The
Connection Machine
System

Programming the NI

Version 7.1
February 1993

Thinking Machines Corporation
Cambridge, Massachusetts

First printing, March 1992
Revised, February 1993

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

Contents

Chapter 2 A Generic Network Interface, cont'd

Chapter 3 The Data Network 21

Appendixes

Indexes

List of Figures

About This Manual

Objectives of This Manual

Programming the NI describes the Network Interface (NI) chip of the Connection Machine CM-5 system. The NI is the component of the CM-5 hardware that manages the machine's internal communications networks.

This manual describes the NI at a level sufficient for applications programmers to write simple programs that directly access the NI chip.

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 5. 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.

Also note that this manual does note cover supervisor-level details of the NI

WARNING: Code that directly accesses the NI chip *will not be supported* by future hardware releases. This manual describes the current version of the NI, but future implementations are free to change any of the NI features described here.

Thus, it is strongly recommended that you use the CMMD software library for code that directly accesses the CM-5 networks.

Revision Information

This is an updated and revised version of the original Programming the NI manual (March 1992).

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 interfaces and features of the Data Network.

Chapter 4 The Control Network
The register interfaces and features of the Control Network, including the broadcast, combine, and global interfaces.

Chapter 5 Writing NI Programs
A brief overview of the process of writing, compiling, and running an NI program.

Chapter 6 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*, November 1992
- *CMMD User's Guide*, Version 2.0, November 1992
- *CMMD Reference Manual*, Version 2.0, November 1992

Notation Conventions

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

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

Customer Support

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

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

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

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

Chapter 1

The Network Interface

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 processing nodes linked together by two main internal networks, the *Data Network* and the *Control Network.*

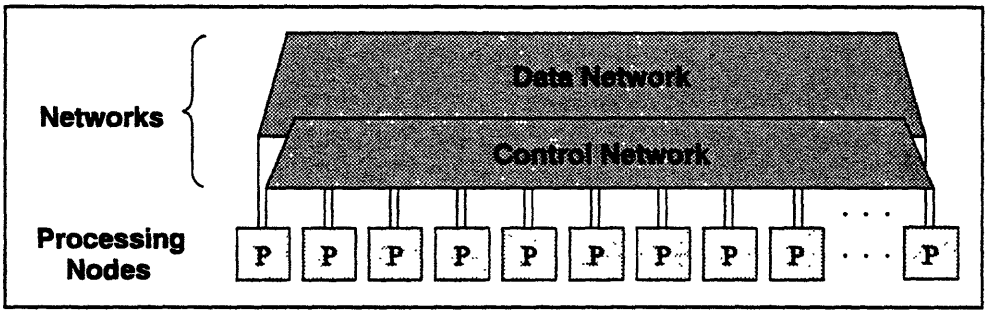

Figure 1. The CM-5 system: Processing nodes linked by Data and Control Networks.

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

1.1.1 The CM-5 Networks

The 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 halves, the *left interface* and the *right interface*, both of which are connected to all processing nodes. The left and right interfaces can be used either independently or together as the combined *Data Network.*

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

The Control Network

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

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

For the Curious: The Diagnostic Network

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

1.1.2 Processing Nodes

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

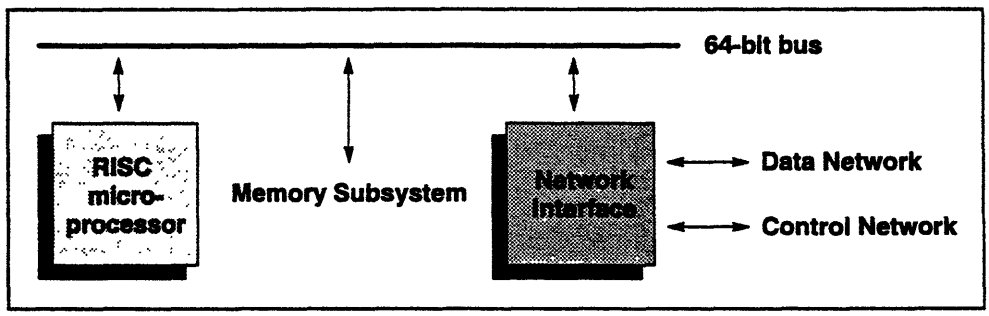

Figure 2. The components of a typical processing node.

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).

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):

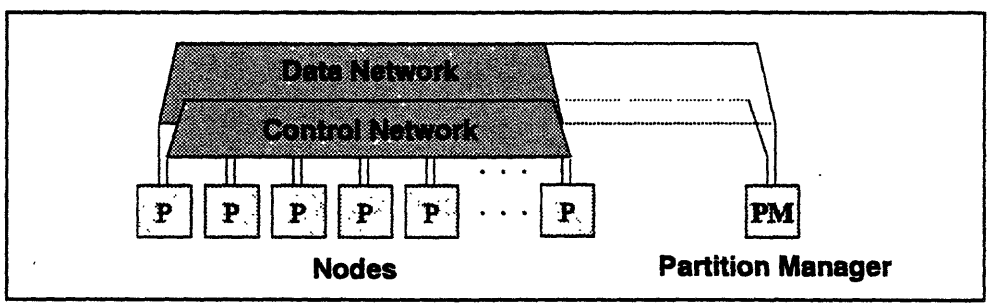

Figure 3. A partition of nodes and its partition manager.

The PM is also attached to the Data and Control Networks, and it can communicate with the processing nodes by sending and receiving messages via its own NI chip. Programs written for the CM-5 normally include two separate files of code, one for the PM and one for the nodes.

