The
Connection Machine
System

Programming the NI

Version 7.1
March 1992

Thinking Machines Corporation
Cambridge, Massachusetts

First printing, March 1992
Revised, March 1992

Thinking Machines Corporation
245 First Street
Cambridge, Massachusetts 02142–1264
(617) 234–1000/876–1111

Contents

Chapter 7, cont.

Appendixes

Appendix A, cont.

Indexes

About This Manual

Objectives of This Manual

This manual shows you how to write programs that directly manipulate the low-level network hardware of the Connection Machine CM-5 system. The main focus in this document is the Network Interface (NI) chip, the component of the CM-5 hardware that manages the machine's internal communications networks.

The code examples throughout this manual are written in C, with **`#define`** macros for simple operations. Most are code fragments illustrating specific examples of NI features. For information about structuring your code and linking it to run on CM-5 hardware, see Chapter 7. A complete description of the macros used in this manual can be found in Appendix A.

Intended Audience

This manual is a guide for experienced programmers, not a tutorial. Some overview information is provided, but this manual is primarily intended to help knowledgeable CM programmers develop special-purpose code.

WARNING: Code that directly accesses the NI chip *will not be supported* by future hardware releases. It's recommended that you use the CMMD software interface for essential code. CMMD also gives you access to the NI, but through a software interface that can be easily ported to future releases.

Revision Information

This manual is new as of Version 7.1. Minor revisions from the first printing are included in this version.

Organization of This Manual

Chapter 1 The Network Interface
An overview of the NI's location and function within the CM-5 hardware.

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 register interface and features of the three Data Networks.

Chapter 4 The Broadcast Network
The register interface and features of the broadcast network.

Chapter 5 The Combine Network
The register interface and features of the combine network.

Chapter 6 The Global Network
The register interface and features of the global network.

Chapter 7 Writing NI Programs
A brief overview of the process of writing, compiling, and running an NI program.

Chapter 8 Programming and Performance Hints
Useful performance techniques, as well as descriptions of potential coding problems.

Appendix A Programming Tools
A complete list of the macro tools used for writing NI programs.

Appendix B CMOS_signal man Page
The **man** page for the CMOST command **`CMOS_signal()`**.

Appendix C Sample NI Programs
A selection of short programs that test the examples presented in the network chapters above.

Related Documents

These documents are part of the Connection Machine documentation set.

- *Connection Machine CM-5 Technical Summary*, October 1991
- *CMMD User's Guide*, Version 1.1, January 1992
- *CMMD Reference Manual*, Version 1.1, January 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 back-trace, 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

First, a word to the wise. You're reading this manual for one of two reasons:

- You absolutely, positively *must* write programs that manipulate the network hardware of the CM-5 at the lowest possible level.
- You've heard about a CM-5 component called the "Network Interface," and think it would be interesting to write a program that manipulates it.

If it's the latter, we strongly suggest that you consider using a higher-level programming method instead. Writing code at the level described in this manual means taking direct control of the Network Interface chip, the part of the CM-5 hardware that manages the machine's internal communications networks. This isn't something that you should be doing unless you have no alternative.

Also, be aware that code that directly accesses the Network Interface chip *will not be supported* in future software and hardware releases — your code may require extensive modification to run. For essential code you should use the CMMD software interface instead. CMMD gives you nearly the same level of access to the CM-5 hardware, but provides it through a standard software interface that will be easily portable to future releases. (For more information, see the *CMMD User's Guide.*)

With this warning out of the way, we'll assume that you're reading this manual for the first reason given above, and show you how to make use of the Network Interface (NI) chip. This manual presents the software tools that you need to program the NI and provides code examples throughout that show you how to do simple network operations on the CM-5.

1.1 The CM-5 System: Nodes and Networks

Because the main focus of this manual is the Network Interface chip, it makes sense to start with an overview of the NI's location and function within the CM-5 system.

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

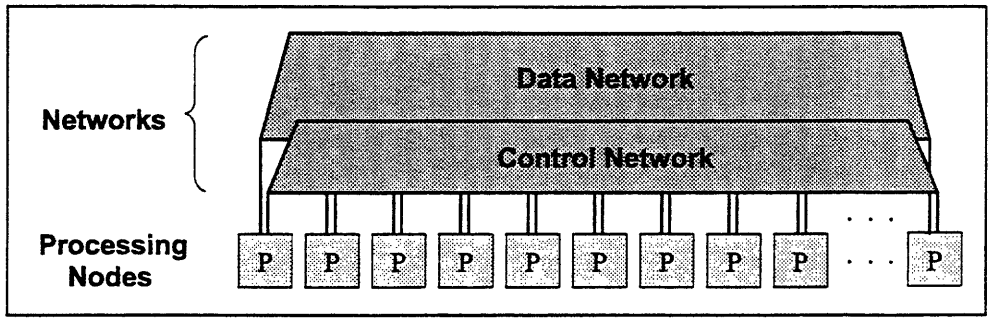

Figure 1. The CM-5 system: Processing nodes linked by Data and Control Networks.

The two networks are similar in design; both are scalable, high-speed data communications networks. The structure and intended purpose of the networks, however, are quite different.

1.1.1 The CM-5 Networks

The Data Network

The Data Network is the data highway of the CM-5. It's 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 half-networks, the *left data network* and *right data network,* both of which are connected to all processing nodes. These two half-networks can either be used independently as separate networks, or together as a single *data network.*

The Control Network

The Control Network is used for control tasks that require the joint cooperation of all nodes. It is composed of three different sub-networks, each with a unique function:

- The *broadcast network* distributes a single numeric value to every node.
- The *combine network* receives a single value from each node, combines the values arithmetically or logically, and then distributes the combined result to all nodes.
- The *global network* handles global synchronization of the nodes.

Note: Because of the unique features of these three sub-networks, NI programmers often refer to them as independent networks in their own right, and they're described independently in this manual. However, there are some interactions between these networks that you'll need to keep in mind. These are explained in later chapters.

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:

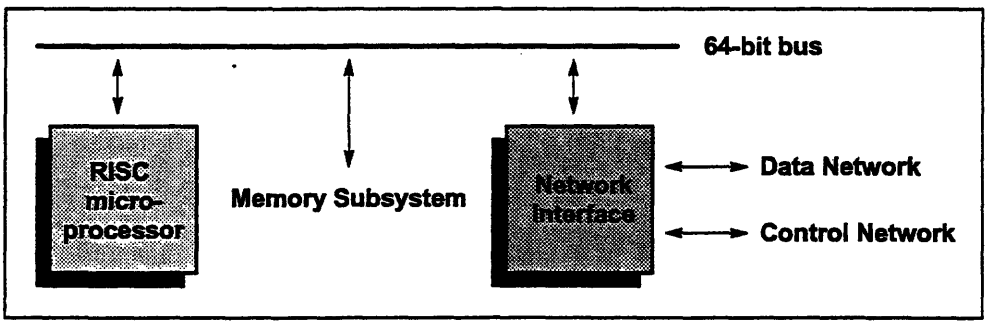

Figure 2. The components of a typical processing node.

The microprocessor (a SPARC chip in the current implementation) executes your code. When this manual speaks of the "node" executing a function or accessing a network, it's really the microprocessor that does the work.

The memory subsystem consists of up to 32 Mbytes of DRAM memory, which is managed either by a memory controller or by a set of four vector units (if your CM-5 has the vector unit option installed). Check with your applications engineer or system administrator to find out what memory hardware is available.

The NI chip serves as an intermediary between the microprocessor and the two networks, providing a standard network interface throughout the CM-5.

1.1.3 Partitions and the Partition Manager

Typically, your code doesn't have access to every processing node in the CM-5. Instead, your code runs on a *partition* of nodes that are monitored by a single *partition manager* (PM) node:

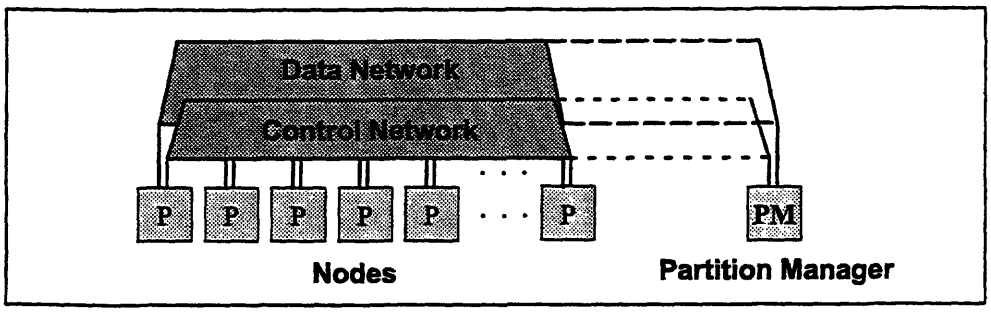

Figure 3. A partition of nodes and its partition manager.

The PM is much like the processing nodes, in that it is attached to the Data and Control Networks and can send messages via the NI. Programs written for the CM-5 normally include two separate files of code, one for the PM and one for the nodes.

Most often, the PM and the nodes operate in a data parallel style: the nodes execute identical programs simultaneously, while the PM controls which function the nodes will execute next. For more information on program structure, see Chapter 7.

1.2 The NI Chip

Although each NI chip in the CM-5 is physically part of its processing node, it's useful to think of the NI as being a separate entity from the rest of the node, because the NI operates independently:

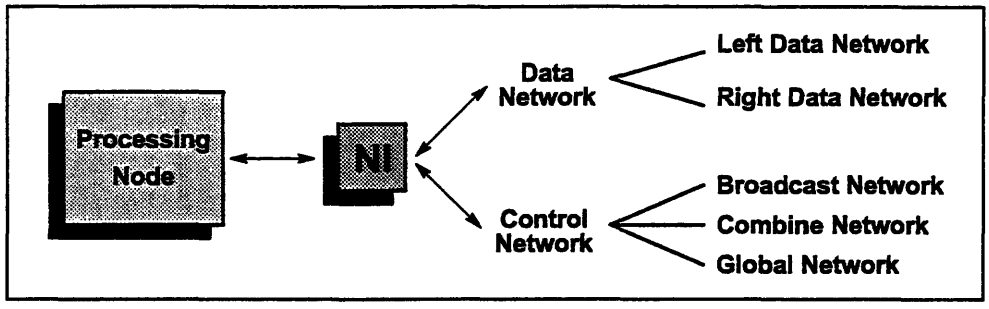

Figure 4. The NI provides access to the facilities of the Data and Control Networks.

When the processing node directs the NI to send a network message, the NI handles the actual dispatching of the message as well as the collection of any replies that arrive from the networks.

The NI uses output buffering to hold messages until they can be sent, and input buffering to hold received messages until the node can read them. The processing node can either examine the registers of the NI to see if a message has arrived, or it can instruct the NI to signal an interrupt when a message arrives.

1.2.1 The NI Registers

The NI chip is register-based — its network functions are controlled entirely by reading and writing registers. For each network the NI manages, there is a unique set of NI registers.

Access to these registers is provided by memory-mapping — the registers are mapped into the memory address space of the microprocessor. This means that from a programmer's point of view the NI is just a region of processor memory with some unique properties.

1.2.2 Types of Registers

There are three basic types of NI control registers:

- First, there are *queue registers*. These "registers" are actually the entry and exit points of message queues associated with the CM-5 networks. Instead of holding values, queue registers act as entry and exit points for network messages.

  By "writing" a message to a queue register, you push the message onto the *send queue* of a network. Likewise, reading the value of a queue register pops a message from the *receive queue* of the network.

- Secondly, there are *status* registers, which are composed of one-bit flags and multi-bit fields. These registers are used to indicate the state of the NI and its message queues.

  For example, most networks have two status flags, **`send_ok`** and **`rec_ok`**, that indicate the current status of messages being sent or received.

- Finally, some status registers act as *control* registers; that is, altering the value of a control register's flags and fields has a corresponding effect on the state of the NI.

  As an example, some networks have an *abstain* flag that you can set or clear to control whether or not the NI will participate in the transactions of one or more networks.

The chapters of this manual that describe each of the networks also describe the NI registers that are associated with them, and describe the programming tools you can use to access these registers.

Implementation Note: Some NI queue registers are mapped onto more than one memory location, and thus appear as regions of memory. Nevertheless, these regions of memory are still considered to be a single "register." The specific memory location that you use in writing to these registers gives the NI additional information about the kinds of network transactions it should perform. (More on this in Section 2.2.1.)

Performance Note: In terms of cycles, reading and writing NI registers is midway between reading the registers of the microprocessor and reading a value from processor memory. See Section 8.1.1 for details on the time taken to read and write NI registers.

1.2.3 Register and Field Names

For consistency of reference in this manual, the names of NI registers and register fields are given in the form

ni_network_purpose

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

`bc`	broadcast network	`com`	combine network
`ldr`	left data network	`rdr`	right data network
`dr`	data network (left and right)	`global`	global data network

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

`send`	The entry-point of a network send queue.
`rec`	The exit-point of a network receive queue.
`send_ok`	Flag indicating that a message was sent successfully.
`send_space`	Field containing amount of space in the send queue.

For conciseness, this manual will occasionally refer to a register by the *purpose* portion of the name alone, but only when this kind of reference is unambiguous.

Names of Register Accessors

The programing tools that you'll use to access NI registers and fields typically have names based on the registers and fields they manipulate.

For example, most networks have an NI register named **`ni_`*`network`*`_status`** that contains the **`ni_`*`network`*`_send_ok`** and **`ni_`*`network`*`_rec_ok`** status flags. Instead of having a separate function for each network to get the value of these flags, there is a single pair of macros, **`SEND_OK()`** and **`REC_OK()`**, that is used to get the **`send_ok`** and **`rec_ok`** flag for any of the networks.

1.3 Writing NI Code

It's possible for you to write NI code using any programming method that allows you to read and write memory addresses. However, this manual assumes that NI programs are written in the C programming language, because there are a large number of existing C macros that you can use to streamline your code. These programming tools fall into two categories:

- Accessor macros that read or write the value of a specified register, flag, or field. (The `SEND_OK` and `REC_OK` macros are good examples.)
- Queue macros that take a number of arguments related to the sending of a single data value, and handle the necessary protocol for sending it.

These tools are introduced individually in the chapters that follow, and there is a complete list of them in Appendix A.

Note: To get access to these predefined macros, your program must `#include` the header file `cmna.h`. (See Chapter 7 for more information.)

1.4 Using This Manual Effectively

The first few chapters of this manual are mainly explanatory, describing the networks of the CM-5 in detail and showing you how to use the NI programming tools associated with them. While these network-specific chapters present some brief code examples, none of these examples constitutes a complete NI program in and of itself. There's a fair amount of information that you simply have to digest before a complete NI program makes sense.

Beginning CM-5 programmers should read through the "generic" network description in Chapter 2, and then read all four network-specific chapters, before turning to the complete sample program presented in Chapter 7.

Experienced CM-5 programmers should read through Chapter 2 and at least one of the network-specific chapters to get a sense of how the networks operate, and then proceed to the sample program in Chapter 7 to see how NI programs are structured.

Whatever your level of experience, Chapter 8 presents a number of important performance strategies and potential sources of programming errors that you should know about.

1.5 WARNING: Experiment at Your Own Risk

In writing code that manipulates the NI chip, you are taking control of the lowest level of the CM-5's hardware. That kind of power does not come without corresponding responsibilities and hazards.

This manual sets strict protocols for reading and writing the registers of the NI. When you use the features of the NI in the manner described here, you should encounter no problems outside of the occasional error message.

If you step outside the bounds, however, the results can be as nasty as they are unpredictable. In some cases reading and writing NI registers incorrectly can even cause your partition of processing nodes to crash, potentially disrupting other timesharing users of the CM-5.

So remember, if you choose to experiment with the NI, you have been warned!

Chapter 2

A Generic Network Interface

Each of the networks accessible through the NI has its own network interface — its own set of control registers that are used to send and receive messages. However, most of the network interfaces have a number of features in common. This chapter presents a "generic" network interface that describes these features. With one exception (the global network), all networks conform to the model described here. Individual variations are described in separate chapters for each network.

Important: The functions described in this chapter are pseudocode representations, not actual functions. You'll get an error if you try to call one of the nonexistent "generic" functions described here.

2.1 Network Messages

For the purposes of this manual, a network *message* is a sequence of word-length values. The length and content of a message depends on the network. Sending a message involves writing this sequence of values to the send queue of a network. As the message is written, the individual values are held in the send queue. When the entire message has been written to the queue, the NI begins trying to send the message through the network.

Similarly, receiving a message involves reading a sequence of values from the receive queue of a network. When the NI receives a message from a network, it accumulates the message in the receive queue. When the entire message has been received, the NI signals that a message is available. Your program can then read the individual words of the message from the receive queue.

2.2 Sending a Message

For each network, two queue registers are used for sending a message:

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

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

- The first value of a message *must* be written to the **`send_first`** queue register. This signals to the NI that a message is being composed.
- The remaining values of the message *must* be written, in order, to the **`send`** queue register.

