The Connection Machine System

CM5

NI Programmer's Handbook

NI Version 2.2
(CM-5E)

Thinking Machines Corporation

The Connection Machine System

NI Programmer's Handbook

NI Version 2.2 (CM-5E),
June 1994

Thinking Machines Corporation

First printing, June 1994

****

The information in this document is subject to change without notice and should not be construed as a commitment by Thinking Machines Corporation. Thinking Machines reserves the right to make changes to any product described herein.

Although the information in this document has been reviewed and is believed to be reliable, Thinking Machines Corporation assumes no liability for errors in this document. Thinking Machines does not assume any liability arising from the application or use of any information or product described herein.

****

Connection Machine® is a registered trademark of Thinking Machines Corporation.
CM, CM-2, CM-200, CM-5, and CM-5e are trademarks of Thinking Machines Corporation.
CM-5 Scale 3, and DataVault are trademarks of Thinking Machines Corporation.
CMOST, CMAX, and Prism are trademarks of Thinking Machines Corporation.
C*® is a registered trademark of Thinking Machines Corporation.
Paris, *Lisp, and CM Fortran are trademarks of Thinking Machines Corporation.
CMHFB, CMMD, CMSSL, and CMX11 are trademarks of Thinking Machines Corporation.
CMview is a trademark of Thinking Machines Corporation.
Scalable Computing (SC) is a trademark of Thinking Machines Corporation.
Scalable Disk Array (SDA) is a trademark of Thinking Machines Corporation.
Thinking Machines® is a registered trademark of Thinking Machines Corporation.
SPARC and SPARCstation are trademarks of SPARC International, Inc.
Sun, Sun-4, and Sun Workstation are trademarks of Sun Microsystems, Inc.
UNIX is a trademark of UNIX System Laboratories, Inc.
The X Window System is a trademark of the Massachusetts Institute of Technology.

Copyright © 1994 by Thinking Machines Corporation. All rights reserved.

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

Contents

Figures xv
About This Manual xvii
Customer Support xxi

Chapter 1 The CM-5 Network Interface **3**

1.1 The CM-5 System: Nodes and Networks 4
1.1.1 The CM-5 Networks 4
The Data Network 4
The Control Network 5
For the Curious: The Diagnostic Network 5
1.1.2 Processing Nodes 5
1.1.3 Partitions and Partition Managers 6
1.1.4 Programming Models 7
User Programming Model 7
OS Programming Model 7
1.2 The NI Chip 8
1.3 The NI Registers 8
1.3.1 For the Curious: The NI Base Address 10
1.3.2 NI Register Types 11
1.3.3 NI Register and Field Names 12
1.3.4 NI Register and Field Programming Constants 13
Finding the Constant You Need 13
Register Constants 13
Field Constants 14
NI Base Address Constant 14
1.3.5 C Macros Useful for Writing NI Code 14
Finding the C Macro You Need 15
1.4 Interrupts 15
1.5 NI Reset 16
1.6 Using This Manual Effectively 16
1.7 WARNING: Experiment at Your Own Risk 17

