interface
3 — broadcast interface
4 — supervisor broadcast interface
5 — combine interface

(The global interface does not conform to the generic interface model, so it does not play a part in this numbering scheme. The values 0, 2, and 4 are reserved.)

The auxiliary data depends on the message, and each interface has its own format for this field. However, all the interfaces have at least one field in common: a *length* field, representing the length of the message in words. This field occupies the low-order 4 bits of the *auxiliary data* field (bits 3 – 6 inclusive).

For the Curious: The auxiliary data is left-shifted three bits to leave sufficient space between **send_first** addresses for doubleword read/write operations. (See Section 2.2.1.)

Send First Address Constants

The following constants are used to construct **send_first** addresses:

NI_BASE	The NI base address.
SF_FIFO_OFFSET	The *interface* field offset (12).
AUXILIARY_START_P	The *auxiliary data* field offset (3).

To construct a **send_first** address, combine the following values, left-shifted as shown:

The NI base address:	**NI_BASE**	
The *interface* number:	*interface_number*	<< **SF_FIFO_OFFSET**
The *auxiliary data* field:	*auxiliary_data*	<< **AUXILIARY_START_P**

The following *interface_number* constants are defined:

`DATA_ROUTER_FIFO`	Data Network interface (1).
`LEFT_DR_FIFO`	Left Data Network interface (6).
`RIGHT_DR_FIFO`	Right Data Network interface (7).
`USER_BC_FIFO`	User broadcast (BC) interface (3).
`SUPERVISOR_BC_FIFO`	Supervisor broadcast (SBC) interface (4).
`COMBINE_FIFO`	Combine (COM) interface (5).

The interface-specific constants defining the *auxiliary data* field format are described together with the corresponding network interfaces in later chapters.

For C Programmers: Appendix E of this manual includes examples of simple C macros that construct `send_first` addresses for each network interface.

2.4 Receiving a Message

For each network interface, the following register is used to receive messages:

`ni_`*interface*`_recv`	FIFO register from which values are read.

A message is received by reading its value(s) in order from the `recv` register, one at a time.

2.4.1 Detecting Arrival of a Message

When a message arrives in the receive FIFO, the NI sets the `rec_ok` flag in the `status` register (see Section 2.5). You can repeatedly test the `rec_ok` flag to determine whether a message has arrived (for example, in a top-level loop).

Alternatively, you can set a flag in the "private" register (See Section 2.7.) that causes the NI to signal an interrupt whenever the `rec_ok` flag is set. You can use this feature to "automate" message reception by having the interrupt trigger an appropriate message-reading routine in your program.

Note: Access to the "private" register is restricted to the supervisor area. User programs, which do not have supervisor access, must make a system call to set the receive interrupt flag.

2.4.2 Simulating the Arrival of a Message

The supervisor has the additional ability to *write* a value to the **recv** register; this pushes a value into the tail end of the FIFO, as if it had arrived from the network. The supervisor can use this method to simulate the arrival of a message from the network (for example, when restoring the networks after a context switch), by writing the values of the message to the **recv** register in the same order as they are to be read out. (An appropriate value should also be written to the **status** register to provide the corresponding auxiliary information.)

Note: An error is signaled if a value is written to the **recv** register when the receive FIFO is full (that is, when the **ni_rec_full** flag in the **private** register is set to 1 — see Section 2.7.5).

2.5 The Status Register

The **ni_***interface*_**status** register can be used to check on the progress of a message that is being sent, to detect when a message has been received, and to retrieve information about a received message. The **status** register includes the following flags and fields, which are the same for each of the network interfaces:

ni_*interface*_**status**	Status register.
ni_send_ok	Flag, status of message being sent.
ni_send_space	Field, space left in send FIFO.
ni_send_empty	Flag, indicates empty send FIFO.
ni_rec_ok	Flag, indicates arrival of a message.
ni_rec_length	Field, total length of received message.
ni_rec_length_left	Field, words left in receive FIFO.

Note: The **rec** status fields always reflect the "current" message in the receive FIFO — the message that includes the next word waiting to be read from the receive FIFO. If there is no pending message, the fields are undefined.

2.5.1 The "Send OK" Flag

If the send FIFO becomes full, all attempts to write a message (either to start or to continue one) cause the message currently being composed to be discarded. You can tell that a message has been discarded by examining the **send_ok** flag.

When the first value of a message is written to the **send_first** register, the **send_ok** flag is set to 1. As long as the message has not been discarded, this flag remains 1, indicating that the message is still being accepted. If the **send_ok** flag is still 1 after you have written the final value of a message, you can assume that that message has been accepted for delivery, and that you can start writing the next one. If the message is discarded, the **send_ok** flag is set to 0, indicating that the message has not been sent, and you should retry sending the entire message.

2.5.2 The "Send Space" Field and "Send Empty" Flag

The **send_space** field contains an *estimate* of the total space (in 32-bit words) left in the FIFO. The actual space remaining may be less; **ni_send_space** is usually correct, but may become invalid because of supervisor activity (such as when processes are swapped in and out). User code should not assume that pushing a message shorter than this value is always successful. The **send_empty** flag is 1 whenever the send FIFO is empty — that is, when there is no pending message in the FIFO.

Programming Note: NI programmers typically write an entire message to the send FIFO and then check the **send_ok** flag to see whether it was accepted, so the **send_space** field and **send_empty** flag typically aren't used.

2.5.3 The "Receive OK" Flag and "Receive Length" Fields

Whenever a message is pending in the receive FIFO, the **rec_ok** flag is set to 1, and remains 1 while any part of the message remains to be read from the FIFO. When no messages are waiting to be read, the flag is set to 0. (Attempting to read from the FIFO when **rec_ok** is 0 signals a Bus Error.)

The **ni_rec_length_left** field contains the number of words of the current message that are left in the receive FIFO. You can assume that it is safe to read this many words from the receive FIFO. If you need the message's original length, the **ni_rec_length** field always contains the total length (in words) of the current message *as it was when it was received.*

2.6 Abstaining from an Interface — The Control Register

Each of the Control Network interfaces has a control register, containing either one or two abstain flags. The names of the register and abstain flag(s) are:

ni_*interface*_control	Control register.
ni_rec_abstain	Normal receive abstain flag.
ni_reduce_rec_abstain	Combine reduction abstain flag.

Note: The global interface, always the exception, uses a different name for this register. See Section 4.3 for more information.

2.6.1 Effect of Abstain Flags

The **rec_abstain** flag, when set to 1, causes the NI to "abstain" from receiving messages via the corresponding interface. That is, the NI does everything necessary to ignore the interface's transactions:

- Arriving messages are simply ignored — they "disappear" with no indication of their arrival, and the **rec_ok** flag remains 0.
- Messages that require the participation of every node (global synch, etc.) are allowed to complete without the abstaining node's participation.
- Messages that require a value (scan messages, for example) are effectively given an appropriate identity value for the type of message being sent.

While the **rec_abstain** flag is set for a given interface, it is an error to try to send a message via that interface from the abstaining node. Attempts to write the **send_first** or **send** registers under these circumstances signals a Bus Error.

2.6.2 Combine Interface Abstain Flags

The **ni_reduce_rec_abstain** flag is only defined for the combine interface, and only applies to reduction operations.

In addition, reduction operations treat the value of the **rec_abstain** flag differently from all other interface operations.

For more information, see Section 4.2.8.

2.6.3 Use the Abstain Flags Safely

The abstain flag for a given interface should only be changed when that interface is not in use. Specifically, when a interface's abstain flag is changed:

- The send FIFO must be empty (that is, the `send_empty` flag must be 1).
- The receive FIFO must be empty (the `rec_ok` flag must be 0).
- There must be no messages in transit via that interface. (There is no flag to detect this; your program must simply be written so that this is the case.)

The effects of changing a interface's abstain flags while the interface is in use are unpredictable — your code may produce erroneous results, or signal an error.

This restriction generally requires that you use one of the interfaces (for example, the global interface) to synchronize the nodes and halt the operations of another interface while you change that interface's abstain flags.

For this reason, most NI programmers set the abstain flags once, at the beginning of a program or routine, and then leave them set that way until the program or routine finishes executing, changing the flags within the routine only where absolutely necessary.

2.6.4 Being a Good Neighbor

Important: Some programming systems (such as CMMD) use the abstain flags for their own purposes. These systems are written with the assumption that the abstain flags do not change unexpectedly, and if the flags do change these systems may not operate correctly.

When you alter the values of the abstain flags, you must take care to save the original settings of these flags and to restore them before handing control back to these systems. Failing to do so can cause either user or OS code to signal obscure errors that are hard to trace.

2.7 The Private Register

Each of the interfaces also has a "private" control register, containing a number of control flags and status fields for supervisor operations. Most of these subfields are interface-dependent; the few that are not are:

ni_*interface*_private	Private register.
ni_rec_ok_ie	Flag, "Receive OK" interrupt enable.
ni_lock	Interface lock flag.
ni_rec_stop	Interface stop flag (except Broadcast intf.).
ni_send_stop	Interface stop flag (Broadcast intf. only).
ni_rec_full	Flag, indicates receive FIFO is full.

The broadcast interface has one exception to the above description: the **ni_rec_stop** flag is not defined; in its place is a flag called **ni_send_stop**, which operates differently. (See Section 2.7.4.)

Usage Note: The private register is only accessible from the supervisor area; users without supervisor access must make a system call to change the flags in this register.

2.7.1 Message Receipt Interrupts — The Rec Interrupt Enable Flag

When the **ni_rec_ok_ie** flag is set to 1, a Green interrupt is signaled whenever a new message becomes available at the front of the interface's receive FIFO (in other words, whenever the **rec_ok** status flag is set to 1 for a new message).

A message may become available either by arriving from the network into an empty FIFO, or by being the next message in the FIFO when the last word of the current message is read out. A different Green interrupt is signaled for each network interface, and the interrupt for each interface can be independently enabled and disabled by setting the **rec_ok_ie** flag for the interface.

The Green interrupts that can be signaled are:

dr rec ok	**ldr rec ok**	**rdr rec ok**
bc rec ok	**sbc rec ok**	**com rec ok**

For more information about these interrupts, and about interrupts in general, see Section 5.1.

2.7.2 Clearing the Interface's Send FIFO — The Lock Flag

The supervisor can use the **ni_lock** flag to temporarily "lock" the interface — that is, prevent use of the interface in a way that is transparent to a user program.

The **lock** flag is normally 0. When it is set to 1, the following effects occur:

- Any message currently being written to the send FIFO is discarded.
- The **send_ok** flag is set to 0 and remains 0 — even if you attempt to write a new message to the send FIFO.
- The value of the **ni_*interface*_space** field is set to 0 and remains 0.

In other words, setting the **lock** flag to 1 clears the send FIFO, and then makes it seem as if the FIFO is permanently full.

2.7.3 Grabbing the Receive FIFO Registers — The Rec Stop Flag

The supervisor can temporarily grab control of a interface's receive FIFO and status register by setting the interface's **ni_rec_stop** flag. Since this involves the joint cooperation of the microprocessor and the NI, a special request/grant protocol is used. Specifically:

- The microprocessor *writes* a 1 to the interface's **rec_stop** flag, indicating it wants direct control of the **recv** and **status** registers. (**Note:** The **rec_stop** flag is not *changed* to 1 until the stop operation is completed.)
- If a message is currently arriving from the interface, the NI finishes receiving the message and stores it in the receive FIFO.
- The NI then stops receiving messages from the interface, and finally *sets* the **rec_stop** flag to 1, indicating that the stop operation is completed.

Once the **rec_stop** flag is set, the supervisor may freely read and write the values of the **recv** and **status** registers (for example, to push additional messages into the FIFO, or to clear the FIFO altogether). When the supervisor is finished with the **recv** and **status** registers, writing a 0 to the interface's **rec_stop** flag restores normal network operations.

Important: It is an error for the supervisor to attempt to write values to the **recv** and **status** registers while the **stop** flag is 0. The effect of doing so is undefined, but is not likely to be pleasant.

2.7.4 Blocking Unsent Broadcast Messages — The Send Stop Flag

The broadcast interface does not have a **rec_stop** flag. Instead, the same position in the **private** register is used for a flag called **ni_send_stop**, which has a different purpose. When the **send_stop** bit is set, it prevents any complete messages waiting in the broadcast send FIFO from being sent over the network. This mechanism is mainly used by the supervisor during process swaps, to hold messages in the interface send FIFO until they can be safely removed and saved.

2.7.5 Detecting a Full Receive FIFO — The Receive Full flag

The **ni_rec_full** flag, when set, indicates that the interface's receive FIFO is full. This is critical to network performance; if too many nodes have full receive FIFOs, the network can become clogged with unreceived messages, and this can prevent new messages from being delivered to their destinations — even if the destination nodes actually have sufficient space in their receive FIFOs.

2.8 Using a Generic Network Interface

To sum up, the strategy to use in accessing a network interface's registers is:

- To send a message, write the first word to the **send_first** register, and any remaining words to the **send** register.
- Check the **send_ok** flag to see if the message was discarded, and if so, retry sending the entire message.
- To receive a message, check the **rec_ok** flag to see if a message is in the FIFO, and if so, use the **length** and **length_left** fields to determine the number of words to read from the **recv** register.
- Use the remaining fields of the **status** register to obtain other interface-specific information about the state of the send and receive FIFOs.
- Use the **abstain** flag(s) in the **control** register to cause individual nodes to ignore the transactions of the interface.
- Use the **private** fields and flags for supervisor features such as disabling send FIFOs, checking for full receive FIFOs, and setting interrupts.

2.9 From the Generic to the Specific

The interface described in this chapter is an idealized view of a network interface, lacking a specific purpose, a detailed description of message protocol, or network-related restrictions on usage of the interface registers.

The next two chapters present a description of the Data Network and Control Network. These chapters present the purpose, protocol, and restrictions of each interface provided by the CM-5 networks, building on the generic interface description presented in this chapter.

Chapter 3

The Data Network

The Data Network consists of two halves, the *left interface* (LDR) and *right interface* (RDR). Each half of the network is connected to all nodes, and can be used independently. The two halves of the network can also be accessed together as the single *Data Network* (DR):

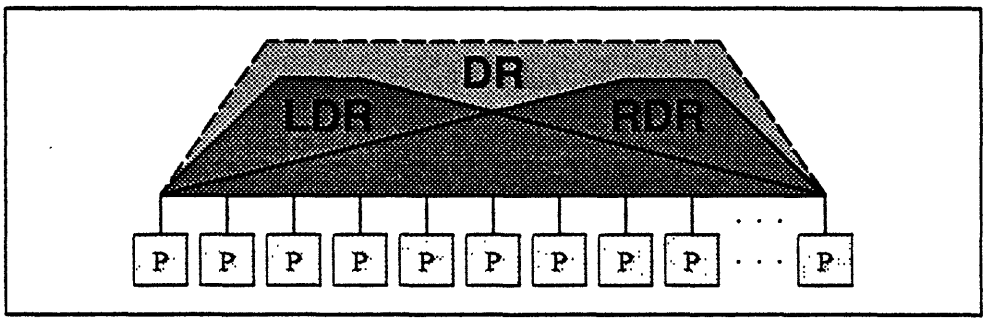


Figure 8. The three interfaces of the Data Network: DR, LDR, and RDR.

For each of these network interfaces there is a separate register interface. This chapter describes these register interfaces, and shows how to use them to send messages through the Data Network.

Terminology Note: The network acronyms (DR, LDR, RDR) are a historical anachronism, and are retained in this manual only because the C constants used to access the Data Network still refer to the three interfaces by the old abbreviations. In addition, the obsolete term "router" is occasionally still used in the programming contants to refer to the Data Network hardware. "Network" is currently preferred, as a more generic and thereby more accurate descriptive term.

3.1 The Data Network Register Interfaces

The three Data Network interfaces are based on the generic model presented in Chapter 2. There are three sets of interface registers: one for each half of the network (LDR and RDR), and one for the combined (DR) network.

Each network interface can be used to send and receive messages, with the following conditions:

- Sending a message via the DR actually sends it by either LDR or RDR, depending on the load of the two interfaces.
- In the current implementation, the DR interface cannot be used to receive any messages.
- The DR interface is mutually exclusive with the two half-network interfaces. In other words:
  - Writing a message to the DR send FIFO excludes using either the LDR or RDR at the same time. Likewise, writing a message to either the LDR or RDR send FIFOs excludes using the DR interface.
  - While a message is being sent, any excluded interface(s) remain excluded until the message has been written and accepted for delivery by the network. Also, the status register(s) of excluded interface(s) are invalidated and should not be used.
- The two half-network interfaces are not mutually exclusive, and in fact can be used simultaneously. In other words, network messages can be sent and received concurrently via both the LDR and RDR.

For each interface, the following registers are used to communicate with the Data Network:

ni_*dinterface*_send_first	Used to send the first value of a message.
ni_*dinterface*_send	Used to send the rest of the message.
ni_*dinterface*_recv	Used to receive a message.
ni_*dinterface*_status	Status register.
ni_*dinterface*_private	Supervisor control register.

The *dinterface* part of these names is a unique abbreviation for each interface:

dr – Data Network **ldr** – left interface **rdr** – right interface

Figure 9 is a memory map indicating the relative locations of these registers in the user and supervisor areas.

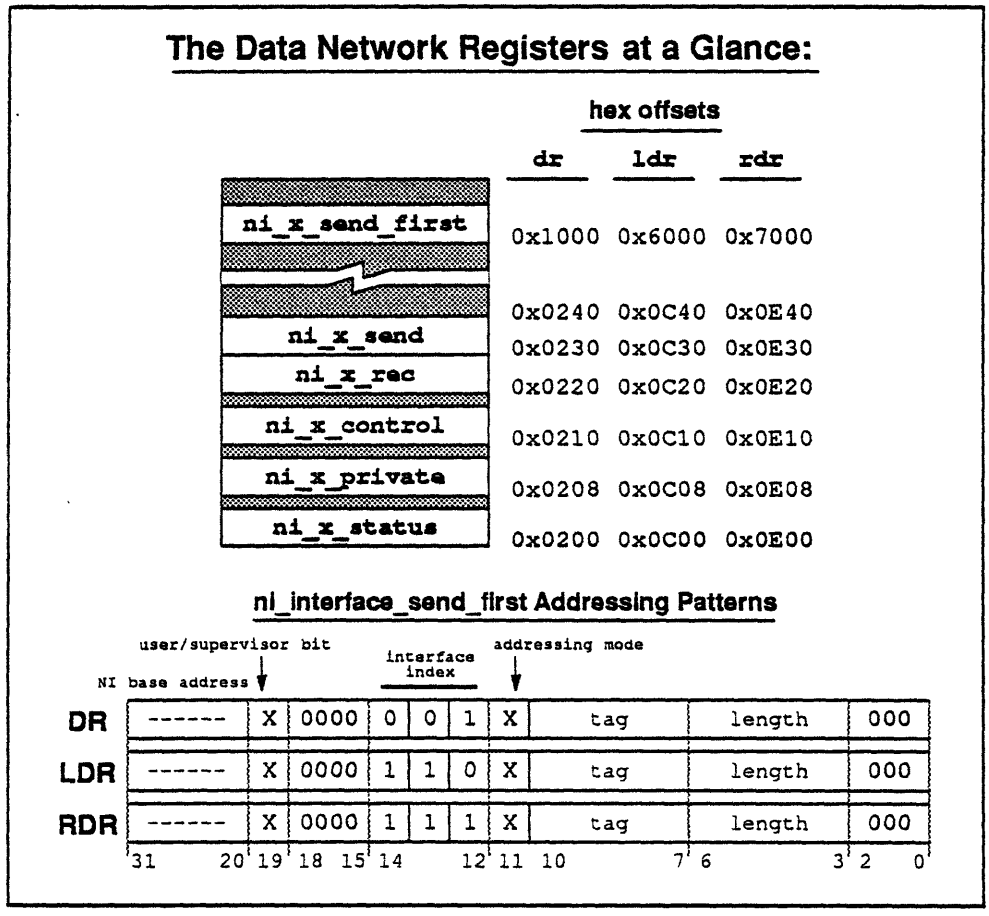


Figure 9. NI registers associated with each of the Data Network interfaces.

The following related registers are also used to control Data Network features:

ni_user_tag_mask	User/supervisor tag reservation register.
ni_rec_interrupt_mask	Contains tag value interrupt flags.
ni_dr_message_count	Contains current message count.
ni_count_mask	Contains tag-count enable flags.

The purpose and use of these registers is described in the sections below.

3.2 Data Network Messages

The Data Network is essentially asynchronous in operation — nodes can send and receive messages freely, so long as enough nodes are receiving messages so that the network does not become clogged (see Section 3.8).

The destination node of a Data Network message is determined by an address word that is added to to the message as it is being written to the send FIFO. (**Note:** The address word is removed in transit. It does *not* count as a message word with reference to the length limits of the send and receive FIFOs.)

Data Network messages are atomic; individual messages are not sent through the network until all the words of each message have been written into the send FIFO, and arrival of each message is not reported until all the words of the message have arrived in the receive FIFO.

The component words of a single Data Network message are always received in the same order as they were sent. However, if you use multiple Data Network messages as "packets" to send long messages from one node to another, the order in which the packets arrive is not guaranteed to be the same as the order in which they were sent.

Your code should not depend on having separate Data Network messages sent to the same node arrive in some predictable order. Instead, your code should include data in the packets (for example, an offset into the original message) that allows the receiving node to arrange the packets into the correct order.

3.3 Data Network Addressing

The Data Network uses two kinds of addressing: *physical* and *relative*. Each node of the CM-5 has a unique physical address based on its location in the CM-5 hardware. This represents an "absolute" address, giving the node's location with respect to the entire machine.

Each node also has a unique relative address based on its location in its partition. Relative addresses run from 0 (for the first node in the partition) up to one less than the total number of nodes in the partition. (See Figure 10.)

Note: The partition manager is always located at an address outside the partition, and so does not occupy any of the relative addresses of the partition. (For more information, see Section 7.1.)

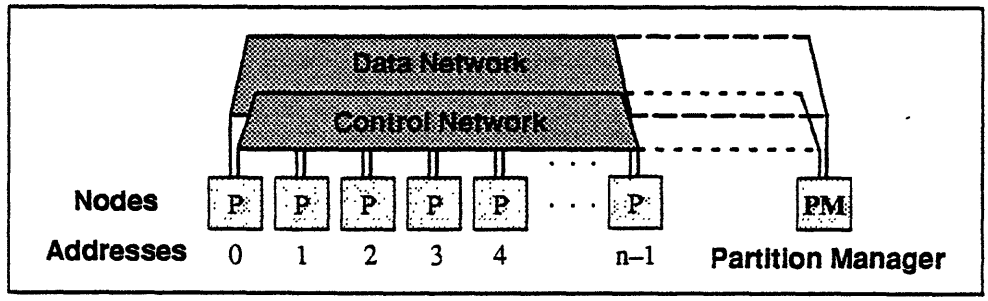


Figure 10. Relative addressing of nodes in a partition.

Just as there are two kinds of addressing, there are also two "modes" of sending a Data Network message: *physical* and *relative*. The mode a message is sent in is determined by a mode flag in the auxiliary data of the message.

When a message is sent in physical mode, its address word is treated as a physical address, and the message can be sent anywhere within the Data Network. (Only the supervisor is allowed to send messages in physical mode.)

When a message is sent in relative mode, the address word is treated as a relative address, and is translated into a physical address based on the current partitioning arrangement. This translation is performed automatically by the NI hardware, using a *chunk table*, described in Section 6.3.

The translation also includes automatic error checking to make certain that the supplied address is a legal relative address for the current partition. Messages that contain illegal relative addresses are not sent through the network; instead, the sending NI signals a Yellow interrupt (`bad relative address`).

For the Curious: The relative addresses in a partition are always contiguous — that is, there are no legal relative addresses in a partition that do not correspond to existing functional nodes. This is in contrast to physical addresses, which can contain gaps corresponding to nonfunctional nodes or to network locations that are not connected to actual CM-5 hardware. (See Section 6.3.)

3.4 Sending and Receiving Messages

The message format for all three Data Network interfaces is the same. The first word of the message is a 20-bit destination address that *must* be zero-extended to 32 bits. Failure to ensure that the address word is zero-extended to the full 32 bits can trigger a serious error, even causing your partition to crash.

The remaining words form the content of the message, which must be no longer than the length limit of the send FIFO.

Programming Note: The length limit of the Data Network send FIFOs is given by the constant `MAX_ROUTER_MSG_WORDS` (currently 5 for all three interfaces).

The auxiliary information of the message consists of the length of the message in words (excluding the address word), a 4-bit tag value, and an addressing mode flag that determines how the address word is interpreted.

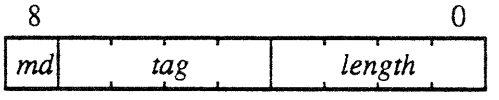
3.4.1 Sending Messages

The following FIFO registers are used to send messages:

`ni_`*dinterface*`_send_first`	Used for first value of a message.
`ni_`*dinterface*`_send`	Used for the rest of the message.

The protocol for sending a message is as described in Chapter 2.

The 9-bit auxiliary information field of the message has the form



where

- *md* is the addressing mode (0 = relative, 1 = physical)
- *tag* is the 4-bit tag value
- *length* is the length of the message in words, excluding address word

The following constants specify the starting bit positions of these fields:

`NI_DR_SEND_AUXILIARY_ADDRESS_MODE_P`	The *md* field offset (8).
`NI_DR_SEND_AUXILIARY_TAG_P`	The *tag* field offset (4).
`NI_DR_SEND_AUXILIARY_LENGTH_P`	The *length* field offset (0).

To construct a **send_first** address, add the following values:

The *md* flag:	*md*	<< **NI_DR_SEND_AUXILIARY_ADDRESS_MODE_P**
The *tag* value:	*tag*	<< **NI_DR_SEND_AUXILIARY_TAG_P**
The *length* value:	*length*	<< **NI_DR_SEND_AUXILIARY_LENGTH_P**

The *md* flag is 0 for a message with a relative destination address, and 1 for a message with a physical destination address.

The following constants can be used to specify the *md* flag:

RELATIVE	Relative node addressing (0).
PHYSICAL	Physical node addressing (1).

Note: Sending messages with physical addresses is reserved for the supervisor. If user code tries to send a message with a *md* flag of 1, a Bus Error is signaled.

The *tag* can be any value from 0 to 7 inclusive for user messages, or from 0 to 15 for supervisor messages. Message tags are described in more detail in Section 3.5.1 below.