If this protocol is not followed, an error is signaled and the message currently being composed is discarded.

When a message is discarded, any values that have been written to the queue are ignored, and the send queue resets itself so that a new message can be started by writing a value to the **`send_first`** register. Writing additional values to the **`send`** register has no effect.

Performance Note: You can use message discarding to your advantage and thereby make your code more efficient. (See Section 8.1.3.)

2.2.1 Auxiliary Information

A network message also includes some *auxiliary information,* such as the length of the message in words. This auxiliary information is transmitted implicitly when you write the first value of a message to the **`send_first`** register.

Each **`send_first`** "register" is 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 specifies the auxiliary information of the message.

Another way of looking at this is that the length of a network message (among other things) determines which **`send_first`** location you must use in sending the message.

2.2.2 Writing a Message

For each *network*, there are two **send_first** macros,

```
CMNA_network_send_first(auxiliary–info, value)
CMNA_network_send_first_double(auxiliary–info, value)
```

that are used to write the first *value* of a message to the **send_first** register. The only difference between them is that the **send_first** macro writes an **unsigned** value, while **send_first_double** writes a **double**. However, for these two macros it's not the *type* of data being sent that's important, only the length.

The **send_first** macro is intended to be used for sending word-length data, and the **send_first_double** macro is intended for sending double-word data. In each case, you should coerce the values you send to the appropriate data type. For example, to send a data value of type **float**, you must first cast it as an **unsigned** value. To send a negative integer value, you must also first coerce it to an **unsigned** value.

Performance Note: There are two kinds of **send_first** macro so that you can use double-word register operations to make your code more efficient. (See Section 8.1.2 for more information.) For the most part, however, this manual focuses on single-word operations for clarity.

For the second and succeeding values of a message there is a different group of macros. For each *network* there are three macros that write values to the **send** register, one for each of the three data types you can send:

```
CMNA_network_send_word(value)
CMNA_network_send_float(value)
CMNA_network_send_double(value)
```

The **send_word** macro writes an **unsigned** word-length value, and the other two macros write values of the indicated data types. Here there are three macros to allow you to send values of differing data types without having to coerce them. You're not restricted to using only one data type, of course; you can use any combination of **send_***type* macro calls when sending a message.

Important: Remember that the **send_***type* macros do not work unless they are preceded by a **send_first** or **send_first_double** call for the same network. You'll get an error if you attempt to use them to send the first value of a message. If you have only one value to send, use the appropriate **send_first** macro.

2.2.3 The Network Status Register

The **ni_*network*_status** register can be used to check on the progress of a message that is being sent. It contains the following flags and fields related to message sending:

ni_*network*_status	Status register, contains the following fields:
ni_*network*_send_ok	Flag, the status of message being sent.
ni_*network*_send_space	Field, the space left in send queue.
ni_*network*_send_empty	Flag, indicates the send queue is empty.

If the send queue becomes full, all attempts to write a message (either to start one or to continue one) cause the message currently being composed to be discarded. You can tell whether 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 message is discarded, the flag is set to 0, indicating that the message has not been sent.

You can check the **send_ok** flag both during and after writing a message to see whether the message is accepted for delivery. If the **send_ok** flag indicates that a message has been discarded, you should retry sending the entire message.

The **send_space** field contains an *estimate* of the total space in words left in the queue. The actual space remaining may be less; **ni_*network*_send_space** is usually correct, but may become invalid in some cases (such as during process swaps). You should not assume that pushing a message shorter than this value will always be successful.

The **send_empty** flag is 1 whenever the send queue is empty — that is, when there is no pending message in the queue.

Programming Note: NI programmers typically write an entire message to the send queue and only *then* check the **send_ok** flag to see whether it was accepted. (See Section 8.1.3 for more information.) For this reason, the **send_space** field and **send_empty** flag typically aren't used by NI programmers. The **send_empty** flag is used by internal NI operations that need to determine the state of the network's send queues.

2.2.4 Reading the Send Status Register Fields

The general method for reading the value of an **ni_*network*_status** field or flag is to read the value of the entire status register, and then extract the required fields from that value. (This cuts down the overhead of repeatedly reading the value of the register.)

Each network has a macro that obtains the current value of the **status** register:

```
int value = CMNA_network_status()
```

Because the position and size of status fields and flags are the same for most of the networks, there is a single set of macros that extract the send status fields from the value returned by **CMNA_*network*_status**:

SEND_OK (*status*)	Gets **send_ok** flag from the given *status* value.
SEND_SPACE (*status*)	Gets **send_space** field.
SEND_EMPTY (*status*)	Gets **send_empty** flag.

For example, to get the three **send** fields from the broadcast network's status register:

```
int value = CMNA_bc_status();
int send_ok = SEND_OK(value);
int space_left = SEND_SPACE(value);
int send_queue_empty = SEND_EMPTY(value);
```

2.3 Receiving a Message

For each network, the following register is used to receive a message:

ni_*network*_rec	Queue register from which values are read.

A message is received by reading its values, in order, from **ni_*network*_rec**. There are three network-specific message-reading macros, one for each network:

```
int value = CMNA_network_receive_word();
int value = CMNA_network_receive_float();
int value = CMNA_network_receive_double();
```

As with the **send_*type*** macros, you are not restricted to reading values of a particular type. You can use any combination of the **rec_*type*** in reading a message.

2.3.1 Detecting and Reading a Received Message

The **ni_*network*_status** register contains additional flags and fields that are used for detecting and receiving incoming messages:

ni_*network*_status	Status register, contains the following fields:
ni_*network*_rec_ok	Flag, indicates arrival of a message.
ni_*network*_rec_length	Field, total length of message received.
ni_*network*_rec_length_left	Field, words left in the receive queue.

The status register's fields always reflect the "current" message, that is, the message that is currently being read (or is waiting to be read) from **ni_*network*_rec**.

Whenever a message is pending in the receive queue, the **rec_ok** flag is set to 1, and remains set while the message is read from the queue. When no messages are waiting to be read, the flag is set to 0. (**Note:** It is an error to try to read from the queue when this flag is 0.)

The field **ni_*network*_rec_length** always contains the total length (in words) of the current message *as it was when it was received.*

The field **ni_*network*_rec_length_left** contains the number of words remaining to be read from the queue. (**Usage Note:** You can assume that it is safe to read this many words from **ni_*network*_rec**.)

2.3.2 Reading the Receive Status Register Fields

Just as with the send status fields, there are generic macros that can be used to extract the values of the receive status fields:

RECEIVE_OK (*status*)	Gets **rec_ok** flag from a *status* value.
RECEIVE_LENGTH (*status*)	Gets **rec_length** field.
RECEIVE_LENGTH_LEFT (*status*)	Gets **rec_length_left** field.

For example, to get the **rec** fields from the right data network's status register:

```
int value = CMNA_RDR_status();
int rec_ok = RECEIVE_OK(value);
int message_length = RECEIVE_LENGTH(value);
int words_to_go = RECEIVE_LENGTH_LEFT(value);
```

2.4 Abstaining from the Network

Some networks have an *abstain flag*, a flag that you can set to cause a node to ignore the transactions of the network. The abstain register and flag typically have names like:

`ni_`*network*`_control`	Abstain status register.
`ni_`*network*`_abstain`	Network abstain flag.

(The global network, always the exception, uses a different name for this register. See Chapter 6 for more information.)

The **`ni_`***network*`_abstain`** flag, when set to 1, causes the NI to "ignore" the transactions of the network. All incoming messages are discarded, and the **`rec_ok`** flag remains 0. Attempts to write a message with the abstain flag set will signal an error.

2.4.1 Reading and Writing the Abstain Flag

To read and write the the abstain flag of a network, use these macros:

```
value = CMNA_read_abstain_flag(register);
CMNA_write_abstain_flag(register, value);
```

The *register* argument is a register address constant, which is defined separately for each network.

2.4.2 Using the Abstain Flag Safely

The abstain flag for a given network should only be changed when that network is not in use. This means that there must be no messages traveling through the network and you must not be either writing to a send queue or reading from a receive queue in any node.

This generally requires that you use one of the NI's global synchronization features to bring operations to a halt in all nodes while the abstain flags are changed. (See Chapter 6 for a discussion of the global network's synchronization features.) The effects of changing a network's abstain flags while the network is in use are unpredictable — your code may run, producing erroneous results, or it may signal an error.

Also, 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 your code exits. Failing to do so can cause your code to signal bizarre errors that are hard to trace.

2.5 Different Networks, Different Purposes

The above description is an idealized view of a network, lacking either a specific purpose or a detailed description of message protocol and the restrictions on usage of the network.

The next four chapters present a description of the Data Network and the three Control sub-networks. Each chapter presents the purpose, protocol, and restrictions of a real CM-5 network, building on the material presented in this chapter.

Chapter 3

The Data Network

The Data Network consists of two half-networks, the *left data network* (LDR) and *right data network* (RDR). The two sub-networks can also be accessed together as a single *data network* (DR).

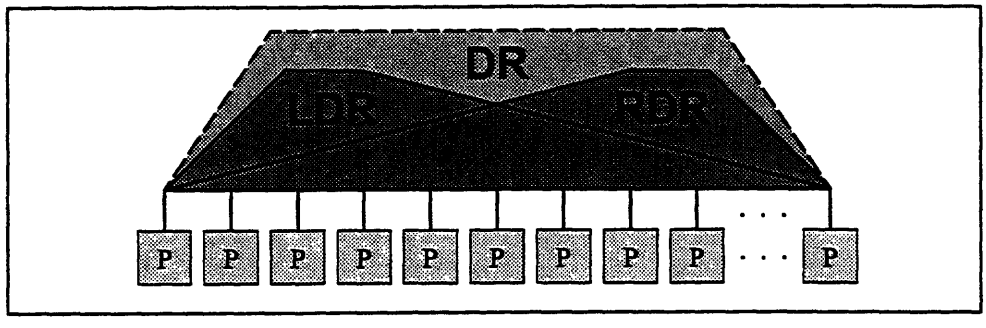

Figure 5. The Data Network consists of independent left and right halves.

There are three sets of control registers: one set for each half-network, and a separate set for the combined (DR) network. Each set of registers is used to send and receive messages via the corresponding network, 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 half-networks. The DR cannot be used to receive messages.
- Sending a message via the DR excludes using either the LDR or RDR individually, and invalidates the LDR and RDR send status fields until the message has been written to the DR send queue and accepted for delivery.

- Likewise, sending a message via either the LDR or RDR excludes using the DR for sending, and invalidates the DR status registers until the message has been written to the LDR or RDR send queue and accepted for delivery.
- The two half-networks are not exclusive, however, and can operate independently — messages can be sent and received concurrently through both the LDR and RDR.

3.1 Data Network Addressing

The Data Network delivers messages to specific processing nodes in the CM-5, as indicated by an address word that is added to each message. Each node has a unique address based on its location in its partition, and these addresses run from 0 (for the first node in the partition) up to one less than the total number of nodes in the partition:

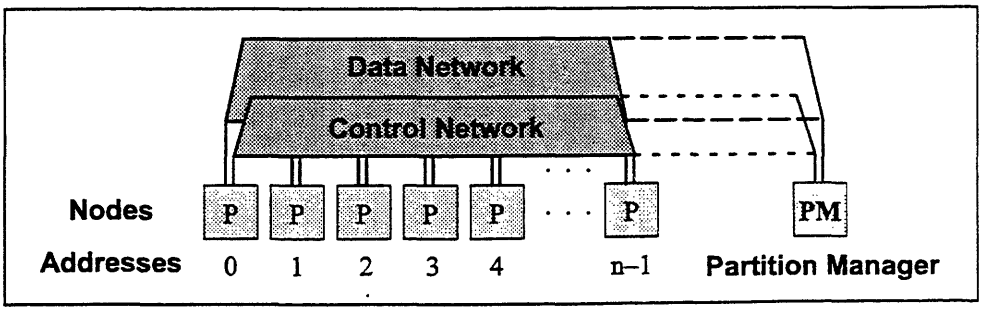

Figure 6. Addressing of nodes in a partition.

You can get the address of the node executing your code, as well as the total number of nodes in the current partition, by examining these variables:

`CMNA_self_address`	Address of current node.
`CMNA_partition_size`	Number of nodes in current partition.

The values of these variables are automatically defined for each of the nodes. The value of **`CMNA_partition_size`** is also defined for the partition manager.

3.2 Sending a Message

The message format for all three Data Networks is the same: The first word of the message is a 20-bit destination address that *must* be zero-extended to 32 bits. **Important:** Failure to do so can trigger a serious error, even causing your partition to crash (See Section 8.2.1).

The remaining words form the content of the message, which must be no longer than the value of **MAX_ROUTER_MSG_WORDS** (currently 5).

The auxiliary information of the message consists of the length of the message in words (excluding the address word), and a four-bit tag value. See Section 3.3.4 for information on the use of tag values.

3.2.1 Message-Sending Macros

The sending interface used for the three Data Networks is the same as the generic interface in Chapter 2. The following queue registers are used to send messages:

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

The *dnetwork* part of these names is a unique abbreviation for each network,

`dr` – data network `ldr` – left data network `rdr` – right data network

and for each *dnetwork* there are corresponding **send_first** and **send** macros:

`CMNA_`*dnetwork*`_send_first(`*tag, length, value*`)`
`CMNA_`*dnetwork*`_send_first_double(`*tag, length, value*`)`

`CMNA_`*dnetwork*`_send_word(`*value*`)`
`CMNA_`*dnetwork*`_send_float(`*value*`)`
`CMNA_`*dnetwork*`_send_double(`*value*`)`

For the **send_first** macros, the *length* argument is the length of the message in words (excluding the address word), the *tag* argument is the message's tag value, and *value* is the first value of the message.

For the **send** macros, *value* is the second and succeeding values of the message.

Note: Currently you are limited to using *tag* values from 0 to 3. All other tags are reserved for supervisor use.

3.2.2 Status Register Fields

The following Data Network status register fields are used for message sending:

ni_*dnetwork*_status	Status register, contains the following fields:
ni_*dnetwork*_send_ok	Flag, the status of message being sent.
ni_*dnetwork*_send_space	Field, the space left in send queue.
ni_send_state	Field, indicates status of three networks.
ni_router_done_complete	Flag, indicates send queue is empty.

The macros used to get the **ni_*network*_status** value for each network are:

```
int value = CMNA_dr_send_status();
int value = CMNA_ldr_status();
int value = CMNA_rdr_status();
```

You can obtain the values of the **send_ok** flag and **send_space** field for each network by using the field extractors described in Chapter 2 (Section 2.2.4).

The **ni_send_state** field and **ni_router_done_complete** flag exist only for the DR interface (that is, they are only accessible from the **ni_dr_status** register). You can obtain the values of these fields by using the following macros:

```
DR_SEND_STATE(status)
DR_ROUTER_DONE(status)
```

For example:

```
int value = CMNA_LDR_status();
int state = DR_SEND_STATE(value);
int network_done = DR_ROUTER_DONE(value);
```

The **ni_router_done_complete** flag is used by the combine network as part of its network-done message function. For more information, see Section 5.5.

3.2.3 Using the send_state Field

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 queue while there is a partially completed message in either the LDR or RDR send queues.

Likewise, having a partially completed message in the DR send queue makes it an error to try to send a message via the LDR or RDR queues. In either case, the

state of the **send** status registers and queues of the excluded network(s) is undefined.

You can use the **ni_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 queue.
1 Partial message in the DR send queue.
2 Partial message in either or both of the LDR or RDR send queues.

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

3.3 Receiving a Message

The message-receiving interface of the Data Networks is as described in Chapter 2. The following register is used for receiving Data Network messages:

`ni_`*dnetwork*`_rec` Queue register from which values are read.

The *dnetwork* abbreviation is the same as for the send registers.

3.3.1 Message-Receiving Macros

To receive a message from the LDR or RDR, use the network-specific reading operations described in Section 2.3:

```
value = CMNA_dnetwork_receive_word();
value = CMNA_dnetwork_receive_float();
value = CMNA_dnetwork_receive_double();
```

Important: There are no message-receiving macros for the DR. You must use the LDR and RDR to receive messages sent via the DR — the DR interface cannot be used to receive messages.

3.3.2 Status Register Fields

The following Data Network status register fields are used in receiving messages:

ni_*dnetwork*_status	Status register, contains the following fields:
ni_*dnetwork*_rec_ok	Flag, indicates receipt of a message.
ni_*dnetwork*_rec_tag	Field, tag value of the message.
ni_*dnetwork*_rec_length	Field, total length of message.
ni_*dnetwork*_rec_length_left	Field, words left in the queue.
ni_rec_state	Field, status of three networks.

You can obtain the values of the **rec_ok** flag and the **rec_length** and **rec_length_left** fields by using the generic field extractors described in Chapter 2 (Section 2.3.2).

The **rec_tag** field always contains the tag value of the current message. To get the **rec_tag** field, use the macro:

RECEIVE_TAG (*status*)

Programming Note: Along with checking the **rec_ok** flag to determine whether there is a message to read, you must also check the tag value of a message before retrieving it. (See Section 3.3.4.)