Copyright © 1994 Thinking Machines Corporation

Chapter 2 A Generic Network Interface 19

2.1 Network Interface Registers 19

2.2 Network Messages 20

2.2.1 Performance Note — Using Doubleword Operations 21

2.3 Sending a Message 21

2.3.1 Message Discarding 22

2.3.2 Auxiliary Information 22

2.3.3 Calculating `ni_`*interface*`_send_first` Addresses 23

Send First Address Constants 23

2.3.4 C Macros for Writing a Message 24

2.4 Receiving a Message 25

2.4.1 C Macros for Reading a Message 25

2.4.2 Detecting Arrival of a Message 26

2.4.3 Simulating the Arrival of a Message 26

2.5 The Status Register 27

2.5.1 The "Send OK" Flag 27

2.5.2 The "Send Space" Field and "Send Empty" Flag 28

2.5.3 The "Receive OK" Flag and "Receive Length" Fields 28

2.5.4 Reading the Status Register Fields 28

2.6 Abstaining from an Interface — The Control Register 29

2.6.1 Effect of Abstain Flags 30

2.6.2 Combine Interface Abstain Flags 30

2.6.3 Reading and Writing the Abstain Flag 30

2.6.4 Use the Abstain Flags Safely 31

2.6.5 Being a Good Neighbor 31

2.7 The Private Register 32

2.7.1 Message Receipt Interrupts — The Rec Interrupt Enable Flag 32

2.7.2 Clearing the Interface's Send FIFO — The Lock Flag 33

2.7.3 Grabbing the Receive FIFO Registers — The Rec Stop Flag . 33

2.7.4 Blocking Unsent Broadcast Messages — The Send Stop Flag 34

2.7.5 Detecting a Full Receive FIFO — The Receive Full Flag 34

2.8 Using a Generic Network Interface 34

2.9 From the Generic to the Specific 35

Chapter 3 The Data Network 37

3.1 The Data Network Register Interfaces 38

3.2 Data Network Messages 40

3.2.1 Short and Long Data Network Messages 40

NI Version 2.2 (CM-5E), June 1994
Copyright © 1994 Thinking Machines Corporation

3.2.2 Long Data Network Message Interrupt 41
3.2.3 Data Network Message-Sending Conventions 41

3.3 Data Network Addressing 41
3.3.1 Physical and Relative Addressing Modes 42

3.4 Sending and Receiving Messages 43
3.4.1 Sending Short Messages 44
Auxiliary Information for Short Messages 44
3.4.2 Sending Long Messages 45
Send First Long Address Format 45
Auxiliary Information for Long Messages 46
3.4.3 Receiving Messages 46

3.5 The Status Registers 47
3.5.1 The Standard Status Registers 47
3.5.2 The "Status All" Alternate Status Register 48
3.5.3 The "Status Pop" Register 49
3.5.4 Message Tags 50
User/Supervisor Tag Reservation 50
Tag Fields and Interrupts 50
Using CMOST Commands
to Set Up NI Interrupt Handlers 53
Tag Fields and the Message-Counting Registers 53
Message Count Disabling 54
Negative Message Count Interrupts 54
3.5.5 The Send and Receive State Fields 55
3.5.6 The Network-Done Flag 56

3.6 The Private Register 56

3.7 All Fall Down Mode 57
3.7.1 Triggering All Fall Down Mode 57
3.7.2 Detecting All Fall Down Mode Messages 57
3.7.3 Resending All Fall Down Mode Messages 58

3.8 Interrupt Enable Flags 59

3.9 Data Network Usage Note: Receive before You Send 59

3.10 Examples 60
Sending and Receiving a Message 60
Sending and Receiving Long Messages 61
Interrupt-Driven Message Retrieval 63
Sending via LDR and RDR Simultaneously 64

Copyright © 1994 Thinking Machines Corporation

Chapter 4 The Control Network **65**

4.1 The Broadcast Interface 66
- 4.1.1 Broadcast Register Interfaces 66
- 4.1.2 Broadcast Messages 67
- 4.1.3 Sending Broadcast Messages 68
- 4.1.4 Auxiliary Information 69
- 4.1.5 Receiving Broadcast Messages 69
- 4.1.6 The Broadcast Status Register 70
- 4.1.7 Abstaining from the Broadcast Interface 71
- 4.1.8 The Broadcast Private Register 71
- 4.1.9 Broadcast Interface Examples 72
  - Sending and Receiving a Message 72

4.2 The Combine Interface 73
- 4.2.1 The Combine Register Interface 74
- 4.2.2 Combine Messages 75
- 4.2.3 Sending Combine Messages 75
- 4.2.4 Auxiliary Information 76
- 4.2.5 Legal Combiner and Pattern Values 77
- 4.2.6 Receiving Combine Message 78
- 4.2.7 The Combine Status Register 78
- 4.2.8 Scanning (Parallel Prefix) and Reduction Operations 79
  - Scanning with Segments 80
  - Addition Scan Overflow 80
- 4.2.9 Network-Done Messages 81
  - How Network-Done Works... 82
  - ...And Why You Should Care 83
- 4.2.10 Abstaining from the Combine Interface 83
- 4.2.11 The Combine Private Register 85
  - Empty Receive FIFO Interrupt 85
  - Clearing the Combine Send FIFO 86
- 4.2.12 Combine Interface Examples 87
  - Sending and Receiving a Combine Message 87
  - Executing Scans and Reduction Scans 88
  - Executing a Network-Done Operation 89