CM-5 Programming Style

The PM and the nodes typically operate in a data parallel style: the nodes execute identical programs simultaneously, and the PM controls which function the nodes will execute next. (For more information on program structure, see Chapter 5.)

1.2 The NI Chip

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

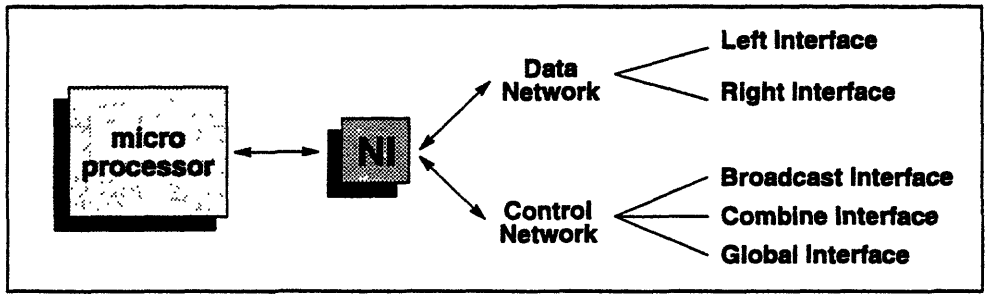

Figure 4. NI provides access to features of the Data Network and Control Network.

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

1.3 The NI Registers

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

The microprocessor can either examine the registers of the NI chip to see whether a message has arrived, or it can instruct the NI to signal an interrupt when a message arrives. Control of the NI is therefore based on register operations and interrupts.

1.3.1 NI Register Types

There are three basic types of NI registers:

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

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

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

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 2.2.2.)

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 6.1.1 for details on the time taken to read and write NI registers.

1.3.2 NI Register and Field Names

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

`ni_`*`interface_purpose`*

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

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

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

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

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

1.4 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.

Finding the C Macro You Need

The predefined C macros typically have names based on the registers and fields they manipulate. For example, most network interfaces have an NI register named `ni_`*interface*`_status` that contains the `ni_`*interface*`_send_ok` and `ni_`*network*`_rec_ok` status 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 interfaces that have a `ni_`*interface*`_status` register.

Note: To get access to these predefined macros, your program must `#include` the header file `cmna.h`. (See Chapter 5 for more information.)

1.5 Using This Manual Effectively

The first few chapters of this manual are mostly 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 both of the network-specific chapters (3 and 4), before turning to the complete sample program presented in Chapter 5.

Experienced CM-5 programmers should read through Chapter 2 and skim chapters 3 and 4 to get a sense of how the networks operate, and then proceed to the sample program in Chapter 5 to see how NI programs are structured.

Whatever your level of experience, Chapter 6 presents a number of important performance strategies and potential sources of programming errors that you should know about.

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

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

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

Similarly, *receiving* a message involves reading its values from the receive FIFO register of the network interface.

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

2.2 Sending a Message

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

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

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

- The first value of a message must be written to the `send_first` FIFO register. This tells the NI that a message is being composed, and also specifies the message's auxiliary information (see Section 2.2.2 below).
- The remaining values (if any) must be written to the `send` FIFO register.

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

2.2.1 Message Discarding

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

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

Whatever the reason, when a message is discarded, it is *completely* discarded. Any previously written values for that message are removed from the send FIFO, and a new message can be started by writing a value to the `send_first` register.

Performance Note: You can use message discarding to your advantage and thereby make your code more efficient. (See Section 6.1.3.)

2.2.2 Auxiliary Information

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

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

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

2.2.3 Writing a Message

For each *interface*, there are two `send_first` macros,

`CMNA_`*interface*`_send_first` (*auxiliary–info*, *value*)
`CMNA_`*interface*`_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 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 doubleword register operations to make your code more efficient. (See Sec-

tion 6.1.2 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 *interface* there are three macros that write values to the **send** register, one for each of the three data types you can send:

CMNA_*interface***_send_word** (*value*)
CMNA_*interface***_send_float** (*value*)
CMNA_*interface***_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.

2.3 Receiving a Message

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

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

A message is received by reading its value(s) in order from the **recv** register, one at a time. There are three network-specific message-reading macros, one for each network *interface*:

```
int value = CMNA_interface_receive_word();
int value = CMNA_interface_receive_float();
int value = CMNA_interface_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.

When a message arrives in the receive FIFO, the NI sets the **rec_ok** flag in the **status** register (see Section 2.4). You can repeatedly test the **rec_ok** flag to determine whether a message has arrived (for example, in a top-level loop).

2.4 The Status Register

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

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

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

2.4.1 Status Fields for Message Sending

The "Send OK" Flag

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

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

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

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

Programming Note: NI programmers typically write an entire message to the send FIFO and only *then* check the **send_ok** flag to see whether it was accepted. (See Section 6.1.3 for more information.) For this reason, the **send_space** field and **send_empty** flag typically aren't used by NI programmers.

2.4.2 Status Fields for Message Receiving

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

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

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

2.4.3 Reading the Status Register Fields

The general method for reading the value of an **ni_***interface***_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_interface_status()
```

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

SEND_OK (*status*)	Gets **send_ok** flag from *status* value.
SEND_SPACE (*status*)	Gets **send_space** field.
SEND_EMPTY (*status*)	Gets **send_empty** flag.
RECEIVE_OK (*status*)	Gets **rec_ok** flag.
RECEIVE_LENGTH (*status*)	Gets **rec_length** field.
RECEIVE_LENGTH_LEFT (*status*)	Gets **rec_length_left** field.