3.3.3 Using the rec_state Field

Just as there is an **ni_send_state** field for the Data Networks, there is a corresponding **ni_rec_state** field that you can use to determine which receive interfaces are in use. You can get the value of this field by using the macro:

DR_RECEIVE_STATE (*status*)

The value of this field is an integer from 0 to 2, with the following meanings:

0 No partial messages in any receive queue.
1 Reserved.
2 Partial message in either or both of the LDR or RDR receive queues.

As with the **ni_send_state** field, the **ni_rec_state** field exists only for the DR interface (it is only accessible from the DR status register).

3.3.4 Tag Fields and Interrupts

The tag values of Data Network messages are used to distinguish different types of messages sent through the network. You can also use them to automate the handling of Data Network messages in your code.

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** operator 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 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 sixteen-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* will cause *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 3.)

So, for example, the function call

```
CMOS_signal( SIGMSG , my_msg_handler , 14 );
```

arranges the NI interrupt system so that when a Data Network message with a tag of 1, 2, or 3 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. (A copy is included as Appendix B of this manual.)

IMPORTANT — Check the Tag before Retrieving a Message

Whether or not you use tag-driven interrupts to receive messages, 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 queue.

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. Because the CM-5 operating system itself sends Data Network messages with interrupt tags, the effect of doing so is unpredictable; an error may be signaled, or your partition may crash.

To avoid this problem, check the tag of a Data Network message *before* retrieving it to make certain that it is a non-interrupting message that you have sent yourself (that is, a message with a tag from 0 to 3 that you have not assigned as an interrupt tag).

3.4 Other Things to Know:

3.4.1 Data Network Message Ordering

The values of a single Data Network message are always retrieved in order. However, because of the way messages are managed while they are being transferred through the network, there is no guarantee that the order in which two separate messages are sent will be the order in which they are received.

If you send multiple messages to the same destination, your code should not depend on having them arrive in any particular order. If you require messages to be received in a particular order, you must wait for each message to be received before sending the next.

3.4.2 "Receive before You Send" — A Data Network Usage Note

An important strategy to keep in mind when using the Data Network is "Receive before you send." That is, 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 your usage of both networks, so that neither network becomes more heavily loaded than the other.

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

3.5 Examples

The examples shown below are code fragments intended to be run on the processing nodes. See Chapter 7 for a discussion of large-scale program structure.

Also, since the interfaces for the DR, LDR, and RDR are virtually identical, the examples below are written for the LDR only. Appropriate functions for the other network interfaces can be obtained by appropriate substitution of names.

Sending and Receiving a Message

Here is a pair of functions that send and receive messages via the LDR interface. The *message* is assumed to be composed of *length* words of data, and is sent with the specified *tag* value to the node with the given *dest_address*.

```
int LDR_send (dest_address, message, length, tag)
        unsigned dest_address, tag;
        int *message;
        int length;
{  int i;
  CMNA_ldr_send_first(tag, length, dest_address);
  while (length--) CMNA_ldr_send_word(*message++);
  return (SEND_OK(CMNA_ldr_status()));
 }

/* Highest tag NOT currently assigned as interrupt */
int tag_limit=0;

int LDR_receive (message, length)
        int *message;
        int length;
{
  int i, tag = 999;
  /* Skip messages currently assigned as interrupts */
  while (tag>tag_limit) {
    if (RECEIVE_OK(CMNA_ldr_status()))
      tag = RECEIVE_TAG(CMNA_ldr_status());
  }
  while (length--)
    *message++ = CMNA_ldr_receive_word();
  return (tag);
}
```

For example, the following code fragment causes each node to send a message to the node with the next-higher node address. (The node with the highest address sends a message to node 0.)

```
int next_node = (CMNA_self_address + 1)
                % CMNA_partition_size;
int i, message[MAX_ROUTER_MSG_WORDS];
for (i=0, i<MAX_ROUTER_MSG_WORDS, i++) message[i]=i;
LDR_send(next_node, message, MAX_ROUTER_MSG_WORDS, 0);
LDR_receive( message, MAX_ROUTER_MSG_WORDS );
```

Sending and Receiving Long Messages

Of course, the above functions are limited by the size restriction on Data Network messages. If you have a lot of data to send, you'll probably want to use a function that can send a message of any word length, breaking it up into chunks as appropriate. Here's such a function, which handles both sending and receiving the message in a single function call:

```
/* Send/Receive function with no length restriction */
LDR_send_receive_msg(dest_address, message, length,
                     tag, dest)
    unsigned dest_address, tag;
    int *message, *dest;
    int length;
{
    int packet_size=MAX_ROUTER_MSG_WORDS-1;
    int send_size, receive_size;
    int offset, source_offset=0, dest_offset;
    int words_to_send=length, words_received=0;
    int count, rec_tag, status;

    while ((words_received<length)||(words_to_send)) {

        /* First try to receive a packet */

        status=CMNA_ldr_status();

        if (words_received<length &&
             RECEIVE_OK(status) &&
             RECEIVE_TAG(status) <= tag_limit) {

         dest_offset = CMNA_ldr_receive_word();

         receive_size=
          RECEIVE_LENGTH_LEFT(CMNA_ldr_status());

         for (count=0; count<receive_size; count++)
         dest[dest_offset++]=CMNA_ldr_receive_word();

         words_received += receive_size;

        }
```

```
            /* Now try sending a packet */

            if (words_to_send) {
             send_size = ((words_to_send < packet_size) ?
                          words_to_send : packet_size);

             do {
              CMNA_ldr_send_first(tag, send_size+1,
                                  dest_address);

              /* Send offset of msg data being sent */
             CMNA_ldr_send_word(source_offset);

              offset=source_offset;
              for (count=0; count<send_size; count++)
                 CMNA_ldr_send_word(message[offset++]);

             } while (!SEND_OK(CMNA_ldr_status()));

             source_offset=offset;
             words_to_send -= send_size;

            } /* if */

        } /* while */

    }
```

Here's an example of how to call this function:

```
#define LONG_FACTOR 5

int mirror_node = (CMNA_partition_size-1) -
CMNA_self_address;

int i, length = MAX_ROUTER_MSG_WORDS*LONG_FACTOR;
int send[MAX_ROUTER_MSG_WORDS*LONG_FACTOR];
int receive[MAX_ROUTER_MSG_WORDS*LONG_FACTOR];

for (i=0, i<length, i++) long_message[i]=i;

LDR_send_receive_msg(mirror_node, send, length, 0, re-
ceive);
```

Interrupt-Driven Message Retrieval

Using interrupt-driven message retrieval simply requires that you define a handler to be called when an interrupting message arrives. The handler should take no arguments, and its returned value is ignored.

```
/* Message-receiving handler for interrupt-driven LDR
test */
#include <cm/cm_signal.h>
int interrupt_done = 0;
int interrupt_expect_length;
int interrupt_receive[MAX_BROADCAST_MSG_WORDS];

void LDR_receive_handler ()
{
    int temp=tag_limit;
    tag_limit=3;
    LDR_receive(interrupt_receive,
                interrupt_expect_length);
    tag_limit=temp;
    interrupt_done=1;
}
```

And that you use **CMOS_signal** to inform the NI that it should signal an interrupt from some or all of the possible tag values. For example:

```
int i, next_node, message_length=MAX_ROUTER_MSG_WORDS;
int message[MAX_ROUTER_MSG_WORDS];
for (i=0, i<message_length, i++) message[i]=i;
next_node = (CMNA_self_address+1)%CMNA_partition_size;

/*  signal interrupts for non-zero tag values   */
CMOS_signal( SIGMSG , LDR_receive_handler , 14 );

/*  Send message with an interrupt tag (3) */
interrupt_done = 0;
interrupt_expect_length = message_length;
LDR_send(next_node, message, message_length, 3);

/* Wait for handler to signal interrupt finished */
while (interrupt_done==0) {};
printf("Received message: ");
for (i=0, i<message_length, i++)
    printf("%d ", message[i]);
```

Sending via LDR and RDR Simultaneously

One advantage to having the two sub-networks in the Data Network is that you can send messages simultaneously through the LDR and RDR. For example, here's a pair of functions that send a single message via both networks, comparing the received results to make sure that the message was received properly:

```
/* Send/Receive functions using LDR and RDR in tandem
*/
void LDR_RDR_send (dest_address, message, length, tag)
     unsigned dest_address, tag;
     int *message, length;
{
  int i;
  CMNA_ldr_send_first(tag, length, dest_address);
  CMNA_rdr_send_first(tag, length, dest_address);
  for (i=0; i<length; i++) {
    CMNA_ldr_send_word(message[i]);
    CMNA_rdr_send_word(message[i]);
  }
}

int LDR_RDR_receive (message, length)
     int *message, length;
{
  int i, ldr_value, rdr_value, length_received_ok=0;
  while (!RECEIVE_OK(CMNA_ldr_status()) ||
         !RECEIVE_OK(CMNA_rdr_status())) {}
  for (i=0; i<length; i++) {
    ldr_value=CMNA_ldr_receive_word();
    rdr_value=CMNA_rdr_receive_word();
    if (ldr_value==rdr_value) {
      message[i]=ldr_value;
      length_received_ok++;
    }
  }
  return(length_received_ok);
}
```

Chapter 4

The Broadcast Network

The broadcast network is one of the components of the Control Network, and is used to broadcast a value from a single source node to all nodes in the same partition.

4.1 Sending a Message

A broadcast message consists of from 1 to **MAX_BROADCAST_MSG_WORDS** (currently 5) words of data.

The only auxiliary information associated with a broadcast message is its length. However, the length of a message is only meaningful for the purpose of sending the message. A broadcast message consisting of more than one word of data may be split in transit into two or more smaller messages. Also, as broadcast messages are received by the NI, they are appended together in the receive queue so that it always appears as if there is one "message" waiting to be read.

Thus, the original length of a broadcast message has no meaning for the nodes receiving it. (Note, however, that the *order* of the words sent in a broadcast message *is* preserved, regardless of how those words are recombined into new messages.)

Important: Each node has a system flag that controls whether broadcast sending is permitted. In the current implementation, this flag is turned off by default. To turn on this flag, you *must* call the following macro prior to any broadcast network operations:

```
CMNA_participate_in(NI_BC_SEND_ENABLE);
```

4.1.1 Message-Sending Macros

The sending interface used for the broadcast network is the same as the generic interface in Chapter 2. The following queue registers are used to send messages:

`ni_bc_send_first`	Used for first value of a message.
`ni_bc_send`	Used for the rest of the message.

and there are corresponding **`send_first`** and **`send`** macros:

`CMNA_bc_send_first(`*length, value*`)`
`CMNA_bc_send_first_double(`*length, value*`)`

`CMNA_bc_send_word(`*value*`)`
`CMNA_bc_send_float(`*value*`)`
`CMNA_bc_send_double(`*value*`)`

For the **`send_first`** macros, the *length* argument is the length of the message in words, and *value* is the first value of the message. For the **`send`** macros, *value* is the second and succeeding values of the message.

Important: To avoid contention for network resources, at most one NI in any partition should broadcast at any time. If two or more NI's attempt to broadcast simultaneously, the effect is unpredictable. An error may be signaled and/or transmitted data may be lost.

Also, because of way the broadcast and combine networks interact, if a node is abstaining from a combine network operation, that node should *not* execute a broadcast operation until the combine operation is completed. (For more information, see Section 8.2.6.)

4.1.2 Status Register Fields

The following broadcast status register fields are used for message sending:

`ni_bc_status`	Status register, contains the following fields:
`ni_bc_send_ok`	Flag, the status of message being sent.
`ni_bc_send_space`	Field, the space left in send queue.
`ni_bc_send_empty`	Flag, indicates that the send queue is empty.

The macro used to get the value of the broadcast status register is:

```
int value = CMNA_bc_status()
```

You can obtain the values of the **send_ok** and **send_empty** flags and the **send_space** field by using the generic field extractors described in Chapter 2 (Section 2.2.4).

4.2 Receiving a Message

The message-receiving interface of the broadcast network is as described in Chapter 2. The following register is used for receiving broadcast messages:

`ni_bc_rec`	Queue register from which values are read.

4.2.1 Message-Receiving Macros

To receive a message from the broadcast network, use the network-specific reading operations described in Section 2.3:

```
value = CMNA_bc_receive_word();
value = CMNA_bc_receive_float();
value = CMNA_bc_receive_double();
```

4.2.2 Status Register Fields

The following broadcast network status register fields are used in receiving messages:

`ni_bc_status`	Status register, contains the following fields:
`ni_bc_rec_ok`	Flag, indicates a message has been received.
`ni_bc_rec_length_left`	Field, number of words left in message.

You can obtain the values of the **rec_ok** flag and the **rec_length_left** field by using the generic field extractors described in Chapter 2 (Section 2.3.2).

4.2.3 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 queue. 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 **ni_bc_rec_length_left** will retrieve the rest of that message. (Remember, however, that this method is not guaranteed to read all of a message that was divided in transit.)

4.3 Abstaining from the Network

The broadcast network has an abstain flag that you can use to cause the NI to ignore incoming broadcast messages. The following registers and flags are used:

ni_bc_control	Status register, contains **rec_abstain** field.
ni_bc_rec_abstain	Flag, broadcast network abstain flag.

Setting the abstain flag, **ni_bc_rec_abstain**, to 1 causes the NI to discard any arriving messages. (The value of **ni_bc_rec_ok** will always be 0, and the receive queue will always be empty.) Setting the flag back to 0 allows the NI to receive messages again.

You can use the macros described in Section 2.4 to read and write the abstain flag. (The abstain register address constant for the broadcast network is **bc_control_reg**.)

```
value = CMNA_read_abstain_flag(bc_control_reg);
CMNA_write_abstain_flag(bc_control_reg, value);
```

4.4 Examples

The examples shown here are fragments of code intended to be run on the processing nodes. See Chapter 7 for a discussion of large-scale program structure.

Sending and Receiving a Message

This function sends a message via the broadcast interface. The message is assumed to be composed of *length* words of data starting at the location specified by *message*.

```
int BC_send(message, length)
    int *message, length;
{
    int i;
    CMNA_bc_send_first(length--, *message++);
    for (i=0; i<length; i++)
         CMNA_bc_send_word(*message++);
    return(SEND_OK(CMNA_bc_status()));
}
```

This function receives a message via the broadcast interface, stores it in memory beginning at the location specified by *message*, and returns the length of the message received.

```
int BC_receive(message, length)
    int *message, length;
{
    int i;
    for(i=0; i<length; i++) {
         while(!RECEIVE_OK(CMNA_bc_status())) {}
         message[i] = CMNA_bc_receive_word();     }
    return(length);
 }
```

For example:

```
int i, message[MAX_BROADCAST_MSG_WORDS];
for (i=0, i<MAX_BROADCAST_MSG_WORDS, i++)
    message[i]=i;

BC_send(message, MAX_BROADCAST_MSG_WORDS);
BC_receive(message, MAX_BROADCAST_MSG_WORDS);
```

Chapter 5

The Combine Network

The combine network is the second component of the Control Network. It is used for executing operations that combine in parallel a single value from each processing node. The supported operations are: parallel prefix (scanning), reduction operations, and network-done tests. (These operations are described individually below.)

5.1 Sending a Message

A combine network message consists of from 1 to **MAX_COMBINE_MSG_WORDS** (currently 5) words of data, with the exception of network-done messages, which are always 1 word in length.

The auxiliary information has three parts:

- The length of the message in words
- A three-bit *combiner* value, determining the combine operation performed
- A two-bit *pattern* value, selecting the order in which values are combined

The sending interface used for the combine network is the same as the generic interface in Chapter 2. The following queue registers are used to send messages:

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

and there are corresponding **send_first** and **send** macros, described in the next section.

5.1.1 Message-Sending Macros

The message-sending macros for the combine network are:

```
CMNA_com_send_first(combiner, pattern, length, value)
CMNA_com_send_first_double(combiner, pattern, length,value)

CMNA_com_send_word(value)
CMNA_com_send_float(value)
CMNA_com_send_double(value)
```

For the **send_first** macros, the *length* argument is the length of the message in words, and *value* is the first value of the message. The *combiner* and *pattern* arguments are described in the sections below, covering each of the possible combine operations.

For the **send** macros, *value* is the second and succeeding values of the message.

Important: Combine operations are not completed until all participating nodes have signaled the *same* type of combine operation. If two nodes attempt to start different combining operations at the same time, an error is signaled.

5.1.2 Status Register Fields

The following combine status register fields are used for message sending:

`ni_com_status`	Status register, contains the following fields:
`ni_com_send_ok`	Flag, the status of message being sent.
`ni_com_send_space`	Field, the space left in send queue.
`ni_com_send_empty`	Flag, indicates the send queue is empty.

The macro used to get the value of the combine status register is:

```
int value = CMNA_com_status()
```

You can obtain the values of the **send_ok**, **send_space**, and **send_empty** fields by using the generic field extractors described in Chapter 2 (Section 2.2.4).