The *length* field can have any value from 1 up to **MAX_ROUTER_MSG_WORDS**.

3.4.2 Receiving Messages

For each interface, the following register is used to receive messages:

ni_*dinterface***_recv**	FIFO register from which values are read.

Data Network messages are received as described in Chapter 2.

Supervisor Usage Note: Currently, a hardware defect in the NI chip does not allow the Data Network **recv** registers to be written by the supervisor to simulate the arrival of messages, etc. The workaround is for a node to send a message into the network using its own address as the destination. Assuming the network is clear (as it is, for example, during context switches) this causes the message to be delivered to the front of the node's receive queue.

3.5 The Status Register

The status register for each of the networks contains the following subfields:

ni_*dinterface*_status	Status register.
ni_send_ok	Flag, status of message being sent.
ni_send_space	Field, space left in send FIFO.
ni_rec_ok	Flag, indicates receipt of message.
ni_rec_length	Field, total length of message.
ni_rec_length_left	Field, words left in the FIFO.
ni_dr_rec_tag	Field, tag value of the message.
ni_dr_send_state	Field, status of send FIFOs.
ni_dr_rec_state	Field, status of receive FIFOs.
ni_router_done_complete	Flag, indicates empty send FIFOs.

The **send_ok**, **send_space**, **rec_ok**, **rec_length**, and **rec_length_left** subfields are as described in Chapter 2. The remaining fields are described in the sections below.

Note: The subfields **ni_dr_send_state** and **ni_dr_rec_state**, and the flag **ni_router_done_complete** apply to all three interfaces. They are only accessible from the DR interface (that is, their values are only defined for the **ni_dr_status** register).

3.5.1 Message Tags

The tag values of Data Network messages are used to distinguish between different types of Data Network messages. The **status** register field **rec_tag** always contains the tag value that was sent with the current message.

Tag values are primarily used for:

- distinguishing between user and supervisor messages
- causing interrupts to be signaled when messages are received
- helping the NI determine when the Data Network is clear of user messages

Some tag values are reserved for supervisor use, to distinguish between supervisor and user messages. The remaining tags can optionally be used in user programs to distinguish different types of user messages.

User/Supervisor Tag Reservation

The NI has a register that controls the reservation of tag values:

`ni_user_tag_mask`	User/supervisor tag reservation register.

Only the low-order 16 bits of this register are used, one for each of the possible tag values (0 to 15). If the *n*th bit of the **`user_tag_mask`** register is 1, then tag value *n* is reserved for supervisor use.

Since the **`tag_mask`** register is only accessible by the supervisor, it effectively acts as a set of permission switches, controlling which tags the supervisor allows user messages to have. If a user program attempts to send a message with a supervisor-reserved tag, a Bus Error is signaled.

Tag Fields and Interrupts

Tag values can be used to trigger interrupts; when a message with an interrupting tag value becomes available for reading in the receive FIFO, the NI signals a Green interrupt (**`dr rec tag`**) to the microprocessor. (A message becomes available either by arriving at an empty receive FIFO, or by being the next message in the FIFO when the current message is read out.) Tag value interrupts can be used to cause the microprocessor to execute a specific section of code whenever a message with an interrupting tag becomes available for reading.

The following register is used to determine which tag values cause interrupts:

`ni_rec_interrupt_mask`	Register, contains tag value interrupt flags.

The **`interrupt_mask`** register contains 16 flags, one for each tag value. If the *n*th bit is 1, then a message with tag value *n* signals a Green interrupt on arrival.

For CMOST Users: You can use CMOST commands to instruct the NI to signal an interrupt when it receives a message with a specific tag. This interrupt causes the processing node to execute a specific routine of your program.

The **`CMOS_signal`** system call is used to set up an interrupt:

```
CMOS_signal( signal, user_function, tag_mask )
```

The *signal* argument is the signal type, and must be the predefined constant **`SIGMSG`**. The *user_function* argument is the name of a user-defined function that should handle receiving and processing the message.

The *tag_mask* argument is a 16-bit field, one bit for each possible value of the tag. If bit *n* in this mask is set, then the receipt of a message with a tag of *n* causes *user_function* to be executed. (Remember that you are limited to using only the first four bits of this mask, corresponding to the tags 0 through 7.)

So, for example, the function call

```
CMOS_signal( SIGMSG , my_msg_handler , 0xFE);
```

arranges the NI interrupt system so that when a Data Network message with a tag from 1 to 7 is received, the user-defined procedure **my_msg_handler** is called.

Note: To use this function, you must **#include** the file **cm/cm_signal.h**. For more information on **CMOS_signal**, see the UNIX manual page for the function.

Tag Fields and the Message-Counting Registers

Tag fields also allow system software to automatically maintain a count of messages sent and received by the NI. This is a key part of the network-done feature of the Control Network (see Section 4.2.7). It allows the NI to determine quickly when the Data Network is clear of user messages. Two registers are used to control this message-counting feature:

ni_dr_message_count	Register, contains current message count.
ni_count_mask	Register, contains tag-count enable flags.

Message Count Disabling

The **ni_dr_message_count** register contains a signed 32-bit integer value that is incremented when a Data Network message is sent (by any of the three interfaces), and decremented when a message is received.

When the **message_count** register becomes zero for all non-abstaining nodes, the NI assumes that there are no countable messages in transit in the Data Network. It is possible to disable message counting for messages with specific tag values. (This is useful, for example, if you only wish to keep a count of user messages, and want supervisor messages to go uncounted.)

The **ni_count_mask** register controls this enabling and disabling of message counting. It contains 16 flags, one for each tag value. If the *n*th **count_mask** bit is 1, then messages with a tag of *n* are counted by **ni_dr_message_count**. If the *nth* bit is zero, messages with that tag are *not* counted.

It's important to be sure that the sending and receiving nodes for a message both agree on whether the message's tag should or should not be counted; if they do not agree, the **ni_dr_message_count** register's value is useless, and can wrap around, becoming negative — see the discussion of this situation below.

Note: The supervisor can write a value to **ni_dr_message_count**, for example, to set the register back to zero, but this should only be done when the Data Network is not in use. Otherwise, there is no way to guarantee that the value of this register remains the same as the value that was written into it.

Negative Message Count Interrupts

If the sum of the **message_count** registers for all nodes becomes negative, it means that either a message was lost in transit or was counted incorrectly. If the global **message_count** sum is negative when a Data Network operation is attempted, a Yellow interrupt (**dr count negative**) is signaled. (See Section D.3.3 in Appendix D.)

Note: If the **message_count** register is incremented or decremented beyond its 32-bit signed value capacity, its value "wraps around," becoming negative. However, the register is large enough that this should not happen unless there is a serious error (a hardware problem that causes messages to be lost, nodes that do not agree on counting of tag messages, etc.).

3.5.2 IMPORTANT — Check the Tag before Receiving a Message

Tag values are not mandatory. You can, for instance, simply supply a tag value of 0 for all Data Network messages. However, this does not mean that you can simply ignore tag values altogether. The CM-5 operating system itself sends Data Network messages with interrupt tags. Whether or not you use tags yourself, you must always check the tag field of a Data Network message before retrieving it, so that you do not accidentally read a message intended as an interrupt.

The Data Network only checks the tag field of a message *after* the message has been delivered to the receive FIFO. If the message has a tag that is set to signal an interrupt (either by the user or by the supervisor), the appropriate interrupt is signaled, with the assumption that the interrupt handler takes care of removing the message from the FIFO.

This means that if you're not careful, you can accidentally read a message with an interrupt-triggering tag value *before* the NI has signaled the interrupt. The effect of doing so is unpredictable; an error may be signaled, or your partition may crash. To avoid this problem, always check the tag of a Data Network message *before* retrieving it, to make certain that it is neither a supervisor message or a message with a tag value that you have assigned to trigger an interrupt.

3.5.3 The Send and Receive State Fields

The DR interface is mutually exclusive with the LDR and RDR interfaces. It is an error to try to write a message to the DR send FIFO while there is a partially completed message in either the LDR or RDR send FIFOs.

Likewise, having a partially completed message in the DR send FIFO makes it an error to try to send a message via the LDR or RDR FIFOs. In either case, the status registers and FIFOs of the excluded interface(s) are invalidated.

You can use the **`ni_dr_send_state`** field to determine which interfaces are in use. The value of this field is an integer from 0 to 2, with the following meanings:

0 No partial messages in any send FIFO.
1 Partial message in the DR send FIFO.
2 Partial message in either or both of the LDR or RDR send FIFOs.

There is also a corresponding **`ni_dr_rec_state`** field that you can use to determine which receive interfaces are in use. (However, because the DR interface cannot be used to receive messages, this field is not as useful as **`ni_dr_send_state`**.)

The value of the **`ni_dr_rec_state`** field is again an integer from 0 to 2:

0 No partial messages in any receive FIFO.
1 Reserved. (The DR interface cannot receive messages.)
2 Partial message in either or both of the LDR or RDR receive FIFOs.

Note: The two half-network interfaces are not mutually exclusive. There is no restriction on having partially completed messages simultaneously in the LDR and RDR FIFOs. (This kind of simultaneous message sending is one reason that the LDR and RDR interfaces exist.)

3.5.4 The Network-Done Flag

The **ni_router_done_complete** flag is used by the Control Network as part of its network-done message function. This feature is designed to make it easy to synchronize the nodes after a Data Network operation.

As noted above, the message-counting register **ni_dr_message_count** also plays a part in the network-done feature. For more information on network-done messages, see Section 4.2.7.

3.6 The Private Register

The **private** register for each of the network interfaces contains the following subfields:

ni_*dinterface*_private	Private register.
ni_rec_ok_ie	Flag, "Receive OK" interrupt enable.
ni_lock	Interface lock flag.
ni_rec_stop	Interface stop flag.
ni_rec_full	Flag, indicates receive FIFO is full.
ni_dr_rec_all_fall_down	Flag, set for All Fall Down message.
ni_all_fall_down_ie	All Fall Down interrupt enable flag.
ni_all_fall_down_enable	Flag, triggers All Fall Down mode.

The **rec_ok_ie**, **lock**, **rec_stop**, and **rec_full** subfields are as described in Chapter 2. The remaining three fields are used to control the All Fall Down mode feature of the Data Network, as described in Section 3.7 below.

Note: The subfield **ni_rec_stop** is only accessible from the DR interface (that is, its value is only defined for the **ni_dr_private** register).

3.7 All Fall Down Mode

All Fall Down mode is a feature of the Data Network that is used primarily by the supervisor for swapping processes out of partitions. When All Fall Down mode is triggered within a partition of the Data Network, all messages currently in transit within that partition are immediately routed downwards through the network to the nearest possible node, regardless of their actual destination. This process clears the Data Network of pending messages as swiftly as possible.

The three `private` register subfields, `ni_dr_rec_all_fall_down`, `ni_all_fall_down_ie`, and `ni_all_fall_down_enable`, are used to trigger All Fall Down mode, as well as to detect when an arriving Data Network message is the result of All Fall Down mode.

3.7.1 Triggering All Fall Down Mode

To trigger All Fall Down mode in a partition, each node in the partition should set its `ni_all_fall_down_enable` flag to 1. This informs the Data Network hardware that the NIs are ready to receive All Fall Down messages.

For the Curious: The Data Network is organized in layers, with each layer managed by internal switching nodes. When All Fall Down mode is started by the nodes, it is broadcast through all the layers of the Data Network, causing the internal switching nodes to begin routing messages downward and out of the network. The Data Network is designed in a fault-tolerant manner, so that even if a given Data Network switching node is not yet in All Fall Down mode, an All Fall Down message sent through it by a higher level node "falls through" and continues moving toward the processing nodes.

3.7.2 Detecting All Fall Down Mode Messages

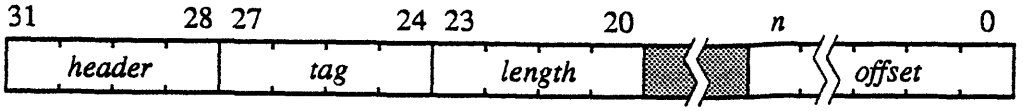
The flag `ni_dr_rec_all_fall_down` is set whenever the current message in the receive FIFO is the result of an All Fall Down operation.

You can also have the NI trigger an interrupt when an All Fall Down message becomes available in the receive FIFO (either by arriving at an empty FIFO, or by being brought forward after a preceding message has been read out). If the interrupt enable flag `ni_all_fall_down_ie` is set, the arrival of an All Fall Down message triggers a Green interrupt (`dr rec all fall down`).

3.7.3 Resending All Fall Down Mode Messages

Each message re-routed by All Fall Down mode carries with it enough information so that the receiving node can resend the message to its intended destination. When an All Fall Down message is read from the receive FIFO, the first word read is not the first word of the message itself, but is an extra address word, containing information about the intended destination of the message.

The All Fall Down address word has the following format:



where

- *header* is a 4-bit header giving the length of the *offset* field
- *tag* is the original tag field of the message
- *length* is the message length in words, excluding the address word
- *offset* is an *n*-bit field used to construct the real address

The *header* field indicates the length of the *offset* field, but in a slightly convoluted manner. The length of the *offset* field, *n*, is 4 times the least integer not less than one-half of the *header* value, *h*. In symbols:

$$n = 4\left\lceil \frac{h}{2} \right\rceil$$

(An algorithmic way to get ths result is to take bits 29 – 31 of the *header* field as an integer, arithmetically add bit 28, and left-shift the result by two bits.)

Once you have the *offset* length, take the physical address of the current node and replace the least significant *n* bits with the *n*-bit value from the *offset* field. This gives the destination physical address. For example, if the *header* value is 1, then the offset is 4 bits in length. If the *offset* value is 0xC, and the physical address of the current node is 0x00111, then the destination physical address is 0x0011C.

The *tag* and *length* fields duplicate the values obtainable from the `rec_tag` and `rec_length` fields in the `status` register. However, these fields are included in the All Fall Down address word because programmers may find them useful.

Note: When an All Fall Down message is received, the value of the `rec_length` field is equal to the original length of the message — the number of data words in the FIFO *not counting* the All Fall Down address word. However, the `rec_length_left` field contains the *total* number of words left in the receive FIFO, and this count *includes* the All Fall Down address word.

3.8 Data Network Usage Note: Receive before You Send

An important strategy to keep in mind when using the Data Network is "Receive before you send." That is, in most cases you should structure your code so that:

- Each node attempts to read a message from the Data Network before sending a new message into it.
- If a node is unable to send a message, the node attempts to read a message to help decrease the network load.

While the Data Network has a large capacity for messages from nodes, the sheer number of nodes connected to it can simply overwhelm it if the nodes repeatedly send messages into the network without attempting to receive them. For this reason, your code should be biased towards removing messages from the network rather than adding them.

However, your code should also provide fair opportunities for both receiving and sending, where "fair" means that the ratio between the two actions should be bounded both below and above, and where "opportunity" means the opportunity to attempt sending or receiving a message, *whether or not* the attempt is successful. Thus, the sending and receiving portions of your code should be called with fairly equal frequency.

When you are using the LDR and RDR concurrently, you should likewise maintain a balance in using both interfaces, so that neither interface becomes more heavily loaded than the other.

In short, the rule of thumb is: "Receive before you send, but receive and send fairly."

Note: Some applications use the LDR and RDR interfaces for completely different purposes, and thus do not normally maintain a load balance between the two halves of the Data Network (that is, one network interface may be used less often than the other). Nevertheless, such application code should still try to maintain a receive/send balance within each of the two network interfaces.

Chapter 4

The Control Network

The Control Network consists of three interfaces, the broadcast interface (BC), the combine interface (COM), and the global interface:

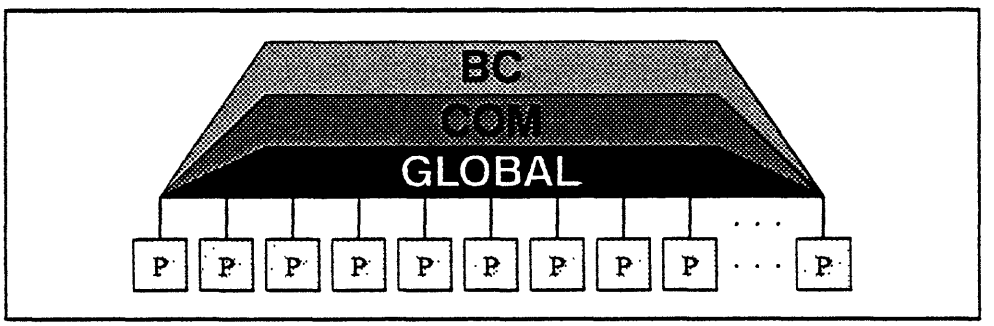


Figure 11. The three interfaces of the Control Network: BC, COM, and global.

The broadcast and combine interfaces are very similar, and there are some internal interactions between these two interfaces that you'll need to keep in mind. The global interface, however, is different in both structure and purpose from either of the other two interfaces.

This chapter describes the three Control Network interfaces, and presents the registers that are used to manipulate them.

4.1 The Broadcast Interface

The broadcast interface is used to broadcast a message from a single source node to all nodes in the same partition (including the broadcasting node).

The broadcast interface provides two separate register interfaces, one for user broadcasts (BC), and one for supervisor broadcasts (SBC). The two register interfaces are completely independent, and can be used concurrently to broadcast messages. Where the sections below refer to "broadcast messages" generically, the description applies equally and independently to both the user and supervisor interfaces.

Implementation Note: Because of the way the broadcast and combine interfaces interact, if a node is abstaining from a combine operation, that node should *not* execute a broadcast operation until the combine operation is completed. (For more information, see Section 7.3.8.)

4.1.1 Broadcast Register Interfaces

The two broadcast register interfaces are based on the generic model presented in Chapter 2. The only difference between them is that the supervisor broadcast registers can only be accessed from the supervisor area.

The following NI registers form the broadcast interface:

`ni_`*`binterface`*`_send_first`	Used to send the first value of a message.
`ni_`*`binterface`*`_send`	Used to send the rest of the message.
`ni_`*`binterface`*`_recv`	Used to receive a message.
`ni_`*`binterface`*`_status`	Status register.
`ni_`*`binterface`*`_control`	Control register.
`ni_`*`binterface`*`_private`	Supervisor control register.

The *binterface* part of these names is a unique abbreviation for each interface:

`bc` – user broadcast interface `sbc` – supervisor broadcast interface

The purpose and use of each of these registers is described in the sections below. Figure 12 contains a memory map showing the relative locations of these registers in the user and supervisor areas.

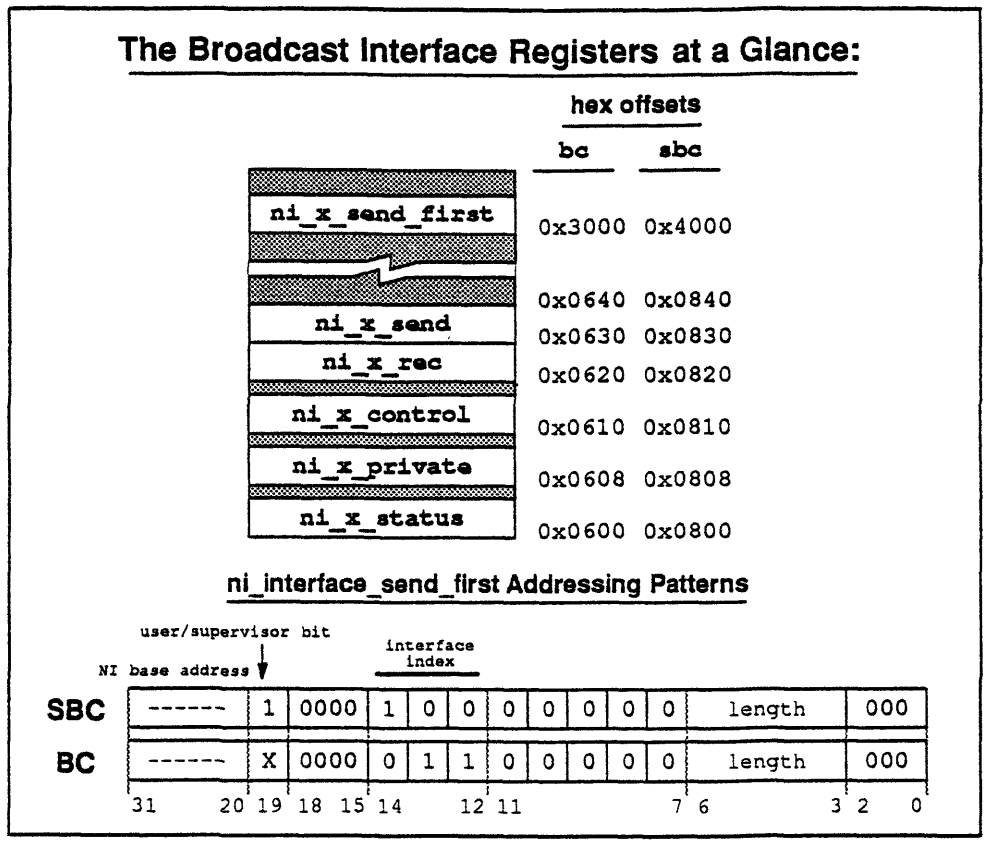


Figure 12. NI registers associated with each of the broadcast interfaces.

4.1.2 Broadcast Messages

The broadcast interface is essentially synchronous in operation — a single node broadcasts a message that is received by all nodes in its partition (including the broadcasting node itself).

Only one node in each partition can broadcast by a given interface at any time. If two or more nodes in the same partition attempt to broadcast simultaneously, via the same interface (user or supervisor), the effect is unpredictable. An error may be signaled and/or transmitted data may be lost. (Remember, however, that the user and supervisor broadcast interfaces operate independently, and can be used concurrently by different nodes in the same partition.)

Broadcast messages are atomic with respect to sending; a broadcast message is not transmitted until all its component words have been written to the send FIFO. Broadcast messages are not atomic in transit, however. A multiword message may be split in transit into two or more smaller messages. Additionally, as broadcast messages arrive at each node they are concatenated together in the receive FIFO.

From the point of view of each receiving node, it always appears as if there is exactly one broadcast "message" waiting to be read from the receive FIFO. (Once a node begins receiving a message, however, the length of the message is fixed, and a new "message" is formed behind it in the FIFO from any words that arrive while the first message is being read out.)

Although the length of a broadcast message is not maintained, the *order* of the words within a message is maintained, as well as the order of messages sent and received via the same interface, user or supervisor. (There is no predictible relationship, however, between the deliveries of user and supervisor messages to the same node. Effectively, the two interfaces act as independent "streams" of messages.)

Implementation Note: The broadcast interface is designed in such a way that a message is not removed from the send FIFO before all non-abstaining nodes have received it. This feature can be used to force synchronization of the nodes.

4.1.3 Sending Broadcast Messages

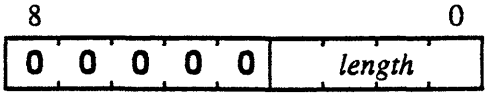
A broadcast message consists of a series of one or more words. The maximum length allowed for a message is determined by the length limit of the send FIFOs. The only auxiliary information associated with a broadcast message is its length. However, the length is only meaningful for the node that sends a message, because of the way messages can be split and concatenated in transit.

Programming Note: The length limit of the broadcast send FIFOs is given by the constants `MAX_BROADCAST_MSG_WORDS` and `MAX_SBC_MSG_WORDS` (currently 4 for both interfaces).

The following FIFO registers are used to send messages:

`ni_`*binterface*`_send_first`	Used to send the first value of a message.
`ni_`*binterface*`_send`	Used to send the rest of the message.

The auxiliary data field of a broadcast message (BC or SBC) has the form



where *length* is the length of the message in words. The *length* field can have any value from 1 up to **MAX_BROADCAST_MSG_WORDS** or **MAX_SBC_MSG_WORDS**. (The high-order bits of the auxiliary data have no useful meaning, but must always be specified as 0.)

The following constant specifies the starting bit position of the *length* field:

NI_BC_SEND_AUXILIARY_LENGTH_P	The *length* field offset (0).

Implementation Note: Each broadcast interface's **private** register includes a supervisor flag, **ni_send_enable**, which controls whether broadcast sending is permitted via that interface. In the current CM-5 OS implementation, these flags are turned off by default, and must be enabled before broadcast sending is attempted. (See Section 4.1.7 for a description of these flags.)

4.1.4 Receiving Broadcast Messages

Broadcast messages are received as described in Chapter 2. For each broadcast interface, the following register is used to receive messages:

ni_*binterface*_recv	FIFO register from which values are read.

4.1.5 The Broadcast Status Register

The status registers for each of the interfaces contain the following subfields:

ni_*binterface*_status	Status register.
ni_send_ok	Flag, status of message being sent.
ni_send_space	Field, space left in send FIFO.
ni_send_empty	Flag, indicates empty send FIFO.
ni_rec_ok	Flag, indicates receipt of message.
ni_rec_length_left	Field, words left in the FIFO.

The meanings of these subfields are as described in Chapter 2.

How to Interpret the Value of the "Length Left" Field

The NI combines broadcast messages as they are received, so there is never more than one "message" waiting to be read from the receive FIFO. However, broadcast messages are never appended to a message that is in the process of being retrieved, so you needn't worry that a message will grow unexpectedly.

Once you have retrieved the first value of a received message, it is safe to assume that reading a number of words equal to the **rec_length_left** value retrieves the rest of the message. (Remember, however, that this method is not guaranteed to read all words of a multiword message that was divided in transit.)

4.1.6 Abstaining from the Broadcast Interface

Each broadcast interface has an abstain flag that you can use to cause the NI to ignore incoming broadcast messages. The abstain flag's effects and use are as described in Section 2.6.

ni_*binterface*_control	Status register, contains **rec_abstain** field.
ni_rec_abstain	Flag, broadcast interface abstain flag.

4.1.7 The Broadcast Private Register

The private register for each broadcast interface contains the following subfields:

ni_*binterface*_private	Private register.
ni_rec_ok_ie	Flag, "Receive OK" interrupt enable.
ni_lock	Interface lock flag.
ni_send_stop	Interface stop flag.
ni_rec_full	Flag, indicates receive FIFO is full.
ni_send_enable	Flag, enables/disables send FIFO.

The **rec_ok_ie**, **lock**, **send_stop**, and **rec_full** subfields are as described in Chapter 2. The remaining field is described below.