4.3 The Global Interface 90
- 4.3.1 The Three Global Register Interfaces 91
- 4.3.2 The Synchronous Global Interface 91
  - Sending and Receiving Messages 92
  - Abstaining from Synchronous Global Messages 92
  - Synchronous Global Receive Interrupt 93
  - Supervisor Operations for the Synchronous Global Interface 93

Copyright © 1994 Thinking Machines Corporation

4.3.3 The Asynchronous Global Interface 93
Sending and Receiving Messages 94
Asynchronous Global Receive Interrupt 95
4.3.4 The Supervisor Asynchronous Global Interface 95
Sending and Receiving Messages 95
Supervisor Asynchronous Global Receive Interrupt 95
4.3.5 Global Interface Examples 96
Using the Synchronous Global Interface 96
Using the Asynchronous Global Interface 96

Chapter 5 NI Interrupts 97

5.1 Interrupt Classes 97
5.1.1 Disabling Bus Errors 100
5.2 Interrupt Pathways 100
5.2.1 Red Interrupts 101
5.2.2 Orange Interrupts 102
5.2.3 Yellow Interrupts 102
5.2.4 Green Interrupts 103
5.3 The Interrupt Cause and Clear Registers 104
5.4 Interrupt Levels 105
5.5 Broadcast Interrupts 106
5.6 Recovering from Interrupts 107

Chapter 6 Other NI Interfaces and Features 109

6.1 The "Hodgepodge" Register 109
6.2 Node Address Registers 110
6.3 NI Chunk Table and Address Translation 110
6.3.1 Node Address Translation 110
6.3.2 Chunk Sizes and Address Allocation 112
6.3.3 Modifying the Chunk Table 114
6.4 Combine Interface Flush 114
6.5 The NI Timer 115
6.6 The Bad Address Register 116
6.7 NI Partition Configuration 117
6.8 Disabling the Control Network 118
6.9 NI Serial Number 118
6.10 NI Reset 119

Copyright © 1994 Thinking Machines Corporation

Chapter 7 Writing NI Programs **121**

7.1 Transferring Data between Nodes and the PM 121
7.1.1 Sending Messages from the PM to Nodes 122
7.1.2 Sending Messages from Nodes to the PM 123
7.1.3 Signaling the PM 124
7.1.4 For the Curious: Using the Data Network 124

7.2 Setting the Abstain Flags 125

7.3 Broadcast Enabling 126

7.4 NI Program Structure 127
7.4.1 The `cmna.h` Header File 127
7.4.2 Partition Manager Code 127
7.4.3 Node Code 128
The Node's "Main" Routine 128
7.4.4 Interface Code 129

7.5 A Sample Program 129

7.6 Compiling and Executing an NI Program 134
7.6.1 A Simple Compiling Script 135
7.6.2 Compiling and Running the Program 136
7.6.3 On-Line Code Examples 136

Chapter 8 NI Programming Issues **137**

8.1 Performance Hints 137
8.1.1 NI Register Operation Times 137
8.1.2 Reading and Writing Registers with Doubleword Values 138
Example: LDR Send/Receive 138
8.1.3 Use Message Discarding for Efficiency 140
8.1.4 Set the Abstain Flags Once and Forget Them 140

8.2 Potential Programming Traps and Snares 141
8.2.1 Address Calculation on the Partition Manager 141
8.2.2 Pay Attention to Data Network Addresses 141
8.2.3 "Middle" Data Network Interface Restrictions 142
8.2.4 Make Sure Doubleword Data Is Doubleword Aligned 142
8.2.5 Order Is Important in Combine Messages 142
8.2.6 Broadcast and Combine Interface Conflicts 142
8.2.7 Broadcast Enabling 143
8.2.8 Combine Interface Pipelining Restriction 143
8.2.9 Restriction on Scan Segment Start Flag 143
8.2.10 Be Careful When Altering Abstain Flags 144
8.2.11 Simulating Receipt of Messages 144