5.1.3 Pipelining Combine Operations

Combine network operations can be pipelined; after starting one combine operation, you can immediately start another without waiting for the first to complete. The length of the pipeline is limited only by the capacity of the message queues. However, all nodes must agree in the type of combine operation they signal.

Note: Pipelining prevents you from using double-word writes to send combine messages — see Section 8.1.2.

5.2 Receiving a Message

The message-receiving interface of the combine network 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`** (see Section 5.5).

The following register is used for receiving combine network messages:

`ni_com_rec`	Queue register from which values are read.

To receive a message from the combine network, use the network-specific reading operations described in Section 2.3:

```
value = CMNA_com_receive_word();
value = CMNA_com_receive_float();
value = CMNA_com_receive_double();
```

5.2.1 Status Register Fields

The following combine status register fields are used in receiving messages:

`ni_com_status`	Status register, contains the following fields:
`ni_com_rec_ok`	Flag, indicates received message.
`ni_com_rec_length`	Field, total length of message received.
`ni_com_rec_length_left`	Field, number of words left in message.

You can obtain the values of these fields by using the generic field extractors described in Chapter 2 (Section 2.3.2).

5.2.2 Abstaining from the Network

The combine network has *two* abstain flags that you can use to cause the NI to abstain from combine network transactions:

`ni_com_control`	Status register, contains combine abstain flags.
`ni_com_abstain`	Flag, combine network abstain flag.
`ni_reduce_rec_abstain`	Flag, special reduction abstain flag.

Setting the **`ni_com_abstain`** flag to 1 causes the NI to discard any arriving combine network messages, and allows any messages signaled by other nodes to complete without the participation of the abstaining node.

In the case of combine operations that expect a value from each node, abstaining nodes effectively supply an appropriate identity value for the operation. However, no result value is written to an abstaining node's receive queue (except for reduce operations, which use the other abstain flag, **`ni_reduce_rec_abstain`**, for this purpose; see Section 5.4).

You can use the abstain flag macros described in Section 2.4 to read and write the abstain flag, using the register address constant **`com_control_reg`**:

```
value = CMNA_read_abstain_flag(com_control_reg);
CMNA_write_abstain_flag(com_control_reg, value);
```

For the **`ni_reduce_rec_abstain`** flag, there is a separate pair of macros:

```
value = CMNA_read_rec_abstain_flag(com_control_reg);
CMNA_write_rec_abstain_flag(com_control_reg, value);
```

Note: Because of way the broadcast and combine networks interact, if a node is abstaining from a combine network operation, that node should *not* execute a broadcast operation until the combine operation is completed. (For more information, see Section 8.2.6.)

Important: As with any other abstain flags, the **`ni_com_abstain`** flag and the **`ni_reduce_rec_abstain`** flag should only be changed when there are no messages pending in the combine network.

5.3 Parallel Prefix (Scanning) Messages

A scan message is from 1 to 5 words in length, representing a value to be combined with the values provided by other nodes on the network. A scanning message can be sent with one of five different combining functions and in one of three different scanning patterns, as determined by the *combiner* and *pattern* values specified when the message is sent.

After all participating nodes have transmitted a scanning value, the values are combined according to the selected *combiner* operation and *pattern*. The result is delivered as a message to all participating nodes after a brief interval. The values are combined cumulatively — that is, the result for each node is the combination of its own transmitted value with the values of all nodes having lower (or higher) addresses.

The legal *combiner* and *pattern* values are specified by symbolic constants. The *combiner* argument must be one of the constants

- `ADD_SCAN` Signed addition.
- `UADD_SCAN` Unsigned addition.
- `OR_SCAN` Bitwise inclusive OR.
- `XOR_SCAN` Bitwise exclusive OR.
- `MAX_SCAN` Signed maximum.

and the *pattern* argument must be one of the constants

- `SCAN_FORWARD` Values are combined in ascending address order.
- `SCAN_BACKWARD` Values are combined in descending address order.
- `SCAN_REDUCE` Reduction operation (see Section 5.4 below).

Important: If you are sending a scan message that is longer than one word, the order in which the words of the message must be 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.

5.3.1 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 combine network interface includes a register used to specify segments:

`ni_scan_start` Status register, indicates start of scan segments.

The one-bit flag in the register **`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 **`SCAN_FORWARD`** operations and downward for **`SCAN_BACKWARD`** operations. If no **`ni_scan_start`** flags are set in a partition, then the entire partition is treated as one segment.

You can read and modify the value of **`ni_scan_start`** by using these macros:

```
int value = CMNA_segment_start();
CMNA_set_segment_start(value)
```

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

5.3.2 Addition Scan Overflow

Addition scans on large values can cause arithmetic overflow in some nodes. The combine network status register includes a flag that you can use to detect such an overflow:

`ni_com_status` Status register.
`ni_com_scan_overflow` Flag, set if add scan had overflow.

The value of the **`ni_scan_overflow`** flag is 1 when the scan message currently being received suffered arithmetic overflow; otherwise, it is 0.

You can obtain the current value of this flag by using the field extraction macro:

```
value = COMBINE_OVERFLOW(status);
```

Note: This flag is only meaningful when the current message being received is an addition scan (an **ADD_SCAN** or **UADD_SCAN** operation).

5.4 Reduction Messages

Reduction is a special case of scanning — a reduction message is simply a scanning message with with a *pattern* value of **SCAN_REDUCE**. The effect of a reduction operation is to combine the values obtained from every node and then to deliver that combined result as a message to all nodes. Every node receives the same combined value.

5.4.1 Abstain Flags for Reduction Messages

In terms of node participation, reduction differs from scanning. The **ni_com_abstain** flag allows a combine operation to proceed without the participation of a given node, but does not prevent the abstaining node from *receiving* the result of the reduction message. There is an additional abstain flag, **ni_reduce_rec_abstain**, that controls whether a node *receives* a reduction result. When **ni_reduce_rec_abstain** is 1, all incoming reduction messages will be discarded.

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. (For an example of just such a situation, see Section 7.1.)

5.5 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* network. All non-abstaining processors *must* signal a network-done message, or else the network-done operation will not complete.

A network-done message is *always* of length 1; the actual value written as the message is ignored. Also, there is a unique pair of *combiner* and *pattern* constants that are used to signal a network-done operation:

combiner: `ASSERT_ROUTER_DONE` *pattern:* `SCAN_ROUTER_DONE`

Network-done messages are an exception to the usual message-reception interface of the combine network. No message is delivered to the combine receive queue as a result of a network-done operation. 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 status register.
`ni_router_done_complete`	Flag, indicates when router is empty.

When a network-done message is sent by a node, the **`ni_router_done_complete`** flag of that node is set to 0. When all 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. (See Section 3.2.2 for the macro that extracts this flag.)

5.5.1 How Network-Done Works, and Why You Should Care

Each node maintains an internal register that is incremented when the node sends a user message, and decremented when the node receives a user message. (System messages are not counted.) When no user messages are being transmitted through the Data Network, the sum of this register across all nodes should be zero.

Network-done messages use an add-scan operation to detect when the Data Network is clear of transmitted messages. Once all non-abstaining nodes have signaled a network-done message, the combine network does a repeated add-scan on the message count registers of the nodes until the sum for all nodes is zero. It then sets the `ni_router_done_complete` flag to 1 in all nodes.

NOTE

Because of a hardware defect, Revision A NI chips don't always execute network-done operations correctly. See Section 8.2.5.

Since network-done operations involve a *combine network* scan of the value of a *Data Network* register, you should be careful about setting and changing the abstain flags of the combine network when you intend to send a network-done message.

For example, if you change the combine abstain flags of one or more nodes while a Data Network operation is in progress, you may exclude nodes with non-zero message count registers from combine network operations. If you then signal a network-done operation, the excluded message count registers may prevent the network-done addition scan from returning zero, and thus the network-done message may never complete.

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.

5.6 Examples

The examples shown here are fragments of code that are intended to be run on the processing nodes. See Chapter 7 for a discussion of large-scale program structure.

Sending and Receiving a Combine Message

This function sends a message via the combine interface. The message is assumed to be composed of *length* words of data starting at the location specified by *message*, and is sent with the given *combiner* and *pattern*.

```
int COM_send(combiner, pattern, message, length)
    int *message, combiner, pattern, length;
{   int i, start, step;
    /* For max scans, send high-order word(s) first */
    if (combiner==MAX_SCAN) {start=length-1; step=-1;}
    else { start=0; step=1; }
    CMNA_com_send_first(combiner, pattern,
                        length, message[start]);
    for (i=1; i<length; i++)
         CMNA_com_send_word(message[(start+=step)]);
    return(SEND_OK(CMNA_com_status())); }
```

This function receives a message, stores it in memory beginning at the location specified by *message*, and returns the length of the message received. (Note that a *combiner* must also be specified, so that maximum scans are retrieved in the right order.)

```
int COM_receive(combiner, message)
    int *message;
{   int i, length, start, step;
    while(!RECEIVE_OK(CMNA_com_status())) {}
    length=RECEIVE_LENGTH(CMNA_com_status());
    /*For max scans,receive high-order word(s) first*/
    if (combiner==MAX_SCAN) {start=length-1; step=-1;}
    else { start=0; step=1; }
    for(i=0; i<length; i++) {
         message[start] = CMNA_com_receive_word();
         start+=step;    }
    return(length);
}
```

Executing Scans and Reduction Scans

This function sends and receives a scan using the given *message* of length *words*, with the specified *combiner* and *pattern*, storing the result in memory starting at *result*.

```
int COM_scan(combiner, pattern, message,
             length, result)
    int *message, *result, combiner, pattern, length;
{
    int status=0, rec_length;
    while (!status)
     status=COM_send(combiner,pattern,message,length);
    rec_length = COM_receive(combiner,result);
    return(rec_length);
}
```

Here's an example of a simple scan using integer values:

```
int send[MAX_COMBINE_MSG_WORDS],
    receive[MAX_COMBINE_MSG_WORDS];

for (i=1; i<MAX_COMBINE_MSG_WORDS; i++)
    send[i]=i;

COM_scan(ADD_SCAN, SCAN_FORWARD, send,
         MAX_COMBINE_MSG_WORDS, receive);
```

As a practical example, you can use a reduction scan on integer values to get the number of non-abstaining processors in the current partition:

```
int send = 1, receive = 0;

COM_scan(ADD_SCAN, SCAN_REDUCE, &send, 1, &receive);

printf("Actual number of processors: %d\n",
         CMNA_partition_size );

printf("Scanned number of processors: %d\n",
         receive );
```

Executing a Network-Done Operation

Here's a simple network-done synchronizing function:

```
void network_done_synch()
{
 CMNA_com_send_first(ASSERT_ROUTER_DONE,
                     SCAN_ROUTER_DONE,1,0);
  while (!DR_ROUTER_DONE(CMNA_dr_status())) {};
}
```

For example:

```
int message = 1;
int network_done_msg = 0;
int next_processor = (CMNA_self_address+1)
                        % CMNA_partition_size;

/* Send a message */
LDR_send (next_processor, &message, 1, 0);

/* Synchronize the nodes */
network_done_synch()

/* Retrieve the message */
LDR_receive (&message, 1);
```

Chapter 6

The Global Network

The global network is the third component of the Control Network. Unlike the broadcast and combine networks, the global network does not use the generic interface model presented in Chapter 2.

The purpose of the global network is to act as a generic synchronization mechanism for the nodes of the CM-5. The global network combines a single bit from every participating node in a logical OR operation, and then returns the result to each node. Thus, it is much like the network-done facility of the combine network, but without the additional condition that the Data Network must be clear before the operation can complete.

A global network message can be sent in either a synchronous or an asynchronous operation, and there is a separate register interface for each of these two methods. The synchronous interface requires that all nodes signal a message before any receive the result. The asynchronous interface permits nodes to send a message and read the result at any time, with the global network continuously monitoring the state of all participating nodes.

6.1 The Synchronous Global Interface

The following registers and flags form the synchronous global network interface:

`ni_sync_global_send`	Status register, contains flag used for sending global messages.
`ni_sync_global`	Status register, contains the following flags:
`ni_sync_global_complete`	Flag, indicates completion of message.
`ni_sync_global_rec`	Flag, indicates global OR of messages.

6.1.1 Sending and Receiving a Message

To start a synchronous global network message, write a value (either 0 or 1) to the the **ni_sync_global_send** register. To do this, use the macro:

```
CMNA_or_global_sync_bit(value)
```

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 signaled a message, the global network 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. To detect when the message has completed and to retrieve the resulting global value, use the macros

```
value = CMNA_global_sync_complete();
value = CMNA_global_sync_rec();
```

6.1.2 Abstaining from the Synchronous Interface

The synchronous global interface includes an abstain flag that can be used to exclude a node from the network'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 will complete without the node (as if the node has set its **ni_sync_global_send** flag to 0). You can use the abstain flag operations described in Chapter 2 (Section 2.4) to read and write the value of the **ni_sync_global_abstain** register. (The address constant for this register is **sync_global_abstain_reg**.) For example:

```
value=CMNA_read_abstain_flag(sync_global_abstain_reg);
CMNA_write_abstain_flag(sync_global_abstain_reg, value);
```

Note: The abstain flag can only be changed when there is no global message pending (that is, an error is signaled if the abstain flag is modified when the **ni_sync_global_complete** flag is 0). Also, an error is signaled if the **ni_sync_global_send** register is written while the abstain flag is 1.

6.2 The Asynchronous Global Interface

The following register and flags form the asynchronous global network interface:

`ni_global`	Status register, contains the following flags:
`ni_global_send`	Flag, used to "send" asynchronous global message.
`ni_global_rec`	Flag, always set to logical OR of **send** flags.

The asynchronous global interface operates continuously — there is no such thing as "sending" or "receiving" a message via this interface. The **ni_global_rec** flag in each node is continuously 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 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 will be no longer than the time taken to transmit a synchronous message.

To set the value of the **ni_global_send** flag, use the macro

```
CMNA_or_global_async_bit(value);
```

and to retrieve the value of the **ni_global_rec** flag, use the macro

```
value = CMNA_global_async_read();
```

6.3 Examples

The examples shown here are fragments of code intended to be run on the processing nodes. See Chapter 7 for a discussion of large-scale program structure.

Using the Synchronous Global Network

Here's a function that executes a simple barrier synchronization using the global network.

```
int global_sync_value(value)
    unsigned int value;
{
    CMNA_or_global_sync_bit(value);
    while (!CMNA_global_sync_complete()) {};
    return(CMNA_global_sync_read());
}
```

All non-abstaining nodes must execute this function for the global message to be completed. If you don't need to send or receive a value, you can rewrite this as:

```
int global_sync()
{
    CMNA_or_global_sync_bit(1);
    while (!CMNA_global_sync_complete()) {};
    (void) CMNA_global_sync_read();
}
```

Using the Asynchronous Global Network

The following function sends a value using the asynchronous global interface, and then immediately reads and returns the current value from the receive register.

```
int CMNA_global_async(value)
  unsigned int value;
{
  CMNA_or_global_async_bit(value);
  return (CMNA_global_async_read());
}
```

Chapter 7

Writing NI Programs

In this chapter we'll start applying some of the tools presented in the preceding chapters. First, we'll cover important small-scale programming issues, such as exchanging data between the nodes in a partition and the partition manager. Next, we'll look at a short program that makes use of every network interface of the NI.

7.1 Transferring Data between Nodes and the PM

As described in Section 3.1, each node in a partition has a unique address based on its location in the partition. However, the PM is not part of this addressing scheme. The PM is always located outside of the address space of the partition that it manages:

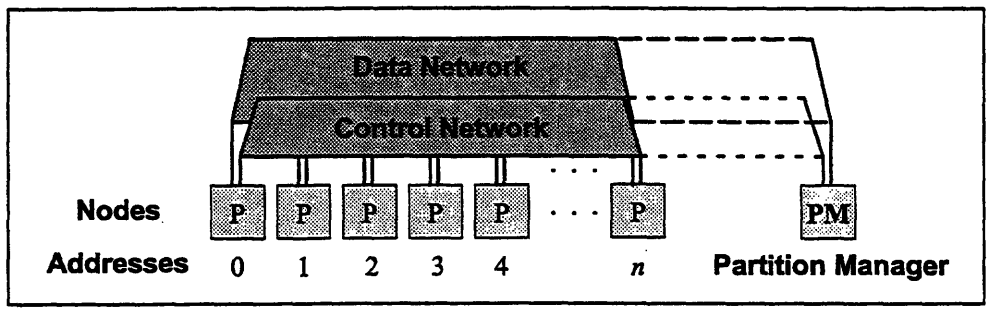

Figure 7. The partition manager stands apart from the partition it manages.

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 from the PM to Nodes

To send a message from the PM to a node, the PM does two broadcast operations: one to send the address of the node that should "receive" the message, and one to transmit the message itself.