The Send Enable Flag

Each broadcast interface has an **`ni_send_enable`** flag, which is used to enable and disable the broadcast send FIFO. When this flag is set to 1, message sending is permitted. When the flag is set to 0, an attempt to write a message to the send FIFO signals a Bus Error. The **`send_enable`** flag should only be changed when there are no broadcast messages pending for the interface.

Usage Note: While this flag can be used as a kind of "send abstain" flag to ensure that only one node broadcasts at any given time (that is, by disabling sending for all nodes but the one making the broadcast), it is much simpler to structure your code so that only one node is permitted to broadcast at any time.

Important: The CMOST operating system sets this flag to 0 by default. The flag must be set to 1 to permit broadcasting of messages.

4.2 The Combine Interface

The combine interface is used for executing operations that combine in parallel a single value from each processing node.

The supported operations are:

- parallel prefix (scanning), which performs a cumulative operation (addition, maximum, logical AND, etc.) over the values from each node in either increasing or decreasing order of send addresses
- reduction, which combines the values from all the nodes and then returns this single combined result to all participating nodes
- network-done, which simplifies the task of synchronizing the nodes after a Data Network operation

Each operation is described in more detail below.

Implementation Note: Because of way the broadcast and combine interfaces interact, if a node is abstaining from a combine operation, that node should *not* execute a broadcast operation until the combine operation is completed. (For more information, see Section 7.3.8.)

4.2.1 The Combine Register Interface

The combine interface's register interface is based on the generic model presented in Chapter 2, and includes the following registers:

ni_com_send_first	Used to send the first value of a message.
ni_com_send	Used to send the rest of the message.
ni_com_recv	Used to receive a message.
ni_com_status	Status register.
ni_com_control	Control register.
ni_com_private	Supervisor control register.
ni_scan_start	Control register used to set scanning segments.

The purpose and use of each of these registers is described in the sections below. Figure 13 contains a memory map showing the relative locations of these registers in the user and supervisor areas.

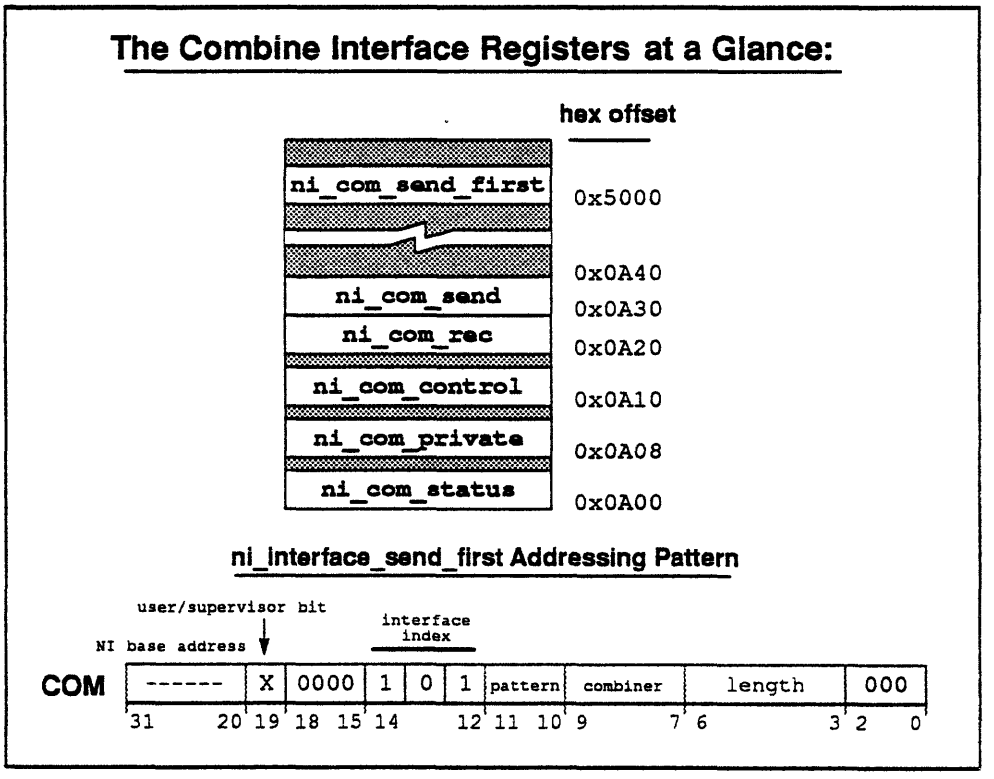


Figure 13. NI registers associated with the combine interface.

4.2.2 Combine Messages

The combine interface is essentially synchronous — combine operations are not completed until all non-abstaining nodes have sent the *same* type of combine operation. If two nodes attempt to start different combining operations at the same time, a Yellow interrupt (`bc or com collision`) is signaled. Once this interrupt has been signaled, combine messages are no longer guaranteed to be valid — it is necessary to flush the Control Network to restore normal operation (see the discussion of Control Network flushing in Section 6.4).

Combine messages are atomic in both sending and receiving; a combine message is not transmitted until all its component words have been written to the send FIFO, and arrival of each message is not reported until all the words of the message have arrived in the receive FIFO.

The order of combine messages is strictly preserved in transit. With the exception of the network-done operation, which uses a different mechanism, the results of combine operations are delivered into the receive FIFO in the same order the operatons were started.

Combine operations can be pipelined. Although all nodes must start the same combine operation in order for that operation to complete, nodes are not required to read the results of each combine message before sending the next. The length of the pipeline is limited only by the capacity of the message FIFOs.

Important: Pipelined messages cannot use doubleword read/write operations.

4.2.3 Sending Combine Messages

A combine message consists of a series of one or more words, with the exception of network-done messages, which are always 1 word in length. The maximum length allowed for a message is determined by the length limit of the send FIFO.

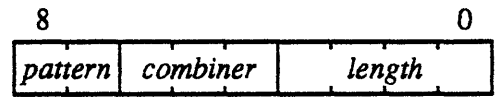
Programming Note: The length limit of the combine interface send FIFO is given by the constant `MAX_COMBINE_MSG_WORDS` (currently 5).

The following FIFO registers are used to send messages:

`ni_com_send_first`	Used to send the first value of a message.
`ni_com_send`	Used to send the rest of the message.

The auxiliary information has three components: the length of the message in words, a three-bit *combiner* value, and a two-bit *pattern* value. (The legal *combiner* and *pattern* values are described below.)

The auxiliary data field of the message has the form



where

- *pattern* is a two-bit value selecting the order in which values are combined
- *combiner* is a three-bit value selecting the combine operation performed
- *length* is the length of the message in words

The following constants specify the starting bit positions of these fields:

`NI_COM_SEND_AUXILIARY_PATTERN_P`	The *pattern* field offset (7).
`NI_COM_SEND_AUXILIARY_COMBINER_P`	The *combiner* field offset (4).
`NI_COM_SEND_AUXILIARY_LENGTH_P`	The *length* field offset (0).

To construct a send_first address, add the following values:

The *pattern* value:	*pattern*	`<< NI_COM_SEND_AUXILIARY_PATTERN_P`
The *combiner* value:	*combiner*	`<< NI_COM_SEND_AUXILIARY_COMBINER_P`
The *length* value:	*length*	`<< NI_COM_SEND_AUXILIARY_LENGTH_P`

For scan and reduction operations, the legal *pattern* and *combiner* values are:

pattern

1 — Backward scan (combine in decending order of node address).
2 — Forward scan (combine in increasing order of node address).
3 — Reduction operations.

combiner:

0 — Bitwise inclusive OR.
1 — Signed addition.
2 — Bitwise exclusive OR.
3 — Unsigned addition.
4 — Signed maximum.

A *pattern* value of 0, together with a *combiner* value of 5, specifies a network-done operation, described later in this chapter.

The *combiner* values 6 and 7 are not currently used.

The following constants can be used to specify the value of the *pattern* field:

SCAN_FORWARD	Forward scan pattern (2).
SCAN_BACKWARD	Backward scan pattern (1).
SCAN_REDUCE	Reduction scan pattern (3).
SCAN_ROUTER_DONE	Network-done operation (0).

The following constants can be used to specify the value of the *combiner* field:

OR_SCAN	Inclusive OR (0).
ADD_SCAN	Signed addition (1).
XOR_SCAN	Exclusive OR (2).
UADD_SCAN	Unsigned add (3).
MAX_SCAN	Signed maximum (4).
ASSERT_ROUTER_DONE	Network-done operation (5).

The *length* field can have any value from 1 up to **MAX_COMBINE_MSG_WORDS**.

4.2.4 Receiving Combine Message

The message-receiving interface of the combine interface is as described in Chapter 2, with the exception of the network-done operation, which is received through the Data Network status field **ni_router_done_complete**.

The following register is used to receive combine messages:

ni_com_recv	FIFO register from which values are read.

4.2.5 The Combine Status Register

The combine status register contains the following subfields:

ni_com_status	Status register.
ni_send_ok	Flag, status of message being sent.
ni_send_space	Field, space left in send FIFO.
ni_send_empty	Flag, indicates empty send FIFO.
ni_rec_ok	Flag, indicates receipt of message.
ni_rec_length	Field, length of message in words.
ni_rec_length_left	Field, words left in the FIFO.
ni_com_scan_overflow	Flag, indicates add-scan overflow.

The `send_ok`, `send_space`, `send_empty`, `rec_ok`, `rec_length`, and `rec_length_left` subfields are as described in Chapter 2. The remaining flag, `com_scan_overflow`, is described in Section 4.2.6.

4.2.6 Scanning (Parallel Prefix) and Reduction Operations

In a scan or reduction operation, each node sends a single value that is combined with the values sent by the other nodes in the partition.

When each participating node has sent a value, the values are combined according to the *combiner* and *pattern* in the auxiliary data of the message, and the result is delivered after a brief interval to the receive FIFOs of the nodes.

For scan operations, the node values are combined cumulatively — that is, the result for each node is the combination of the values transmitted by all nodes having lower (or higher) relative addresses. Forward scans combine values in order of ascending node addresses. Backward scans combine values in order of descending node addresses.

Reduction is a special case of scanning. When a reduction message is sent, the values from all participating nodes are combined into a single value, and then this single result is sent to all the nodes.

Important: If you are sending a message that is longer than one word, the order in which the words of the message are written depends on the *combine* operation:

- Maximum operations require the most significant word to be written first.
- Both types of addition require the least significant word to be written first.
- Inclusive and exclusive OR have no word-ordering requirement.

Scanning with Segments

You can use segmented scanning to divide a partition into *segments* of nodes — regions of nodes within which forward and backward scanning is done independently of all other nodes in the partition. The scan values obtained within each segment do not affect the values obtained in any other segment.

Note: Reduction operations do not use segmented scanning. Reduction scans ignore the current segment settings.

The following control register is used to read and set the current segmentation:

ni_scan_start One-bit control register, indicates start of scan segments.

The one-bit flag in **ni_scan_start** is used to indicate the starting points of segments. Segments begin in each node where **ni_scan_start** is 1, and extend through the nodes in order of node address — upward for forward scans, downward for backward scans. If no **ni_scan_start** flags are set in a partition, then the entire partition is treated as one segment.

Important: If you are sending a multiword message, the value of **ni_scan_start** when the first value is written applies to the entire message. Altering the flag after the first value is written has no effect on the message.

Addition Scan Overflow

Addition scans on large values can cause arithmetic overflow in some nodes. The **ni_com_scan_overflow** flag in the **status** register indicates whether the current scan result has suffered arithmetic overflow. This flag is 1 if the current message in the receive FIFO sufferred arithmetic overflow; otherwise, it is 0.

Note: The **com_scan_overflow** flag's value is only defined when the current message in the receive FIFO is the result of a scan or reduction operation with a combiner of addition or unsigned addition.

You can also instruct the NI to signal an interrupt for scan overflow. The **private** register contains a flag, **ni_com_scan_overflow_ie**, that when set to 1 causes an a Green interrupt (**scan overflow**) to be signaled when a scan result that overflowed is read from **ni_com_recv**.

4.2.7 Network-Done Messages

Network-done messages are used to synchronize the processing nodes after a Data Network operation. A network-done message is sent by a node when it has completed sending its Data Network messages and is waiting for the other nodes to finish. (Of course, even after a node has sent a network-done message, it may still *receive* Data Network messages.)

Important: Although network-done messages are directly related to the operation of the Data Network, they are a feature of the combine interface of the *Control Network.* All non-abstaining processors *must* start a network-done message before the network-done operation can be completed.

A network-done message is always of length 1, and the actual word written is ignored — all that matters is the sending of the message itself. Network-done messages have a unique pair of *combiner* and *pattern* values: the *combiner* field for the message must be 5, and the *pattern* field must be 0.

Network-done messages are an exception to the usual message-reception interface of the combine interface. The result of a network-done message is not delivered as a value in the receive FIFO.

Instead, the Data Network flag **ni_router_done_complete** is used to indicate when the network-done message has been sent by all nodes:

ni_dr_status	Data Network (DR) status register.
ni_router_done_complete	Network-done completion flag.

When a node sends a network-done message, the **ni_router_done_complete** flag of that node is set to 0. When all non-abstaining nodes have sent a network-done message, and when the Data Network has no pending messages for any node, the **ni_router_done_complete** flag is set to 1 for all nodes.

Usage Note: An attempt to send a network-done message with a length other than 1, or to send a network-done message while another such message is still in progress (that is, while the **ni_router_done_complete** flag is zero) signals a Bus Error.

How Network-Done Works...

Network-done messages continually use the combine interface hardware until they are completed, so any combine operations started after a network-done won't complete until after the network-done message is completed.

The network-done operation makes use of the **ni_dr_message_count** register of the Data Network to determine when the Data Network is clear. As described in Section 3.5.1, each node increments this register when it sends a message, and decrements the register when it receives a message. (Not counting, of course, messages for which counting is disabled by a 0 flag in **ni_count_mask**.)

When the `ni_dr_message_count` register is zero for all non-abstaining nodes, there should be no messages in transit through the Data Network. (Again, this may not be the case if there are messages for which message-counting is disabled, but this does not prevent the use of the network-done operation.)

A network-done message basically does a repeated addition scan on the values of the `ni_dr_message_count` register for all non-abstaining nodes. When the global result of this scan is zero, then the NI assumes that the Data Network is clear, and sets the `ni_router_done_complete` flag to 1.

...And Why You Should Care

Since network-done operations involve a *combine interface* scan of the value of a *Data Network* register, you should be careful about setting and changing the abstain flags of the combine interface when you intend to send a network-done message. (See Section 4.2.8 for a discussion of the combine interface's abstain flags.)

For example, if you change the combine abstain flags of one or more nodes while a Data Network operation is in progress, you may inadvertently exclude one or more nodes that have non-zero `message_count` registers. If you then start a network-done operation, these registers are ignored by the implied addition scan. In most cases, this prevents the result of the scan from ever becoming zero, and thus prevents the network-done message from completing.

To send a network-done message safely, make sure that the combine abstain flags of all nodes that might send or receive a message via the Data Network are cleared before starting the Data Network operation, and make sure those abstain flags remain cleared until after the network-done message has been completed.

NOTE

Because of a hardware defect, Revision A NI chips don't always execute network-done operations correctly. For more information, see Section 7.3.5.

4.2.8 Abstaining from the Combine Interface

The combine interface has two abstain flags that you can use to cause the NI to abstain from combine interface transactions. The use of these flags differs slightly from the description in Chapter 2, as described below.

`ni_com_control`	Status register, contains combine abstain flags.
`ni_rec_abstain`	Flag, combine interface abstain flag.
`ni_reduce_rec_abstain`	Flag, special reduction abstain flag.

Setting the `ni_rec_abstain` flag to 1 causes the NI to discard any arriving combine interface messages, and allows any messages sent by other nodes to complete without the participation of the abstaining node. In effect, abstaining nodes provide an appropriate identity value for any type of combine message.

Important: As with all abstain flags, the `ni_rec_abstain` flag and the `ni_reduce_rec_abstain` flag should only be changed when there are no messages pending in the combine interface. If a message is currently being written to the send FIFO when either abstain flag is changed, a Yellow interrupt (`com abstain changed`) is signaled.

Implementation Note: Because of way the broadcast and combine interfaces interact, if a node is abstaining from a combine operation, that node should *not* execute a broadcast operation until the combine operation is completed. (For more information, see Section 7.3.8.)

The Reduction Receive Abstain Flag

For scan operations, no result value is written to an abstaining node's receive FIFO. For reduction operations, however, there is an additional abstain flag, `ni_reduce_rec_abstain`, that controls whether or not the abstaining node receives the result.

Setting this flag to 1 causes a node to ignore the results of reduction operations. If `ni_rec_abstain` is 1 and `ni_reduce_rec_abstain` is 0, the node receives the results of reduction operations without having to supply a value for them. (For more detail, see the section on reduction operations below.)

For the Curious: The reason for this distinction is that there are important cases where it is necessary for a node to receive the result of a reduction without having to participate in it. For example, when you want to transfer a value from the nodes of a partition to the partition manager, you can set the combine abstain flags so that the nodes transmit a reduction message and only the PM receives it.

4.2.9 The Combine Private Register

The combine interface's private register contains the following subfields:

`ni_com_private`	Private register.
`ni_rec_ok_ie`	Flag, "Receive OK" interrupt enable.
`ni_lock`	Interface lock flag.
`ni_rec_stop`	Interface stop flag.
`ni_rec_full`	Flag, indicates receive FIFO is full.
`ni_com_scan_overflow_ie`	Flag, scan overflow interrupt enable.
`ni_com_rec_empty_ie`	Flag, empty rec. FIFO inter. enable.
`ni_com_send_length`	Field, send message length.
`ni_com_send_combiner`	Field, send message combine value.
`ni_com_send_pattern`	Field, send message pattern value.
`ni_com_send_start`	Flag, scan segmentation flag.

The `rec_ok_ie`, `lock`, `rec_stop`, and `rec_full` subfields are as described in Chapter 2. The `ni_com_scan_overflow_ie` flag is described in Section 4.2.6. The remaining fields are described in the sections below.

Empty Receive FIFO Interrupt

When the `ni_com_rec_empty_ie` flag is set to 1, the NI signals a Green interrupt (`com rec empty`) if the receive FIFO ever becomes empty (that is, when the `rec_ok` flag becomes 0). This allows the supervisor to insert one or more messages into the empty receive FIFO, so that from a user program's point of view, the FIFO is never empty. (This is used by the OS in context switching.)

Clearing the Combine Send FIFO

The pipelining feature of the combine interface means that when the supervisor needs to swap a process out, there may be several complete messages pending in the combine send FIFO, each of which has its own auxiliary information (each message may have different *combine* and *pattern* values, for instance).

The supervisor extracts messages from the send FIFO by reading them, one at a time, from the `ni_com_send` register. Reading a value from this register extracts the word (or doubleword) that was most recently pushed into the FIFO.

Important: Once the supervisor begins reading words from the send FIFO, the FIFO must be emptied before a new message can be written to it. (This avoids

the potential for accidentally pushing a new message on top of a half-extracted old message.) The effect of violating this restriction is undefined.

Usage Note: It is only legal to read a value from the **ni_com_send** register when the combine interface is not being used (that is, when the receive FIFO is empty and no node in the partition is or will be in the process of writing a combine message while the contents of the send FIFO are being read out.

The four **private** register fields **send_length**, **send_combiner**, **send_pattern**, and **send_start** contain the auxiliary data and segmentation information for the most recent message in the send FIFO (that is, the message that includes the next word that the supervisor can read from the send FIFO).

Specifically:

ni_com_send_length	Field, send message length.
ni_com_send_combiner	Field, send message combine value.
ni_com_send_pattern	Field, send message pattern value.
ni_com_send_start	Flag, scan segmentation flag.

- **send_length** is the number of words in the entire message.
- **send_combiner** is the combine value for the message.
- **send_pattern** is the pattern value.
- **send_start** is the **ni_scan_start** register value for the message.

The supervisor can use these fields like the corresponding **status** register fields to obtain the auxiliary data for messages extracted from the send FIFO. The **send_length** field is undefined for a network-done message. (The message is always one word in length.) The value of **scan_start** is undefined for reduction and network-done messages, which ignore the segmentation flag.

4.3 The Global Interface

The global interface provides a generic synchronization mechanism for the CM-5's processing nodes. It is much like the network-done feature of the combine interface, but without the additional condition that the Data Network must be clear before the operation can complete.

The global interface combines a single bit from every participating node in a logical OR operation, and then returns the result to each node. The actual values sent by the nodes, however, can be completely arbitrary. The sending of the message itself is sufficient to provide synchronization of the nodes.

A global interface message can be sent by one of three subinterfaces:

- the synchronous global interface, which requires that all nodes send a message before any receive the result
- the asynchronous global interface, which permits nodes to send a message and read the result at any time, with the network continually monitoring the state of all participating nodes
- the supervisor asynchronous global interface, which is identical to the asynchronous global interface save that its registers are accessible only from the supervisor area

There is a separate register set for each of these three methods. Each of these interfaces is described in more detail in the sections below.

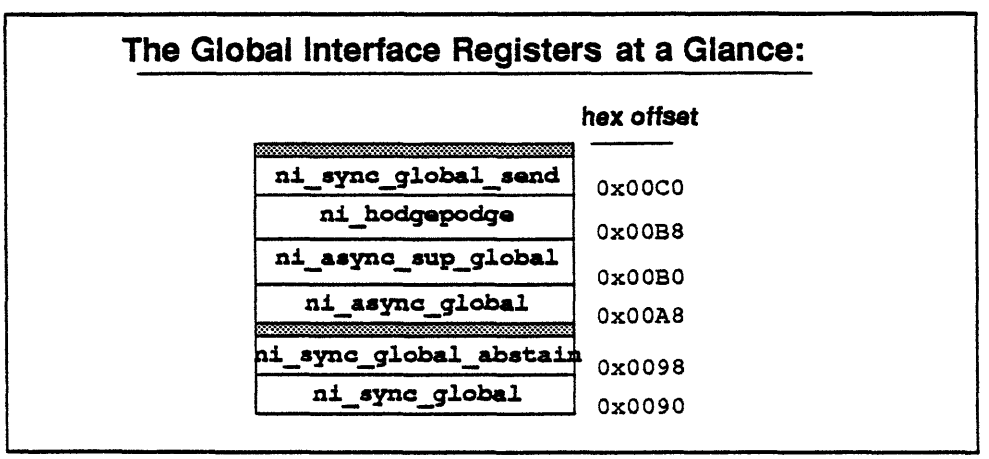


Figure 14. NI registers associated with the global interface.

4.3.1 The Three Global Register Interfaces

Unlike the broadcast and combine interfaces, the global interface does not use the generic interface model presented in Chapter 2. The following registers are used for the three interfaces:

Synchronous global interface:

`ni_sync_global_send`	Used to send the first value of a message.
`ni_sync_global_abstain`	Used to abstain from synch global msgs.
`ni_sync_global`	Used to receive a message.
`ni_hodgepodge`	Contains interrupt enable flag.

Aynchronous global interface:

`ni_async_global`	Asynchronous send and receive flags
`ni_hodgepodge`	Contains interrupt enable flag.

Supervisor aynchronous global interface:

`ni_async_sup_global`	Supervisor asynch. send and receive flags
`ni_hodgepodge`	Contains interrupt enable flag.

The purpose and use of these registers is described in the sections below, and Figure 14 contains a memory map showing their relative locations in NI memory.

4.3.2 The Synchronous Global Interface

The synchronous global interface takes the global OR of a flag set by each node. Each non-abstaining node must set its synchronous global flag (and thereby send a synchronous global message) before the result of the operation is reported to any node.

The following registers and flags form the synchronous global interface:

`ni_sync_global_send`	Used to send the first value of a message.
`ni_sync_global_abstain`	Used to abstain from synch. global msgs.
`ni_sync_global`	Used to receive a message.
`ni_sync_global_rec`	Synchronous global receive flag.
`ni_sync_global_complete`	Synchronous global completion flag.
`ni_hodgepodge`	Contains interrupt enable flag.
`ni_sync_global_rec_ie`	Receive interrupt enable flag.

Sending and Receiving Messages

To start a synchronous global interface message, write a value (either 0 or 1) to the the **ni_sync_global_send** register.

When you write a value to the **global_send** register, the **ni_sync_global_complete** flag is set to 0, indicating that a message is in progress. (**Note:** It is an error to write to the **ni_sync_global_send** register when the **ni_sync_global_complete** flag is 0.)

When all participating nodes have sent a message, the global interface takes the logical OR of the **ni_sync_global_send** flag in each node, and then sets the **ni_sync_global_rec** flag of every participating node to the result. At the same time, the **ni_sync_global_complete** flag is set back to 1 to indicate completion of the message.

Abstaining from Synchronous Global Messages

The synchronous global interface includes an abstain flag that can be used to exclude a node from the interface's operations:

ni_sync_global_abstain Status register, contains global abstain flag.

When the **ni_sync_global_abstain** flag is set to 1, synchronous global messages complete without the node's participation (as if the node has sent the message with its **ni_sync_global_send** flag set to 0).

Note: As with all abstain flags, **ni_sync_global_abstain** should only be changed when there is no global message pending. A Bus Error is signaled if the abstain flag is modified when the **ni_sync_global_complete** flag is 0.

Also, a Bus Error is signaled if the **ni_sync_global_send** register is written while the abstain flag is 1.

Synchronous Global Receive Interrupt

If the **ni_sync_global_rec_ie** flag in the **hodgepodge** register is set to 1, then a Green interrupt (**sync global rec**) is signaled whenever the **ni_sync_global_rec** flag changes from 0 to 1.

4.3.3 The Asynchronous Global Interface

The asynchronous global interface is not so much a node synchronization tool as a means for determining whether all the nodes are still operating properly, or whether some global action needs to be taken. As with the synchronous interface, the asynchronous interface takes the global OR of a flag set by each node. However, this global OR is performed continually, so that a change of a flag by any node is reported almost immediately to the other nodes.

For example, each node can set its flag to 1 before performing an operation, and set the flag to 0 when the operation is completed. The global interface returns a 1 value until all nodes have set their flags to 0, guaranteeing that all nodes have completed the operation.

The following registers and flags form the asynchronous global interface:

ni_async_global	Control register, contains the following flags:
ni_global_send	Flag, used to "send" asynchronous messages.
ni_global_rec	Flag, always set to logical OR of **send** flags.
ni_hodgepodge	Control register, includes the following flag:
ni_global_rec_ie	Flag, global receive interrupt enable.

Sending and Receiving Messages

Because the asynchronous global interface operates continually, there really is no such thing as "sending" or "receiving" a message via this interface.