For example, to get the three **send** fields from the broadcast interface status register, you could use the following C code:

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

And to get the **rec** fields from the right data interface status register, you could use the following code:

```
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);
```

2.5 Abstaining from an Interface — The Control Register

Each of the Control Network interfaces has a control register containing at least one *abstain flag*, a flag that you can set to cause a node to ignore the transactions of the network. The control register and abstain flag(s) typically have names like:

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

Note: The global interface, always the exception, uses a different name for the control register. See Section 4.3.2 for more information.

2.5.1 Effect of Abstain Flags

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

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

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

2.5.2 Combine Interface Abstain Flags

The **ni_reduce_rec_abstain** flag is only defined for the combine interface, and only applies to reduction operations. In addition, reduction operations treat the value of the **rec_abstain** flag differently from all other interface operations. For more information, see Section 4.2.9.

2.5.3 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.

2.5.4 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 Section 4.3 for a discussion of the global interface 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.

2.5.5 Being a Good Neighbor

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

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

2.6 Using a Generic Network Interface

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

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

2.7 From the Generic to the Specific

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

The next four chapters present a description of the Data Network interfaces and the three Control Network interfaces. 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 halves, the *left interface* (LDR) and *right interface* (RDR). Each half of the network is connected to all nodes, and can be used independently. The two halves of the network can also be accessed together as the single *Data Network* (DR):

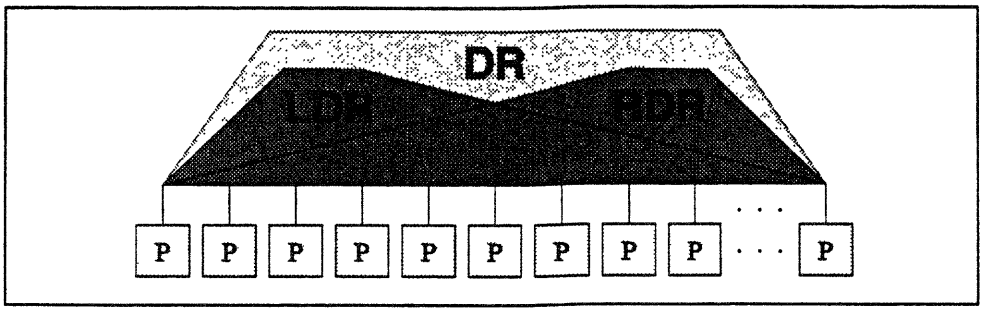

Figure 5. The three interfaces of the Data Network: DR, LDR, and RDR.

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

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

3.1 The Data Network Register Interfaces

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

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

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

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

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

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

`dr` — Data Network `ldr` — left interface `rdr` — right interface

3.2 Data Network Messages

The Data Network is essentially asynchronous in operation — nodes can send and receive messages freely, so long as enough nodes are receiving messages so that the network does not become clogged (see Section 3.6).

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

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

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

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

3.3 Data Network Addressing

The Data Network 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. (See Figure 6.)

Note: The partition manager is always located at an address outside the partition, and so does not occupy any of the relative addresses of the partition. (For more information, see Section 5.1.)

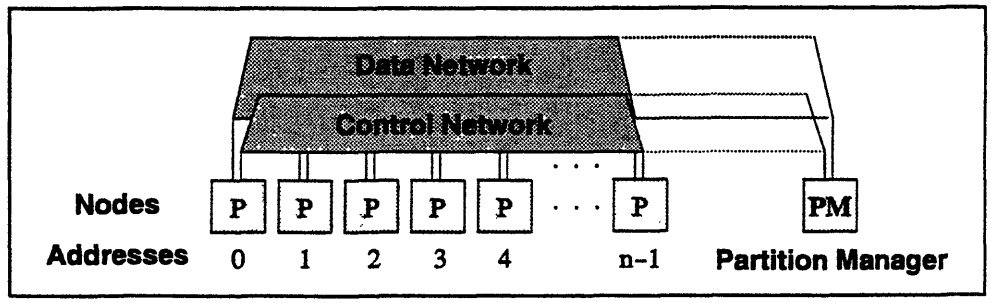

Figure 6. Addressing of nodes in a partition.

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.

3.4 Sending and Receiving Messages

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

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

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

The auxiliary information of the message consists of the length of the message in words (excluding the address word), and a four-bit tag value. See Section 3.5.1 for information on the use of tag values.

3.4.1 Sending Messages

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

ni_*dinterface*_**send_first**	Used for first value of a message.
ni_*dinterface*_**send**	Used for the rest of the message.

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

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

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

CMNA_*dinterface*_**send_first**(*tag, length, value*)
CMNA_*dinterface*_**send_first_double**(*tag, length, value*)

CMNA_*dinterface*_**send_word**(*value*)
CMNA_*dinterface*_**send_float**(*value*)
CMNA_*dinterface*_**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 7. All other tags are reserved for supervisor use.

3.4.2 Receiving Messages

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

ni_*dinterface*_**recv**	FIFO register from which values are read.