For example:

```
void PM_send_to_NODE(node_address, value)
    int node_address, value;
{
    CMNA_bc_send_first(1, node_address);
    CMNA_bc_send_first(1, value);
}
```

Each of the nodes should perform two broadcast network reads, one to determine whether the address of the message matches the node's own address, and one to either receive and store the message or to ignore it, based on the supplied node address:

```
int NODE_get_from_PM(dest)
    int *dest;
{
    int address, value;

    while (!RECEIVE_OK(CMNA_bc_status())) {};
    address=CMNA_bc_receive_word();

    while (!RECEIVE_OK(CMNA_bc_status())) {};
    value=CMNA_bc_receive_word();

    if (address == CMNA_self_address) *dest=value;
}
```

Notice that the node waits until the **rec_ok** flag is set *each* time it tries to receive a value from the broadcast network. This is important — while these routines are written so that the PM's two broadcast values should arrive in the node's receive queue nearly simultaneously, it's still necessary to check the **rec_ok** flag before each broadcast read, because the two values are still separate messages.

Also, notice that in this example only one node "accepts" the value sent from the PM, but there's no reason why you can't have more than one node "accept" the value — you can use any test you like to decide whether the nodes keep or discard the values they receive.

7.1.2 Sending Messages from Nodes to the PM

Sending a message from a node to the PM is almost as straightforward, but involves two networks this time, the broadcast network and the combine network.

First, the PM sets its **ni_com_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. (**Note:** We'll handle this step separately in Section 7.2, below.)

Next, the PM broadcasts a message containing the address of a processing node, as in the **PM_send_to_NODE** example above. The nodes respond by signaling a combine network message (a **UADD_SCAN** reduction), in which only the node with the address specified by the PM transmits a value. (The other nodes supply an identity value of 0 for the reduction.) The PM then receives this message to get the requested value.

Here's the function that handles the PM side of this transaction:

```
int PM_get_from_NODE(node_address)
    int node_address;
{
    CMNA_bc_send_first(1, node_address);
    while (!RECEIVE_OK(CMNA_com_status())) {};
    return(CMNA_com_receive_word());
}
```

And here's the corresponding node function:

```
void NODE_send_to_PM(value)
   int value;
{
    int address;
    while (!RECEIVE_OK(CMNA_bc_status())) {};
    address = CMNA_bc_receive_word();
    if (address != CMNA_self_address) value = 0;
  CMNA_com_send_first(UADD_SCAN,SCAN_REDUCE,1,value);
    while (!RECEIVE_OK(CMNA_com_status())) {};
   (void) CMNA_com_receive_word();
}
```

Notice that immediately after the nodes send a combine message, they perform a combine network read to discard the resulting value. You might think it would be a good idea to temporarily toggle the combine abstain flags for the nodes, so that they will simply ignore the result. However, this is not such a good strategy. (Why not? See Section 7.2.)

7.1.3 Signaling the PM

Because the above PM/node communication functions use both the broadcast and combine networks, we'll want a function that forces the PM to wait until the nodes have finished their computations before the PM broadcasts a request for the results. A single function will suffice for both the PM and the nodes:

```
void PM_NODE_synch()
{
    CMNA_or_global_sync_bit(1);
    while (!CMNA_global_sync_complete()) {};
    (void) CMNA_global_sync_read();
}
```

This function uses the global network to create a simple barrier synchronization.

7.1.4 For the Curious: Using the Data Network

You can also use the Data Network to send messages between the partition manager and the nodes. However, owing to the distinction between addressing on the nodes and on the partition manager, it's not as clear-cut an operation as using the broadcast and combine methods described above.

To send a message from the partition manager to a specific node via the Data Network, you can use the methods presented in Chapter 3, using the node's address as the destination for the message.

To send a message from a node to the partition manager, however, you must make a system function call:

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

where the *network* abbreviation is **dr**, **ldr**, or **rdr**, depending on the network, and the other arguments are as noted in Chapter 3. The partition manager can then receive this message as usual. There is a catch, however — this system call is currently implemented as a trap instruction, which in practical terms means it is much less efficient than the combine network method shown in Section 7.1.2.

Sending messages to and from the PM via the Data Network is primarily useful in cases where you want to send a message to a specific node without requiring all the other nodes to stop and do a network operation at the same time.

7.2 Setting the Abstain Flags

Both the PM and the nodes will need to modify their abstain flags in order to use the above functions. Since they will also need to restore the previous values of these flags afterwards, it makes sense to use a single pair of functions to handle saving and restoring the flags, rather than individually toggling flags within a program.

Also, while changing abstain flags in the middle of a program does work, it's error-prone because it requires that you ensure the corresponding network(s) are empty before changing the abstain flag settings. It's much more straightforward to simply set the abstain flags appropriately at the beginning of your program, and then leave them alone as much as possible.

With these factors in mind, here are a pair of functions that handle saving and restoring the abstain flags, giving them whatever intermediate settings you select.

First, a routine that saves the current values of the abstain flags and then sets them to new values.

```
int bc_abstain_flag,
    com_abstain_flag,
    com_rec_abstain_flag,
    sync_global_abstain_flag;

void save_and_set_abstain_flags
       (new_bc, new_com, new_com_rec, new_sync_global)
    int new_bc, new_com, new_com_rec, new_sync_global;
{
    bc_abstain_flag =
     CMNA_read_abstain_flag(bc_control_reg);
    com_abstain_flag =
     CMNA_read_abstain_flag(com_control_reg);
    com_rec_abstain_flag =
     CMNA_read_rec_abstain_flag(com_control_reg);
    sync_global_abstain_flag =
     CMNA_read_abstain_flag(sync_global_abstain_reg);

    CMNA_write_abstain_flag(bc_control_reg, new_bc);
    CMNA_write_abstain_flag(com_control_reg, new_com);
    CMNA_write_rec_abstain_flag(com_control_reg,
                                new_com_rec);
    CMNA_write_abstain_flag(sync_global_abstain_reg,
                                new_com);
}
```

Next, a function that restores the old values:

```
void restore_abstain_flags()
{
    CMNA_write_abstain_flag(bc_control_reg,
                            bc_abstain_flag);
    CMNA_write_abstain_flag(com_control_reg,
                            com_abstain_flag);
    CMNA_write_rec_abstain_flag(com_control_reg,
                               com_rec_abstain_flag);
    CMNA_write_abstain_flag(sync_global_abstain_reg,
                           sync_global_abstain_flag);
}
```

One caveat about these functions: they assume that none of the Control sub-networks are in use when you call them. This should be the case if you call them at the beginning and end of your program, as they are intended to be used. If you need to use functions like these within the body of a program, you should precede and follow them with code (function calls, etc.) that synchronizes the nodes, thus ensuring that none of the affected networks are in use.

For example, you can use the global network to synchronize the nodes while you change the abstain flags for the other networks, and then use the network-done operation of the combine network to synchronize while you change the abstain flags for the global network. (You can probably now see why it's easier just to set these flags once and then ignore them!)

7.3 Broadcast Enabling

Along the setting the abstain flags, there's one other important operation that needs to be included in any NI program. As noted in Section 4.1.1, you need to call the macro

```
CMNA_participate_in(NI_BC_SEND_ENABLE);
```

to enable broadcast sending — *even if you clear the broadcast abstain flag.* The best point in your program to do this is the same place you set the abstain flags.

7.4 NI Program Structure

Now, with these tools in hand we can turn to the task of designing an NI program.

An NI program consists of three files:

- Code to be run on the partition manager
- Code to be run on the nodes (one program executed by all nodes)
- An interface file defining the node routines that are callable from the PM

The sections below describe each of these parts in detail, and show you how to bring them together into a working program.

7.4.1 The `cmna.h` Header File

Important: Both the partition manager code file and the node code file must **`#include`** the header file **`cmna.h`**., as follows:

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

This header file contains **`#include`** directives that load the other files needed to define the NI programming tools described in this manual.

7.4.2 Partition Manager Code

Code that runs on the PM may contain anything ordinarily included in a program running on a Sun computer. This includes **`printf`** calls, system calls, I/O calls, and calls to other specialized libraries. The simplest PM program might look something like this:

```
#include <cm/cmna.h>
void main() {
    /* start node program running */
    node_program();
    }
```

This program does nothing more than call the corresponding node program defined below. Typically, however, the PM code will include operations that send data to the nodes and retrieve the results of the node computations.

7.4.3 Node Code

Code written for execution on the nodes consists of one or more subroutines that perform local computations and make NI calls to send messages through the networks. Node programs can also include simple I/O calls to display intermediate results.

In particular, the output of **`printf`** calls from all nodes is collected and saved in a file (typically named "**`CMTSD_printf.pn.`***pid*") that you can examine during and/or after execution of your program. However, the handling of **`printf`** calls from the nodes slows down program execution considerably, so this method of output is best used only for debugging your program.

Note: As of this release, many UNIX system calls are not supported on the nodes. If node programs invoke these unsupported calls, segmentation violations may be signaled. You should use node subroutines primarily for computations and NI operations, and use the PM code for system calls and external I/O.

The Node's "Main" Routine

The first subroutine in the node file must be the one initially called by the PM. This routine serves much the same function as the "main" routine in standard C programming — it is the trigger that starts everything else running.

While you can give a node subroutine any name that you wish, if it is to be called from the PM, then you must add the prefix **`CMPE_`** to the subroutine name when defining it and when calling it from another node subroutine. This prefix is used by the compiler to determine which subroutines will be called from the PM. You do *not* have to use the **`CMPE_`** prefix anywhere outside of the node subroutine file.

The simplest node program, corresponding to the PM program given above, is:

```
#include <cm/cmna.h>
void CMPE_node_program() {
 /* Node program, does nothing, just an entry point */
    }
```

As you can see, this is less than the bare bones of a subroutine — it does nothing at all. We'll see an example of a complete node program below.

7.4.4 Interface Code

The "interface code" file is nothing more than a file of function prototypes, as might appear in a header file. It is used in the compilation process to produce special declaration code that allows the nodes to respond correctly to subroutine calls from the PM.

The interface code file for the skeletal program given above has just one line:

```
void node_program();
```

Important: Before you compile it, the interface code file must be preprocessed by the utility program **sp-pe-stubs**.This utility program translates your interface prototypes into complete subroutine calls that can be compiled with the PM and node code files to produce an executable NI program.

This is the reason why node functions callable from the PM require the **CMPE_** prefix — the **sp-pe-stubs** utility adds this prefix to the name of each function so that there's no collision with the names of functions that you have defined.

7.5 A Sample Program

As an example, here's a simple NI program that uses each of the CM-5 networks. First, the partition manager source file:

Filename: `NI_test.c`

```
/* Sample NI program - PM program */
#include <cm/cmna.h>
#include "utils.h"

void main () {
    int input, result, high_node;

    printf("\nSimple NI test program, by W.R.Swanson,\n");
    printf("Thinking Machines Corporation--1/31/92.\n\n");

    /* Enable broadcast sending */
    CMNA_participate_in(NI_BC_SEND_ENABLE);
```

```
    /*Abstain from broadcast reception, combine sending */
    save_and_set_abstain_flags(1,1,0,0);

    /* Start node programs running */
    node_main();

    /* Get value from the user, send it to the nodes. */
    printf("This CM-5 partition has %d nodes.\n",
              CMNA_partition_size);

    printf("Please type an integer to send: ");
    scanf("%d", &input);

    PM_send_to_NODE(0, input);
    printf("Sent value %d to node 0...\n",input);

    /* Wait for the nodes to finish juggling numbers */
    PM_NODE_synch();

    /* Get value from high-address node */
    /* (size - 2, because scan result starts with 0) */
    high_node = CMNA_partition_size-2;

    result = PM_get_from_NODE(high_node);
    printf("Got value %d (should be %d) from node %d.\n",
           result, input, high_node);
    result = PM_get_from_NODE(0);
    printf("Got value %d (should be %d) from node 0.\n",
           result, (input*(high_node+1)));

    restore_abstain_flags();
}
```

Next, the corresponding code for the processing nodes:

Filename: NI_test.node.c

```
/* Sample NI program - node program */
#include <cm/cmna.h>
#include "utils.h"

void CMPE_node_main () {
    int value=0, scan_value, flipped_value;
    int mirror_node_addr;
    CMNA_participate_in(NI_BC_SEND_ENABLE);
    save_and_set_abstain_flags(0,0,0,0);
```

```
/* Node 0 gets the value sent by the PM... */
NODE_get_from_PM(&value);

/* and broadcasts it to all nodes */
if (CMNA_self_address==0) CMNA_bc_send_first(1,value);
while (!RECEIVE_OK(CMNA_bc_status())) {};
value = CMNA_bc_receive_word();

/* Do an addition scan to put a different value
     in each node */
CMNA_com_send_first(UADD_SCAN,SCAN_FORWARD,1,value);
while (!RECEIVE_OK(CMNA_com_status())) {};
scan_value = CMNA_com_receive_word();

/* Use LDR to "flip" order of values in nodes */
mirror_node_addr =
     (CMNA_partition_size-1) - CMNA_self_address;
CMNA_ldr_send_first(0, 1, mirror_node_addr);
CMNA_ldr_send_word(scan_value);
while (!RECEIVE_OK(CMNA_ldr_status())) {};
flipped_value = CMNA_ldr_receive_word();

/* Signal to PM that answer is ready */
PM_NODE_synch();

/* Send value from high-order node back to PM */
NODE_send_to_PM(flipped_value);
/* Send value from node 0 back to PM */
NODE_send_to_PM(flipped_value);

restore_abstain_flags();
}
```

And the interface code file:

Filename: `NI_test.proto`

```
/* Sample NI program - interface code */
node_main();
```

Finally, both the PM and node programs include a utilities file, which includes such tools as the abstain-flag functions and the PM/node communications functions:

Filename: `utils.h`

```
/* Utility code */
int bc_abstain_flag, com_abstain_flag,
    com_rec_abstain_flag, sync_global_abstain_flag;

void save_and_set_abstain_flags(new_bc, new_com,
                                new_com_rec,
new_sync_global)
    int new_bc, new_com, new_com_rec, new_sync_global;
{
    bc_abstain_flag =
     CMNA_read_abstain_flag(bc_control_reg);
    com_abstain_flag =
     CMNA_read_abstain_flag(com_control_reg);
    com_rec_abstain_flag =
     CMNA_read_rec_abstain_flag(com_control_reg);
    sync_global_abstain_flag =
     CMNA_read_abstain_flag(sync_global_abstain_reg);

    CMNA_write_abstain_flag(bc_control_reg, new_bc);
    CMNA_write_abstain_flag(com_control_reg, new_com);
    CMNA_write_rec_abstain_flag(com_control_reg,
                                new_com_rec);
    CMNA_write_abstain_flag(sync_global_abstain_reg,
                            new_sync_global);
}

void restore_abstain_flags()
{
    CMNA_write_abstain_flag(bc_control_reg,
                            bc_abstain_flag);
    CMNA_write_abstain_flag(com_control_reg,
                            com_abstain_flag);
    CMNA_write_rec_abstain_flag(com_control_reg,
                             com_rec_abstain_flag);
    CMNA_write_abstain_flag(sync_global_abstain_reg,
                        sync_global_abstain_flag);
}
```

```
void PM_send_to_NODE(node_address, value)
    int node_address, value;
{
    CMNA_bc_send_first(1, node_address);
    CMNA_bc_send_first(1, value);
}

int NODE_get_from_PM(dest)
    int *dest;
{
    int address, value;
    while (!RECEIVE_OK(CMNA_bc_status())) {};
    address=CMNA_bc_receive_word();
    while (!RECEIVE_OK(CMNA_bc_status())) {};
    value=CMNA_bc_receive_word();
    if (address == CMNA_self_address) *dest=value;
}


int PM_get_from_NODE(node_address)
    int node_address;
{
    CMNA_bc_send_first(1, node_address);
    while (!RECEIVE_OK(CMNA_com_status())) {};
    return(CMNA_com_receive_word()); }

void NODE_send_to_PM(value)
    int value;
{
    int address;
    while (!RECEIVE_OK(CMNA_bc_status())) {};
    address = CMNA_bc_receive_word();
    if (address != CMNA_self_address) value = 0;
    CMNA_com_send_first(UADD_SCAN,SCAN_REDUCE,
                        1,value);
    while (!RECEIVE_OK(CMNA_com_status())) {};
    (void) CMNA_com_receive_word();
}

void PM_NODE_synch()
{
    CMNA_or_global_sync_bit(1);
    while(!CMNA_global_sync_complete()) {};
    (void) CMNA_global_sync_read();
}
```

7.6 Compiling and Executing an NI Program

Note: This section presents a brief overview of the process of compiling and executing an NI program. It's very much like the procedure used in compiling and executing a CMMD program — so much so that you should also read the *CMMD User's Guide* for more information. (In particular, the *CMMD User's Guide* includes examples of using a generic makefile to compile your code. This may be more appropriate to your needs and inclinations than the script example shown below.)

To compile an NI program you must:

- Preprocess the interface file by calling **sp-pe-stubs**.
- Compile the resulting file, as well as the PM and node routine files.
- Link the three object files together with the CM linking program **cmld**.

To illustrate this, here are the steps you would take in compiling the sample program shown above.