The **ni_global_rec** flag in each node is continually updated to reflect the "current" logical OR of the **ni_global_send** flag in all nodes. When any node writes a new value into its **ni_global_send** flag, the change is propagated to the **ni_global_rec** flag of all other nodes after a brief interval.

Important: Because this is an asynchronous mechanism, the **ni_global_rec** flag may not always reflect the present state of the **ni_global_send** flags in all the nodes. There is always a delay between the instant any node changes its **ni_global_send** flag and the instant that all nodes receive the result of the change. You should not write code that depends on this delay having any exact length, but you can assume that the delay is no longer than the time taken to transmit a synchronous message.

Asynchronous Global Receive Interrupt

If the **ni_global_rec_ie** flag in the **hodgepodge** register is set to 1, then a Green interrupt (**global rec**) is signaled whenever the **ni_global_rec** flag changes from 0 to 1.

4.3.4 The Supervisor Asynchronous Global Interface

The supervisor asynchronous global interface is identical to the asynchronous interface described above, except that its registers are only accessible from the supervisor area. This interface is typically used by the operating system to synchronize the nodes during OS operations such as context switching.

For example, if each node sets its flag to 0, then the global interface returns a 0 value until one of the nodes signals a 1 instead. If any node reaches a point in its operations where OS intervention is required, the node can set its flag to 1, signaling a 1 value to all the other nodes, and also indicating to the OS that some global action must be taken.

The following register and flags form the supervisor asynchronous interface:

ni_async_sup_global	Control register, contains these flags:
ni_supervisor_global_send	Flag, used to "send" messages.
ni_supervisor_global_rec	Flag, logical OR of **send** flags.
ni_hodgepodge	Control register, includes the flag:
ni_supervisor_global_rec_ie	Supervisor receive interrupt enable.

Sending and Receiving Messages

The **ni_supervisor_global_send** and **ni_supervisor_global_rec** flags are used to send and receive messages the same way that the asynchronous interface does (described above).

Supervisor Asynchronous Global Receive Interrupt

If the **ni_supervisor_global_rec_ie** flag in the **hodgepodge** register is set to 1, then a Green interrupt (**supervisor global rec**) is signaled whenever the **ni_supervisor_global_rec** flag changes from 0 to 1.

Chapter 5

NI Interrupts

The NI chip is, in many ways, the "interrupt gateway" of the CM-5. Most node hardware and software exceptions, whether or not they originate in the NI chip, are signaled to the node microprocessor via NI interrupts.

The NI is capable of signaling an interrupt in any of five classes and at any of a number of levels of severity. Interrupts can be signaled by events beyond the programmers's control (such as hardware failures), or by fatal errors in the way a program uses the NI, or deliberately, under program control.

Interrupts are signaled by one of two different methods:

- as a local interrupt to the NI's associated microprocessor
- as a broadcast interrupt to the other NIs in the partition

This chapter describes the kinds of interrupts available on the NI, their causes, the registers used to determine their type and severity when they are signaled, and the mechanism used to signal a broadcast interrupt.

5.1 Interrupt Classes

The NI can signal five different classes of interrupt: Red, Orange, Yellow, Green, and Bus Errors. Red interrupts tend to be the most severe and green interrupts the least severe. The five types are distinguished as follows:

- **Red interrupts** indicate a failure of the hardware, such as checksum violations and message format errors.

  They occur at unpredictable times relative to the instruction stream and are usually irrecoverable. Determining the precise cause of a Red interrupt may require the use of the Diagnostic Network.

  The possible Red interrupts are:

`internal fault`	Failure detected in NI chip itself.
`dr checksum error`	Data Network checksum failure.
`cn checksum error`	Control Network checksum failure.
`cn hard error`	Control Network hardware failure.
`mc error`	Error detected in memory subsystem.
`cmu error`	Cache/MMU error.
`bc interrupt red`	Red broadcast interrupt.

- **Orange interrupts** indicate that the attention of the operating system is required, as in timer interrupts and broadcast interrupt messages.

  They occur at unpredictable times relative to the instruction stream and do not destroy any information that might be needed to determine the cause of the interrupt.

  The possible Orange interrupts are:

`timer interrupt`	NI timer reached `interrupt_now`.
`bc interrupt orange`	Orange broadcast interrupt.

- **Yellow interrupts** indicate that the software has made an error. They are usually irrecoverable, as they indicate that your program is doing something illegal and must be rewritten. Sufficient information is retained in the NI to permit isolation of the cause of the interrupt, but it is not always possible to recover all the information relating to the cause of the interrupt.

  Yellow interrupts are associated with particular instructions, but usually are not signaled at the exact point of the offending instruction, because of the loose coupling between the NI and the microprocessor.

  The possible Yellow interrupts are:

`dr count negative`	Negative DR message count.
`bc or com collision`	Conflict in broadcast/combine ops.
`com abstain changed`	Flag changed while interface in use.
`bad relative address`	Address outside partition, etc.
`bc interrupt yellow`	Yellow broadcast interrupt.

- **Green interrupts** indicate the occurrence of common events for which the software has requested notification, such as the arrival of messages, the signaling of broadcast interrupts, arithmetic overflow in a scan, etc. There is one interrupt for each event, and each event's interrupt can be enabled and disabled independently under the control of the supervisor.

  Depending on the type of event, the interrupt may or may not occur synchronously with a particular instruction. No information is lost by a Green interrupt.

  The possible Green interrupts are:

`scan overflow`	Overflow in combine interface scan.
`dr rec ok`	DR message received.
`ldr rec ok`	LDR message received.
`rdr rec ok`	RDR message received.
`bc rec ok`	Broadcast received.
`sbc rec ok`	Supervisor broadcast received.
`com rec ok`	Combine message received.
`com rec empty`	Empty combine receive FIFO.
`dr rec tag`	Message with interrupt tag received.
`dr rec all fall down`	All Fall Down message received.
`sync global rec`	Synchronous global msg received.
`global rec`	Asynchronous global msg received.
`supervisor global rec`	Supervisor asynch. msg received.
`bc interrupt green`	Green broadcast interrupt.

- **Bus Errors** indicate that a bus transaction cannot be completed, as in an attempt to read an address that does not correspond to a register, or to write a message that does not conform to sending protocol (**`send_first`**, then **`send`**). Bus Errors are signaled asynchronously, and are irrecoverable.

  There is basically one flavor of Bus Error:

`bad memory access`	Meaningless or illegal reference.

  Bus Errors are treated differently from the four colored interrupts. Bus Errors are always handled as traps, primarily because they occur only on read operations, and do not involve the NI chip.

  Note: Bus Errors are distinct from segmentation violation errors. Segmentation errors result from attempting to read an unmapped virtual address, and are signaled synchronously with the offending instruction. Bus Errors result from errors with physical addresses, once the address has been transmitted to the Mbus itself.

5.2 Interrupt Pathways

The four colored interrupts (Red, Orange, Yellow, and Green) result from a number of different causes. Figure 15 shows the pathways followed by the various types of interrupts on their way to the microprocessor. These pathways are described in detail in the sections below.

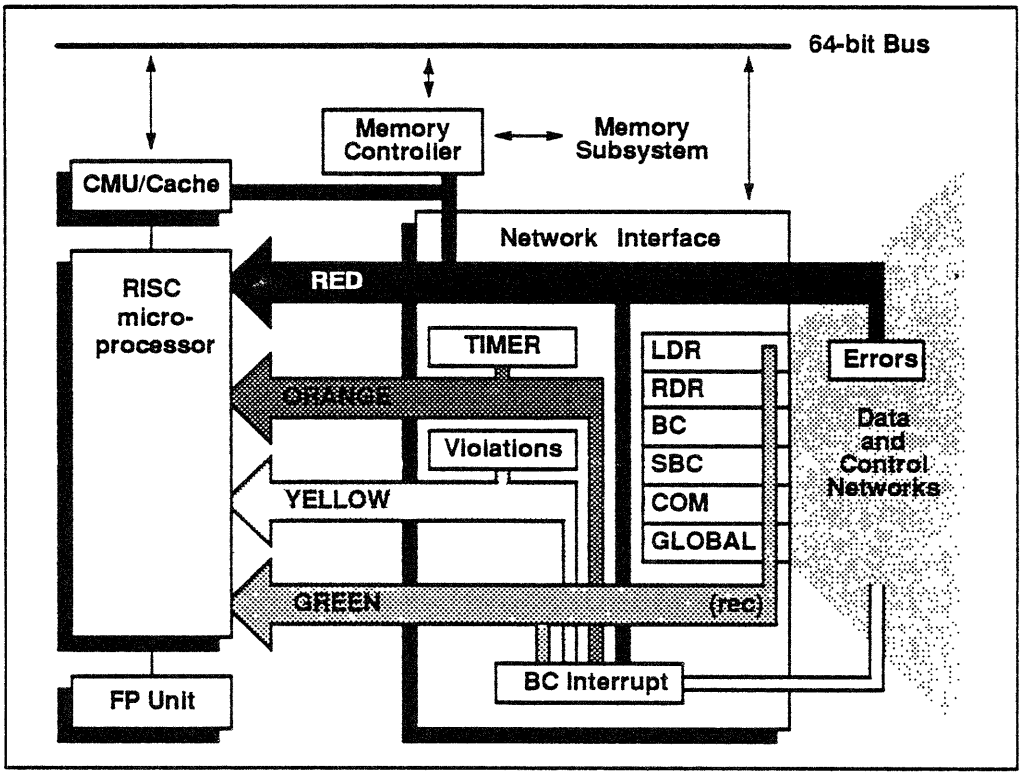


Figure 15. The possible pathways for colored interrupts.

5.2.1 Red Interrupts

The Red interrupts are of two varieties:

- *On-chip faults* — hardware errors detected by the NI itself
- *Off-chip faults* — problems on other devices that are signaled via the NI

On-chip faults are universally fatal — that is, they always cause the OS to halt (usually forcefully). It is then necessary to use diagnostic measures to determine the cause of the problem.

Off-chip faults are caused by problems on other components, and it is necessary for the OS to poll those devices to find out what happened.

Of the red interrupts, the following are off-chip faults:

mc error	— error in MC (memory controller)
cmu error	— error in CMU (cache and memory unit)

The cause of these faults can only be determined by examining the state of the appropriate hardware:

- MC errors are caused by either a fault in the MC itself (usually fatal), or (if the CM-5 has the vector unit option installed) by an error signaled from one or more of the vector units. In either case, it is necessary to examine the state of the appropriate hardware to determine the actual cause of the interrupt.
- CMU errors are only caused by bad memory writes (typically memory writes to illegal addresses) and are always fatal. CMU errors are asynchronous, so that the error is not signaled until some time after the offending write instruction.

All the remaining Red interrupts are on-chip faults. Three are caused by network problems:

dr checksum error	— Data Network fault.
cn checksum error	— Control Network fault.
cn hard error	— Control Network hardware fault.

One is caused by NI chip problems:

internal fault	— NI chip fault.

And one can be signaled by software:

bc interrupt red	— Red broadcast interrupt.

Warning: A Red broadcast interrupt is functionally equivalent to deliberately causing a fatal error, so use it with caution — if you use it at all!

5.2.2 Orange Interrupts

There are only two Orange interrupts. One is caused by the NI timer:

`timer interrupt` — Timer alarm interrupt.

And the other can be signaled by software:

`bc interrupt orange` — Orange broadcast interrupt.

5.2.3 Yellow Interrupts

The Yellow interrupts are, with one exception (the Yellow broadcast interrupt), caused by NI violations produced in user code:

`com abstain changed` — Illegal abstain flag change.
`bc or com collision` — Multiple message collision.
`bad relative address` — Illegal DR destination.
`dr count negative` — Negative DR message count.

There is also a Yellow broadcast interrupt that can be signaled by software:

`bc interrupt yellow` — Yellow broadcast interrupt.

5.2.4 Green Interrupts

The Green interrupts are, for the most part, indications of non-error network events for which the user may want to assign a specific code handler.

For example, there are nine Green interrupts, one for each major network interface, that indicate when a message has arrived in the interface's **recv** register:

`bc rec ok` — BC interface message received.
`sbc rec ok` — SBC interface message received.
`com rec ok` — COM interface message received.
`dr rec ok` — DR interface message received.
`ldr rec ok` — LDR interface message received.
`rdr rec ok` — RDR interface message received.
`global rec` — Asynchronous global message received.
`sync global rec` — Synchronous global message received.
`supervisor global rec` — Supervisor asynchronous global message.

In addition, there is a Green interrupt for an important combine interface event:

scan overflow	— Combine interface add-scan overflow.

There are a number of interrupts for OS-related events:

dr rec tag	— DR message arrived with interrupting tag.
dr rec all fall down	— DR All Fall Down mode message received.
com rec empty	— Combine receive FIFO empty.

And as usual there is a broadcast interrupt that can be signaled by software:

bc interrupt green	— Green broadcast interrupt.

5.3 The Interrupt Cause and Clear Registers

Once an interrupt has been signaled, there are four NI registers that you can use to determine which interrupt it is, and also to clear it once you have finished handling it:

ni_interrupt_cause	Flags set by non-Green interrupts.
ni_interrupt_clear	Flags used to clear non-Green interrupts.
ni_interrupt_cause_green	Flags set by Green interrupts.
ni_interrupt_clear_green	Flags used to clear Green interrupts.

When an event causing an interrupt occurs, a bit in the **ni_interrupt_cause** or **ni_interrupt_cause_green** register is set. Which bit is set indicates what the event was. If more than one interrupt occurs before any are cleared, several bits in these registers may be set simultaneously.

The **interrupt_cause** and **interrupt_cause_green** registers may also be written explicitly (by the supervisor, not user code) to cause interrupts to be signaled without their normal triggering event occurring.

Interrupts can be cleared by writing a value to the **ni_interrupt_clear** or **ni_interrupt_clear_green** registers. Any value written to these registers should contain 1's in locations corresponding to the interrupts that are to be cleared. It is not possible to read the value of the **ni_interrupt_clear** or **ni_interrupt_clear_green** registers — use the corresponding **cause** register to determine whether any interrupts have not yet been cleared.

Note: If a given interrupt has an interrupt enable flag (a flag with a name ending in `_ie`) and the flag is set to 0, then the interrupt is not signaled and the corresponding `ni_interrupt_cause` or `ni_interrupt_cause_green` flag is not set.

5.4 Interrupt Levels

Each of the four color classes of interrupt include a "level" or "priority" value that can be used to provide the software with information about the relative importance or priority of interrupts of various colors.

Any interrupt level can be assigned to each color of interrupt. It is, for example, permissible to give Green interrupts a level of 15 while Red interrupts have a level of 4. However, the relative interrupt levels are intended to indicate priority or severity; for example, there are mechanisms for masking all interrupts (of any color) below a given level.

The following register is used to set the priority value for each interrupt color:

`ni_interrupt_level`	Control register, contains these fields:
`ni_interrupt_level_red`	Red interrupt priority level.
`ni_interrupt_level_orange`	Orange interrupt priority level.
`ni_interrupt_level_yellow`	Yellow interrupt priority level.
`ni_interrupt_level_green`	Green interrupt priority level.

The four eight-bit fields, `level red` through `level green`, each indicate the level at which the corresponding color of interrupt is signaled. For example, if the `level red` field is set to 13, all red interrupts from that point onwards are signaled to the microprocessor with a level of 13.

If more than one color of interrupt is signaled simultaneously, the interrupt level signaled to the processor is the inclusive OR of the levels for each interrupt color.

If any of the `interrupt_level` fields is set to 0, then all interrupts of the corresponding color are suppressed. (When the NI is reset, for example, all four interrupt level fields are set to 0.)

Implementation Note: Currently, only the low-order bit of each interrupt level field is used. The other bits are required to be 0.

5.5 Broadcast Interrupts

The broadcast interrupt mechanism allows an interrupt to be signaled from one NI to all other NIs in the current partition. Each NI receiving the broadcast immediately signals an interrupt to its associated microprocessor.

Important: Only one NI in each partition can use the broadcast interrupt facility. If two or more NIs try to broadcast simultaneously in the same partition, a Yellow interrupt (`bc or com collision`) is signaled to all nodes in the partition, and the broadcast interrupt messages that are received are undefined.

The broadcast interrupt can be of any color, Red, Orange, Yellow, or Green. A unique flag exists in the `cause` and `clear` registers for each color of broadcast interrupt. Only Bus Errors cannot be broadcast — mainly because it is not useful (and doesn't really make sense) to do so.

The following register and flags are used to send a broadcast interrupt:

`ni_interrupt_send`	Register used to send broadcast interrupt.
`ni_hodgepodge`	Control register, includes the flags:
`ni_interrupt_send_ok`	Flag, set when broadcast is sent.
`ni_interrupt_rec_enable`	Flag, enables receipt of interrupts.

To send a broadcast interrupt, write a value to the `ni_interrupt_send` register indicating the color of interrupt to be signaled. The permissible values for each color of interrupt are:

Value	Interrupt	Description
8	`bc interrupt red`	Red broadcast interrupt.
4	`bc interrupt orange`	Orange broadcast interrupt.
2	`bc interrupt yellow`	Yellow broadcast interrupt.
1	`bc interrupt green`	Green broadcast interrupt.

Note: More than one color of interrupt can be broadcast at a time (for example, by combining the above values with a logical-OR operation). Multi-colored broadcast interrupts are signaled by the hardware exactly as if each colored interrupt was signaled separately. The software effects of such multi-colored interrupts are determined entirely by the current interrupt handlers on the nodes.

Writing a value to `ni_interrupt_send` sets the `ni_interrupt_send_ok` flag to 0 until the interrupt has been successfully broadcast, at which point the flag is set back to 1. An attempt to write a value to `ni_interrupt_send` while `ni_interrupt_send_ok` is 0 signals a Bus Error.

Any NI can disable broadcast interrupts by setting its **ni_interrupt_rec_enable** flag to 0. Doing so causes all broadcast interrupts received by that NI chip to be ignored. Setting the flag back to 1 re-enables broadcast interrupts.

Note: There is a special class of broadcast interrupt, the Reset interrupt, which cannot be disabled. See Section 6.10 for more information about the cause and effects of an NI Reset.

5.6 Recovering from Interrupts

The methods used to recover from an interrupt depend heavily on the type of interrupt itself. Appendix D of this manual provides guidelines describing the steps needed to recover from each of the possible interrupts.

Chapter 6

Other NI Interfaces and Features

This chapter describes the remaining NI registers and features not covered in the preceding chapters. Except as noted, all registers and features described in this chapter are accessible only to the supervisor.

6.1 The "Hodgepodge" Register

The `ni_hodgepodge` register, as its name suggests, contains a collection of miscellaneous flags that are used by various features of the NI.

`ni_hodgepodge`	Register with "hodgepodge" of flags:
`ni_sync_global_rec_ie`	Sync global receive interrupt enable.
`ni_global_rec_ie`	Asynch global receive intrpt. enable.
`ni_supervisor_global_rec_ie`	Supervisor asynch. rec. intrpt. enable.
`ni_interrupt_send_ok`	Broadcast interrupt send ok flag.
`ni_interrupt_rec_enable`	Broadcast interrupt receive enable.
`ni_flush_complete`	Combine flush complete flag.
`ni_timer_ie`	NI timer interrupt enable flag.
`ni_configuration_complete`	Configuration complete flag.
`ni_cn_stop_send`	Control Network disable flag.

For more information on the meaning and use of these flags, refer to the sections describing the NI features that use them. (Look up the individual flags by name in the Index.)

6.2 Node Address Registers

There are three NI registers that provide information about the physical address of the current node within the CM-5, as well as the size and location of the current partition:

`ni_physical_self`	20-bit physical address of current node.
`ni_partition_base`	20-bit address of first node in partition.
`ni_partition_size`	Number of nodes in current partition.

These registers are used by other NI chip features, such as the chunk table address translation mechanism described in Section 6.3 below.

6.3 NI Chunk Table and Address Translation

The NI *chunk table* is a small array stored in the NI itself that determines the locations of the "chunks" of processing nodes that make up a Data Network partition on the CM-5. A *chunk* is a contiguous sequence of physical addresses that correspond to real, working processing nodes. Addresses corresponding to broken or missing hardware are isolated by not being included in any chunk.

Important: The chunk table specifies chunks of *node addresses* — the chunk table has nothing to do with memory allocation on the nodes.

6.3.1 Node Address Translation

The chunk table is used to convert from relative node addresses used within a partition to the physical addresses required by the Data Network.

For the Curious: A side effect of the use of the chunk table is that it implicitly divides the Data Network up into "partitions" of nodes. That is, there is no hardware restriction preventing a Data Network message from traveling between partitions; it is the chunk tables that determine whether a relative address is legal for a given partition of nodes.

The mapping from relative to physical addresses is performed in three steps:

First, the relative address is compared with the `ni_partition_size` register, to determine whether it is legal for the current partition. (If the relative address

is greater than or equal to `ni_partition_size`, the address is guaranteed not to correspond to a node in the current partition, and an error is signaled.)

Next, the relative address is split into two parts (see Figure 16).

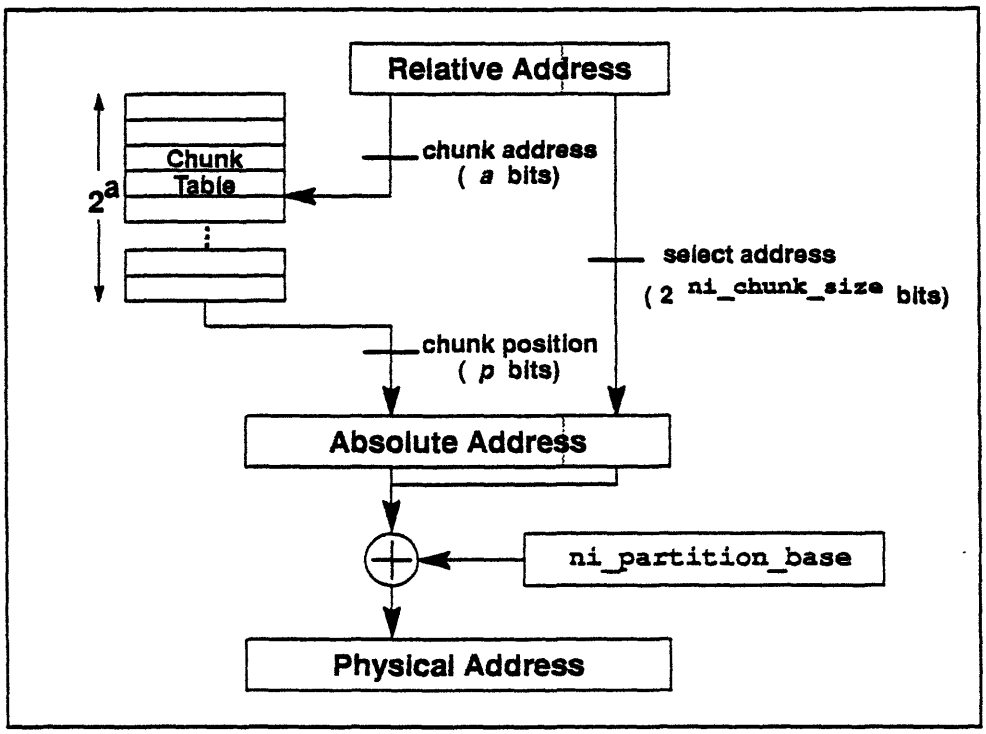


Figure 16. Translation from relative addresses to physical addresses.

The two parts of the address are:

- the high-order bits of the address, known as the *chunk address*
- the low-order bits of the address, known as the *select address*

The chunk address is used as a pointer into the NI's chunk table. The referenced chunk table entry, known as the *chunk position*, is recombined with the select address to form an *absolute address* — essentially an offset from the address of the first processor in the current partition.

Finally, the absolute processor address is added to the value of the register `ni_partition_base` to get the required physical address.

6.3.2 Chunk Sizes and Address Allocation

The size of the chunk table is determined by the number of bits in a chunk address (call this a), and the number of bits in a chunk position (call this p). The chunk table consists of 2^a entries, each p bits long. The values of a and p are currently fixed by hardware at $a = 6$ and $p = 8$. Thus, the chunk table contains 64 entries, each 8 bits long.

However, while the size of the chunk table is fixed, the size of the chunks it references (that is, the number of physical addresses per chunk) is under supervisor control. The following register is used to set the chunk size:

`ni_chunk_size` Size of chunks referenced by the chunk table.

The **`ni_chunk_size`** register contains a three-bit value that determines the number of bits in the select address part of a relative address, and thus sets the number of addresses per chunk.

The number of bits in a select address is $2^{\texttt{ni\_chunk\_size}}$. As a result, the number of physical addresses in a chunk is $4^{\texttt{ni\_chunk\_size}}$, and this means that the number of possible relative addresses (in other words, the number of accessible nodes) is $2^a * 4^{\texttt{ni\_chunk\_size}}$. This also means that the total physical address space accessible through the chunk table is $2^p * 4^{\texttt{ni\_chunk\_size}}$. Thus, the accessible physical address space is always 2^{p-a} times the size of the relative address space. This extra "unused" space between chunks is used to isolate regions of broken or missing hardware. (See Figure 17.)

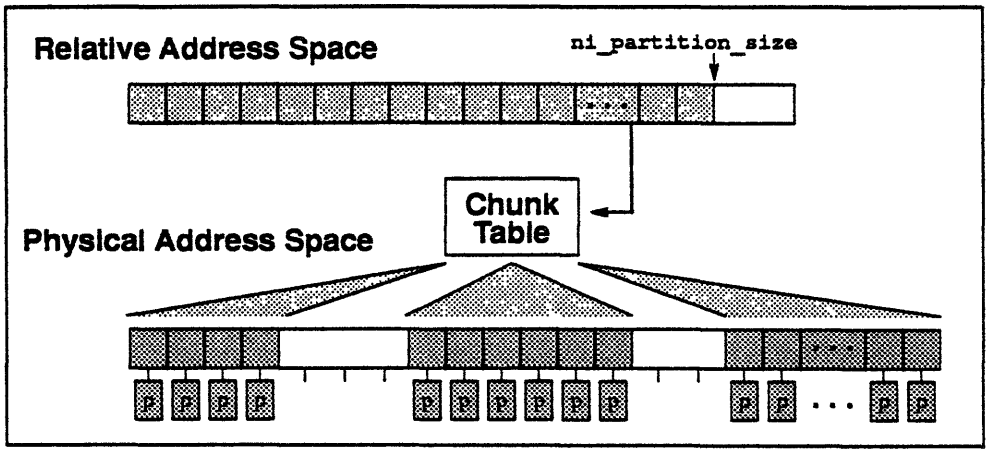


Figure 17. The chunk table is used to map contiguous relative addresses onto discontiguous physical addresses.