Copyright © 1994 Thinking Machines Corporation

8.2 Potential Programming Traps and Snares (*cont'd*)
8.2.12 Message Too Long Interrupt Restriction 144
8.2.13 All Fall Down Restriction 144
8.2.14 Send/Receive and FIFO Locking Restrictions.............. 144

Appendixes

Appendix A NI Registers, Fields, and Constants 147

A.1 NI Registers 147
A.1.1 Global and System Registers 148
A.1.2 Network Interface Registers 149
Combined Data Network Interface (DR) 149
Left Data Network Interface (LDR) 149
Right Data Network Interface (RDR) 149
Broadcast Interface (BC) 150
Supervisor Broadcast Interface (SBC) 150
Combine Interface (COM) 150

A.2 NI Message Length Limit Constants 151

A.3 Send First Register Addresses 151
Data Network (DR/LDR/RDR) Auxiliary Data Fields .. 152
Broadcast (BC/SBC) Auxiliary Data Fields 153
Combine Auxiliary Data Fields 153

A.4 Send First Long (Data Network) Register Addresses 154

A.5 NI Fields 155
A.5.1 Combined Data Network (DR) Fields 155
The `ni_dr_status` Register 155
The `ni_dr_status_long` Register 156
The `ni_dr_status_{all/pop}` Registers 156
The `ni_dr_private` Register 156
A.5.2 Left Data Network Interface (LDR) Fields 157
The `ni_ldr_status` Register 157
The `ni_ldr_status_long` Register 157
The `ni_ldr_status_{all/pop}` Registers 157
The `ni_ldr_private` Register 158
A.5.3 Right Data Network Interface (RDR) Fields 158
The `ni_rdr_status` Register 158
The `ni_rdr_status_long` Register 158
The `ni_rdr_status_{all/pop}` Registers 159
The `ni_rdr_private` Register 159

Copyright © 1994 Thinking Machines Corporation

A.5.4 Broadcast Interface (BC) Fields 159
The ni_bc_status Register 159
The ni_bc_private Register 160
The ni_bc_control Register 160
A.5.5 Supervisor Broadcast Interface (SBC) Fields 160
The ni_sbc_status Register 160
The ni_sbc_private Register 160
The ni_sbc_control Register 161
A.5.6 Combine Interface (COM) Fields 161
The ni_com_status Register 161
The ni_com_private Register 161
The ni_com_control Register 162
A.5.7 Global Interface Fields 162
The ni_sync_global Register 162
The ni_async_global Register 162
The ni_async_sup_global Register 162
A.5.8 Interrupt Register Fields 162
The ni_interrupt_cause Register 163
The ni_interrupt_cause_green Register 163
The ni_interrupt_{clear,set} Registers 164
The ni_interrupt_{clear,set}_green Registers 164
A.5.9 Other Register Fields and Constants 165
The ni_interrupt_level Register 165
The ni_hodgepodge Register 165
The ni_bad_address Register 166

Appendix B NI Interrupts 167

B.1 Red Interrupts 168
B.1.1 Internal Fault .... Red Interrupt ... 168
B.1.2 CN Checksum Error, DR Checksum Error .... Red Interrupt ... 168
B.1.3 CN Hard Error .... Red Interrupt ... 169
B.1.4 MC Error, CMU Error .... Red Interrupt ... 169
B.1.5 BC Interrupt Red .... Red Interrupt ... 170
B.2 Orange Interrupts 170
B.2.1 Timer Interrupt .... Orange Interrupt 170
B.2.2 Router Done Complete .... Orange Interrupt 171
B.2.3 BC Interrupt Orange .... Orange Interrupt 171
B.3 Yellow Interrupts 171
B.3.1 BC Interrupt Yellow .... Yellow Interrupt . 172
B.3.2 Bad Memory Access .... Yellow Interrupt . 172