First, preprocess the interface code file:

```
/usr/bin/sp-pe-stubs < NI_test.proto > NI_test.intf.c
```

Next, compile the three code files:

```
cc  NI_test.c -c -g -DCM5 -DMAIN=main
   -I/usr/include
cc NI_test.node.c -c -g -DCM5 -dalign -Dpe_obj
   -I/usr/include
cc NI_test.intf.c -c -g -DCM5 -DMAIN=main
   -I/usr/include
```

Finally, link everything together. For this purpose, you *must* use the CM-specific linking program **cmld**:

```
/usr/bin/cmld -o NI_test
    NI_test.o NI_test.intf.o
         -L/usr/lib -lcmna_sp -lcmrts -lm
    -pe NI_test.node.o
         -L/usr/lib -lcmna_pe -lcmrts_pe -lm
```

The result is a single executable file, **NI_test**, which you can run by logging onto one of the partition managers of a CM-5 and executing the file.

7.6.1 A Simple Compiling Script

Here's a short UNIX script that automates this process. It takes as its single argument the name of an NI program, constructs the names of the three component files from the program name, compiles the files, and links them together as shown above.

Note: This script assumes that the program files are all present in the current directory.

```
#! /usr/bin/csh -e -f
echo "Script: $0, Compiling $1 for the NI..."

set PMFILE      = "$1.c"
set PMOFILE     = "$1.o"
set NODEFILE    = "$1.node.c"
set NODEOFILE   = "$1.node.o"
set INTFFILE    = "$1.proto"
set INTFCFILE   = "$1.intf.c"
set INTFOFILE   = "$1.intf.o"
set OUTFILE     = "$1"
set NODEOUTFILE = "$1.pn"
set EXECUTABLE      = "a.out"
set NODEEXECUTABLE = "a.out.pn"

echo 'Preprocessing interface code file: ' $INTFFILE
/usr/bin/sp-pe-stubs < $INTFFILE > $INTFCFILE

echo 'Compiling PM code file: ' $PMFILE
cc -c -g -DCM5 -DMAIN=main -I/usr/include $PMFILE -o
$PMOFILE
echo 'Compiling node code file: ' $NODEFILE
cc -c -g -Dpe_obj -DPE_CODE -I/usr/include $NODEFILE
    -o $NODEOFILE
echo 'Compiling interface code file: ' $INTFCFILE
cc -c -g -DCM5 -DMAIN=main -I/usr/include $INTFCFILE
    -o $INTFOFILE

echo 'Linking it all together...'
/usr/bin/cmld -lg $PMOFILE $INTFOFILE -o $OUTFILE \
    -L/usr/lib -lcmna_sp -lm \
    -pe -lg $NODEOFILE -L/usr/lib -lcmna_pe -lm

echo 'Done. Executable written to: ' $OUTFILE
```

7.6.2 Compiling and Running the Program

Note: The following assumes that you have previously logged into one of the partition managers of a CM-5.

The output of the script for the **NI_test** sample program looks like this:

```
% nicc2 NI_test
Script: nicc2, Compiling NI_test for the NI...
Preprocessing interface code file:  NI_test.proto
Compiling PM code file:  NI_test.c
Compiling node code file:  NI_test.node.c
Compiling interface code file:  NI_test.intf.c
Linking it all together...
Done. Executable written to:  NI_test
```

The script produces a single executable file **NI_test**, which can be executed as follows:

```
50: NI_test

Simple NI test program, by W.R.Swanson,
Thinking Machines Corporation -- 1/31/92.

This CM-5 partition has 32 nodes.
Please type an integer to send to the nodes: 42
Sent value 42 to node 0...
Received value 42 (should be 42) from node 30.
Received value 1302 (should be 1302) from node 0.
```

7.6.3 Online Code Examples

As of Version 7.1.3 of the CM system software, there are online copies of the sample program and script in this chapter, along with copies of the programming examples in Appendix C.

Depending on where your system administrator has chosen to store the CM software, these files may be located under the pathname

```
/usr/cm/src/ni-examples
```

or they may also be located somewhere else entirely. Check with your system administrator for help in locating these files.

Chapter 8

Programming and Performance Hints

This chapter describes the ways you can make your NI programs more efficient, and also points out a few potential programming traps that you may encounter.

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

8.1 Performance Hints

8.1.1 NI Register Operation Times

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

register access	(register variable):	1 cycle
cache memory	(previously accessed variable):	2–3 cycles
NI register access	(`CMNA_`*network*`_status()`):	4–5 cycles
memory access	(newly accessed variable):	~20 cycles

The time taken to perform an NI register read/write operation is longer than the time taken for cached memory accesses, but much shorter than the time for full memory accesses. For efficiency's sake, you should read and write NI registers as sparingly as possible and rely on cached values wherever possible.

For the Curious: This is why the NI status register tools are designed so that you can read an NI status register once and then extract fields from the retrieved value. Once you have retrieved the value of the NI register and stored it in cached memory, the access time for extracting multiple fields decreases substantially.

8.1.2 Reading and Writing Registers with Double-Word Values

While this document focuses for the most part on reading and writing network messages in terms of single words, you can also use double-word operations in reading and writing network registers. This may be more efficient, depending on your application.

Writing a double word to a register has the same effect as writing two single-word values, but involves only one register operation. Likewise, reading a double word from a register is the same as reading two single words. However, you can always overdo a good thing. Attempting a double-word read or write for a message that consists of only one word (as is the case for network-done tests) signals an error.

Usage Note: The combine network, because of its pipelining feature, is an exception. You can't use double-word writes when using pipelined combine operations. However, you *can* always use double-word reads when accessing the combine network.

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

```
cc -c -g -DCM5 -dalign -Dpe_obj -I/usr/include
NI_test.node.c
```

Example: LDR Send/Receive

Here's the **`LDR_send_receive_msg`** function from the Data Network chapter, rewritten to use double-word writes:

```
int tag_limit = 3;

LDR_send_receive_msg(dest_address,message,length,tag,dest)
    unsigned dest_address, tag;
    int *message, *dest, length;
{
    int send_size, send_size2, receive_size,receive_size2;
    int offset, source_offset=0, dest_offset;
    int words_to_send=length, words_received=0;
    int packet_size, count, rec_tag, status;
    double *dbl;
```

```
    if (((int)message & 3) || ((int)dest & 3))
     CMPN_panic("Message or dest not doubleword aligned");
    packet_size = (MAX_ROUTER_MSG_WORDS-1) & ~1;

    while ((words_received < length) || (words_to_send)) {
     /* First try to receive a packet */
     status=CMNA_ldr_status();
     if (words_received<length && RECEIVE_OK(status) &&
         RECEIVE_TAG(status)<=tag_limit) {
      dest_offset = CMNA_ldr_receive_word();
      receive_size =
         RECEIVE_LENGTH_LEFT(CMNA_ldr_status());
      for (count=0; count<(receive_size>>1); count++) {
         dbl = (double *)(&dest[dest_offset++]);
         dest_offset++;
         *dbl = CMNA_ldr_receive_double();
         dbl++;    }
      if (receive_size & 1) /* If word left over */
         dest[dest_offset++] = CMNA_ldr_receive_word();
      words_received += receive_size;
      } /* if */

      /* Now try sending a packet */
      if (words_to_send) {
         send_size = ((words_to_send < packet_size) ?
                      words_to_send : packet_size);
         send_size2 = send_size >> 1;
         do {
          CMNA_ldr_send_first(tag,send_size+1,
                                  dest_address);
          CMNA_ldr_send_word(source_offset);
          offset=source_offset;
          /* Send as many doubles as possible */
          for (count=0; count<send_size2; count++){
              dbl = (double *)(&message[offset++]);
              offset++;
              CMNA_ldr_send_double(*dbl++); }
          if (send_size & 1) /* If a word is left over */
              CMNA_ldr_send_word(message[offset++]);
          } while (!SEND_OK(CMNA_ldr_status()));
         source_offset=offset;
         words_to_send -= send_size;
         } /* if */
     } /* while */
}
```

8.1.3 Use Message Discarding for Efficiency

When a message you are writing to a network send queue 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 queue, 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.

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 will be of any benefit. In addition, repeatedly checking the **send_ok** flag while you are writing a message can slow your code down considerably.

8.1.4 Set the Abstain Flags Once and Forget Them

In most cases, abstain flags of a network can only be changed when the network is not in use — that is, when there are no messages pending in either the send or receive queues, 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. There are examples of utility functions that handle the abstain flags in this way in Section 7.2.

8.2 Potential Programming Traps and Snares

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

8.2.1 Pay Attention to Data Network Addresses

When sending a Data Network message from one node to another, the address of the destination node must be a valid address within the current partition. If an address higher than `CMNA_partition_size` is supplied, the NI will signal 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 will signal 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.

8.2.2 Check the Tag before Retrieving a Data Network Message

As described in Section 3.3.4, 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 queue. 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 3 that you have not assigned as an interrupt tag.)

8.2.3 Make Sure Double-Word Data Is Double-Word Aligned

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

```
cc -c -g -DCM5 -dalign -Dpe_obj -I/usr/include
NI_test.node.c
```

If the double-word values in your code are not properly aligned, the nodes will most likely signal "illegal address" errors, and your code will not run.

8.2.4 Order Is Important in Combine Messages

As noted in Section 5.3, if you are sending a scan 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.

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

Because of a hardware defect, Revision A NI chips do not always transmit network-done messages correctly. As described in Section 5.5.1, an internal register in each NI is used to keep track of the number of messages sent and received through the Data Network, and a combine network add-scan on the value of these registers is used to determine when the network is empty.

Rev A NI chips, however, do not correctly increment and decrement this register. This defect has been corrected in later revisions of the chip, 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. (Consult your applications engineer or systems manager to find out whether this is the case.) 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 zero at the beginning of the node program (for example, at the same point that you set the abstain flags):

```
CMNA_router_msg_count = 0;
```

- Every time the node program successfully sends a message via the Data Network (that is, writes a message to the send queue and detects that the **`send_ok`** flag is set), it should increment the count variable:

```
do { CMNA_ldr_send_first(0, 1, dest_address);
     CMNA_ldr_send_word(message);
   } while (!SEND_OK(CMNA_ldr_status()));
CMNA_router_msg_count++;
```

- 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 of the values of the message), it should decrement the count variable:

```
status = CMNA_ldr_status();
if (RECEIVE_OK(status) && RECEIVE_TAG(status)<4) {
    message = CMNA_ldr_receive_word();
    CMNA_router_msg_count--; }
```

- Just before the node program signals a network-done message, it should use the system function **`CMOS_set_dr_msg_count_reg()`** to write the current value of the count variable into the count register.

```
CMOS_set_dr_msg_count_reg(CMNA_router_msg_count);
do { CMNA_com_send_first
          (ASSERT_ROUTER_DONE,SCAN_ROUTER_DONE,
           1,CMNA_force_read);
  } while (!(SEND_OK(CMNA_com_status())));
```

- **Important:** While waiting for the network-done operation to complete, the node program must write the current value of the count variable into the register *before* examining the **`ni_router_done_complete`** flag:

```
do {status = CMNA_ldr_status();
 if (RECEIVE_OK(status)&&RECEIVE_TAG(status)<4) {
        message = CMNA_ldr_receive_word();
        CMNA_router_msg_count--; }
    CMOS_set_dr_msg_count_reg
        (CMNA_router_msg_count);
    } while (!DR_ROUTER_DONE(CMNA_dr_status())) {};
```

8.2.6 Broadcast and Combine Network Collisions

Because of the way the broadcast and combine networks interact, you should take care in using the abstain flags of these two networks.

In particular, if your code causes a node (processing node or PM) to abstain from the combine network, and if the following conditions hold:

- The abstaining node is signalling a broadcast message
- Simultaneously, the other nodes are signalling 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 partition manager to abstain from the combine network (so that it can receive the results of a scan or reduction operation, for example), yet at the same time you want the PM to broadcast messages to the processing nodes telling them what to do. The conflict arises when the PM needs to broadcast a message at the same time as the nodes are signalling a scan or reduction message.

There is a software workaround to use in this situation. When the abstaining node (the PM in this case) needs to send a broadcast message, use the function

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

instead of the sending methods described in Chapter 4. This function includes internal safety checks that prevent broadcast and combine network messages from colliding.

Appendixes

Appendix A

Programming Tools

A.1 Generic Variables and Macros

To determine the address of a node, and its place within its partition, use these variables:

```
int CMNA_self_address     –  Relative address of current node.
int CMNA_partition_size   –  Number of nodes in partition.
```

These are the macros used to examine the fields of the **ni_*network*_status** register for any *network* that has such a register:

Field Name:	Macro used to read value of field:
ni_*network*_send_ok	SEND_OK (*status_value*)
ni_*network*_send_space	SEND_SPACE (*status_value*)
ni_*network*_send_empty	SEND_EMPTY (*status_value*)
ni_*network*_rec_ok	RECEIVE_OK (*status_value*)
ni_*network*_rec_length	RECEIVE_LENGTH (*status_value*)
ni_*network*_length_left	RECEIVE_LENGTH_LEFT (*status_value*)

For networks that have an abstain flag, there is a pair of macros that can be used to read and write the value of the flag:

```
value = CMNA_read_abstain_flag(register_address);
CMNA_write_abstain_flag(register_address, value);
```

For both macros, *register_address* is a symbolic constant giving the address of the abstain flag register (this is defined separately for each network that has such a register).

For the **write** macro, *value* is the new value (0 or 1) to be written to the flag.

A.2 Data Network Constants and Macros

Send and Receive Register Macros

The **send_first** registers for the Data Networks are accessed via the macros below:

Register Name:	Macros used to write first value of message to register:
ni_dr_send_first	CMNA_dr_send_first(*tag, length, value*) CMNA_dr_send_first_double(*tag, length, value*)
ni_ldr_send_first	CMNA_ldr_send_first(*tag, length, value*) CMNA_ldr_send_first_double(*tag, length, value*)
ni_rdr_send_first	CMNA_rdr_send_first(*tag, length, value*) CMNA_rdr_send_first_double(*tag, length, value*)

The *length* argument in each case is the total length in words of the message to be sent (excluding the address word), and the *tag* argument is the message's tag value.

The **send** and **rec** registers of the Data Networks can be written to and read from by the generic register macros in Section A.1, and by the following special purpose macros:

Register Name:	Macros used to access register:
ni_dr_send	CMNA_dr_send_word(*word_value*) CMNA_dr_send_float(*float_value*) CMNA_dr_send_double(*double_value*)
ni_ldr_send	CMNA_ldr_send_word(*word_value*) CMNA_ldr_send_float(*float_value*) CMNA_ldr_send_double(*double_value*)

ni_ldr_rec	word_value = CMNA_ldr_receive_word();
	float_value = CMNA_ldr_receive_float();
	double_value = CMNA_ldr_receive_double();

ni_rdr_send	CMNA_rdr_send_word(*word_value*) CMNA_rdr_send_float(*float_value*) CMNA_rdr_send_double(*double_value*)

ni_rdr_rec	word_value = CMNA_rdr_receive_word();
	float_value = CMNA_rdr_receive_float();
	double_value = CMNA_rdr_receive_double();

Status Register Macros

The values of the Data Network status registers can be obtained by using the macros:

```
int dr_status = CMNA_dr_send_status();
int ldr_status = CMNA_ldr_status();
int rdr_status = CMNA_rdr_status();
```

You can extract the fields of the status registers by applying the following macros:

Register/Field Name:	Macros used to access fields:
ni_dr_status	
ni_dr_send_ok	SEND_OK(dr_status)
ni_dr_send_space	SEND_SPACE(dr_status)
ni_send_state	DR_SEND_STATE(dr_status)
ni_rec_state	DR_RECEIVE_STATE(dr_status)
ni_router_done_complete	DR_ROUTER_DONE(dr_status)
ni_ldr_status	
ni_ldr_send_ok	SEND_OK(ldr_status)
ni_ldr_send_space	SEND_SPACE(ldr_status)
ni_ldr_rec_ok	RECEIVE_OK(ldr_status)
ni_ldr_rec_tag	RECEIVE_TAG(ldr_status)
ni_ldr_rec_length	RECEIVE_LENGTH(ldr_status)
ni_ldr_rec_length_left	RECEIVE_LENGTH_LEFT(ldr_status)
ni_rdr_status	
ni_rdr_send_ok	SEND_OK(rdr_status)
ni_rdr_send_space	SEND_SPACE(rdr_status)
ni_rdr_rec_ok	RECEIVE_OK(rdr_status)
ni_rdr_rec_tag	RECEIVE_TAG(rdr_status)
ni_rdr_rec_length	RECEIVE_LENGTH(rdr_status)
ni_rdr_rec_length_left	RECEIVE_LENGTH_LEFT(rdr_status)

Message Length Limit

The maximum length of a Data Network message (not counting the address word attached in sending it) is given by the constant

```
MAX_ROUTER_MSG_WORDS
```

A.3 Broadcast Network Constants and Macros

Send and Receive Register Macros

The **send_first** register for the broadcast network is accessed via the macros listed here:

Register Name:	Macros used to write first value of message to register:
ni_bc_send_first	CMNA_bc_send_first(*length, value*) CMNA_bc_send_first_double(*length, value*)

The **send** and **rec** registers of the broadcast network can be written to and read from by the following special purpose macros:

Register Name:	Macros used to access register:
ni_bc_send	CMNA_bc_send_word(*word_value*) CMNA_bc_send_float(*float_value*) CMNA_bc_send_double(*double_value*)

ni_bc_rec	word_value = CMNA_bc_receive_word();
	float_value = CMNA_bc_receive_float();
	double_value = CMNA_bc_receive_double();

Status Register Macros

The value of the broadcast network status register can be obtained by using the macro:

```
int bc_status = CMNA_bc_status();
```

You can extract the fields of the status register by applying the following macros:

Register/Field Name:	Macros used to access fields:
ni_bc_status	
ni_bc_send_ok	SEND_OK(bc_status)
ni_bc_send_space	SEND_SPACE(bc_status)
ni_bc_send_empty	SEND_EMPTY(bc_status)
ni_bc_rec_ok	RECEIVE_OK(bc_status)
ni_bc_rec_length_left	RECEIVE_LENGTH_LEFT(bc_status)

Abstain Register Macros

The broadcast abstain register contains a single flag bit, which can be read and written using the generic abstain bit operations described in Section A.1.