In the simplest case, the chunk table is set up to map all relative addresses to a contiguous region of $2^a * 4^{\texttt{ni\_chunk\_size}}$ physical addresses. In this case, chunk table entry *n* simply has the value *n*.

The table below lists the permissible values for the **ni_chunk_size** register, along with the corresponding number of relative addresses (nodes) per chunk, and the maximum size of the physical address space in nodes and addresses.

ni_chunk_size	Addresses/chunk	Nodes	Phys. address space
1	4	256	1K
2	16	1K	4K
3	64	4K	16K
4	256	16K	64K
5	1K	64K	256K
6	4K	256K	1M

Note: The effects of writing **ni_chunk_size** with a value not listed in this table are undefined, but almost certainly disastrous.

6.3.3 Modifying the Chunk Table

The following registers are used to read and write chunk table entries:

ni_chunk_table_data	Location used to read/write table entries.
ni_chunk_table_address	Chunk table location that is read/written.

Note: The chunk table is set up by the OS when the nodes are grouped into partitions, and from then on the chunk table is normally not modified. Accordingly, the registers listed above are accessible only from the supervisor area.

When the **ni_chunk_table_data** register is written, the value written is stored in the chunk table entry indicated by **ni_chunk_table_address**. When the **table_data** register is read, the value read is the current contents of that chunk table entry.

The **ni_chunk_table_address** register determines the entry of the chunk table that is affected by reading or writing the **ni_chunk_table_data** register. The size of the values that are read from and written to this register depends on the size of chunk addresses (see the discussion in Section 6.3.2).

Important: In order for the Control Network to operate correctly, the entries of the chunk table must be in ascending order. In other words, each chunk table entry must contain a larger address than the entry that precedes it.

Note: The effects of reading or writing the **table_data** register while the Data Network is in use are undefined, and best avoided.

6.4 Combine Interface Flush

The combine interface flush operation is used to reset the hardware of the combine interface, canceling any uncompleted combine operations. As with all other Control Network operations, a combine flush must started in unison by all of the nodes in a partition — nodes cannot "abstain" from a flush. Also, flushes only affect the single partition in which they are started; they don't cross partition boundaries.

Important: The broadcast and global interfaces are not affected by flushing, and must be cleared separately.

The combine flush interface consists of the following registers and flags:

ni_com_flush_send	Single-flag register used to start a flush.
ni_hodgepodge	Control register, includes the flag:
ni_flush_complete	Flag, set when flush is completed.

To start a flush operation, write any value (either 0 or 1) to the **ni_com_flush_send** register. This sets **ni_flush_complete** to 0, and then starts the interface flush. When the flush is completed, the **flush_complete** flag is set back to 1. Attempting to write the **ni_com_flush_send** register while **ni_flush_complete** is 0 or **ni_com_abstain** is 1 signals a Bus Error.

Important: A flush operation should be executed only when there are no messages in transit through the combine interface, that is, when **ni_com_send_empty** is 1, and **ni_com_rec_ok** is 0.

Usage Note: The combine flush operation is only useful when the send and receive FIFOs of the combine interface are empty. The combine flush operation does *not* clear out the FIFOs — it merely resets the communications hardware of the interface itself. The flush operation is only intended to be used in context switches, after the FIFOs have been cleared and saved.

6.5 The NI Timer

The NI contains a simple timing mechanism that can be used to measure the time between two events and to interrupt the microprocessor after a specific interval.

The following registers and flags form the timer interface:

`ni_time`	Timer register, regularly incremented.
`ni_interrupt_now`	Register, timer value that triggers interrupt.
`ni_hodgepodge`	Control register, includes the flag:
`ni_timer_ie`	Timer interrupt enable flag.

The 32-bit register `ni_time` contains an unsigned value that is incremented at every microprocessor clock cycle. When the timer value exceeds the register's capacity, it wraps around to 0.

The value of the `ni_time` register can be read at any time, and can be written by the supervisor to set the NI's timer to a chosen value.

The NI timer can signal an interrupt at a specific timer value. When the value of `ni_time` equals the value stored in the `ni_interrupt_now` register, an Orange interrupt (`timer interrupt`) is signaled.

This interrupt can be enabled and disabled by setting the `ni_timer_ie` flag in the `hodgepodge` register. When this flag is 1, timer interrupts are enabled. When this flag is 0, timer interrupts are disabled.

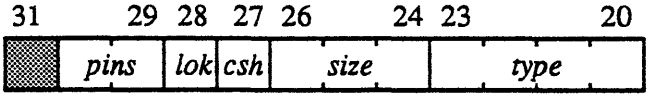
6.6 The Bad Address Register

When a Bus Error is signaled as the result of an illegal memory reference, the `ni_bad_address` register contains the illegal address, the data size, and the type (read or write) of the transaction. The data returned by a read from an illegal memory address is undefined. Data written to an illegal memory address is lost.

`ni_bad_address`	Bad address register, contains the fields:
`ni_bad_address_low`	Low 20 bits of illegal address.
`ni_bad_address_type`	Size and type of transaction.

Usage Note: The **ni_bad_address** register is updated *every* time a memory transaction is made, not just when an error occurs. Thus, its value is only valid when a Bus Error (**ni bad memory access**) has actually been signaled. If more than one illegal access is performed before the first one is handled, the value of the **ni_bad_address** register is the most recent bad memory address.

Currently, the format of the **ni_bad_address_type** field is:



where

- *type* indicates the transaction type (0 = write, 1 = read)
- *size* gives the data size (2 = word, 3 = doubleword)
- *csh, lok* are the MBUS cacheable and lock bits
- *pins* is the setting of the NI's two physical base address pins

Values for the *type* and *size* fields other than those shown above are reserved. The *csh*, *lok*, and *pins* fields are hardware-related and not useful to NI programmers.

6.7 NI Partition Configuration

The NI has a register that can be used to change the partitioning of the CM-5. The following register and flag are used to control the partitioning feature:

ni_configuration	Partition configuration control register.
ni_hodgepodge	Control register, includes the flag:
ni_configuration_complete	Flag, set when partitioning is done.

The **ni_configuration** is a five-bit register that controls the *configuration*, or set of processor partitions, that is in use. The value in this register is actually the "height" (number of layers) of the Control Network partition to which the node belongs. Control Network operations use this value to determine the maximum height of the network to which a message needs to be sent.

By writing a value to the **configuration** register, you can temporarily change the size of the current partition. (Since the actual size of the partition is currently determined by the state of the Control Network itself, you can only reduce the size of the partition.)

Note: Only one NI per partition needs to write a value to the **configuration** register — the configuration operation includes all nodes in the same partition.

The actual value written to the **ni_configuration** register is an encoded version of the new partition size:

```
configuration = log2( partition_size ) + 2
```

Extra for Experts: By writing a 0 to the **configuration** register, you can temporarily isolate each node in the partition in its own "mini-partition," so that network operations performed by each node apply only to that node. Obviously, you should restore the original value of the **configuration** register when you are finished using this "mini-partition" effect.

The flag **ni_configuration_complete** is set to 0 while the repartitioning is in progress, and then set back to 1 to indicate its completion. At the same time, the **ni_configuration** register of the NI that sent the message is updated to the new partitioning value. The configuration registers and flags of the other NIs are not affected. An attempt to write a value to the **ni_configuration** register while **ni_configuration_complete** is 0 signals a Bus Error.

Important: A partition change should not be done when the Control Network is in use — the effect of doing so is undefined, but certainly disastrous.

6.8 Disabling the Control Network

There is one last flag in the **hodgepodge** register that has not yet been described:

ni_hodgepodge	Control register, includes the flag:
ni_cn_stop_send	Flag, disables Control Network sending.

This flag is used to completely disable the Control Network, preventing any messages from being sent into it — including the periodic "idle" packets that are sent when the network is not otherwise being used.

The **stop_send** flag is generally used only during an NI Reset (see Section 6.10) when it is necessary to totally disable the Control Network. When the

stop_send flag is 1, the Control Network is disabled. When the **stop_send** flag is set to 0, normal network operations resume.

For the Curious: The Control Network is designed in such a way that packets are periodically sent into it even when the network is not in use. When no message is being sent by the user or by the OS, these "idle" packets simply contain no data, and have no effect on the nodes. However, idle packets *can* affect the state of the Control Network itself in unwelcome ways, especially during a reset operation, when it is important for the state of the network to remain unchanged.

For the Even More Curious: Because the Data Network operates in an essentially asynchronous manner, with messages being sent from the nodes "on demand," the Data Network does not transmit idle packets, and thus has nothing analogous to the Control Network's **stop_send** flag.

6.9 NI Serial Number

Finally, one NI register contains the hardware serial number of the NI chip:

ni_serial_number Version serial number of NI chip.

This serial number identifies the version of NI chip that is installed.

Usage Note: Most revisions of the NI chip do not have usefully distinguishable serial numbers, so this register is not particularly valuable.

6.10 NI Reset

Under the following conditions, the NI chip is completely reset:

- The system administrator requests a repartitioning of the CM-5.
- The system administrator uses the diagnostic hardware of the CM-5 to reset the processing nodes and networks.

When the NI is reset, a number of its register fields and flags are set to known states. The following events occur on an NI Reset:

- All abstain and lock flags are set to 1, thus isolating the NI from all networks. These flags are:

  ```
  ni_dr_lock      ni_ldr_lock     ni_rdr_lock
  ni_bc_lock      ni_sbc_lock     ni_com_lock
  ni_reduce_rec_abstain    ni_com_abstain
  ni_bc_rec_abstain        ni_sbc_rec_abstain
  ni_sync_global_abstain
  ```

- `ni_interrupt_level` is set to 0. This disables all colored interrupts.
- All sending and receiving FIFOs are cleared.
- `ni_flush_complete` and `ni_sync_global_complete` are set to 1.

The values of all other NI registers are undefined, and must be set by software.

NI Reset is triggered by a special broadcast interrupt, the Reset interrupt, that can be sent from another NI or from the partition manager. This interrupt is always effective and cannot be disabled.

Chapter 7

NI Programming Issues

This chapter presents a number of NI programming issues that you should keep in mind, as well as important performance and programming hints and warnings.

7.1 The Partition Manager

As described in Section 1.1.3, each node in a partition has a unique address in its partition. However, the PM is not part of this addressing scheme. The PM is always located outside the address space of the partition that it manages.

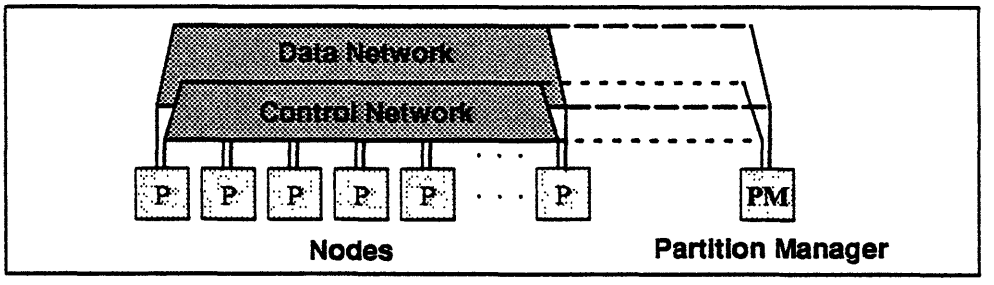


Figure 18. The partition manager stands apart from the partition it manages.

This means that sending messages to and from the partition manager involves some careful coordination between the PM and the nodes.

7.1.1 Sending Messages between the PM and the Nodes

To send a message from the PM to a node, use one of the broadcast interface interfaces. A common strategy is for the PM to send a broadcast message with two pieces of data: the address of the node that should "receive" the message, and the actual message itself. Each node does two broadcast interface reads, one to determine whether the address of the message matches the node's own address, and one to receive the message itself (or to discard it, if the address doesn't match).

To send a message from one or more nodes back to the PM, use the combine interface. The PM should set its **ni_rec_abstain** flag to 1 and its **ni_reduce_rec_abstain** flag to 0, so that it can receive a combine message without having to send a value. The nodes send a combine interface reduction message (for example, a **UADD_SCAN** reduction), and the PM, because of the settings of its abstain flags, receives the result as a combine interface message.

7.1.2 For the Curious: Using the Data Network

You can use the Data Network to send messages between the PM and the nodes. This is primarily useful in cases where you want to send a message to a specific node without forcing the other nodes to do a network operation at the same time. However, owing to the distinction between node and PM addressing, it's not as clear-cut an operation as using the combine interface.

To send a message from the partition manager to a specific node via the Data Network, you can simply use the node's relative address within the partition as the destination address for the message. To send a message from a node to its partition manager, the node must send a message outside of its partition. This can only be accomplished via an OS function call.

For example, in the CMOST operating system, the following function is used to send a message from a node to its PM:

```
int *source, length, tag
CMNA_interface_send_packet_to_scalar(source, length, tag)
```

where the *interface* abbreviation is **dr**, **ldr**, or **rdr**, depending on the interface involved. The partition manager can then receive this message as usual. The **send_packet_to_scalar** system call is currently implemented as a trap instruction, so it is much less efficient than using the combine interface.

7.2 Performance Hints

7.2.1 NI Register Operation Times

Here are some rough estimates of the time taken by a number of basic operations:

register access	(register variable):	1 cycle
cache memory	(previously accessed variable):	2–3 cycles
NI register read	(**ni_*interface*_status**, etc.):	7–8 cycles
NI register write	(**ni_*interface*_status**, etc.):	3–4 cycles
memory access	(newly accessed variable):	~25 cycles

The time taken to perform an NI register read or write operation is longer than the time taken for cached memory accesses, but much shorter than the time for full memory accesses. (NI register writes are faster than reads because an NI read operation requires that the node microprocessor wait for the read operation to move through the Mbus buffer before a value is actually read and returned.)

7.2.2 Reading and Writing Registers with Doubleword Values

While this document focuses for the most part on reading and writing network messages in terms of single (32-bit) words, you can also use doubleword (64-bit) operations in reading and writing network registers.

Writing a doubleword to a register has the same effect as writing two single-word values, but involves only one register operation. Likewise, reading a doubleword from a register is the same as reading two single words.

The combine interface is an exception to this rule, because of its pipelining feature. You can't use doubleword writes when you are pipelining combine operations. However, you *can* use doubleword reads with pipelined operations, and doubleword writes *are* permitted for non-pipelined combine operations.

In addition, attempting a doubleword read or write for a message that consists of only one word (as is the case for network-done tests) signals an error.

For C Programmers: To use doubleword read and write operations, the values you send must be doubleword aligned in memory. To ensure that this is the case, use the compiler switch **-dalign** when compiling any file that includes doubleword function calls or variable definitions. For example:

```
cc -c -g -DCM5 -dalign -I/usr/include ni_code.c
```

7.2.3 Use Message Discarding for Efficiency

When a message you are writing to a network send FIFO is discarded, it is completely discarded — effectively, it is as if you never began writing the message.

Many NI programmers take advantage of this property by writing a complete message to a network FIFO, and only then checking to see whether it was discarded (and if so, writing it again). This might seem a sloppy practice, but it is actually a safe and efficient strategy.

Because messages are typically only a few words long, and because the NI completely ignores a discarded message, it's perfectly reasonable to check the **send_ok** flag just once, after you've written the entire message. Also, if your code is properly written it should be rare for a message to be discarded, and thus unlikely that checking the **send_ok** flag after writing each value of the message provides any benefit. In fact, checking the **send_ok** flag after you write each value of a message can slow your code down considerably.

7.2.4 Set the Abstain Flags Once and Forget Them

In most cases, abstain flags of a network interface can be changed only when the network is not in use — that is, when there are no messages pending in either the send or receive FIFOs, and no messages in transit in the network. While this certainly does not prevent you from toggling the state of the abstain flags within your code, it does make this kind of flag-toggling more prone to programming errors.

A more straightforward strategy to use is to set the values of the abstain flags once, at the beginning of your program, leave them alone while the program runs, and then restore their original values before your program exits.

Note: This last point is important. As noted in Section 2.6.4, some programming systems (such as CMMD) use the abstain flags for their own purposes. These systems are written with the assumption that the abstain flags won't change unexpectedly, so if the flags do change these systems may not operate correctly.

When you alter the values of the abstain flags, you must take care to save the original settings of these flags and to restore them before your code exits. Failing to do so can cause your code to signal obscure errors that are hard to trace.

7.3 Potential Programming Traps and Snares

Here are some potential sources of serious errors that you should keep in mind.

Note: Some of the notes and warnings below are included in earlier chapters. They are repeated here so that you can find them quickly.

7.3.1 Pay Attention to Data Network Addresses

When sending a Data Network message with a relative address, the address must be valid within the current partition. If an address higher than `CMNA_partition_size` is supplied, the NI signals an error.

Also, there is currently a 20-bit limit on the length of a Data Network address, and the remaining high-order bits in a 32-bit address value must be 0. If any of these high-order bits are nonzero, the NI signals a serious error, and in some cases the entire partition of nodes may crash. You should either write your code so that the high-order bits of a network address can never be other than zero, or failing that mask out the top 12 bits of an address before using it.

7.3.2 Check the Tag before Retrieving a Data Network Message

As described in Section 3.5.2, whether or not you use tag-driven interrupts to receive messages, you must take care not to accidentally read a message intended as an interrupt, because the operating system of the CM-5 itself sends Data Network messages with interrupt tags.

The Data Network only checks the tag field of a message after the message has been delivered to the receive FIFO. This means that if you're not careful, you can accidentally read a message with an interrupt-triggering tag value before the NI has signaled the interrupt. The effect of doing so is unpredictable. An error may be signaled, or your partition may crash.

To avoid this problem, check the tag value of a Data Network message before retrieving it to make certain that it is a non-interrupting message (that is, a message with a tag value from 0 to 7 that you have not assigned as an interrupt tag.)

7.3.3 Make Sure Doubleword Data Is Doubleword Aligned

C Programmers: This is also mentioned in the performance section above, but it's as well to re-emphasize it. When you use doubleword read and write operations in your C code, you must compile your code with the **`-dalign`** compiler switch, so that doubleword values are properly aligned in memory:

```
cc -c -g -DCM5 -dalign -I/usr/include ni_code.c
```

If the doubleword values in your code are not properly aligned, the nodes will most likely signal "illegal address" errors, and your code won't run.

7.3.4 Order Is Important in Combine Messages

As noted in Section 4.2.6, for scan messages longer than one word, the order in which the words of the message are written depends on the combine operation:

- Maximum operations require the most significant word to be written first.
- Both types of addition require the least significant word to be written first.
- Inclusive and exclusive OR have no word-ordering requirement.

7.3.5 Restriction on Network-Done Operations for Rev A NI Chips

As described in Section 4.2.7, the **`ni_dr_message_count`** register is used to keep track of the number of Data Network messages sent and received, and also to determine when a network-done operation has completed.

Revision A NI chips, however, do not correctly increment and decrement this register. This defect has been corrected in later revisions, but to run code on a machine that includes *any* Rev A chips, you must use a software workaround: you must yourself use a program variable to keep track of the number of messages sent and received, and you must "force" the NI message-count register to have this value during a network-done operation.

Note: This software workaround is necessary if and only if the CM-5 on which you execute your code contains any Rev A NI chips in its processing nodes. On CM-5 systems with no Rev A NI chips, this workaround is not needed (and is inefficient, as well).

The recommended variable to use is **CMNA_router_msg_count** (this variable is predefined for you in the header files loaded by **cmna.h**). The workaround strategy is as follows:

- Set **CMNA_router_msg_count** to 0 at the beginning of the node program (for example, at the same point that you set the values of the abstain flags).
- Every time the node program successfully sends a message via the Data Network (that is, writes a message to the send FIFO and detects that the **send_ok** flag is set), it should increment the **msg_count** variable.
- Likewise, whenever the node program receives a message from the Data Network (that is, detects that the **rec_ok** flag is set and reads all the values of the message), it should decrement the **msg_count** variable.
- Just before sending a network-done message, write the current value of the **msg_count** variable into the **msg_count** register.

  Note: Because the **msg_count** register is restricted to the supervisor, user code must make an OS call to set its value. In the CMOST operating system, the following system call is used:

```
CMOS_set_dr_msg_count_reg(CMNA_router_msg_count);
```

- While waiting for the network-done operation to complete, repeatedly write the current value of the **msg_count** variable into the register. This must be done before checking the **ni_router_done_complete** flag. Otherwise, the flag may not be correct.

7.3.6 Simulating Receipt of Messages

As noted in Section 3.4.2, a hardware defect in the NI chip does not allow **recv** registers to be written by the supervisor to simulate the arrival of messages. The workaround is for a node to send a message into the network using its own address as the destination. Assuming the network is clear (as it is, for example, during context switches) this causes the message to be delivered to the front of the node's receive queue.

7.3.7 Broadcast Enabling

As noted in Section 4.1.7, each broadcast interface has a **send_enable** flag. These flags are set to 0 by default in the CMOST operating system, and must be set to 1 before broadcasts are used. The CMOST system call to set these flags is:

```
CMNA_participate_in(NI_BC_SEND_ENABLE);
CMNA_participate_in(NI_SBC_SEND_ENABLE);
```

7.3.8 Broadcast and Combine Interface Conflicts

Because of the way the broadcast and combine interfaces interact, you should be careful in using the abstain flags of these interfaces. If your code causes a node (processing node or PM) to abstain from the combine interface, and if:

- the abstaining node is sending a broadcast message
- simultaneously, the other nodes are sending a combine message,

then because of timing conflicts in the Control Network hardware, the two types of messages can collide, possibly causing your partition to crash. This situation most often occurs when you have instructed the PM to abstain from the combine interface so that it can receive the results of a scan or reduction operation, yet at the same time you want the PM to broadcast messages to the nodes telling them what to do. The conflict arises when the PM needs to broadcast a message at the same time that the nodes are sending a combine message. To avoid this problem, your code must include safety checks that prevent a broadcast message from being sent at the same time that other nodes are sending a combine message. The CMOST operating system includes a function you can call to send a broadcast message that implicitly performs this safety checking:

```
int *msg, length;
CMNA_bc_send_msg(msg, length);
```

7.3.9 Be Careful When Altering Abstain Flags

As mentioned in Section 2.6.4, some programming systems (such as CMMD) use the abstain flags for their own purposes. When you alter the values of the abstain flags, you must take care to save the original settings of these flags and to restore them before handing control back to these systems. Failing to do so can cause either user or OS code to signal obscure errors that are hard to trace.

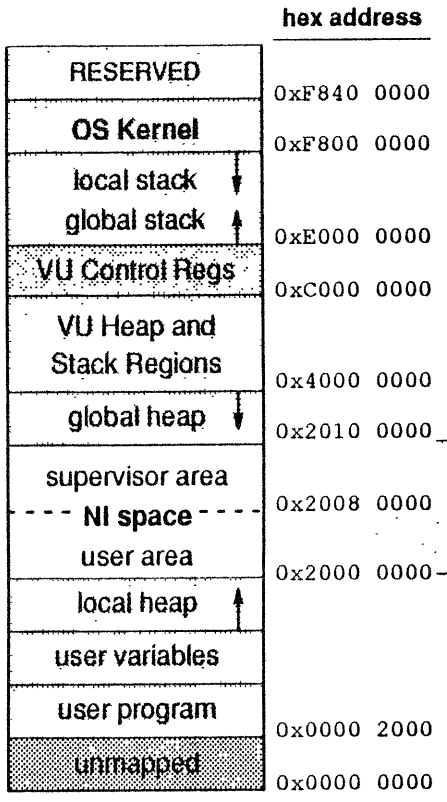
Appendixes

Appendix A

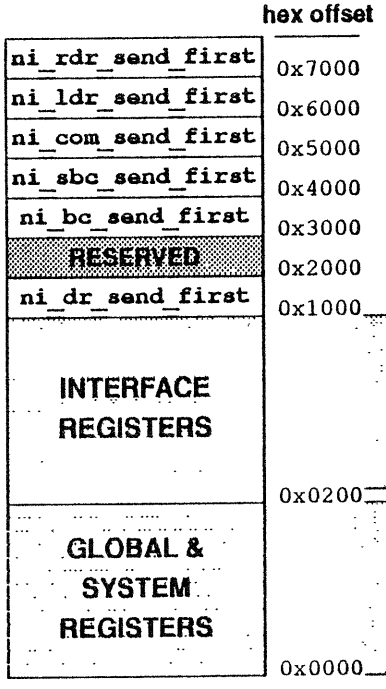
NI Memory Map

On the following page is a two-sided memory and register map, showing the overall arrangement of the NI's registers, as well as the layout of subfields within those registers. .