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

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

```
value = CMNA_dinterface_receive_word();
value = CMNA_dinterface_receive_float();
value = CMNA_dinterface_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.

3.5 The Status Register

The status register for each of the Data Network interfaces contains the following sub-fields:

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

The macros used to get the **ni_*interface*_status** for each network interface 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**, **send_space**, **rec_ok** and **rec_length** flags and fields for each network by using the field extractors described in Chapter 2 (Section 2.4.3).

The remaining flags and fields are described in the sections below.

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 the section below on message tags.)

Implementation Note: The `ni_dr_send_state` and `ni_dr_rec_state` fields, as well as the flag `ni_router_done_complete`, are intended to apply to all three interfaces at once, and thus are only accessible from the DR interface (that is, they are only defined for the `ni_dr_status` register).

3.5.1 Message Tags

The tag values of Data Network messages are used to distinguish between different types of Data Network messages. The `status` register field `rec_tag` always contains the tag value that was sent with the current message. To get the `rec_tag` field, use the macro:

`RECEIVE_TAG` (*status*)

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

IMPORTANT — Check the Tag before Receiving a Message

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

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

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

3.5.2 Message Tags and Interrupts

Tag values can be used to trigger interrupts; when a message with an interrupting tag value becomes available for reading in the receive FIFO, the NI signals an interrupt to the microprocessor. Tag value interrupts can be used to cause the microprocessor to execute a specific section of code.

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

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

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

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

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

So, for example, the function call

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

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

Note: To use this function, you must **`#include`** the file **`cmsys/cm_signal.h`**. For more information on **`CMOS_signal`**, see the UNIX manual page for the function. (This is included as an appendix to this document.)

3.5.3 The Send and Receive State Fields

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

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

You can use the **ni_dr_send_state** and **ni_dr_rec_state** fields to determine which interfaces are in use.

ni_dr_send_state is an integer from 0 to 2, with the following meanings:

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

ni_dr_rec_state is also an integer from 0 to 2:

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

You can obtain the values of these fields by using the following macros:

DR_SEND_STATE (*status*)
DR_RECEIVE_STATE (*status*)

For example:

```
int value = CMNA_LDR_status();
int send_state = DR_SEND_STATE(value);
int rec_state  = DR_RECEIVE_STATE(value);
```

Implementation Note: The **ni_dr_send_state** and **ni_dr_rec_state** fields exist only for the DR interface (that is, are only accessible from the **ni_dr_status** register).

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

3.5.4 The Network-Done Flag

The `ni_router_done_complete` flag is used by the Control Network as part of its network-done message function. This feature is designed to make it easy to synchronize the nodes after a Data Network operation. For more information, see Section 4.2.8.

You can use the following macro to access this flag:

DR_ROUTER_DONE (*status*)

For example:

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

3.6 Data Network Usage Note: Receive before You Send

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

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

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

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

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

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

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

3.7 Examples

The examples shown below are code fragments intended to be run on the processing nodes. See Chapter 5 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, receive);
```

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

You use **CMOS_signal** to inform the NI that it should signal an interrupt from some or all of the possible tag values. (Remember that you must **#include** the header file **cmsys/cm_signal** to have access to **CMOS_signal**.) 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-interfaces 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 interfaces, comparing the received results to make sure that the message was received properly:

```
/* Send/Receive functions using LDR/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 Control Network

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

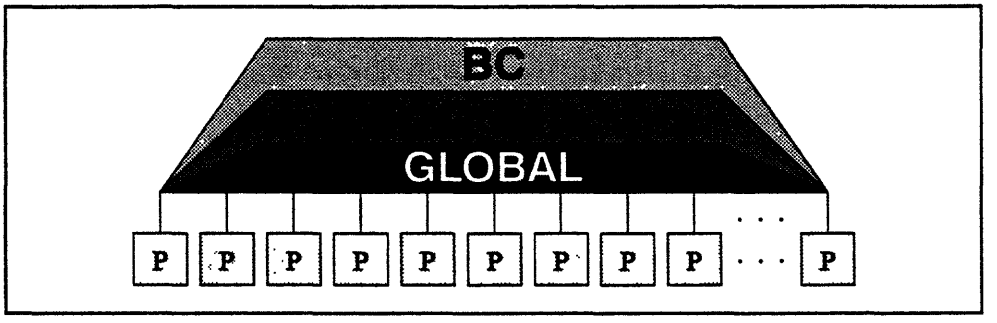

Figure 7. The three interfaces of the Control Network: BC, COM, and global.

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

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

4.1 The Broadcast Interface

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

Implementation Note: Because of the way the broadcast and combine interfaces interact, if a node is abstaining from a combine operation, that node should *not* execute a broadcast operation until the combine operation is completed. (For more information, see Section 6.2.6.)

4.1.1 Broadcast Register Interface

The broadcast register interface is based on the generic model presented in Chapter 2. The following NI registers form the broadcast interface:

`ni_bc_send_first`	Used to send the first value of a message.
`ni_bc_send`	Used to send the rest of the message.
`ni_bc_recv`	Used to receive a message.
`ni_bc_status`	Status register.
`ni_bc_control`	Control register.

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

4.1.2 Broadcast Messages

A broadcast message is essentially synchronous — a single node broadcasts a message that is received by all nodes in its partition (including the broadcasting node itself). Only one node in each partition can broadcast by a given interface at any time. If two or more nodes in the same partition attempt to broadcast simultaneously the effect is unpredictable. An error may be signaled and/or transmitted data may be lost.

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

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

Although the length of a broadcast message is not maintained, the *order* of the words within a message *is* maintained. Also, while messages can be combined and fragmented, the order in which entire messages are sent and received is unaltered.

4.1.3 Sending Broadcast Messages

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

Programming Note: The length limit of the broadcast send FIFO is given by the constant `MAX_BROADCAST_MSG_WORDS` (currently 4).

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

`ni_bc_send_first`	Used to send the first value of a message.
`ni_bc_send`	Used to send 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: 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* include the following macro call prior to any broadcast interface operations:

```
CMNA_participate_in(NI_BC_SEND_ENABLE);
```

4.1.4 Receiving Broadcast Messages

Broadcast messages are received as described in Chapter 2. The following register is used to receive messages:

`ni_bc_recv`	FIFO register from which values are read.

To receive a message from the broadcast interface, 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();
```

4.1.5 The Broadcast Status Register

The broadcast status register contains the following sub-fields:

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

The meanings of these sub-fields are as described in Chapter 2. You can obtain the values of these sub-fields by using the generic field extractors described in Chapter 2 (Section 2.4.3).