Register/Field Name:	Macros used to access fields:
ni_bc_control	
ni_bc_rec_abstain	value=CMNA_read_abstain_flag (bc_control_reg); CMNA_write_abstain_flag (bc_control_reg,value);

Message Length Limit

The maximum length of a broadcast network message is given by the constant

```
MAX_BROADCAST_MSG_WORDS
```

A.4 Combine Network Constants and Macros

Send and Receive Register Macros

The **send_first** register for the combine network is accessed via the macros below:

Register Name:	Macros used to write first value of message to register:
ni_com_send_first	CMNA_com_send_first (*combiner, pattern, length, value*) CMNA_com_send_first_double (*combiner, pattern, length, value*)

For scan operations, the *combiner* argument can be any one of the constants:

ADD_SCAN MAX_SCAN OR_SCAN UADD_SCAN XOR_SCAN

and the *pattern* argument can be any one of the constants:

SCAN_BACKWARD SCAN_FORWARD SCAN_REDUCE

For network-done operations there is a unique *combiner* and *pattern* pair:

combiner: ASSERT_ROUTER_DONE *pattern:* SCAN_ROUTER_DONE

The **send** and **rec** registers of the combine network can be written to and read from by the generic register macros in Section A.1, and by the following special purpose macros:

Register Name:	Macros used to access register:
ni_com_send	CMNA_com_send_word(*word_value*) CMNA_com_send_float(*float_value*) CMNA_com_send_double(*double_value*)

ni_com_rec	word_value = CMNA_com_receive_word();
	float_value = CMNA_com_receive_float();
	double_value = CMNA_com_receive_double();

Message Length Limit

The maximum length of a combine network message (with the exception of network-done messages, which are always 1 word in length) is given by the constant:

MAX_COMBINE_MSG_WORDS

Segment Start Register Macros

The **ni_scan_start** register is accessed by the following special purpose macros:

Register Name:	Macros used to access register:
ni_scan_start	CMNA_set_segment_start(*value*) value = CMNA_segment_start();

Status Register Macros

The value of the combine network status register can be obtained by using the macro:

```
int com_status = CMNA_com_status();
```

You can extract the fields of the status register by applying the following macros:

Register/Field Name:	Macros used to access fields:
ni_com_status	
ni_com_send_ok	SEND_OK(com_status)
ni_com_send_space	SEND_SPACE(com_status)
ni_com_send_empty	SEND_EMPTY(com_status)
ni_com_rec_ok	RECEIVE_OK(com_status)
ni_com_rec_length	RECEIVE_LENGTH(com_status)
ni_com_rec_length_left	RECEIVE_LENGTH_LEFT(com_status)
ni_com_scan_overflow	COMBINE_OVERFLOW(com_status)

Abstain Register Macros

The combine abstain register contains two single-bit flags, which can be read and written by the macros listed below:

Register/Field Name:	Macros used to access fields:
ni_com_control	
ni_com_abstain	value=CMNA_read_abstain_flag(com_control_reg); CMNA_write_abstain_flag(com_control_reg,*value*);
ni_reduce_rec_abstain	value=CMNA_read_rec_abstain_flag(com_control_reg); CMNA_write_rec_abstain_flag(com_control_reg,*value*);

A.5 Global Network Constants and Macros

Synchronous Global Register Macros

The synchronous global registers are read and written by the following macros:

Register Name:	Macros used to access register:
ni_sync_global_send	CMNA_or_global_sync_bit(*value*)
ni_sync_global	
ni_sync_global_complete	value = CMNA_global_sync_complete()
ni_sync_global_rec	value = CMNA_global_sync_rec()
ni_sync_global_abstain	value=CMNA_read_abstain_flag(sync_global_abstain_reg); CMNA_write_abstain_flag(sync_global_abstain_reg,*value*);

Asynchronous Global Register Macros

The two flags of the asynchronous global register are read and written by these macros:

Register/Flag Name:	Macros used to access register:
ni_global	
ni_global_send	CMNA_or_global_async_bit(*value*)
ni_global_rec	value = CMNA_global_async_read()

Appendix B

CMOS_signal man page

CMOS_signal – asynchronous event handlers on the nodes

```
#include <cm/cm_signal.h>
(*CMOS_signal(sig, func, mask))()
int sig;
void (*func)();
int mask;
```

CMOS_signal allows code on the nodes to specify software handlers for certain asynchronous events. It is the responsibility of the user to ensure that the signal handler does not change the state of the node in any way that will disrupt execution of the interrupted code.

A node program can specify that the arrival of data router messages with a certain set of tags will generate an interrupt. The code specifies the message handler and the set of tags with a call to **CMOS_signal()** with **sig = SIGMSG**, ***func** set to the address of the user-written handler function, and **mask** set to a bit mask specifying which tags will interrupt. (Bit 0 corresponds to tag 0, bit 1 corresponds to tag 1, and so forth.) Currently, tags 0 to 3 are reserved for user messages. Bits 4 and up are reserved for system messages, and may not be used or referenced by user code.

The context of the node is saved before the user message handler is called. Thus, use of floating-point instructions in the user message handler will cause unpredictable errors in the interrupted code. Also, the network state of the CM is not altered before entering the user message handler. Thus, the message(s) that produced the interrupt will still be in the receiving FIFO when the user message handler is invoked. It is the responsiblity of the user message handler to empty these messages.

The handler routine can be declared:

```
void handler()
```

The routine is not passed any parameters relating to the received message. The user message handler must read the NI registers to determine such details as the tag of the message and whether the message has arrived via the left or right data network, etc.

Message interrupts are disabled while user code is in a user message handler. Thus, user message handlers need not be reentrant. Also, if the user code anticipates a series of interrupting messages, the arrival of the first message can be used to invoke the message handler and the remaining messages can be received via polling within the handler, thus saving the overhead of an interrupt for all but the first message.

Appendix C

Sample NI Programs

This appendix contains a series of NI programs that test all the programming examples shown in the chapters of this manual. For each program, only the PM and node code files are given. The interface file for each program is identical to that given for the sample program in Chapter 7, and these test programs `#include` the same `utils.h` file as is used in Chapter 7.

As of Version 7.1.3 of the CM system software, there are on-line copies of the sample programs presented here. Depending on where your system administrator has stored the CM software, these files may be located under the pathname `/usr/cm/src/ni-examples`. Check with your system administrator for help in locating these files.

Note: You should view the examples presented here as a cookbook of possible coding ideas, not a hard-and-fast rulebook on network protocol. These examples are written for clarity, not efficiency, and your own individual application should be your guide as to how to rearrange the code fragments presented here, and how best to trim them for speed.

C.1 Data Network Test

This program presents examples of a number of different kinds of Data Network operations, including:

- Sending and receiving messages limited by the length of the network queues.
- Sending and receiving unlimited-length messages.
- Using interrupt-driven message retrieval.
- Sending and receiving by the LDR and RDR simultaneously.

Filename: LDR_test.c

```
/* LDR test program - PM program */
#include <cm/cmna.h>
#include "utils.h"

#define LONG_FACTOR 5

void main () {
    int input, result, high_node;
    printf("\nLDR test program, by William R. Swanson,\n");
    printf("Thinking Machines Corporation -- 2/3/92.\n\n");

    /* Enable broadcast sending */
    CMNA_participate_in(NI_BC_SEND_ENABLE);
    /* Abstain from broadcast reception and combine sending */
    save_and_set_abstain_flags(1,1,0,0);
    /* Start node programs running */
    node_main();

    /* Get a value from the user and send it to the nodes. */
    printf("This CM-5 partition has %d nodes.\n",
            CMNA_partition_size);
    printf("Please type an integer to send to the nodes: ");
    scanf("%d", &input);
    PM_send_to_NODE(0, input);
    printf("Sent value %d to node 0...\n",input);
    /* Wait for the nodes to finish juggling numbers */
    PM_NODE_synch();

    /* Get value from high node */
    high_node = CMNA_partition_size - 1;
    result = PM_get_from_NODE(high_node);
    printf("Short send:\n");
    printf("Received value %d (should be %d) from node %d.\n",
           result, input+MAX_BROADCAST_MSG_WORDS-1, high_node);
    result = PM_get_from_NODE(high_node);
    printf("Long send:\n");
    printf("Received value %d (should be %d) from node %d.\n",
           result, input+(MAX_BROADCAST_MSG_WORDS*
                          LONG_FACTOR)-1, high_node);
    result = PM_get_from_NODE(high_node);
    printf("Interrupt-driven send:\n");
    printf("Received value %d (should be %d) from node %d.\n",
           result, input+MAX_BROADCAST_MSG_WORDS-1, high_node);
```

```
    result = PM_get_from_NODE(0);
    printf("Dual-network send:\n");
    printf("Received value %d (should be %d) from node %d.\n",
           result, MAX_BROADCAST_MSG_WORDS, 0);
    restore_abstain_flags();
}
```

Filename: LDR_test.node.c

```
/* LDR test program - node program */
#define NI_ROUTER_DONE_P NI_ROUTER_DONE_COMPLETE_P
#include <cm/cmna.h>
#include <cm/cm_signal.h>
#include "utils.h"

#define LONG_FACTOR 5

/* Send/Receive functions limited by length restriction */
int LDR_send (dest_address, message, length, tag)
    unsigned dest_address, tag;
    int *message;
    int length;
{
  int i;
  CMNA_ldr_send_first(tag, length, dest_address);
  while (length--) CMNA_ldr_send_word(*message++);
  return (SEND_OK(CMNA_ldr_status())); }

int tag_limit=0;

int LDR_receive (message, length)
    int *message;
    int length;
{
  int i, tag = 999;
  /* Skip messages currently assigned as interrupts */
  while (tag>tag_limit) {
    if (RECEIVE_OK(CMNA_ldr_status()))
      tag = RECEIVE_TAG(CMNA_ldr_status());
  }
  while (length--)
    *message++ = CMNA_ldr_receive_word();
  return (tag);
}
```

```
/* Send/Receive function with no length restriction */
LDR_send_receive_msg(dest_address, message, length, tag, dest)
    unsigned dest_address, tag;
    int *message, *dest;
    int length;
{
    int packet_size=MAX_ROUTER_MSG_WORDS-1;
    int send_size, receive_size;
    int offset, source_offset=0, dest_offset;
    int words_to_send=length, words_received=0;
    int count, rec_tag, status;

 while ((words_received < length) || (words_to_send)) {

   /* First try to receive a packet */
   status=CMNA_ldr_status();
   if (words_received<length &&
       RECEIVE_OK(status) &&
       RECEIVE_TAG(status) <= tag_limit) {
     dest_offset = CMNA_ldr_receive_word();
     receive_size = RECEIVE_LENGTH_LEFT(CMNA_ldr_status());
     for (count=0; count<receive_size; count++)
       dest[dest_offset++] = CMNA_ldr_receive_word();
     words_received += receive_size;
     }

   /* Now try sending a packet */
   if (words_to_send) {
     send_size = ((words_to_send < packet_size) ?
                  words_to_send : packet_size);
     do {
       CMNA_ldr_send_first(tag, send_size + 1, dest_address);
       /* Send offset to indicate part of message being sent */
       CMNA_ldr_send_word(source_offset);
       offset=source_offset;
       for (count=0; count<send_size; count++)
     CMNA_ldr_send_word(message[offset++]);
     } while (!SEND_OK(CMNA_ldr_status()));
     source_offset=offset;
     words_to_send -= send_size;
   }
 }
}
```

```
/* Message-receiving handler for interrupt-driven LDR test */
int interrupt_done=0;
int interrupt_expect_length;
int interrupt_receive[MAX_BROADCAST_MSG_WORDS];

void LDR_receive_handler ()
{
    int temp=tag_limit;
    tag_limit=3;
    LDR_receive(interrupt_receive, interrupt_expect_length);
    tag_limit=temp;
    interrupt_done=1;
}

/* Send/Receive functions using LDR and RDR in tandem */
void LDR_RDR_send (dest_address, message, length, tag)
     unsigned dest_address, tag;
     int *message, length;
{
    int i;
    CMNA_ldr_send_first(tag, length, dest_address);
    CMNA_rdr_send_first(tag, length, dest_address);
    for (i=0; i<length; i++) {
         CMNA_ldr_send_word(message[i]);
         CMNA_rdr_send_word(message[i]);
  }
}

int LDR_RDR_receive (message, length)
     int *message, length;
{
  int i, ldr_value, rdr_value, length_received_ok=0;
  while (!RECEIVE_OK(CMNA_ldr_status()) ||
     !RECEIVE_OK(CMNA_rdr_status())) {}
  for (i=0; i<length; i++) {
    ldr_value=CMNA_ldr_receive_word();
    rdr_value=CMNA_rdr_receive_word();
    if (ldr_value==rdr_value) {
      message[i]=ldr_value;
      length_received_ok++;
    }
  }
  return(length_received_ok);
}
```

```
/* Combine "network-done" Function */
void network_done_synch()
{
 CMNA_com_send_first(ASSERT_ROUTER_DONE,SCAN_ROUTER_DONE,1, 0);
 while (!DR_ROUTER_DONE(CMNA_dr_status())) {};
}

/* Tool to ensure there's nothing in the receive queues */
/* Not used here, but you may find it handy */
void LDR_empty_network() {
  int status, length, i;
  while (status=CMNA_ldr_status(), RECEIVE_OK(status))
    if (RECEIVE_TAG(status) <= tag_limit) {
      length = RECEIVE_LENGTH(status);
      for (i=0; i<length; i++)
    (void) CMNA_ldr_receive_word();
    }
}

void CMPE_node_main () {
    int value=0, i, length=MAX_BROADCAST_MSG_WORDS;
    int long_length=length*LONG_FACTOR;
    int next_node, mirror_node;
    int received_ok;
    int  send[MAX_BROADCAST_MSG_WORDS*LONG_FACTOR],
         receive[MAX_BROADCAST_MSG_WORDS],
         long_receive[MAX_BROADCAST_MSG_WORDS*LONG_FACTOR],
         dual_receive[MAX_BROADCAST_MSG_WORDS];

  /* signal interrupts for non-zero tag values */
  CMOS_signal( SIGMSG , LDR_receive_handler , 14 );
  CMNA_participate_in(NI_BC_SEND_ENABLE);
  save_and_set_abstain_flags(0,0,0,0);

  /* All nodes get the value sent by the PM... */
  All_NODES_get_from_PM(&value);

  for( i=0; i<long_length; i++) {
    send[i]=value+i;
    long_receive[i]=-999;
  }
  for( i=0; i<length; i++) {
    receive[i]=-999;
    interrupt_receive[i]=-999;
    dual_receive[i]=-999;
  }
```

```
    /* Calculate some useful addresses */
    next_node = (CMNA_self_address + 1) % CMNA_partition_size;
    mirror_node = (CMNA_partition_size-1) - CMNA_self_address;

    /* Do an ordinary, length-limited send */
    LDR_send(next_node, send, length, 0);
    network_done_synch();
    LDR_receive(receive, length);
    network_done_synch();

    /* Do an unlimited-length send */
    LDR_send_receive_msg(mirror_node, send,
                         long_length, 0, long_receive);
    network_done_synch();

    /* Do an interrupt-driven send with a tag of 3*/
    interrupt_expect_length=length;
    LDR_send(next_node, send, length,3);
    while (!interrupt_done) {}
    network_done_synch();

    /* Send via both LDR and RDR, and check results */
    LDR_RDR_send (mirror_node, send, length, 0);
    network_done_synch();
    received_ok=LDR_RDR_receive (dual_receive, length);

    /* Signal to PM that answer is ready */
    PM_NODE_synch();

    /* Send check values back to PM */
    NODE_send_to_PM(receive[length-1]);
    NODE_send_to_PM(long_receive[long_length-1]);
    NODE_send_to_PM(interrupt_receive[length-1]);
    NODE_send_to_PM(received_ok);

    restore_abstain_flags();
}
```