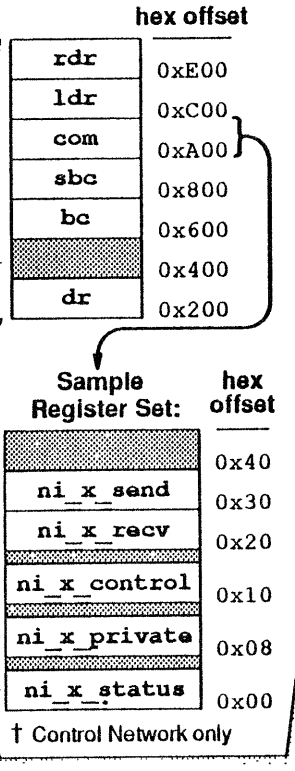
Node Virtual Memory Map
(with or without VUs installed)



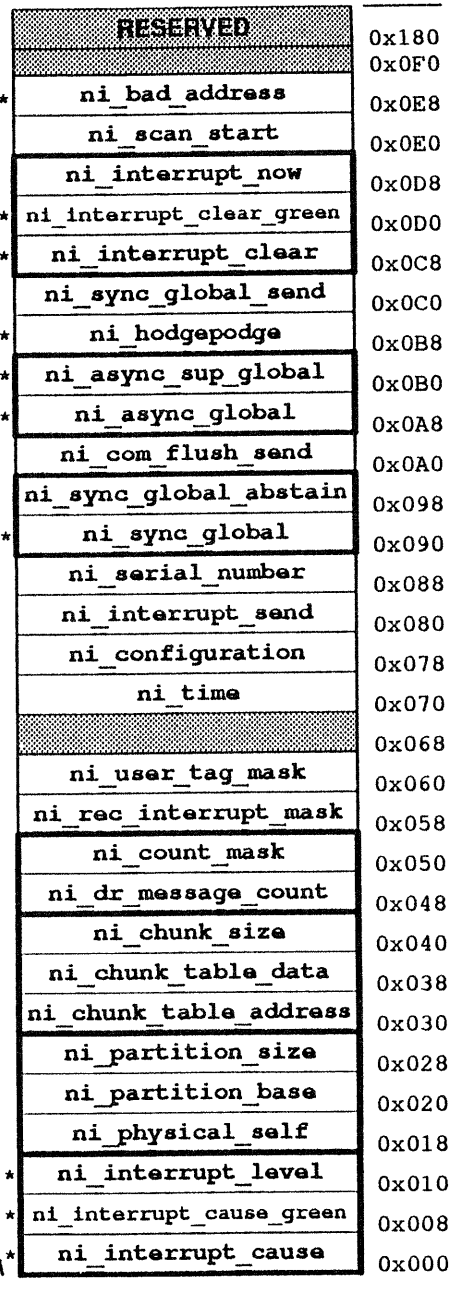
NI Virtual Memory Area
(user or supervisor)



Interface Registers

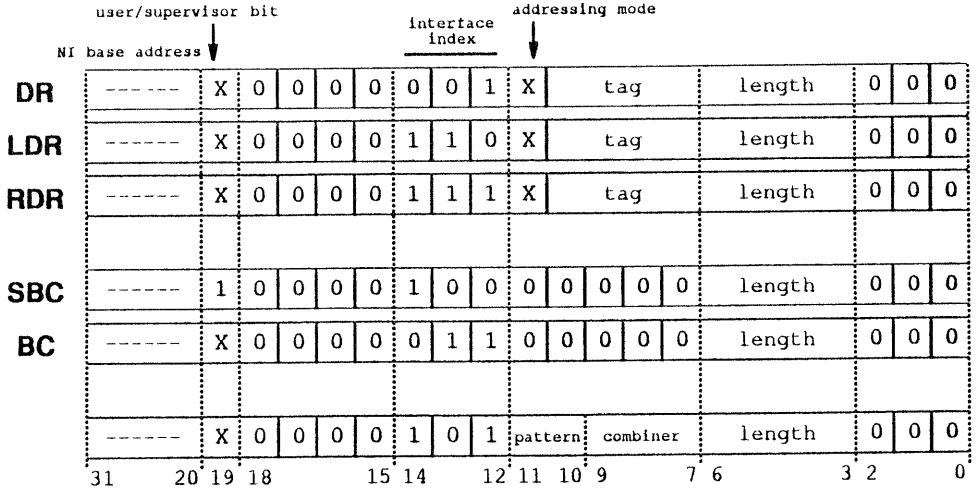


Global & System Registers



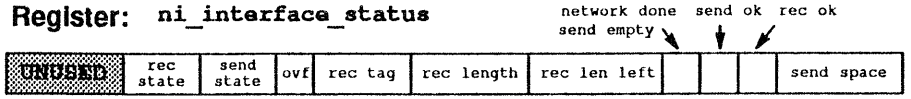
* Indicates register with subfields
(See listings on reverse side)

ni_interface_send_first Addressing Patterns



Register & Memory Map

Register: ni_interface_status

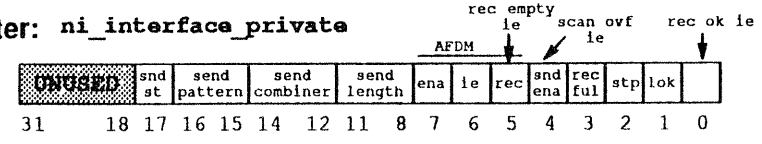


Field Name:	Pos:	Size:	DR	L/RDR	S/BC	COM
ni_send_space	0	4	√	√	√	√
ni_rec_ok	4	1	√	√	√	√
ni_send_ok	5	1	√	√	√	√
ni_router_done_complete	6	1	√			
ni_send_empty	6	1			√	√
ni_rec_length_left	7	4	√	√	√	√
ni_rec_length	11	4	√	√		√
ni_dr_rec_tag	15	4	√	√		
ni_com_scan_overflow	20	1				√
ni_dr_send_state	21	2	√			
ni_dr_rec_state	23	2	√			

Register: ni_interface_control

Field Name:	Pos:	Size:	DR	L/RDR	S/BC	COM
ni_rec_abstain	0	1			√	√
ni_reduce_rec_abstain	1	1				√

Register: ni_interface_private



Field Name:	Pos:	Size:	DR	L/RDR	S/BC	COM
ni_rec_ok_ie	0	1	√	√	√	√
ni_lock	1	1	√	√	√	√
ni_rec_stop	2	1	√			√
ni_send_stop	2	1			√	
ni_rec_full	3	1	√	√	√	√
ni_send_enable	4	1			√	
ni_com_scan_overflow_ie	4	1				√
ni_dr_rec_all_fall_down	5	1	√	√		
ni_com_rec_empty_ie	5	1				√
ni_all_fall_down_ie	6	1	√			
ni_all_fall_down_enable	7	1	√			
ni_com_send_length	8	4				√
ni_com_send_combiner	12	3				√
ni_com_send_pattern	15	2				√
ni_com_send_start	17	1				√

Registers: (same bit positions, all flags)
ni_interrupt_cause/clear

Field Name:	Pos:
ni_cause/clear_internal_fault	0
ni_cause/clear_mc_error	1
ni_cause/clear_cmu_error	2
ni_cause/clear_bc_interrupt_red	3
ni_cause/clear_cn_checksum_error	4
ni_cause/clear_cn_hard_error	5
ni_cause/clear_dr_checksum_error	6
ni_cause/clear_timer_interrupt	7
ni_cause/clear_bc_interrupt_orange	8
ni_cause/clear_bc_interrupt_yellow	9
ni_cause/clear_bc_or_com_collision	10
ni_cause/clear_com_abstain_changed	11
ni_cause/clear_dr_count_negative	12
ni_cause/clear_bad_relative_address	13
ni_cause/clear_bad_memory_access	14

Registers: (same bit positions, all flags)
ni_interrupt_cause/clear_green

Field Name:	Pos:
ni_cause/clear_bc_interrupt_green	0
ni_cause/clear_scan_overflow	1
ni_cause/clear_bc_rec_ok	2
ni_cause/clear_sbc_rec_ok	3
ni_cause/clear_com_rec_ok	4
ni_cause/clear_com_rec_empty	5
ni_cause/clear_sync_global_rec	6
ni_cause/clear_global_rec	7
ni_cause/clear_supervisor_global_rec	8
ni_cause/clear_dr_rec_ok	9
ni_cause/clear_ldr_rec_ok	10
ni_cause/clear_rdr_rec_ok	11
ni_cause/clear_dr_rec_tag	12
ni_cause/clear_dr_rec_all_fall_down	13

Register: ni_hodgepodge

Field Name:	Pos:	Size:
ni_global_rec_ie	0	1
ni_supervisor_global_rec_ie	1	1
ni_flush_complete	2	1
ni_interrupt_send_ok	3	1
ni_configuration_complete	4	1
ni_interrupt_rec_enable	5	1
ni_sync_global_rec_ie	6	1
ni_timer_ie	7	1
ni_cn_stop_send	8	1

Register: ni_interrupt_level

Field Name:	Pos:	Size:
ni_interrupt_level_green	0	1
ni_interrupt_level_yellow	8	1
ni_interrupt_level_orange	16	1
ni_interrupt_level_red	24	1

Register: ni_bad_address

Field Name:	Pos:	Size:
ni_bad_address_low	0	20
ni_bad_address_type	20	12

Register: ni_sync_global

Field Name:	Pos:	Size:
ni_sync_global_rec	0	1
ni_sync_global_complete	1	1

Register: ni_async_global

Field Name:	Pos:	Size:
ni_global_send	0	1
ni_global_rec	1	1

Register: ni_async_sup_global

Field Name:	Pos:	Size:
ni_supervisor_global_send	0	1
ni_supervisor_global_rec	1	1

Appendix B

NI Registers, Fields, and Constants

This appendix presents a tabular summary of the registers and fields of the NI chip, as well as the programming constants that can be used to locate them.

Note: To get access to these constants, your program must either include the header file `cmna.h` (see Section 1.3.3), or include the appropriate header file from the CMNA header file set (see Appendix F).

B.1 NI Registers

For each register the following information is provided:

- the name of the register
- the hex offset of the register from the user or supervisor base address
- the size of the register in bits
- the length (number of memory words to which the register is mapped)
- the read/write permissions of the register for both user and supervisor

Register Constants

Note: With the exception of the `send_first` registers (which are described in Section B.3 below), the names of the constants used to access NI registers are derived from the names of the registers themselves by uppercasing the register name and adding the suffix "`_A`".

Each register constant provides the absolute address of the register, in either the user or supervisor memory area, depending on which header file (`cmna.h` or `cmna_sup.h`) has been included.

B.1.1 Global and System Registers

Register Name:	Address:	Size:	Len:	Permissions: Super:	User:
ni_interrupt_cause	0x0000	15	1	R/W	None
ni_interrupt_cause_green	0x0008	14	1	R/W	None
ni_interrupt_level	0x0010	32	1	R/W	None
ni_physical_self	0x0018	20	1	R/W	None
ni_partition_base	0x0020	20	1	R/W	None
ni_partition_size	0x0028	20	1	R/W	None
ni_chunk_table_address	0x0030	6	1	R/W	None
ni_chunk_table_data	0x0038	8	1	R/W	None
ni_chunk_size	0x0040	3	1	R/W	None
ni_dr_message_count	0x0048	32	1	R/W	None
ni_count_mask	0x0050	16	1	R/W	None
ni_rec_interrupt_mask	0x0058	16	1	R/W	None
ni_user_tag_mask	0x0060	16	1	R/W	None
ni_time	0x0070	32	1	R/W	R
ni_configuration	0x0078	5	1	R/W	None
ni_interrupt_send	0x0080	5	1	R/W	None
ni_serial_number	0x0088	32	1	R	None
ni_sync_global	0x0090	2	1	R	R
ni_sync_global_abstain	0x0098	1	1	R/W	R/W
ni_com_flush_send	0x00A0	1	1	W	None
ni_async_global	0x00A8	2	1	R/W	R/W
ni_async_sup_global	0x00B0	2	1	R/W	None
ni_hodgepodge	0x00B8	6	1	R/W	None
ni_sync_global_send	0x00C0	1	1	R/W	R/W
ni_interrupt_clear	0x00C8	15	1	W	None
ni_interrupt_clear_green	0x00D0	14	1	W	None
ni_interrupt_now	0x00D8	32	1	R/W	None
ni_scan_start	0x00E0	1	1	R/W	R/W
ni_bad_address	0x00E8	32	1	R/W	None

B.1.2 Network Interface Registers

Combined Data Network Interface (DR)

Register Name:	Address:	Size:	Len:	Permissions: Super:	User:
`ni_dr_status`	0x0200	24	1	R/W	R
`ni_dr_private`	0x0208	10	1	R/W	None
`ni_dr_recv`	0x0220	32	16	R/W	R
`ni_dr_send`	0x0230	32	16	W	W
`ni_dr_send_first` (block)	0x1000	32	2	W	W

Left Data Network Interface (LDR)

Register Name:	Address:	Size:	Len:	Permissions: Super:	User:
`ni_ldr_status`	0x0c00	32	1	R/W	R
`ni_ldr_private`	0x0c08	24	1	R/W	None
`ni_ldr_recv`	0x0c20	32	16	R/W	R
`ni_ldr_send`	0x0c30	32	16	W	W
`ni_ldr_send_first` (block)	0x6000	32	2	W	W

Right Data Network Interface (RDR)

Register Name:	Address:	Size:	Len:	Permissions: Super:	User:
`ni_rdr_status`	0x0e00	32	1	R/W	R
`ni_rdr_private`	0x0e08	24	1	R/W	None
`ni_rdr_recv`	0x0e20	32	16	R/W	R
`ni_rdr_send`	0x0e30	32	16	W	W
`ni_rdr_send_first` (block)	0x7000	32	2	W	W

Broadcast Interface (BC)

Register Name:	Address:	Size:	Len:	Permissions: Super:	User:
`ni_bc_status`	0x0600	6	1	R	R
`ni_bc_private`	0x0608	17	1	R/W	None
`ni_bc_control`	0x0610	1	1	R/W	R/W
`ni_bc_recv`	0x0620	32	16	R/W	R
`ni_bc_send`	0x0630	32	16	W	W
`ni_bc_send_first` (block)	0x3000	32	2	W	W

Supervisor Broadcast Interface (SBC)

Register Name:	Address:	Size:	Len:	Permissions: Super:	User:
`ni_sbc_status`	0x0800	6	1	R	None
`ni_sbc_private`	0x0808	17	1	R/W	None
`ni_sbc_control`	0x0810	1	1	R/W	None
`ni_sbc_recv`	0x0820	32	16	R/W	None
`ni_sbc_send`	0x0830	32	16	W	None
`ni_sbc_send_first` (block)	0x4000	32	2	W	None

Combine Interface (COM)

Register Name:	Address:	Size:	Len:	Permissions: Super:	User:
`ni_com_status`	0x0a00	12	1	R/W	R
`ni_com_private`	0x0a08	6 (18)	1	R/W	None
`ni_com_control`	0x0a10	2	1	R/W	R/W
`ni_com_recv`	0x0a20	32	16	R/W	R
`ni_com_send`	0x0a30	32	16	R/W	W
`ni_com_send_first` (block)	0x5000	32	2	W	W

B.2 NI Message Length Limit Constants

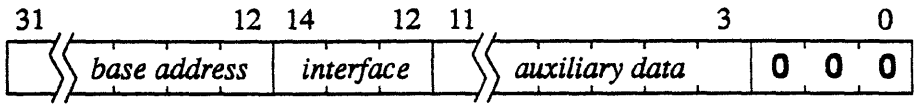
The following constants give the message length limits of the network interfaces:

`MAX_ROUTER_MSG_WORDS`	DR/LDR/RDR interface length limit.
`MAX_COMBINE_MSG_WORDS`	Combine (COM) interface length limit.
`MAX_BROADCAST_MSG_WORDS`	Broadcast (BC) interface length limit.
`MAX_SBC_MSG_WORDS`	Supervisor broadcast (SBC) length limit.

These constants determine the maximum values that can be supplied in the *length* component of the auxiliary data of a network message. (See the descriptions of the auxiliary data formats for the various interfaces below.)

B.3 Send First Register Addresses

The **send_first** address for a network message is a 32-bit value of the form:



where *interface* is the interface number (an integer from 0 to 7 representing the interface being used), *auxiliary data* is the auxiliary information of the message. (The *base address* portion is the base address of the NI memory area, either user or supervisor.)

The following constants are used to construct **send_first** addresses:

NI_BASE	The NI base address.
SF_FIFO_OFFSET	The *interface* field offset (12).
AUXILIARY_START_P	The *auxiliary data* field offset (3).

To construct a **send_first** address, add the following values, left-shifted as shown:

The NI base address:	**NI_BASE**	
The *interface* constant:	*interface_number*	<< **SF_FIFO_OFFSET**
The auxiliary data:	*auxiliary_data*	<< **AUXILIARY_START_P**

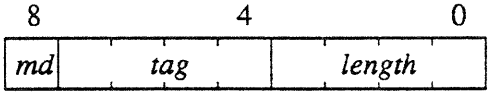
The following *interface_number* constants are defined:

DATA_ROUTER_FIFO	DR network interface (1).
LEFT_DR_FIFO	LDR network interface (6).
RIGHT_DR_FIFO	RDR network interface (7).
USER_BC_FIFO	User broadcast (BC) interface (3).
SUPERVISOR_BC_FIFO	Supervisor broadcast (SBC) interface (4).
COMBINE_FIFO	Combine (COM) interface (5).

The constants specifying the *auxiliary data* format for each interface are listed in the sections below.

Data Network (DR/LDR/RDR) Auxiliary Data Fields

The format of the auxiliary data of a Data Network message is:



where

- *md* is the addressing mode (0 = relative, 1 = physical).
- *tag* is the 4-bit tag value.
- *length* is the length of the message in words, excluding address word.

The following constants specify the starting bit positions of these fields:

`NI_DR_SEND_AUXILIARY_ADDRESS_MODE_P`	The *md* field offset (8).
`NI_DR_SEND_AUXILIARY_TAG_P`	The *tag* field offset (4).
`NI_DR_SEND_AUXILIARY_LENGTH_P`	The *length* field offset (0).

To construct a send_first address, add the following values:

The *md* flag:	*md*	`<< NI_DR_SEND_AUXILIARY_ADDRESS_MODE_P`
The *tag* value:	*tag*	`<< NI_DR_SEND_AUXILIARY_TAG_P`
The *length* value:	*length*	`<< NI_DR_SEND_AUXILIARY_LENGTH_P`

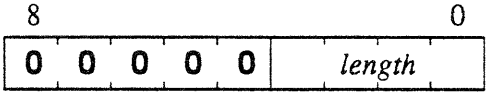
The following constants can be used to specify the *md* flag:

`RELATIVE`	Relative node addressing (0).
`PHYSICAL`	Physical node addressing (1).

The *tag* can be any value from 0 to 3 inclusive for user messages, or from 0 to 15 for supervisor messages. (The *length* value limit is given in Section B.2.)

Broadcast (BC/SBC) Auxiliary Data Fields

The format of the auxiliary data of a broadcast message is:

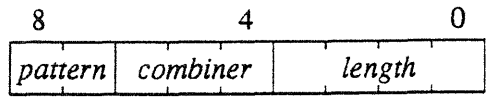


where *length* is the length of the message in words. (The high-order bits of the auxiliary data have no useful meaning, but must always be 0.) The following constant specifies the starting bit position of the *length* field:

`NI_BC_SEND_AUXILIARY_LENGTH_P`	The *length* field offset (0).

Combine Auxiliary Data Fields

The format of the auxiliary data of a combine interface message is:



where

- *pattern* is a two-bit value selecting the order in which values are combined
- *combiner* is a three-bit value selecting the combine operation performed
- *length* is the length of the message in words

The following constants specify the starting bit positions of these fields:

`NI_COM_SEND_AUXILIARY_PATTERN_P`	The *pattern* field offset (7).
`NI_COM_SEND_AUXILIARY_COMBINER_P`	The *combiner* field offset (4).
`NI_COM_SEND_AUXILIARY_LENGTH_P`	The *length* field offset (0).

To construct a **`send_first`** address, add the following values:

The *pattern* value:	*pattern*	<< `NI_COM_SEND_AUXILIARY_PATTERN_P`
The *combiner* value:	*combiner*	<< `NI_COM_SEND_AUXILIARY_COMBINER_P`
The *length* value:	*length*	<< `NI_COM_SEND_AUXILIARY_LENGTH_P`

The following constants can be used to specify the value of the *pattern* field:

`SCAN_FORWARD`	Forward scan pattern (2).
`SCAN_BACKWARD`	Backward scan pattern (1).
`SCAN_REDUCE`	Reduction scan pattern (3).
`SCAN_ROUTER_DONE`	Network-done operation (0).

The following constants can be used to specify the value of the *combiner* field:

`OR_SCAN`	Forward scan pattern (0).
`ADD_SCAN`	Backward scan pattern (1).
`XOR_SCAN`	Reduction scan pattern (2).
`UADD_SCAN`	Network-done operation (3).
`MAX_SCAN`	Reduction scan pattern (4).
`ASSERT_ROUTER_DONE`	Network-done operation (5).

B.4 NI Fields

The register subfields of the NI are presented below, grouped by register. For each field, the following information is provided:

- the name of the field
- the name of the position constant used to access the field (see note below)
- the starting position and bit length of the field
- the read/write permissions of the field for both user and supervisor

Note: The programming constants used to access NI fields come in pairs.

One constant, with a suffix of "`_P`", gives the starting bit position of the field. In the tables below, this value appears in the **Pos:** (position) column.

The other constant, with a suffix of "`_L`", gives the length of the field. In the tables below, this value appears in the **Len:** (length) column.

Only the "`_P`" constant name is shown in the tables below. Unless otherwise noted, you can assume that the "`_L`" constant exists as well.

B.4.1 Combined Data Network (DR) Fields

The `ni_dr_status` Register

Field Name:	Constant:	Pos:	Len:	Permissions: Super:	User:
`ni_send_space`	`NI_SEND_SPACE_P`	0	4	R	R
`ni_rec_ok`	`NI_REC_OK_P`	4	1	R	R
`ni_send_ok`	`NI_SEND_OK_P`	5	1	R	R
`ni_router_done_complete`	`NI_ROUTER_DONE_COMPLETE_P`	6	1	R	R
`ni_rec_length_left`	`NI_REC_LENGTH_LEFT_P`	7	4	R/W	R
`ni_rec_length`	`NI_REC_LENGTH_P`	11	4	R/W	R
`ni_dr_rec_tag`	`NI_DR_REC_TAG_P`	15	4	R/W	R
`ni_dr_send_state`	`NI_DR_SEND_STATE_P`	21	2	R	R
`ni_dr_rec_state`	`NI_DR_REC_STATE_P`	23	2	R	R

The ni_dr_private Register

Field Name:	Constant:	Pos:	Len:	Permissions: Super:	User:
ni_rec_ok_ie	NI_REC_OK_IE_P	0	1	R/W	None
ni_lock	NI_LOCK_P	1	1	R/W	None
ni_rec_stop	NI_REC_STOP_P	2	1	R/W	None
ni_rec_full	NI_REC_FULL_P	3	1	R	None
ni_dr_rec_all_fall_down	NI_DR_REC_ALL_FALL_DOWN_P	5	1	R/W	None
ni_all_fall_down_ie	NI_ALL_FALL_DOWN_IE_P	6	1	R/W	None
ni_all_fall_down_enable	NI_ALL_FALL_DOWN_ENABLE_P	7	1	R/W	None

B.4.2 Left Data Network Interface (LDR) Fields

The ni_ldr_status Register

Field Name:	Constant:	Pos:	Len:	Permissions: Super:	User:
ni_send_space	NI_SEND_SPACE_P	0	4	R	R
ni_rec_ok	NI_REC_OK_P	4	1	R	R
ni_send_ok	NI_SEND_OK_P	5	1	R	R
ni_rec_length_left	NI_REC_LENGTH_LEFT_P	7	4	R/W	R
ni_rec_length	NI_REC_LENGTH_P	11	4	R/W	R
ni_dr_rec_tag	NI_DR_REC_TAG_P	15	4	R/W	R

The ni_ldr_private Register

Field Name:	Constant:	Pos:	Len:	Permissions: Super:	User:
ni_rec_ok_ie	NI_REC_OK_IE_P	0	1	R/W	None
ni_lock	NI_LOCK_P	1	1	R/W	None
ni_rec_full	NI_REC_FULL_P	3	1	R	None
ni_dr_rec_all_fall_down	NI_DR_REC_ALL_FALL_DOWN_P	5	1	R/W	None

B.4.3 Right Data Network Interface (RDR) Fields

The `ni_rdr_status` Register

Field Name:	Constant:	Pos:	Len:	Permissions: Super:	User:
`ni_send_space`	`NI_SEND_SPACE_P`	0	4	R	R
`ni_rec_ok`	`NI_REC_OK_P`	4	1	R	R
`ni_send_ok`	`NI_SEND_OK_P`	5	1	R	R
`ni_rec_length_left`	`NI_REC_LENGTH_LEFT_P`	7	4	R/W	R
`ni_rec_length`	`NI_REC_LENGTH_P`	11	4	R/W	R
`ni_dr_rec_tag`	`NI_DR_REC_TAG_P`	15	4	R/W	R

The `ni_rdr_private` Register

Field Name:	Constant:	Pos:	Len:	Permissions: Super:	User:
`ni_rec_ok_ie`	`NI_REC_OK_IE_P`	0	1	R/W	None
`ni_lock`	`NI_LOCK_P`	1	1	R/W	None
`ni_rec_full`	`NI_REC_FULL_P`	3	1	R	None
`ni_dr_rec_all_fall_down`	`NI_DR_REC_ALL_FALL_DOWN_P`	5	1	R/W	None

B.4.4 Broadcast Interface (BC) Fields

The `ni_bc_status` Register

Field Name:	Constant:	Pos:	Len:	Permissions: Super:	User:
`ni_send_space`	`NI_SEND_SPACE_P`	0	4	R	R
`ni_rec_ok`	`NI_REC_OK_P`	4	1	R	R
`ni_send_ok`	`NI_SEND_OK_P`	5	1	R	R
`ni_send_empty`	`NI_SEND_EMPTY_P`	6	1	R	R
`ni_rec_length_left`	`NI_REC_LENGTH_LEFT_P`	7	4	R	R

The ni_bc_private Register

Field Name:	Constant:	Pos:	Len:	Permissions: Super:	User:
ni_rec_ok_ie	NI_REC_OK_IE_P	0	1	R/W	None
ni_lock	NI_LOCK_P	1	1	R/W	None
ni_rec_stop	NI_REC_STOP_P	2	1	R/W	None
ni_rec_full	NI_REC_FULL_P	3	1	R	None
ni_send_enable	NI_SEND_ENABLE_P	4	1	R/W	None

The ni_bc_control Register

Field Name:	Constant:	Pos:	Len:	Permissions: Super:	User:
ni_rec_abstain	NI_REC_ABSTAIN_P	0	1	R/W	R/W

B.4.5 Supervisor Broadcast Interface (SBC) Fields

The ni_sbc_status Register

Field Name:	Constant:	Pos:	Len:	Permissions: Super:	User:
ni_send_space	NI_SEND_SPACE_P	0	4	R	None
ni_rec_ok	NI_REC_OK_P	4	1	R	None
ni_send_ok	NI_SEND_OK_P	5	1	R	None
ni_send_empty	NI_SEND_EMPTY_P	6	1	R	None
ni_rec_length_left	NI_REC_LENGTH_LEFT_P	7	4	R	None

The ni_sbc_private Register

Field Name:	Constant:	Pos:	Len:	Permissions: Super:	User:
ni_rec_ok_ie	NI_REC_OK_IE_P	0	1	R/W	None
ni_lock	NI_LOCK_P	1	1	R/W	None
ni_rec_stop	NI_REC_STOP_P	2	1	R/W	None
ni_rec_full	NI_REC_FULL_P	3	1	R	None
ni_send_enable	NI_SEND_ENABLE_P	4	1	R/W	None

The ni_sbc_control Register

Field Name:	Constant:	Pos:	Len:	Permissions: Super:	User:
ni_rec_abstain	NI_REC_ABSTAIN_P	0	1	R/W	None

B.4.6 Combine Interface (COM) Fields

The ni_com_status Register

Field Name:	Constant:	Pos:	Len:	Permissions: Super:	User:
ni_send_space	NI_SEND_SPACE_P	0	4	R	R
ni_rec_ok	NI_REC_OK_P	4	1	R	R
ni_send_ok	NI_SEND_OK_P	5	1	R	R
ni_send_empty	NI_SEND_EMPTY_P	6	1	R	R
ni_rec_length_left	NI_REC_LENGTH_LEFT_P	7	4	R/W	R
ni_rec_length	NI_REC_LENGTH_P	11	4	R/W	R
ni_com_scan_overflow	NI_COM_SCAN_OVERFLOW_P	20	1	R/W	R

The ni_com_private Register

Field Name:	Constant:	Pos:	Len:	Permissions: Super:	User:
ni_rec_ok_ie	NI_REC_OK_IE_P	0	1	R/W	None
ni_lock	NI_LOCK_P	1	1	R/W	None
ni_rec_stop	NI_REC_STOP_P	2	1	R/W	None
ni_rec_full	NI_REC_FULL_P	3	1	R	None
ni_com_scan_overflow_ie	NI_COM_SCAN_OVERFLOW_IE_P	4	1	R/W	None
ni_com_rec_empty_ie	NI_COM_REC_EMPTY_IE_P	5	1	R/W	None
ni_com_send_length	NI_COM_SEND_LENGTH_P	8	4	R	None
ni_com_send_combiner	NI_COM_SEND_COMBINER_P	12	3	R	None
ni_com_send_pattern	NI_COM_SEND_PATTERN_P	15	2	R	None
ni_com_send_start	NI_COM_SEND_START_P	17	1	R	None

The ni_com_control Register

Field Name:	Constant:	Pos:	Len:	Permissions: Super:	User:
ni_rec_abstain	NI_REC_ABSTAIN_P	0	1	R/W	R/W
ni_reduce_rec_abstain	NI_REDUCE_REC_ABSTAIN_P	1	1	R/W	R/W

B.4.7 Global Interface Fields

The ni_sync_global Register

Field Name:	Constant:	Pos:	Len:	Permissions: Super:	User:
ni_sync_global_rec	NI_SYNC_GLOBAL_REC_P	0	1	R	R
ni_sync_global_complete	NI_SYNC_GLOBAL_COMPLETE_P	1	1	R	R

The ni_async_global Register

Field Name:	Constant:	Pos:	Len:	Permissions: Super:	User:
ni_global_send	NI_GLOBAL_SEND_P	0	1	R/W	R/W
ni_global_rec	NI_GLOBAL_REC_P	1	1	R	R

The ni_async_sup_global Register

Field Name:	Constant:	Pos:	Len:	Permissions: Super:	User:
ni_supervisor_global_send	NI_SUPERVISOR_GLOBAL_SEND_P	0	1	R/W	None
ni_supervisor_global_rec	NI_SUPERVISOR_GLOBAL_REC_P	1	1	R	None

B.4.8 Interrupt Register Fields

Note: The position ("`_P`") constants for these flags are as described above. The length for all flags (1) is given by the single constant `NI_INTERRUPT_L`.