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

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

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

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

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

4.1.6 Abstaining from the Broadcast Interface

The broadcast interface has an abstain flag that you can use to cause the NI to ignore incoming broadcast messages. The abstain flag's effects and use are as described in Section 2.5.

ni_bc_control	Status register, contains **rec_abstain** field.
ni_rec_abstain	Flag, broadcast interface abstain flag.

The address constant for the abstain register is **bc_control_reg**. You can use the macros described in Section 2.5.3 to read and write the abstain flag:

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

4.1.7 Broadcast Interface Examples

The examples shown here are fragments of code intended to be run on the processing nodes. See Chapter 5 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);
```

4.2 The Combine Interface

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

The supported operations are:

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

Each operation is described in more detail below.

Implementation Note: Because of way the broadcast and combine interfaces interact, if a node is abstaining from a combine operation, that node should *not* execute a broadcast operation until the combine operation is completed. (For more information, see Section 6.2.6.)

4.2.1 The Combine Register Interface

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

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

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

4.2.2 Combine Messages

The combine interface is essentially synchronous — combine operations are not completed until all non-abstaining nodes have started the *same* type of combine operation. If two nodes attempt to start different combining operations at the same time, an error is signaled.

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

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

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

Important: Pipelined messages cannot use doubleword read/write operations — see Section 6.1.2.

4.2.3 Sending Combine Messages

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

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

The 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 legal *combiner* and *pattern* values are described in Section 4.2.4 below.

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

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

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

```
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.

4.2.4 Legal Combiner and Pattern Values

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

pattern

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

combiner:

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

Network-done operations are specified by a *pattern* value of 0 together with a *combiner* value of 5.

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

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

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

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

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

4.2.5 Receiving Combine Message

The message-receiving interface of the combine interface is as described in Chapter 2, with the exception of the network-done operation, which is received through the Data Network status field **ni_router_done_complete** (see Section 4.2.8).

The following register is used to receive combine messages:

ni_com_recv	FIFO 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();
```

4.2.6 The Combine Status Register

The combine status register contains the following sub-fields:

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

The **send_ok**, **send_space**, **send_empty**, **rec_ok**, **rec_length**, and **rec_length_left** sub-fields are as described in Chapter 2. You can obtain the values of these sub-fields by using the generic field extractors described in Section 2.4.3.

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

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

The flag **com_scan_overflow** is described in Section 4.2.7.1.

4.2.7 Scanning (Parallel Prefix) and Reduction Operations

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

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

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

A scan or reduction 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. The 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.

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

The legal *combiner* and *pattern* values can be specified as 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.

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.

Scanning with Segments

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

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

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

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

The one-bit flag in 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.

4.2.7.1 Addition Scan Overflow

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

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

This flag is 1 if the current message in the receive FIFO 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: The **com_scan_overflow** flag is only meaningful when the current message being received is a signed or unsigned addition scan (an **ADD_SCAN** or **UADD_SCAN** operation).

4.2.8 Network-Done Messages

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

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

A network-done message is *always* of length 1; 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 interface. The result of a network-done message is not delivered as a value in the receive FIFO.

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

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

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

You can use the following macro to access this flag:

`DR_ROUTER_DONE` (*status*)

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

How Network-Done Works...

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

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.

...And Why You Should Care

Since network-done operations involve a *combine interface* scan of the value of a *Data Network* register, you should be careful about setting and changing the abstain flags of the combine interface when you intend to send a network-done message. (See Section 4.2.9 for a discussion of the combine interface abstain flags.)

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

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

NOTE

Because of a hardware defect, Revision A NI chips don't always execute network-done operations correctly. For more information, see Section 6.2.5.

4.2.9 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_rec_abstain	Flag, combine network abstain flag.
ni_reduce_rec_abstain	Flag, special reduction abstain flag.

The effect and use of these abstain flags is as described in Section 2.5.

Note: Because of way the broadcast and combine interfaces 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 6.2.6.)

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 4.2.10).

You can use the abstain flag macros described in Section 2.5.3 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);
```

4.2.10 Abstain Flags and Reduction Messages

Reduction operations differ from scans in terms of node participation. 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 combine abstain flag, **ni_reduce_rec_abstain**, that controls whether a node *receives* the result of a reduction operation. When **ni_reduce_rec_abstain** is 1, all incoming reduction results are discarded.

You can use the following macros to read and write the receive abstain flag:

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

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 5.1.)