Copyright © 1994 Thinking Machines Corporation

B.3.3 COM Abstain Changed .... Yellow Interrupt . 172
B.3.4 DR Count Negative .... Yellow Interrupt . 173
B.3.5 BC or COM Collision .... Yellow Interrupt . 173
B.3.6 Bad Relative Address .... Yellow Interrupt . 174
B.3.7 Message Too Long .... Yellow Interrupt.. 174

B.4 Green Interrupts 175
B.4.1 BC Interrupt Green .... Green Interrupt . 175
B.4.2 DR Receive Tag .... Green Interrupt . 176
B.4.3 DR Receive All Fall Down .... Green Interrupt . 176
B.4.4 Interface (DR, BC, COM, etc.) Receive OK .... Green Interrupt . 176
B.4.5 Global Rec (Sync, Global, or Supervisor) Green Interrupt . 177
B.4.6 Com Receive Empty .... Green Interrupt . 177
B.4.7 Scan Overflow .... Green Interrupt . 178
B.4.8 DP Error (Vector Unit Error) .... Green Interrupt . 178
B.4.9 Send FIFO Empty (Data Network Only) Green Interrupt . 179
B.4.10 LDR/RDR Tag, LDR/RDR User Tag ... Green Interrupt . 179

B.5 Bus Errors 180
B.5.1 Bad Memory Access .... Bus Error 180

Appendix C Programming Tools 183

C.1 Generic Variables and Macros 183
C.2 Data Network Constants and Macros 184
Send and Receive Register Macros 184
Status Register Macros 185
Message Length Limit 186
C.3 Broadcast Interface Constants and Macros 186
Send and Receive Register Macros 186
Status Register Macros 186
Abstain Register Macros 187
Message Length Limit 187
C.4 Combine Interface Constants and Macros 188
Send and Receive Register Macros 188
Message Length Limit 189
Segment Start Register Macros 189
Status Register Macros 189
Abstain Register Macros 190
C.5 Global Interface Constants and Macros 190
Synchronous Global Register Macros 190
Asynchronous Global Register Macros 191

Copyright © 1994 Thinking Machines Corporation

Appendix D Predefined Low-Level NI Constants **193**

Appendix E CMOS_signal Man Page **201**

Appendix F NI Accessor Examples **203**

F.1 Reading and Writing Registers 203

F.2 Reading and Writing Subfields 204

F.3 Constructing Send-First Addresses 205

Data Network Send-First Macros 205

Broadcast Interface Send-First Macros 206

Combine Interface Send-First Macros 206

Appendix G Sample NI Programs **207**

G.1 Data Network Test 207

G.2 Data Network Doubleword Messages Test 214

G.3 Broadcast Interface Test 217

G.4 Combine Interface Test 220

G.5 Global Network Test 224

Appendix H CMNA Header Files **227**

H.1 What Is CMNA? 227

H.2 CMNA Header Files 228

H.2.1 The Main CMNA Header File: **cm/cmna.h** 229

H.2.2 The User Header File: **cmsys/cmna.h** 229

H.2.3 The Supervisor Header File: **cmsys/cmna_sup.h** 229

H.2.4 The NI Interface Header File: **ni_interface.h** 230

H.2.5 The NI Macros Header File: **ni_macros.h** 230

H.2.6 The NI Constants Header Files: **ni_constants.h**, **ni_defines.h** 230

H.3 CMNA Functions 231

H.3.1 CMNA Version 231

H.3.2 Activity Functions 231

H.3.3 DR Interface Functions 232

H.3.4 LDR Interface Functions 232

H.3.5 RDR Interface Functions 233

H.3.6 BC Interface Functions 234

H.3.7 SBC Interface Functions 235

NI Version 2.2 (CM-5E), June 1994
Copyright © 1994 Thinking Machines Corporation

H.3.8 COM Interface Functions 237
H.3.9 Global Interface Functions 237

Appendix I NI Chip Version 2.2 Changes 239

I.1 Long Data Network Messages 239
I.2 New Data Network Status Interface 240
I.3 New Data Network Tag Interrupt Interface 240
I.4 Non-Compatible Change to Broadcast Interface 240
I.5 New Interrupts 241
I.6 New Data Network Interrupt Enable Flags 241
I.7 New Bus Error Conditions 241
I.8 Disabling Bus Errors 242
I.9 Manually Triggering Interrupts 242
I.10 Global Interface Context-Switching 242
I.11 New Hodgepodge Register Fields 242