The `ni_interrupt_cause` Register

Flag Name:	Pos:	Len:	Permissions: Super:	User:
`ni_cause_internal_fault`	0	1	R/W	None
`ni_cause_mc_error`	1	1	R/W	None
`ni_cause_cmu_error`	2	1	R/W	None
`ni_cause_bc_interrupt_red`	3	1	R/W	None
`ni_cause_cn_checksum_error`	4	1	R/W	None
`ni_cause_cn_hard_error`	5	1	R/W	None
`ni_cause_dr_checksum_error`	6	1	R/W	None
`ni_cause_timer_interrupt`	7	1	R/W	None
`ni_cause_bc_interrupt_orange`	8	1	R/W	None
`ni_cause_bc_interrupt_yellow`	9	1	R/W	None
`ni_cause_bc_or_com_collision`	10	1	R/W	None
`ni_cause_com_abstain_changed`	11	1	R/W	None
`ni_cause_dr_count_negative`	12	1	R/W	None
`ni_cause_bad_relative_address`	13	1	R/W	None
`ni_cause_bad_memory_access`	14	1	R/W	None

The `ni_interrupt_cause_green` Register

Flag Name:	Pos:	Len:	Permissions: Super:	User:
`ni_cause_bc_interrupt_green`	0	1	R/W	None
`ni_cause_scan_overflow`	1	1	R/W	None
`ni_cause_bc_rec_ok`	2	1	R/W	None
`ni_cause_sbc_rec_ok`	3	1	R/W	None
`ni_cause_com_rec_ok`	4	1	R/W	None
`ni_cause_com_rec_empty`	5	1	R/W	None
`ni_cause_sync_global_rec`	6	1	R/W	None
`ni_cause_global_rec`	7	1	R/W	None
`ni_cause_supervisor_global_rec`	8	1	R/W	None
`ni_cause_dr_rec_ok`	9	1	R/W	None
`ni_cause_ldr_rec_ok`	10	1	R/W	None
`ni_cause_rdr_rec_ok`	11	1	R/W	None
`ni_cause_dr_rec_tag`	12	1	R/W	None
`ni_cause_dr_rec_all_fall_down`	13	1	R/W	None

The `ni_interrupt_clear` Register

Field Name:	Pos:	Len:	Permissions: Super:	User:
`ni_clear_internal_fault`	0	1	W	None
`ni_clear_mc_error`	1	1	W	None
`ni_clear_cmu_error`	2	1	W	None
`ni_clear_bc_interrupt_red`	3	1	W	None
`ni_clear_cn_checksum_error`	4	1	W	None
`ni_clear_cn_hard_error`	5	1	W	None
`ni_clear_dr_checksum_error`	6	1	W	None
`ni_clear_timer_interrupt`	7	1	W	None
`ni_clear_bc_interrupt_orange`	8	1	W	None
`ni_clear_bc_interrupt_yellow`	9	1	W	None
`ni_clear_bc_or_com_collision`	10	1	W	None
`ni_clear_com_abstain_changed`	11	1	W	None
`ni_clear_dr_count_negative`	12	1	W	None
`ni_clear_bad_relative_address`	13	1	W	None
`ni_clear_bad_memory_access`	14	1	W	None

The `ni_interrupt_clear_green` Register

Field Name:	Pos:	Len:	Permissions: Super:	User:
`ni_clear_bc_interrupt_green`	0	1	W	None
`ni_clear_scan_overflow`	1	1	W	None
`ni_clear_bc_rec_ok`	2	1	W	None
`ni_clear_sbc_rec_ok`	3	1	W	None
`ni_clear_com_rec_ok`	4	1	W	None
`ni_clear_com_rec_empty`	5	1	W	None
`ni_clear_sync_global_rec`	6	1	W	None
`ni_clear_global_rec`	7	1	W	None
`ni_clear_supervisor_global_rec`	8	1	W	None
`ni_clear_dr_rec_ok`	9	1	W	None
`ni_clear_ldr_rec_ok`	10	1	W	None
`ni_clear_rdr_rec_ok`	11	1	W	None
`ni_clear_dr_rec_tag`	12	1	W	None
`ni_clear_dr_rec_all_fall_down`	13	1	W	None

Note: To locate the flags in the `interrupt_clear` registers, use the constants defined for the `interrupt_cause` registers — the flag positions are the same.

B.4.9 Other Register Fields and Constants

Note: The programming constants for these flags are obtained by uppercasing the name of the flag, then adding "`_P`" for the position, or "`_L`" for the length.

The `ni_interrupt_level` Register

Field Name:	Pos:	Len:	Permissions: Super:	User:
`ni_interrupt_level_green`	0	1	R/W	None
`ni_interrupt_level_yellow`	8	1	R/W	None
`ni_interrupt_level_orange`	16	1	R/W	None
`ni_interrupt_level_red`	24	1	R/W	None

The `ni_hodgepodge` Register

Field Name:	Pos:	Len:	Permissions: Super:	User:
`ni_global_rec_ie`	0	1	R/W	None
`ni_supervisor_global_rec_ie`	1	1	R/W	None
`ni_flush_complete`	2	1	R	None
`ni_interrupt_send_ok`	3	1	R	None
`ni_configuration_complete`	4	1	R	None
`ni_interrupt_rec_enable`	5	1	R/W	None
`ni_sync_global_rec_ie`	6	1	R/W	None
`ni_timer_ie`	7	1	R/W	None
`ni_cn_stop_send`	8	1	R/W	None

The `ni_bad_address` Register

Field Name:	Pos:	Len:	Permissions: Super:	User:
`ni_bad_address_low`	0	20	R/W	None
`ni_bad_address_type`	20	12	R/W	None

Note: The contents of the `ni_bad_address` register are implementation-dependent, so there are no predefined constants for this register.

Appendix C

Predefined Low-Level NI Constants

For ease of reference, here are the low-level programming constants defined in the header files **cmsys/ni_constants.h**, and **cmsys/ni_defines.h** (see Appendix F), grouped by register and field.

Note for C Programmers: These constants are defined as raw, unsigned integer values. If you use them in C code, you must recast them as pointer values of type **(unsigned *)**. Otherwise, the C compiler will treat them as integers, possibly causing "illegal pointer operation" errors.

```
=== Send First Register Constants ===
Field Offsets:
SF_FIFO_OFFSET      (12)
AUXILIARY_START_P   (3)

Length Constant:    NI_SEND_FIRST_L (32)

Interface Number constants:
DATA_ROUTER_FIFO    (1)
LEFT_DR_FIFO        (6)
RIGHT_DR_FIFO       (7)
USER_BC_FIFO        (3)
SUPERVISOR_BC_FIFO  (4)
COMBINE_FIFO        (5)

=== Auxiliary Data Field Constants ===
--- DR/LDR/RDR Interface ---
NI_DR_SEND_AUXILIARY_ADDRESS_MODE_P (8)
RELATIVE (0)
PHYSICAL (1)
NI_DR_SEND_AUXILIARY_TAG_P (4)          NI_DR_TAG_L (4)
NI_DR_SEND_AUXILIARY_LENGTH_P (0)       NI_DR_LENGTH_L (4)
```

```
=== Auxiliary Data Field Constants, cont. ===
--- BC/SBC Interface ---
NI_BC_SEND_AUXILIARY_LENGTH_P (0)      (no length constant)

--- COM Interface ---
NI_COM_SEND_AUXILIARY_PATTERN_P (7)
NI_COM_SEND_PATTERN_L (2)
SCAN_ROUTER_DONE    (0)
SCAN_BACKWARD       (1)
SCAN_FORWARD        (2)
SCAN_REDUCE         (3)
NI_COM_SEND_AUXILIARY_COMBINER_P (4)
NI_COM_SEND_COMBINER_L (3)
OR_SCAN             (0)
ADD_SCAN            (1)
XOR_SCAN            (2)
UADD_SCAN           (3)
MAX_SCAN            (4)
ASSERT_ROUTER_DONE (5)
NI_COM_SEND_AUXILIARY_LENGTH_P (0)
NI_COM_SEND_LENGTH_L (4)

=== Interface send/receive FIFO size limits ===
MAX_ROUTER_MSG_WORDS     (5)
MAX_COMBINE_MSG_WORDS    (5)
MAX_BROADCAST_MSG_WORDS (4)
MAX_SBC_MSG_WORDS   (4)

=== Send Registers ===
NI_DR_SEND_A        (NI_BASE | 0x0230)
NI_LDR_SEND_A       (NI_BASE | 0x0c30)
NI_RDR_SEND_A       (NI_BASE | 0x0e30)
NI_BC_SEND_A        (NI_BASE | 0x0630)
NI_SBC_SEND_A       (NI_BASE | 0x0830)
NI_COM_SEND_A       (NI_BASE | 0x0a30)
NI_SEND_L (32)

=== Receive Registers ===
NI_DR_RECV_A        (NI_BASE | 0x0220)
NI_LDR_RECV_A       (NI_BASE | 0x0c20)
NI_RDR_RECV_A       (NI_BASE | 0x0e20)
NI_BC_RECV_A        (NI_BASE | 0x0620)
NI_SBC_RECV_A       (NI_BASE | 0x0820)
NI_COM_RECV_A       (NI_BASE | 0x0a20)
NI_REC_L (32)
```

```
=== Status Register ===
NI_DR_STATUS_A          (NI_BASE | 0x0200)
NI_LDR_STATUS_A         (NI_BASE | 0x0c00)
NI_RDR_STATUS_A         (NI_BASE | 0x0e00)
NI_XDR_STATUS_L (19)

NI_BC_STATUS_A          (NI_BASE | 0x0600)
NI_SBC_STATUS_A         (NI_BASE | 0x0800)
NI_BC_STATUS_L (11)

NI_COM_STATUS_A         (NI_BASE | 0x0a00)
NI_COM_STATUS_L (21)

NI_STATUS_L (25)

Field Constants:
NI_SEND_SPACE_P (0)           NI_SEND_SPACE_L (4)
NI_REC_OK_P (4)               NI_REC_OK_L (1)
NI_SEND_OK_P (5)              NI_SEND_OK_L (1)
NI_ROUTER_DONE_COMPLETE_P(6)  NI_ROUTER_DONE_COMPLETE_L (1)
NI_SEND_EMPTY_P (6)           NI_SEND_EMPTY_L (1)
NI_REC_LENGTH_LEFT_P (7)      NI_REC_LENGTH_LEFT_L (4)
NI_REC_LENGTH_P (11)          NI_REC_LENGTH_L (4)
NI_DR_REC_TAG_P (15)          NI_DR_REC_TAG_L (4)
NI_COM_SCAN_OVERFLOW_P (20)   NI_COM_SCAN_OVERFLOW_L (1)
NI_DR_SEND_STATE_P (21)       NI_DR_SEND_STATE_L (2)
NI_DR_REC_STATE_P (23)        NI_DR_REC_STATE_L (2)

=== Control Registers ===
NI_BC_CONTROL_A         (NI_BASE | 0x0610)
NI_SBC_CONTROL_A        (NI_BASE | 0x0810)
NI_BC_CONTROL_L (1)

NI_COM_CONTROL_A        (NI_BASE | 0x0a10)
NI_COM_CONTROL_L (2)

NI_CONTROL_L (2)

Field Constants:
NI_REC_ABSTAIN_P (0)          NI_REC_ABSTAIN_L (1)
NI_REDUCE_REC_ABSTAIN_P (1)   NI_REDUCE_REC_ABSTAIN_L (1)
```

```
=== Private Registers ===
NI_DR_PRIVATE_A          (NI_BASE | 0x0208)
NI_DR_PRIVATE_L (10)

NI_LDR_PRIVATE_A         (NI_BASE | 0x0c08)
NI_RDR_PRIVATE_A         (NI_BASE | 0x0e08)
NI_XDR_PRIVATE_L (6)

NI_BC_PRIVATE_A          (NI_BASE | 0x0608)
NI_SBC_PRIVATE_A         (NI_BASE | 0x0808)
NI_BC_PRIVATE_L (5)

NI_COM_PRIVATE_A         (NI_BASE | 0x0a08)
NI_COM_PRIVATE_L (18)

NI_PRIVATE_L (18)

=== Private Registers, cont. ===
Field Constants:
NI_REC_OK_IE_P (0)            NI_REC_OK_IE_L (1)
NI_LOCK_P (1)                 NI_LOCK_L (1)
NI_REC_STOP_P (2)             NI_REC_STOP_L (1)
NI_REC_FULL_P (3)             NI_REC_FULL_L (1)
NI_SEND_ENABLE_P (4)          NI_SEND_ENABLE_L (1)
NI_BC_SEND_ENABLE_P (4)       NI_BC_SEND_ENABLE_L (1)
NI_COM_SCAN_OVERFLOW_IE_P(4)  NI_COM_SCAN_OVERFLOW_IE_L (1)
NI_DR_REC_ALL_FALL_DOWN_P(5)  NI_DR_REC_ALL_FALL_DOWN_L (1)
NI_COM_REC_EMPTY_IE_P (5)     NI_COM_REC_EMPTY_IE_L (1)
NI_ALL_FALL_DOWN_IE_P (6)     NI_ALL_FALL_DOWN_IE_L (1)
NI_ALL_FALL_DOWN_ENABLE_P(7)  NI_ALL_FALL_DOWN_ENABLE_L (1)
NI_COM_SEND_LENGTH_P (8)      NI_COM_SEND_LENGTH_L (4)
NI_COM_SEND_COMBINER_P (12)   NI_COM_SEND_COMBINER_L (3)
NI_COM_SEND_PATTERN_P (15)    NI_COM_SEND_PATTERN_L (2)
NI_COM_SEND_START_P (17)      NI_COM_SEND_START_L (1)

=== Global and System Registers ===
NI_INTERRUPT_CAUSE_A                (NI_BASE | 0x0000)
NI_CAUSE_INTERNAL_FAULT_P (0)
NI_CAUSE_MC_ERROR_P (1)
NI_CAUSE_CMU_ERROR_P (2)
NI_CAUSE_BC_INTERRUPT_RED_P (3)
NI_CAUSE_CN_CHECKSUM_ERROR_P (4)
NI_CAUSE_CN_HARD_ERROR_P (5)
NI_CAUSE_DR_CHECKSUM_ERROR_P (6)
          (cont.)
```

```
NI_INTERRUPT_CAUSE_A
NI_CAUSE_TIMER_INTERRUPT_P (7)
NI_CAUSE_BC_INTERRUPT_ORANGE_P (8)
NI_CAUSE_BC_INTERRUPT_YELLOW_P (9)
NI_CAUSE_BC_OR_COM_COLLISION_P (10)
NI_CAUSE_COM_ABSTAIN_CHANGED_P (11)
NI_CAUSE_DR_COUNT_NEGATIVE_P (12)
NI_CAUSE_BAD_RELATIVE_ADDRESS_P (13)
NI_CAUSE_BAD_MEMORY_ACCESS_P (14)
NI_INTERRUPT_TYPE_L (15)
NI_INTERRUPT_L (1)

NI_INTERRUPT_CAUSE_GREEN_A          (NI_BASE | 0x0008)
NI_CAUSE_BC_INTERRUPT_GREEN_P (0)
NI_CAUSE_SCAN_OVERFLOW_P (1)
NI_CAUSE_BC_REC_OK_P (2)
NI_CAUSE_SBC_REC_OK_P (3)
NI_CAUSE_COM_REC_OK_P (4)
NI_CAUSE_COM_REC_EMPTY_P (5)
NI_CAUSE_SYNC_GLOBAL_REC_P (6)
NI_CAUSE_GLOBAL_REC_P (7)
NI_CAUSE_SUPERVISOR_GLOBAL_REC_P (8)
NI_CAUSE_DR_REC_OK_P (9)
NI_CAUSE_LDR_REC_OK_P (10)
NI_CAUSE_RDR_REC_OK_P (11)
NI_CAUSE_DR_REC_TAG_P (12)
NI_CAUSE_DR_REC_ALL_FALL_DOWN_P (13)
NI_INTERRUPT_GREEN_TYPE_L (14)
NI_INTERRUPT_L (1)

NI_INTERRUPT_LEVEL_A                (NI_BASE | 0x0010)

NI_INTERRUPT_LEVEL_L (32)
NI_INTERRUPT_LEVEL_COLOR_L (8)

NI_PHYSICAL_SELF_A                  (NI_BASE | 0x0018)
NI_PARTITION_BASE_A                 (NI_BASE | 0x0020)
NI_PARTITION_SIZE_A                 (NI_BASE | 0x0028)
NI_PHYSICAL_ADDRESS_L (20)

NI_CHUNK_TABLE_ADDRESS_A            (NI_BASE | 0x0030)
NI_CHUNK_TABLE_ADDRESS_L (6)

NI_CHUNK_TABLE_DATA_A               (NI_BASE | 0x0038)
NI_CHUNK_TABLE_DATA_L (8)
```

```
NI_CHUNK_SIZE_A                    (NI_BASE | 0x0040)
NI_CHUNK_SIZE_L (3)

NI_DR_MESSAGE_COUNT_A              (NI_BASE | 0x0048)
NI_DR_MESSAGE_COUNT_L (32)

NI_COUNT_MASK_A                    (NI_BASE | 0x0050)
NI_REC_INTERRUPT_MASK_A            (NI_BASE | 0x0058)
NI_USER_TAG_MASK_A                 (NI_BASE | 0x0060)
NI_TAG_MASK_L (16)

NI_TIME_A                          (NI_BASE | 0x0070)
NI_TIME_L (32)

NI_CONFIGURATION_A                 (NI_BASE | 0x0078)
NI_CONFIGURATION_L (5)

NI_INTERRUPT_SEND_A                (NI_BASE | 0x0080)
NI_INTERRUPT_SEND_L (5)

NI_SERIAL_NUMBER_A                 (NI_BASE | 0x0088)
NI_SERIAL_NUMBER_L (32)

NI_SYNC_GLOBAL_A                   (NI_BASE | 0x0090)
NI_SYNC_GLOBAL_REC_P (0)
NI_SYNC_GLOBAL_REC_L (1)
NI_SYNC_GLOBAL_COMPLETE_P(1)
NI_SYNC_GLOBAL_COMPLETE_L (1)
NI_SYNC_GLOBAL_L (2)

NI_SYNC_GLOBAL_ABSTAIN_A           (NI_BASE | 0x0098)
NI_SYNC_GLOBAL_ABSTAIN_L (1)

NI_COM_FLUSH_SEND_A                (NI_BASE | 0x00a0)
NI_FLUSH_SEND_L (1)

NI_ASYNC_GLOBAL_A                  (NI_BASE | 0x00a8)
NI_GLOBAL_SEND_P (0)     NI_GLOBAL_SEND_L (1)
NI_GLOBAL_REC_P (1)      NI_GLOBAL_REC_L (1)
NI_GLOBAL_L (2)

NI_ASYNC_SUP_GLOBAL_A              (NI_BASE | 0x00b0)
NI_SUPERVISOR_GLOBAL_SEND_P (0)
NI_SUPERVISOR_GLOBAL_SEND_L (1)
NI_SUPERVISOR_GLOBAL_REC_P (1)
NI_SUPERVISOR_GLOBAL_REC_L (1)
NI_GLOBAL_L (2)
```

```
NI_HODGEPODGE_A                    (NI_BASE | 0x00b8)
NI_GLOBAL_REC_IE_P (0)
NI_GLOBAL_REC_IE_L (1)
NI_SUPERVISOR_GLOBAL_REC_IE_P (1)
NI_SUPERVISOR_GLOBAL_REC_IE_L (1)
NI_FLUSH_COMPLETE_P (2)
NI_FLUSH_COMPLETE_L (1)
NI_INTERRUPT_SEND_OK_P (3)
NI_INTERRUPT_SEND_OK_L (1)
NI_CONFIGURATION_COMPLETE_P (4)
NI_CONFIGURATION_COMPLETE_L (1)
NI_INTERRUPT_REC_ENABLE_P (5)
NI_INTERRUPT_REC_ENABLE_L (1)
NI_SYNC_GLOBAL_REC_IE_P (6)
NI_SYNC_GLOBAL_REC_IE_L (1)
NI_TIMER_IE_P (7)
NI_TIMER_IE_L (1)
NI_CN_STOP_SEND_P (8)
NI_CN_STOP_SEND_L (1)
NI_HODGEPODGE_L (9)

NI_SYNC_GLOBAL_SEND_A              (NI_BASE | 0x00C0)
NI_SYNC_GLOBAL_SEND_L (1)

NI_INTERRUPT_CLEAR_A               (NI_BASE | 0x00c8)
NI_INTERRUPT_CLEAR_GREEN_A         (NI_BASE | 0x00d0)
(use same constants as for CAUSE register)

NI_INTERRUPT_NOW_A                 (NI_BASE | 0x00d8)
NI_INTERRUPT_NOW_L (32)

NI_SCAN_START_A                    (NI_BASE | 0x00e0)
NI_SCAN_START_L (1)

NI_BAD_ADDRESS_A                   (NI_BASE | 0x00e8)
NI_BAD_ADDRESS_L (32)
```

Appendix D

NI Interrupts

The methods used to recover from an NI interrupt depend heavily on the type of interrupt itself. This appendix describes each of the possible interrupts in detail, and provides guidelines describing how you can and should recover from them.

For each interrupt, the following information is provided:

- the name and color of the interrupt
- the `ni_interrupt_cause` or `ni_interrupt_cause_green` flag that is set when the interrupt is signaled
- the `ni_interrupt_clear` or `ni_interrupt_clear_green` flag that is used to clear the interrupt when it has been handled
- the triggering event that causes the interrupt to be signaled
- the effect of the interrupt on the NI and the networks
- the correct method for handling the interrupt

Note: It is possible for the supervisor to trigger an interrupt artificially, by setting the appropriate `ni_interrupt_cause` or `ni_interrupt_cause_green` flag. Since this can be done for any interrupt, it is not documented under the triggering events given below for each interrupt.

Also, since the `ni_interrupt_clear` and `ni_interrupt_clear_green` flags must be used to clear every interrupt once the required handling operations have been performed, this step is assumed, and is not listed under the handling guidelines for each interrupt.

D.1 Red Interrupts

Red interrupts indicate a failure of the hardware, such as checksum violations and message format errors. They occur at unpredictable times relative to the instruction stream and are usually irrecoverable. Determining the precise cause of a Red interrupt may require the use of the Diagnostic Network.

The cause and clear flags listed for each interrupt are found in these registers:

`ni_interrupt_cause` `ni_interrupt_clear`

D.1.1 Internal Fault ..... Red Interrupt

Flags: `ni_cause/clear_internal_fault`

Cause: A fault has been detected in the NI chip.

Effect: The effects are undefined and irrecoverable.

Handling: No software-serviceable parts inside. Please report this fault to your applications engineer or systems manager for correction.

D.1.2 CN Checksum Error, DR Checksum Error ..... Red Interrupt

Flags: `ni_cause/clear_cn_checksum_error`
`ni_cause/clear_dr_checksum_error`

Cause: A message with a bad checksum value was received from either the Control Network or Data Network. This interrupt is signaled as soon as the bad checksum value is received by the NI.

Effect: None. The received message(s) may still be read. However, they will almost certainly contain an error either in data or address.

Handling: This interrupt indicates that a network chip (or the NI chip itself) has failed. The failed chip must be tracked down with the Diagnostic Network. Please report this fault to your applications engineer or systems manager for correction.