4.2.11 Combine Interface Examples

The examples shown here are fragments of code that are intended to be run on the processing nodes. See Chapter 5 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);
```

4.3 The Global Interface

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

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

A global interface message can be sent by either of the following interfaces:

- the synchronous global interface, which requires that all nodes send a message before any receive the result
- the asynchronous global interface, which permits nodes to send a message and read the result at any time, with the network continually monitoring the state of all participating nodes

There is a separate register set for each of these interfaces. Both are described in more detail in the sections below.

4.3.1 The Global Register Interfaces

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

Synchronous global interface:

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

Asynchronous global interface:

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

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

4.3.2 The Synchronous Global Interface

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

The following registers and flags form the synchronous global interface:

ni_sync_global_send	Used to send the first value of a message.
ni_sync_global_abstain	Used to abstain from synch. global msgs.
ni_sync_global	Used to receive a message.
ni_sync_global_rec	Synchronous global receive flag.
ni_sync_global_complete	Synchronous global completion flag.

Sending and Receiving Messages

To start a synchronous global interface message, write a value (either 0 or 1) to the the **ni_sync_global_send** register. 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 sent a message, the global interface takes the logical OR of the **ni_sync_global_send** flag in each node, and then sets the **ni_sync_global_rec** flag of every participating node to the result. At the same time, the **ni_sync_global_complete** flag is set back to 1 to indicate completion of the message.

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();
```

Abstaining from Synchronous Global Messages

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

ni_sync_global_abstain	Status register, contains global abstain flag.

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

You can use the abstain flag operations described in Chapter 2 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.

4.3.3 The Asynchronous Global Interface

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

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

The following registers and flags form the asynchronous global interface:

ni_async_global	Control register, contains the following flags:
ni_global_send	Flag, used to "send" asynchronous messages.
ni_global_rec	Flag, always set to logical OR of **send** flags.

Sending and Receiving Messages

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

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

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

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();
```

4.3.4 Global Interface Examples

The examples shown here are fragments of code intended to be run on the processing nodes. See Chapter 5 for a discussion of large-scale program structure.

Using the Synchronous Global Interface

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

```
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 Interface

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 5

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 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.

5.1 Transferring Data between Nodes and the PM

As described in Section 3.3, 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:

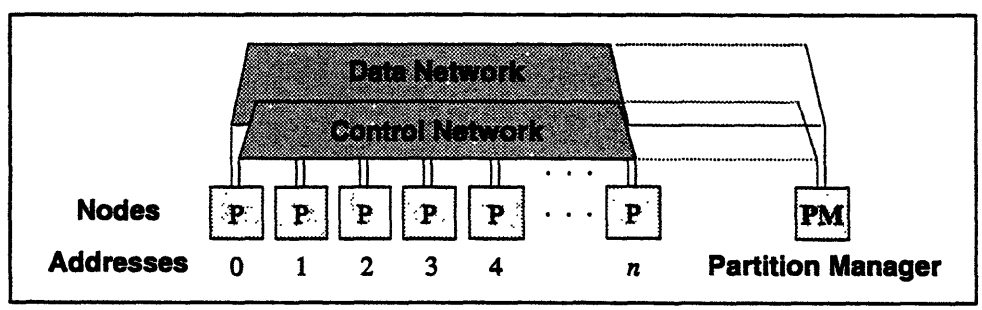

Figure 8. The partition manager stands apart from the partition it manages.

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

5.1.1 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 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 interface. 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.

5.1.2 Sending Messages from Nodes to the PM

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

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 5.2, 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 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 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 5.2.)

5.1.3 Signaling the PM

Because the above PM/node communication functions use both the broadcast and combine interfaces, 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 interface to create a simple barrier synchronization.

5.1.4 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_interface_send_packet_to_scalar(source, length, tag)
```

where the *interface* abbreviation is `dr`, `ldr`, or `rdr`, depending on the network interface you wish to use, 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 5.1.2.

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.

5.2 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 interfaces 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 interfaces are in use.

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

5.3 Broadcast Enabling

Along with setting the abstain flags, there's one other important operation that needs to be included in any NI program. As noted in Section 4.1.3, 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.

5.4 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.

5.4.1 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. **`Note:`** If you plan to call **`CMOS_signal()`** (see Section 3.5.2), you must also **`#include`** the header file **`<cmsys/cm_signal.h>`**.)

5.4.2 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.

5.4.3 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.

5.4.4 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 host-callable function, so that there's no possibility of collision with names of node functions that you have not defined as host entry-points.

5.5 A Sample Program

As an example, here's a simple NI program that uses each of the CM-5 network interfaces. 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();
}
```

5.6 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.

5.6.1 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
# nicc2 -- Compiles an NI program
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
```

5.6.2 Compiling and Running the Program

Note: The following examples assume that you are currently logged into one of the partition managers of a CM-5.

The output of the compiling script for the **NI_test** 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.
```

5.6.3 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/examples/ni-examples
```

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

Chapter 6

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.

6.1 Performance Hints

6.1.1 NI Register Operation Times

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

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

The time taken to perform an NI register read/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.

6.1.2 Reading and Writing Registers with Doubleword Values

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

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

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

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

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

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

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 */
}
```

6.1.3 Use Message Discarding for Efficiency

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

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

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

6.1.4 Set the Abstain Flags Once and Forget Them

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

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

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

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

6.2 Potential Programming Traps and Snares

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

6.2.1 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.

Implementation Note: Currently, there is an additional restriction — the most significant (20th) bit of the address must also be 0, or an error will result.

6.2.2 Check the Tag before Retrieving a Data Network Message

As described in Section 3.5.1, 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.

6.2.3 Make Sure Double-Word Data Is Doubleword Aligned

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

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

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

6.2.4 Order Is Important in Combine Messages

As noted in Section 4.2.7, 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.