C.2 Data Network Double-Word Messages Test

This program demonstrates the use of double-word read and write operations for Data Network transmissions:

Filename: `dbl_test.c`

```
/* Double-word ops test program - PM program */
#include <cm/cmna.h>
#include "utils.h"

#define LONG_FACTOR 5

void main () {
    int input, result, high_node;
    printf("\nDouble-word test program, by W. R. Swanson,\n");
    printf("Thinking Machines Corporation -- 2/3/92.\n\n");

    /* Enable broadcast sending */
    CMNA_participate_in(NI_BC_SEND_ENABLE);
    /* Abstain from broadcast reception and combine sending */
    save_and_set_abstain_flags(1,1,0,0);
    /* Start node programs running */
    node_main();

    /* Get a value from the user and send it to the nodes. */
    printf("This CM-5 partition has %d nodes.\n",
              CMNA_partition_size);
    printf("Please type an integer to send to the nodes: ");
    scanf("%d", &input);
    PM_send_to_NODE(0, input);
    printf("Sent value %d to node 0...\n",input);
    /* Wait for the nodes to finish juggling numbers */
    PM_NODE_synch();

    /* Get value from high node */
    high_node = CMNA_partition_size - 1;
    result = PM_get_from_NODE(high_node);
    printf("Long send using double-word ops:\n");
    printf("Received value %d (should be %d) from node %d.\n",
           result, input+(MAX_BROADCAST_MSG_WORDS*
                          LONG_FACTOR)-1, high_node);
    restore_abstain_flags();
}
```

Filename: dbl_test.node.c

```
/* Double-word ops test program - PM program */
#include <cm/cmna.h>
#include <cm/cm_signal.h>
#include "utils.h"
#define LONG_FACTOR 5
int tag_limit = 3;

/* Send/Receive function using double-words */
LDR_send_receive_msg_double(dest_address, message,
                            length, tag, dest)
     unsigned dest_address, tag;
     int *message, *dest;
     int length;
{
 int packet_size;
 double *dbl;
 int send_size, send_size2, receive_size, receive_size2;
 int offset, source_offset=0, dest_offset;
 int words_to_send=length, words_received=0;
 int count, rec_tag, status;

 if ((int)message & 3)
 CMPN_panic("Error: Message array not double-word aligned!");

 if ((int)dest & 3)
 CMPN_panic("Error: Dest array not double-word aligned!");

 packet_size = (MAX_ROUTER_MSG_WORDS-1) & ~1;

 while ((words_received < length) || (words_to_send)) {

   /* First try to receive a packet */
   status=CMNA_ldr_status();
   if (words_received<length &&
       RECEIVE_OK(status) &&
       RECEIVE_TAG(status) <= tag_limit) {
     dest_offset = CMNA_ldr_receive_word();
     receive_size = RECEIVE_LENGTH_LEFT(CMNA_ldr_status());
     printf("received offset %d, size %d.\n",
              dest_offset, receive_size);
```

```
      for (count=0; count<(receive_size>>1); count++) {
        dbl = (double *)(&dest[dest_offset++]);
        dest_offset++;
        *dbl = CMNA_ldr_receive_double();
        dbl++;
      }
      if (receive_size & 1) /* If word left over */
        dest[dest_offset++] = CMNA_ldr_receive_word();
      words_received += receive_size;
    }

    /* Now try sending a packet */
    if (words_to_send) {
      send_size = ((words_to_send < packet_size) ?
                   words_to_send : packet_size);
      send_size2 = send_size >> 1;
      do {
        CMNA_ldr_send_first(tag, send_size + 1, dest_address);
        CMNA_ldr_send_word(source_offset);
        offset=source_offset;
        /* Send as many doubles as possible */
        for (count=0; count<send_size2; count++){
      dbl = (double *)(&message[offset++]);
      offset++;
      CMNA_ldr_send_double(*dbl++);
        }
        if (send_size & 1) /* If a word is left over */
      CMNA_ldr_send_word(message[offset++]);
      } while (!SEND_OK(CMNA_ldr_status()));
      printf("sent offset %d, size %d.\n",
               source_offset, send_size);
      source_offset=offset;
      words_to_send -= send_size;
    }
  }
}

/* Combine "network-done" Function */
void network_done_synch()
{
 CMNA_com_send_first(ASSERT_ROUTER_DONE,SCAN_ROUTER_DONE,1,0);
  while (!DR_ROUTER_DONE(CMNA_dr_status())) {};
}
```

```
void CMPE_node_main () {
  int value=0, i;
  int length=MAX_BROADCAST_MSG_WORDS*LONG_FACTOR;
  int mirror_node;

  /* These variabes MUST be double-word aligned! */
  double temp_dalign_send;
  int send[MAX_BROADCAST_MSG_WORDS*LONG_FACTOR];
  double temp_dalign_rec;
  int receive[MAX_BROADCAST_MSG_WORDS*LONG_FACTOR];

  CMNA_participate_in(NI_BC_SEND_ENABLE);
  save_and_set_abstain_flags(0,0,0,0);

  /* All nodes get the value sent by the PM... */
  All_NODES_get_from_PM(&value);

  for( i=0; i<length; i++) {
    send[i]=value+i;
    receive[i]=-999;
  }

  mirror_node = (CMNA_partition_size-1) - CMNA_self_address;

  /* Do an unlimited-length send using double-word ops */
 LDR_send_receive_msg_double(mirror_node, send,
                             length, 0, receive);
  network_done_synch();

  /* Signal to PM that answer is ready */
  PM_NODE_synch();

  /* Send check value back to PM */
  NODE_send_to_PM(receive[length-1]);

  restore_abstain_flags();
}
```

C.3 Broadcast Network Test

This program presents a simple test of broadcast network transmission:

Filename: `BC_test.c`

```
/* Broadcast examples program - PM program */
#include <cm/cmna.h>
#include "utils.h"

void main () {
    int input, result, high_node;
    printf("\nBroadcast test program, by W. R. Swanson,\n");
    printf("Thinking Machines Corporation -- 2/1/92.\n\n");

    /* Enable broadcast sending */
    CMNA_participate_in(NI_BC_SEND_ENABLE);
    /* Abstain from broadcast reception and combine sending */
    save_and_set_abstain_flags(1,1,0,0);
    /* Start node programs running */
    node_main();

    /* Get a value from the user and send it to the nodes. */
    printf("This CM-5 partition has %d nodes.\n",
          CMNA_partition_size);
    printf("Please type an integer to send to the nodes: ");
    scanf("%d", &input);

    PM_send_to_NODE(0, input);
    printf("Sent value %d to node 0...\n",input);

    /* Wait for the nodes to finish juggling numbers */
    PM_NODE_synch();

    /* Get value from high node */
    high_node = CMNA_partition_size - 1;
    result = PM_get_from_NODE(high_node);
    printf("Received value %d (should be %d) from node %d.\n",
           result, input+MAX_BROADCAST_MSG_WORDS-1, high_node);

    restore_abstain_flags();
}
```

Filename: BC_test.node.c

```
/* Broadcast examples program - node program */
#include <cm/cmna.h>
#include "utils.h"

int BC_send(message, length)
     int *message, length;
{
  int i;
  CMNA_bc_send_first(length--, *message++);
  for (i=0; i<length; i++) CMNA_bc_send_word(*message++);
  return(SEND_OK(CMNA_bc_status()));
}

int BC_receive(message, length)
     int *message, length;
{
  int i;
  for(i=0; i<length; i++) {
    while(!RECEIVE_OK(CMNA_bc_status())) {}
    message[i] = CMNA_bc_receive_word();
  }
  return(length);
}

void CMPE_node_main () {
    int value=0, i, length=MAX_BROADCAST_MSG_WORDS;
    int send[MAX_BROADCAST_MSG_WORDS],
        receive[MAX_BROADCAST_MSG_WORDS];
    int status, rec_length;

    CMNA_participate_in(NI_BC_SEND_ENABLE);
    save_and_set_abstain_flags(0,0,0,0);

    /* Node 0 gets the value sent by the PM... */
    NODE_get_from_PM(&value);

    for( i=0; i<length; i++) {
      send[i]=value+i;
      receive[i]=-999;
    }
```

```
if (CMNA_self_address==0) {
  status=0;
  while(!status) status = BC_send(send, length);
}
rec_length = BC_receive(receive);

/* Signal to PM that answer is ready */
PM_NODE_synch();

/* Send value from high-order node back to PM */
NODE_send_to_PM(receive[length-1]);

restore_abstain_flags();
}
```

C.4 Combine Network Test

This program presents examples of a number of different kinds of combine network operations, including:

- Scanning messages, with and without segments
- Reduction messages
- Network-done messages

Filename: `COM_test.c`

```
/* Combine examples program - PM program */
#include <cm/cmna.h>
#include "utils.h"

void main () {
    int input, result, segment_size, high_node, i, expected;
    printf("\nCombine test program, by William R. Swanson,\n");
    printf("Thinking Machines Corporation -- 2/1/92.\n\n");

    /* Enable broadcast sending */
    CMNA_participate_in(NI_BC_SEND_ENABLE);

    /* Abstain from broadcast reception and combine sending */
    /* Abstain from combine reception, too, for a while... */
    save_and_set_abstain_flags(1,1,1,0);
```

```
    /* Start node programs running */
    node_main();

    /* Get a value from the user and send it to the nodes. */
    printf("This CM-5 partition has %d nodes.\n",
           CMNA_partition_size);
    printf("Please type a positive integer: ");
    scanf("%d", &input);

    high_node = CMNA_partition_size-1;
    PM_send_to_NODE(high_node, input);
    printf("Sent value %d to node %d...\n", input, high_node);

    /* Wait for the nodes to finish juggling numbers */
    PM_NODE_synch();
    /* Turn combine reception back on */
    CMNA_write_rec_abstain_flag(com_control_reg, 0);

    /* Get check values */
    result = PM_get_from_NODE(0);
    printf("Received value %d (should be %d) from node %d.\n",
           result, (input+MAX_BROADCAST_MSG_WORDS-1), 0);

    result = PM_get_from_NODE(high_node);
    printf("Received value %d (should be %d) from node %d.\n",
           result, (input*high_node), high_node);

    segment_size = PM_get_from_NODE(0);
    result = PM_get_from_NODE(0);
    printf("Received value %d (should be %d) from node %d.\n",
           result, (input+MAX_BROADCAST_MSG_WORDS-1)
                   * (segment_size-1), 0);

    result = PM_get_from_NODE(0);
    printf("Network done for node 0 got %d (should be %d).\n",
           result, high_node);

    result = PM_get_from_NODE(0);
    printf("Scanning counted %d nodes (should be %d).\n",
           result, CMNA_partition_size);

    restore_abstain_flags();
}
```

Filename: COM_test.node.c

```
/* Combine examples program - node program */
#define NI_ROUTER_DONE_P NI_ROUTER_DONE_COMPLETE_P
#include <cm/cmna.h>
#include "utils.h"
int COM_send(combiner, pattern, message, length)
     int *message, combiner, pattern, length;
{
  int i, start, step;
  /* For max scans, send high-order word(s) first */
  if (combiner==MAX_SCAN) { start=length-1; step=-1; }
  else { start=0; step=1; }
  CMNA_com_send_first(combiner, pattern, length,
                      message[start]);
  for (i=1; i<length; i++)
    CMNA_com_send_word(message[(start+=step)]);
  return(SEND_OK(CMNA_com_status()));
}

int COM_receive(combiner, message)
     int *message;
{
  int i, length, start, step;
  while(!RECEIVE_OK(CMNA_com_status())) {}
  length=RECEIVE_LENGTH(CMNA_com_status());
  /* For max scans, send high-order word(s) first */
  if (combiner==MAX_SCAN) { start=length-1; step=-1; }
  else { start=0; step=1; }
  for(i=0; i<length; i++) {
    message[start] = CMNA_com_receive_word();
    start+=step;
  }
  return(length);
}

int COM_scan(combiner, pattern, message, length, result)
     int *message, *result, combiner, pattern, length;
{
  int status=0, rec_length;
  while (!status) status =
    COM_send(combiner, pattern, message, length);
  rec_length = COM_receive(combiner, result);
  return(rec_length);
}
```

```
void CMPE_node_main () {
    int value=0, i, length=MAX_BROADCAST_MSG_WORDS;
    int send[MAX_BROADCAST_MSG_WORDS],
        result[MAX_BROADCAST_MSG_WORDS],
        seg_result[MAX_BROADCAST_MSG_WORDS];
    int rec_length, segment_size, high_node;
    int one, node_count;
    int message, network_done_msg, next_processor;

    CMNA_participate_in(NI_BC_SEND_ENABLE);
    save_and_set_abstain_flags(0,0,0,0);
    /* Make sure segmenting is turned off to begin with */
    CMNA_set_segment_start(0);
    high_node = CMNA_partition_size - 1;

    /* High node gets the value sent by the PM... */
    NODE_get_from_PM(&value);

    /* Fill send array based on supplied value */
    for( i=0; i<length; i++) {
      send[i]=((CMNA_self_address==high_node) ? value+i : 0);
      result[i]=-999;
      seg_result[i]=-999;
    }

    /* Do a max scan to distribute send values to all nodes */
    rec_length = COM_scan(MAX_SCAN, SCAN_BACKWARD, send,
                          length, send);

    /* Scan overwrites high node -- put back original value */
    if (CMNA_self_address==high_node)
      for(i=0; i<length; i++) send[i] = value+i;

    /* Do an add scan to make different values */
    rec_length = COM_scan(ADD_SCAN, SCAN_FORWARD, send,
                          length, result);

    /* Do a backwards segmented reduction */
    segment_size=(CMNA_partition_size<5 ?
                  CMNA_partition_size : 5);
    CMNA_set_segment_start(((CMNA_self_address % segment_size)
                    == segment_size-1));
    rec_length = COM_scan(MAX_SCAN, SCAN_BACKWARD, result,
                          length, seg_result);
    CMNA_set_segment_start(0);
```

```
/* Try network-done feature */
message=CMNA_self_address;
network_done_msg=0;
next_processor = (CMNA_self_address+1)%CMNA_partition_size;
CMNA_ldr_send_first(0,1,next_processor);
CMNA_ldr_send_word(message);

COM_send(ASSERT_ROUTER_DONE, SCAN_ROUTER_DONE,
        &network_done_msg, 1);

while (!DR_ROUTER_DONE(CMNA_dr_status())) {};
while (!RECEIVE_OK(CMNA_ldr_status())) {};

message=CMNA_ldr_receive_word();

/* Use reduction to do a processor "roll-call" */
one=1;
node_count=-999;
rec_length = COM_scan(ADD_SCAN, SCAN_REDUCE,
                      &one, 1, &node_count);

/* Signal to PM that answers are ready */
PM_NODE_synch();

/* Send check values back to PM */
NODE_send_to_PM(send[length-1]);
NODE_send_to_PM(result[0]);
NODE_send_to_PM(segment_size);
NODE_send_to_PM(seg_result[length-1]);
NODE_send_to_PM(message);
NODE_send_to_PM(node_count);

restore_abstain_flags();
}
```

C.5 Global Network Test

This program presents a quick example of asynchronous and synchronous global network transmission:

Filename: GLOBAL_test.c

```
/* Global network test program - node program */
#include <cm/cmna.h>
#include "utils.h"

void main () {
    int value;
    printf("\nGlobal test program, by William R. Swanson,\n");
    printf("Thinking Machines Corporation -- 2/6/92.\n\n");

    /* Enable broadcast sending */
    CMNA_participate_in(NI_BC_SEND_ENABLE);
    /* Abstain from broadcast reception and combine sending */
    save_and_set_abstain_flags(1,1,0,0);

    printf("This CM-5 partition has %d nodes.\n",
           CMNA_partition_size);

    /* Start node programs running */
    printf("Starting node programs...\n");
    node_main();

    /* Test asynchronous global network */
    CMNA_or_global_async_bit(0);

    PM_NODE_synch();

    value = CMNA_global_async_read();
    printf("Received async bit %d (should be 0).\n", value);

    restore_abstain_flags();
}
```

Filename: GLOBAL_test.node.c

```
/* Global network test program - node program */
#include <cm/cmna.h>
#include "utils.h"

void CMPE_node_main () {
        int value;
    CMNA_participate_in(NI_BC_SEND_ENABLE);
    save_and_set_abstain_flags(0,0,0,0);

    CMNA_or_global_async_bit(0);

    /* Signal to PM that answer is ready */
    PM_NODE_synch();

    value = CMNA_global_async_read();

    if (value)
         printf("Error: node got non-zero global value.");

    restore_abstain_flags();
    }
```

Indexes

Language Index

This index lists the macros, system functions, and constants referred to in this document.

C, cont.

D

M

N

N, cont.

O

R

S

U, X

Concept Index

This index lists the major terms and topics found in this document.

C, cont.

D

E

F

G

I

L

M

N

N, cont

O

P

Q

R

R, cont.

S

T

W