D.1.3 CN Hard Error ..... Red Interrupt

Flags: `ni_cause/clear_cn_hard_error`

Cause: A hardware error occurred in the Control Network.

Effect: The effects are undefined and irrecoverable.

Handling: This interrupt indicates one of two things: either a hardware problem in the Control Network, which must be located by use of the Diagnostic Network; or a serious software problem (specifically, a double trap forcing a processor (IU) reset). Please report this fault to your applications engineer or systems manager for correction.

D.1.4 MC Error, CMU Error ..... Red Interrupt

Flags: `ni_cause/clear_mc_error`
`ni_cause/clear_cmu_error`

Cause: An interrupt is being signaled by either the memory controller, or by the CMU (cache and memory management unit). These two kinds of external interrupt are signaled to the microprocessor by way of the NI chip.

Effect: None, aside from the interrupt itself.

Handling: These interrupts continue to be signaled until they are cleared on the memory controller or CMU.

Note: Unlike most NI interrupts, these two interrupts are not cleared by writing the corresponding `ni_interrupt_clear` flag. Instead, a flag on the memory controller or CMU must be reset.

Nevertheless, it *is* legal to write a 1 to the `ni_interrupt_clear` flags for these interrupts. While this has no effect, it is permitted so that you can write uniform interrupt handler code.

D.1.5 BC Interrupt Red ..... Red Interrupt

Flags: `ni_cause/clear_bc_interrupt_red`

Cause: The NI received a Red broadcast interrupt, and the broadcast interrupt enable flag `ni_interrupt_rec_enable` was set to 1.

Effect: None, aside from the interrupt itself.

Handling: This is a software-signaled interrupt. Your interrupt handler should detect and handle this interrupt as appropriate for your program.

D.2 Orange Interrupts

Orange interrupts indicate that the attention of the operating system is required, as in timer interrupts and broadcast interrupt messages. They occur at unpredictable times relative to the instruction stream and do not destroy any information that might be needed to determine the cause of the interrupt.

The cause and clear flags listed for each interrupt are found in these registers:
`ni_interrupt_cause` `ni_interrupt_clear`

D.2.1 Timer Interrupt ..... Orange Interrupt

Flags: `ni_cause/clear_timer_interrupt`

Cause: The `ni_time` register is equal to the `ni_interrupt_now` register, and the timer interrupt flag `ni_timer_ie` flag is 1.

Effect: None, aside from the interrupt itself.

Handling: This interrupt is software-controlled, and should be handled by your interrupt handler.

D.2.2 BC Interrupt Orange ..... Orange Interrupt

Flags: `ni_cause/clear_bc_interrupt_orange`

Cause: The NI received a Orange broadcast interrupt, and the broadcast interrupt enable flag `ni_interrupt_rec_enable` was set to 1.

Effect: None, aside from the interrupt itself.

Handling: This is a software-signaled interrupt. Your interrupt handler should detect and handle this interrupt as appropriate for your program.

D.3 Yellow Interrupts

Yellow interrupts indicate that the software has made an error. They are usually irrecoverable, as they indicate that your program is doing something illegal and will have to be rewritten. Sufficient information is retained in the NI to permit isolation of the cause of the interrupt, but it is not always possible to recover all the information relating to the cause of the interrupt.

Yellow interrupts are associated with particular instructions, but usually are not signaled at the exact point of the offending instruction, because of the loose coupling between the NI and the microprocessor.

The cause and clear flags listed for each interrupt are found in these registers:
`ni_interrupt_cause` `ni_interrupt_clear`

D.3.1 BC Interrupt Yellow ..... Yellow Interrupt

Flags: `ni_cause/clear_bc_interrupt_yellow`

Cause: The NI received a Yellow broadcast interrupt, and the broadcast interrupt enable flag `ni_interrupt_rec_enable` was set to 1.

Effect: None, aside from the interrupt itself.

Handling: This is a software-signaled interrupt. Your interrupt handler should detect and handle this interrupt as appropriate for your program.

D.3.2 COM Abstain Changed ..... Yellow Interrupt

Flags: `ni_cause/clear_com_abstain_changed`

Cause: The `ni_com_abstain` or `ni_reduce_rec_abstain` flags were changed while the combiner send FIFO was not empty.

Effect: The attempted change does not occur. Whether execution is allowed to continue depends on the interrupt handler.

Handling: Your interrupt handler should decide whether to signal this as an error, or to quietly recover from it, perhaps displaying a warning message.

D.3.3 DR Count Negative ..... Yellow Interrupt

Flags: `ni_cause/clear_dr_count_negative`

Cause: The combined value of all `ni_dr_message_count` registers in the Data Network has become negative, indicating a mismatch in the sending and/or receiving of Data Network messages.

Effect: None, but this interrupt is signaled repeatedly until the situation is corrected.

Handling: This may occur either when a failure in a Data Network or NI chip causes the annihilation of a message, or when an OS error causes a countable Data Network message to be sent out of its partition. This interrupt may also occur if two or more nodes in a paritition do not agree on which Data Network message tags are to be counted (that is, their `ni_count_mask` registers are not equal).

To restore the Data Network to a proper state, make sure that the partition is empty of Data Network messages, and then set all the `ni_dr_message_count` registers in the partition to 0.

Note: It may be that by the time the interrupt is signaled, the values of one or more of the `ni_dr_message_count` registers will have changed. This may make it difficult to locate the error, since the sum of the `ni_dr_message_count` registers may be positive by the time the interrupt is signaled.

D.3.4 BC or COM Collision ..... Yellow Interrupt

Flags: `ni_cause/clear_bc_or_com_collision`

Cause: Three separate conditions cause this interrupt:

- Two NIs attempted to broadcast at the same time.
- Two different combine operations signaled at the same time.
- Two NIs simultaneously attempted a broadcast interrupt.

Effect: No combining or broadcast operations can proceed while the `ni_cause_bc_or_com_collision` flag is set. If the error was colliding broadcast interrupts, the broadcast is not signaled.

Handling: If the error was colliding combine messages, the messages are still in the combine send FIFO. The supervisor should take control of this FIFO and read out the messages to determine where the collision occurred. If the error was colliding broadcast messages, the `ni_bc_send_empty` (or `ni_sbc_send_empty`) flags will be set to 0 in the contending processors. If the error was colliding broadcast interrupts, the `ni_interrupt_send_ok` will be 0 in the processors that sent the colliding broadcast interrupts.

Note: When the `ni_clear_bc_or_com_collision` flag is written, all messages in the broadcast and supervisor broadcast send FIFOs disappear, and the `ni_interrupt_send_ok` flag is set to 1. None of the other FIFOs, either send or receive, are affected.

D.3.5 Bad Relative Address ..... Yellow Interrupt

Flags: `ni_cause/clear_bad_relative_address`

Cause: An attempt was made to send a Data Network message with a relative address that is illegal for the current partition.

Effect: The message with the bad address is discarded and the appropriate `ni_`*interface*`_send_ok` flag is set to 0, indicating that the attempt to send the message failed.

Handling: Your interrupt handler should decide whether to signal this as an error, or to quietly recover from it, perhaps displaying a warning message.

D.4 Green Interrupts

Green interrupts indicate the occurrence of common events for which the software has requested notification, such as the arrival of messages, the signaling of broadcast interrupts, arithmetic overflow in a scan, etc. There is one interrupt for each event, and each event's interrupt can be enabled and disabled independently under the control of the supervisor.

Depending on the type of event, the interrupt may or may not occur synchronously with a particular instruction. No information is lost by a Green interrupt.

The cause and clear flags listed for each interrupt are found in these registers:
`ni_interrupt_cause_green` `ni_interrupt_clear_green`

D.4.1 BC Interrupt Green ..... Green Interrupt

Flags: `ni_cause/clear_bc_interrupt_green`

Cause: The NI received a Green broadcast interrupt, and the broadcast interrupt enable flag `ni_interrupt_rec_enable` was set to 1.

Effect: None, aside from the interrupt itself.

Handling: This is a software-signaled interrupt. Your interrupt handler should detect and handle this interrupt as appropriate for your program.

D.4.2 DR Receive Tag ..... Green Interrupt

Flags: `ni_cause/clear_dr_rec_tag`

Cause: A message arrived at the front of a Data Network receive FIFO that has an interrupting tag (a tag corresponding to a set flag in the register `ni_rec_interrupt_mask`).

Effect: None, aside from the interrupt itself.

Handling: This interrupt is software-controlled, and should be handled by your interrupt handler.

D.4.3 DR Receive All Fall Down ..... Green Interrupt

Flags: `ni_cause/clear_dr_rec_all_fall_down`

Cause: An All Fall Down mode message arrived at the front of a Data Network receive FIFO, while `ni_all_fall_down_ie` is 1.

Effect: The first word read from the FIFO is the All Fall Down mode address word, which is used to determine the correct destination address for the message. The `rec_length` field contains the length of the message *not counting* the address word, while the `rec_length_left` field contains the total length of the message *counting* the address word.

Handling: Your handler should receive and store the message in such a way that it can later be resent to its correct destination.

D.4.4 Interface (DR, BC, COM, etc.) Receive OK ... Green Interrupt

Flags:
`ni_cause/clear_bc_rec_ok`
`ni_cause/clear_sbc_rec_ok`
`ni_cause/clear_com_rec_ok`
`ni_cause/clear_dr_rec_ok`
`ni_cause/clear_ldr_rec_ok`
`ni_cause/clear_rdr_rec_ok`

Cause: A new message became available from the receive FIFO of one of the interfaces while the corresponding `ni_`*interface*`_rec_ok_ie` flag was set to 1.

Effect: While enabled, each of these interrupts is signaled once for each arriving message in the appropriate interface's receive FIFO.

Handling: This interrupt is software-controlled, and should be handled by your interrupt handler. (Typically, your handler reads the interrupting message from the FIFO, but you can decide to do otherwise.)

D.4.5 Global Rec (Sync, Global, or Supervisor) .... Green Interrupt

Flags: `ni_cause/clear_sync_global_rec`
`ni_cause/clear_global_rec`
`ni_cause/clear_supervisor_global_rec`

Cause: One of the following events happened:

A synchronous global operation completed with a result of 1, and the `ni_sync_global_rec_ie` flag is 1.

The asynchronous global receive flag `ni_global_rec` changed from 0 to 1, and the `ni_global_rec_ie` flag is 1.

The supervisor asynchronous receive flag `ni_supervisor_global_rec` changed from 0 to 1, and the `ni_supervisor_global_rec_ie` flag is 1.

Effect: None, aside from the interrupts themselves.

Handling: These interrupts are software-controlled, and should be handled by your interrupt handler.

D.4.6 Com Receive Empty ..... Green Interrupt

Flags: `ni_cause/clear_com_rec_empty`

Cause: The combine receive FIFO became empty while the empty receive FIFO interrupt flag `ni_com_rec_empty_ie` is 1.

Effect: None, aside from the interrupt itself.

Handling: This interrupt is software-controlled, and should be handled by your interrupt handler.

D.4.7 Scan Overflow ..... Green Interrupt

Flags: `ni_cause/clear_scan_overflow`

Cause: The first word of a scan or reduce message that suffered arithmetic overflow was read from the combine receive FIFO, and the `ni_scan_overflow_ie` interrupt enable flag is 1. This can only happen if the message combiner is a signed or unsigned addition.

Effect: None. The arrived message may be read normally.

Handling: Your interrupt handler should decide whether to signal this as an error, or to quietly recover from it, perhaps displaying a warning message.

D.5 Bus Errors

Bus Errors indicate that a bus transaction cannot be completed, as in an attempt to read a virtual address that does not correspond to a register, or to write a message that doesn't conform to protocol. Bus Errors are signaled asynchronously and are usually irrecoverable. Bus Errors are distinct from segmentation violation errors, which result from attempting to read an unmapped virtual address, and are signaled synchronously with the offending instruction.

The cause and clear flags listed for each interrupt are found in these registers:

`ni_interrupt_cause` `ni_interrupt_clear`

D.5.1 Bad Memory Access ..... Bus Error

Flags: `ni_cause/clear_bad_memory_access`

Cause: Bus Errors are signaled for number of reasons, including:

- Attempting to read a read-protected address.
- Attempting to write a write-protected address.
- Attempting to read or write a value that does not fit in a register.
- Attempting to read or write an address that is not a register.

Some specific examples are:

Bus Errors caused by reads or writes:

- reading or writing a supervisor-only register from the user area
- reading the **ni_*interface*_rec** register of an empty receive FIFO
- attempting to read a doubleword from a FIFO that has only a word left, or attempting to use a doubleword operation to write a single-word message
- writing the **send_first** register of a network interface while there is an incomplete message pending in the send FIFO
- writing the **send** register of a network interface without having first written a value to the corresponding **send_first** register

Bus Errors caused by sending a message:

- attempting to send a message longer than the entire send FIFO
- attempting to send a message via a network interface for which the corresponding abstain flag is set
- attempting to send a user message with a supervisor-reserved tag
- attempting to send or receive a message through an excluded Data Network interface.
- attempting to send a combine message with an illegal combiner or pattern value.
- attempting to send a network-done message with a length greater than 1, or attempting to send any network-done message while the **ni_network_done** flag is 0 or the **ni_com_abstain** flag is 1
- attempting to send a synchronous global message or to change the **ni_sync_global_abstain** flag while the **ni_sync_global_complete** flag is 0

Bus Errors caused by other operations:

- attempting to start a flush operation while the **ni_flush_complete** flag is 0
- attempting to start a configuration operation while the **ni_configuration_complete** flag is 0
- attempting to send a broadcast interrupt while the **ni_interrupt_send_ok** flag is 0
- attempting to write a value to the **ni_*interface*_rec** register when the receive FIFO is full.

Effect: The address, size and type of the offending memory transaction is be stored in the **ni_bad_address** register.

Any data written by the offending transaction is lost. Any side-effects that would have been triggered by the offending transaction (such as the initiation of a synchronous global operation) do not occur. In particular, an attempted doubleword read from a receiving FIFO containing only one word will not result in popping the word.

Handling: Examine the **ni_bad_address** register to determine what memory transaction caused the error.

Appendix E

NI Programming Examples

For C programmers, here are some examples of macros that you can use to access the registers and fields of the NI. In most cases, these macros take as arguments the register and field constants defined previously in this manual.

E.1 Reading and Writing Registers

The simplest NI register operations involve reading and writing the value of a register, typically with one of three types of values: unsigned, float, and double. The macros below provide a simple register reading/writing interface.

```
#define ni_register(type,reg)      *((type *) (reg))
#define ni_read_reg(reg)       ni_register(unsigned, reg)
#define ni_read_reg_f(reg)     ni_register(float, reg)
#define ni_read_reg_d(reg)     ni_register(double, reg)

#define ni_set_register(type,reg,value)
              ni_register(type, reg) = ((type) (value))
#define ni_write_reg(reg)
                    ni_set_register(unsigned, reg, value)
#define ni_write_reg_f(reg)
                    ni_set_register(float, reg, value)
#define ni_write_reg_d(reg)
                    ni_set_register(double, reg, value)
```

In these examples the *reg* argument is the address constant of the appropriate register, and the *value* argument is the word, float, or double to be written.

E.2 Reading and Writing Subfields

Often, you'll want to read or write the value of a register subfield. Here's a set of macros that efficiently extract a field from a register. (Note that the *field* argument in these examples is the name of the field constant *without* the `_P` or `_L` suffixes — these are added automatically by the macros themselves.)

```
/* mask for values that will fit into the given field */
#define ni_mask_field_values (field_length) \
    (~(~0 << field_length))

/* mask that extracts a field from the register */
#define ni_mask_field (position, length) \
    (ni_mask_field_values (length) << position)

/* right-shift register value, mask out the field */
#define ni_get_field(register_val, pos, len) \
    ((register_val >> pos) & ni_mask_field_values(len))

#define ni_read_field(register, pos, len) \
    ni_get_field(ni_read_reg(register), pos, len)
```

And here's a set of macros that efficiently modify the value of a register field:

```
/* mask that is ANDed with register to change field */
#define ni_new_value_mask(pos, len, new_value) \
    ~((new_value ^ ni_mask_field_values(len)) << pos)

/* Logical AND register with mask that changes field */
#define ni_set_field(reg_val, pos, len, new_value) \
  (reg_val & ni_new_value_mask(pos, len, new_value))

#define ni_write_field(reg, pos, len, new_value) \
  ni_write_reg(register, \
    ni_set_field(ni_read_reg(reg), pos, len, new_value))
```

You may also want to simply set or clear an arbitrary set of register bits:

```
#define ni_set_bits_in_register(reg, bitmask) \
  ni_write_reg(reg, ni_read_reg(reg) | (bitmask))

#define ni_clear_bits_in_register(reg, bitmask)\
  ni_write_reg(reg, ni_read_reg(reg) & ~(bitmask))
```

E.3 Constructing Send-First Addresses

The only other major set of programming tools that you might need are macros that construct a **send_first** address for a given interface. For example:

```
#define ni_send_first_a(interface,auxiliary_data) \
  ((unsigned *) (  NI_BASE | \
                   interface      << SF_FIFO_OFFSET | \
                   auxiliary_data << AUXILIARY_START_P))

#define ni_send_first(interface,auxiliary_data,value) \
  ni_write_reg(ni_send_first_a(interface,auxiliary_data), \
               value)
```

Data Network Send-First Macros

Here's a set of macros that constructs the **send_first** addresses for the three Data Network interfaces:

```
#define ni_xdr_auxiliary_data(mode,tag,length) \
    ( mode    << NI_DR_SEND_AUXILIARY_ADDRESS_MODE_P | \
      tag     << NI_DR_SEND_AUXILIARY_TAG_P | \
      length  << NI_DR_SEND_AUXILIARY_LENGTH_P )

#define ni_dr_send_first(mode, tag, length, value) \
  ni_send_first(DATA_ROUTER_FIFO, \
                ni_xdr_auxiliary_data(mode,tag,length), \
                value)

#define ni_ldr_send_first(mode, tag, length, value) \
  ni_send_first(LEFT_DR_FIFO, \
                ni_xdr_auxiliary_data(mode,tag,length), \
                value)

#define ni_rdr_send_first(mode, tag, length, value) \
  ni_send_first(RIGHT_DR_FIFO, \
                ni_xdr_auxiliary_data(mode,tag,length), \
                value)
```

Broadcast Interface Send-First Macros

Here's a set of macros that constructs the **send_first** addresses for the two broadcast interfaces:

```
#define ni_xbc_auxiliary_data(length) \
    ( length  << NI_BC_SEND_AUXILIARY_LENGTH_P )

#define ni_bc_send_first(length, value) \
  ni_send_first(USER_BC_FIFO, \
                ni_xbc_auxiliary_data(length), \
                value)

#define ni_sbc_send_first(length, value) \
  ni_send_first(SUPERVISOR_BC_FIFO, \
                ni_xbc_auxiliary_data(length), \
                value)
```

Combine Interface Send-First Macros

Finally, here's a set of macros that constructs the **send_first** addresses for the combine interface:

```
#define ni_com_auxiliary_data(pattern,combiner,length) \
    ( pattern  << NI_COM_SEND_AUXILIARY_PATTERN_P | \
      combiner << NI_COM_SEND_AUXILIARY_COMBINER_P | \
      length   << NI_COM_SEND_AUXILIARY_LENGTH_P )

#define ni_bc_send_first(pattern,combiner,length,value)\
  ni_send_first(COMBINE_FIFO, \
                ni_com_auxiliary_data(pattern,combiner,\
                                      length) \
                value)
```

Appendix F

CMNA Header Files

To access the NI constants described in this document, you must **#include** the header file **cm/cmna.h**:

```
#include <cm/cmna.h>
```

This file **#include**s many other header files that provide access to NI constants, register macros, and accessor functions. These constants, macros, and functions are collectively referred to as CMNA (CM Network Accessors), and can serve as a basis for your own NI accessor code.

Note: The functions and macros in CMNA are designed to be very generic in operation. As such, they are much less efficient than the special-purpose macros and functions you'll probably write on your own. Nevertheless, you can use the operations defined in CMNA as a jumping-off point for your own code, to help you understand what needs to be done to get your code to run correctly.

F.1 What is CMNA?

There are two main parts to CMNA:

- The NI Interface — Constants and macros used to manipulate NI registers.
- CnC ("C-and-C") — C functions that perform NI operations such as reading and writing messages of arbitrary length.

The CMNA header files define the NI interface explicitly, in terms of register accessor macros and constants. The header files also provide C prototypes for the CnC functions, which are part of the CMOST operating system code.

F.2 CMNA Header Files

The following header files are part of CMNA:

/usr/include/

cm/cmna.h	— Main CMNA header file.
cmsys/cmna.h	— CMNA user header file.
cmsys/cmna_sup.h	— CMNA supervisor header file.
cmsys/ni_interface.h	— Main NI interface header file.
cmsys/ni_macros.h	— NI macro definitions.
cmsys/ni_constants.h	— NI register/flag constant definitions.
cmsys/ni_defines.h	— Low-level NI constant definitions.

The following diagram shows the relationship among the header files that make up CMNA:

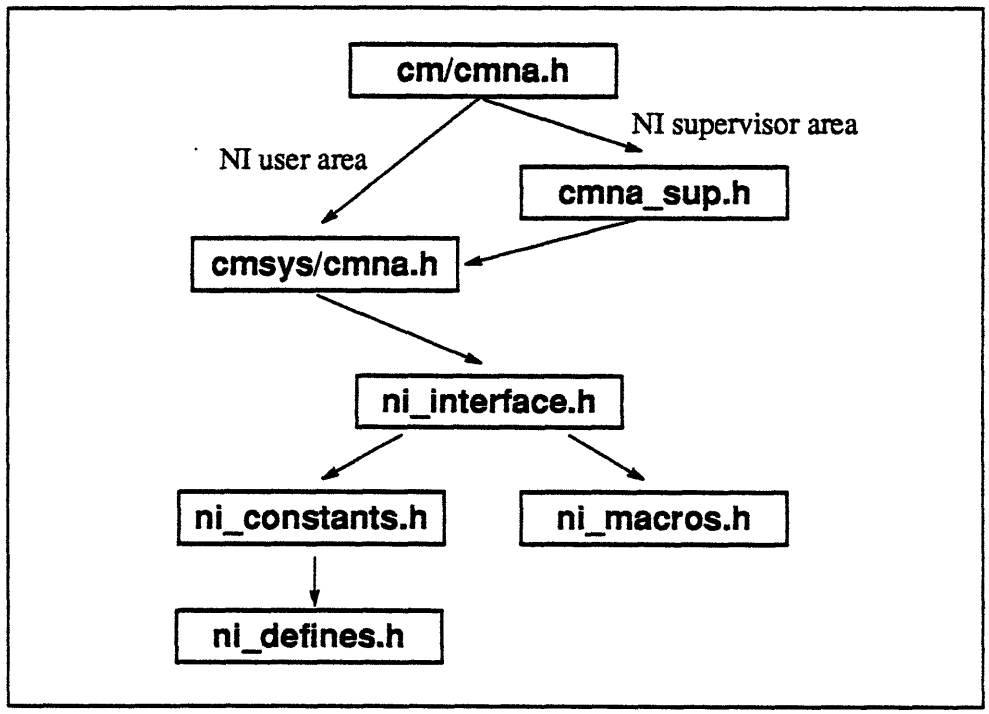


Figure 19. Relationship between CMNA and NI header files.

F.2.1 The Main CMNA Header File: cm/cmna.h

This single file `#includes` all the header files that are needed to define CMNA. However, it contains virtually no definitions of its own. It simply `#includes` either of the two header files `cmsys/cmna.h` or `cmsys/cmna_sup.h`, according to which NI register area (user or supervisor) the `#includeing` code needs.

Implementation Note: At present, `cmsys/cmna_sup.h` is only `#included` for diagnostic code (that is, code that defines the symbol `CMDIAG`).

F.2.2 The User Header File: cmsys/cmna.h

This file `#includes` the NI constant and macro files described below, and also defines a number of useful C mask constants and C macros that are used in CMNA. However, the constants and macros defined here are only sufficient for the needs of CMNA, and are not by any means a complete set. (See the description of the `ni_constants.h`, and `ni_defines` files below.)

F.2.3 The Supervisor Header File: cmsys/cmna_sup.h

This file modifies a few key constant definitions so that any absolute memory address constants defined in the other header files will refer to the NI supervisor area, rather than the NI user area. It then `#includes` `cmsys/cmna.h`, so it has much the same effect as that header file.

Note: The `cmsys/cmna_sup.h` file is only of use to programmers with legal access to the NI supervisor area. Including this file does *not* in itself grant access to the NI's supervisor area; it simply redefines many CMNA constants to have address values that are only legal for supervisor code.

F.2.4 The NI Interface Header File: ni_interface.h

This file defines the NI accessor interface. It `#includes` the file `ni_constants.h`, and defines a number of basic NI register macros that are used by CMNA. It then `#includes` `ni_macros.h` to define the remainder of the CMNA macros.

This file also defines a number of NI register constants that are suitable for use in C code. (That is, constants that have been cast as `(unsigned *)` values. See the description of `ni_constants.h` and `ni_defines.h` below.)

F.2.5 The NI Macros Header File: ni_macros.h

This file defines a number of C macros that perform sterotypical NI operations such as sending and receiving messages via a specific network interface.

F.2.6 The NI Constants Header Files: ni_constants.h, ni_defines.h

These files define a number of register constants and masks that are used by CMNA. In particular, `ni_constants.c` includes definitions of constants specifying the absolute memory address for each of the NI's registers. The file `ni_defines.h` defines hundreds of constants that give the size and offset of the register fields of the NI. These two sets of constants provide a complete interface for NI operations written in assembly code. Appendix C provides a complete list of these constants, grouped by register and category.

Note For C Programmers: The register address constants are unsigned pointer values. To use them in C code, you must first cast them to type `(unsigned *)`. For example:

```
unsigned *ni_dr_status = ((unsigned *) NI_DR_STATUS);
```

If you don't perform this casting step, the C compiler by default treats the constants as signed integers, possibly causing your code to fail. Many of these constants are recast in just this fashion in the header file `ni_interface.c`, so you may be able to just use those constants without having to do any recasting yourself.

Indexes

Programming Tools Index

This index lists the register names and fields, programming constants, functions and macros referred to within this document. Bold page numbers indicate a defining reference or important description.

G – L

M

N

O

P

R

S

T

U

X

Concepts Index

This index lists the essential concepts referred to within this document. Bold page numbers indicate a defining reference or important description.

A

B

C

D

E

F

G

H

I

N

O

P

S

T

U

V

W

X

Y