6.2.5 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. 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())) {};
```

6.2.6 Broadcast and Combine Interface Collisions

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

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

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

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

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_*interface*_status** register for any *interface* that has such a register:

Field Name:	Macro used to read value of field:
ni_send_ok	SEND_OK(*status_value*)
ni_send_space	SEND_SPACE(*status_value*)
ni_send_empty	SEND_EMPTY(*status_value*)
ni_rec_ok	RECEIVE_OK(*status_value*)
ni_rec_length	RECEIVE_LENGTH(*status_value*)
ni_length_left	RECEIVE_LENGTH_LEFT(*status_value*)

For interfaces 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 interface 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 Network interfaces 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 Network interfaces 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_recv	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_send_ok	SEND_OK(dr_status)
ni_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_send_ok	SEND_OK(ldr_status)
ni_send_space	SEND_SPACE(ldr_status)
ni_rec_ok	RECEIVE_OK(ldr_status)
ni_ldr_rec_tag	RECEIVE_TAG(ldr_status)
ni_rec_length	RECEIVE_LENGTH(ldr_status)
ni_rec_length_left	RECEIVE_LENGTH_LEFT(ldr_status)
ni_rdr_status	
ni_send_ok	SEND_OK(rdr_status)
ni_send_space	SEND_SPACE(rdr_status)
ni_rec_ok	RECEIVE_OK(rdr_status)
ni_rdr_rec_tag	RECEIVE_TAG(rdr_status)
ni_rec_length	RECEIVE_LENGTH(rdr_status)
ni_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 Interface Constants and Macros

Send and Receive Register Macros

The **send_first** register for the broadcast interface 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 interface 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_recv	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 interface 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_send_ok	SEND_OK(bc_status)
ni_send_space	SEND_SPACE(bc_status)
ni_send_empty	SEND_EMPTY(bc_status)
ni_rec_ok	RECEIVE_OK(bc_status)
ni_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_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 message is given by the constant

```
MAX_BROADCAST_MSG_WORDS
```

A.4 Combine Interface Constants and Macros

Send and Receive Register Macros

The **send_first** register for the combine interface 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 interface 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_recv	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 message (with the exception of network-done messages, which are always 1 word) 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 interface 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_send_ok	SEND_OK(com_status)
ni_send_space	SEND_SPACE(com_status)
ni_send_empty	SEND_EMPTY(com_status)
ni_rec_ok	RECEIVE_OK(com_status)
ni_rec_length	RECEIVE_LENGTH(com_status)
ni_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_rec_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 Interface 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_async_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

Syntax:

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

Description:

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 program 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 except for the floating point context and the global registers **g5**, **%g6**, and **%g7** 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) which produced the interrupt will still be in the receiving FIFO when the user message handler is invoked. It is the responsibility of the user message handler to empty these messages.

Return Values:

`CMOS_signal()` returns the previous action on success. On failure, it returns -1 and sets `errno` to indicate the error.

Errors:

`CMOS_signal()` will fail and no action will take place if one of the following occurs:

`EINVAL`	`sig` was not a valid signal number.

Notes:

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 interface, etc.

Message interrupts are disabled while user code is in a user message handler. Thus, user message handlers need not be reentrant. However, the message handler should not enable interrupts (via a call to `CMOS_signal()`.) If it does, the results are unpredictable. Also, note that 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. Message interrupts are disabled by a call to `CMOS_signal()` with `func` set to `CM_SIG_IGN`. The `mask` argument is ignored. (Note that all user tag interrupts are disabled by this call.)

Appendix C

CMNA Header Files

To access the NI macros described in this document, you must **#include** the header file **cm/cmna.h**:

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

This file **#includes** many other header files that provide access to NI register macros and accessor functions. These macros and functions are collectively referred to as CMNA (CM Network Accessors), and can serve as a basis for your own NI accessor code.

Note: The functions and macros in CMNA are designed to be very generic in operation. As such, they are much less efficient than the special-purpose macros and functions you'll probably write on your own. Nevertheless, you can use the operations defined in CMNA as a jumping-off point for your own code, to help you understand what needs to be done to get your code to run correctly.

C.1 What is CMNA?

There are two main parts to CMNA:

- The NI Interface — Constants and macros used to manipulate NI registers.
- CnC ("C-and-C") — C functions that perform NI operations such as reading and writing messages of arbitrary length.

The CMNA header files define the NI interface explicitly, in terms of register accessor macros and constants. The header files also provide C prototypes for the CnC functions, which are part of the CMOST operating system code.

C.2 CMNA Header Files

The following header files are part of CMNA:

/usr/include/

cm/cmna.h	— Main CMNA header file.
cmsys/cmna.h	— CMNA user header file.
cmsys/cmna_sup.h	— CMNA supervisor header file.
cmsys/ni_interface.h	— Main NI interface header file.
cmsys/ni_macros.h	— NI macro definitions.
cmsys/ni_constants.h	— NI register/flag constant definitions.
cmsys/ni_defines.h	— Low-level NI constant definitions.

The following diagram shows the relationship among the header files that make up CMNA:

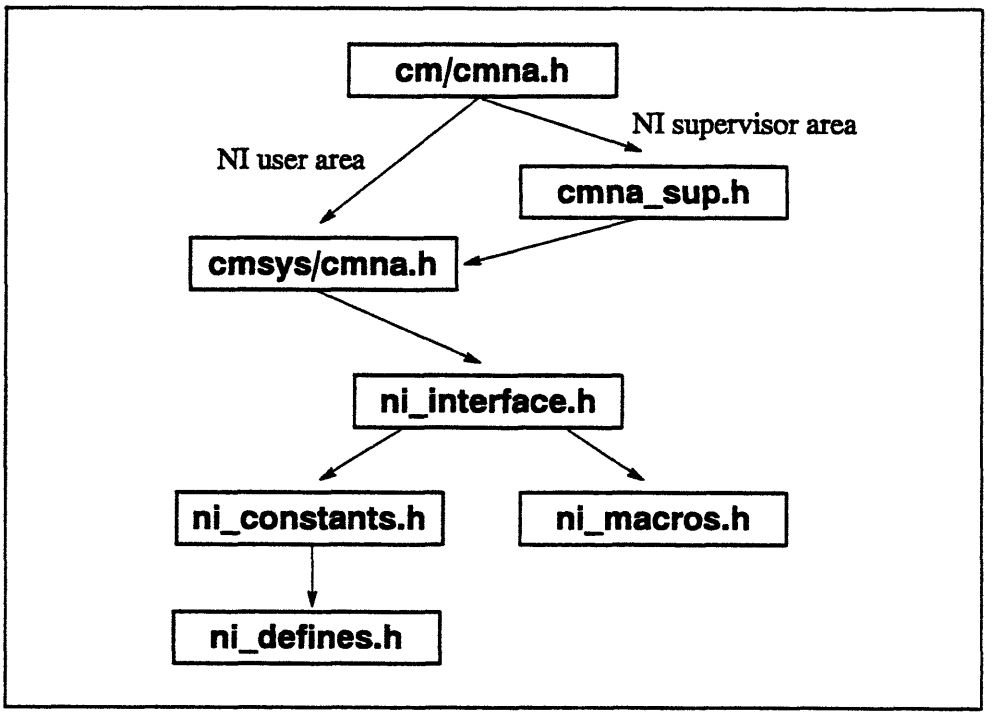

Figure 9. Relationship between CMNA and NI header files.

C.2.1 The Main CMNA Header File: cm/cmna.h

This single file `#includes` all the header files that are needed to define CMNA. However, it contains virtually no definitions of its own. It simply `#includes` either of the two header files `cmsys/cmna.h` or `cmsys/cmna_sup.h`, according to which NI register area (user or supervisor) the `#includeing` code needs.

Implementation Note: At present, `cmsys/cmna_sup.h` is only `#included` for diagnostic code (that is, code that defines the symbol `CMDIAG`).

C.2.2 The User Header File: cmsys/cmna.h

This file `#includes` the NI constant and macro files described below, and also defines a number of useful C mask constants and C macros that are used in CMNA. However, the constants and macros defined here are only sufficient for the needs of CMNA, and are not by any means a complete set. (See the description of the `ni_constants.h`, and `ni_defines` files below.)

C.2.3 The Supervisor Header File: cmsys/cmna_sup.h

This file modifies a few key constant definitions so that any absolute memory address constants defined in the other header files will refer to the NI supervisor area, rather than the NI user area. It then `#includes` `cmsys/cmna.h`, so it has much the same effect as that header file.

Note: The `cmsys/cmna_sup.h` file is only of use to programmers with legal access to the NI supervisor area. Including this file does *not* in itself grant access to the NI's supervisor area; it simply redefines many CMNA constants to have address values that are only legal for supervisor code.

C.2.4 The NI Interface Header File: ni_interface.h

This file defines the NI accessor interface. It `#includes` the file `ni_constants.h`, and defines a number of basic NI register macros that are used by CMNA. It then `#includes` `ni_macros.h` to define the remainder of the CMNA macros.

This file also defines a number of NI register constants that are suitable for use in C code. (That is, constants that have been cast as `(unsigned *)` values. See the description of `ni_constants.h` and `ni_defines.h` below.)

C.2.5 The NI Macros Header File: ni_macros.h

This file defines a number of C macros that perform sterotypical NI operations such as sending and receiving messages via a specific network interface.

C.2.6 The NI Constants Header Files: ni_constants.h, ni_defines.h

These files define a number of register constants and masks that are used by CMNA. In particular, `ni_constants.c` includes definitions of constants specifying the absolute memory address for each of the NI's registers. The file `ni_defines.h` defines hundreds of constants that give the size and offset of the register fields of the NI. These two sets of constants provide a complete interface for NI operations written in assembly code.

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

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

If you don't perform this casting step, the C compiler by default treats the constants as signed integers, possibly causing your code to fail. Note that many of these constants are recast in just this fashion in the `ni_interface.c` header file , so you may be able to just use those constants without having to do any recasting yourself.

Appendix D

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 5, and these test programs `#include` the same `utils.h` file as is used in Chapter 5.

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.

Important: You should view the examples presented here as merely a cookbook of possible 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.

D.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 <cmsys/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();
}
```

D.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 <cmsys/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 variables 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();
}
```

D.3 Broadcast Interface Test

This program presents a simple test of broadcasting:

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();
}
```

D.4 Combine Interface Test

This program presents examples of a number of different kinds of combine 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. Swan-
son,\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);

    /* Make sure all results are in */
    PM_NODE_synch();

    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);

/* Make sure all results have been received */
PM_NODE_synch();

restore_abstain_flags();
}
```

D.5 Global Network Test

This program presents a quick example of asynchronous and synchronous global interface operations:

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. **Boldface** page numbers indicate a definition or other important reference.

A, B

C

D

M

N

O

R

S

U, X

Concept Index

This index lists the major terms and topics found in this document. **Boldface** page numbers indicate a definition or other important reference.

D

E

F

G, H

I

L

M

R

S

T, W, X