Index

Programming Tools Index 245
Concepts Index 255

NI Memory Map

NI Memory Map 269

Copyright © 1994 Thinking Machines Corporation

Figures

Figure 1. The CM-5 system: Processing nodes linked by Data and Control Networks. .. 4
Figure 2. The components of a typical processing node. 5
Figure 3. A partition of nodes and its partition manager. 6
Figure 4. NI provides access to features of the Data Network and Control Network. 8
Figure 5. The NI registers are mapped into user and supervisor memory areas. 9
Figure 6. Sample virtual memory maps showing location of NI memory region. 10
Figure 7. NI registers associated with each interface. 20
Figure 8. The three interfaces of the Data Network: DR, LDR, and RDR. 37
Figure 9. NI registers associated with each of the Data Network interfaces. 39
Figure 10. Relative addressing of nodes in a partition. 42
Figure 11. Data Network message format 43
Figure 12. Tag value interrupt paths for Data Network messages 52
Figure 13. The three interfaces of the Control Network: BC, COM, and global. 65
Figure 14. NI registers associated with each of the broadcast interfaces. 67
Figure 15. NI registers associated with the combine interface. 74
Figure 16. NI registers associated with the global interface. 90
Figure 17. The possible pathways for colored interrupts. 100
Figure 18. Translation from relative addresses to physical addresses. 111
Figure 19. The chunk table is used to map contiguous relative addresses onto discontiguous physical addresses. 113
Figure 20. The partition manager stands apart from the partition it manages. 121
Figure 21. Relationship between CMNA and NI header files. 228

Copyright © 1994 Thinking Machines Corporation

About This Manual

Objectives of This Manual

This manual describes in detail the design, features, and correct use of the Network Interface (NI) chip of the Connection Machine CM-5 system. This description is at a level sufficient for low-level CM-5 coders to make full use of the NI's features. Both user- and supervisor-level information is included, as well as numerous programming examples written in the C programming language.

Intended Audience

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

Revision Information

This manual is a consolidation of two previously published manuals: *Programming the NI* and *NI Systems Programming*.

This manual replaces both of the earlier books. This manual contains the same information and examples, and reflects the most recent revision of the NI chip, used in the CM-5E version of the Connection Machine system node hardware.

Copyright © 1994 Thinking Machines Corporation

Organization of This Manual

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

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

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

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

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

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

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

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

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

Appendix C Programming Tools
A list of NI macros and constants defined by the CMNA software layer.

Copyright © 1994 Thinking Machines Corporation

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

Appendix E **CMOS_signal Man Page**
The UNIX manual page for the `CMOS_signal` system call.

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

Appendix G **Sample NI Programs**
C code examples demonstrating the NI features described in the chapters of this manual.

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

Appendix I **NI Chip Version 2.2 Changes**
A quick-reference list of the changes to the NI chip as of Version 2.2, with references into the main text of this manual.

NI Memory Map
A two-sided memory map of the NI registers and fields.

Related Documents

These documents are part of the Connection Machine documentation set:

- *Connection Machine CM-5 Technical Summary*, November 1993
- *VU Programmer's Handbook*, CMOST Version 7.2 August 1993

Copyright © 1994 Thinking Machines Corporation

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.

Copyright © 1994 Thinking Machines Corporation

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:

Internet
Electronic Mail: `customer-support@think.com`

uucp
Electronic Mail: `ames!think!customer-support`

U.S. Mail: Thinking Machines Corporation
Customer Support
245 First Street
Cambridge, Massachusetts 02142-1264

Telephone: (617) 234-4000

Copyright © 1993 Thinking Machines Corporation

Chapter 1

The CM-5 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.

Copyright © 1994 Thinking Machines Corporation

1.1 The CM-5 System: Nodes and Networks

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

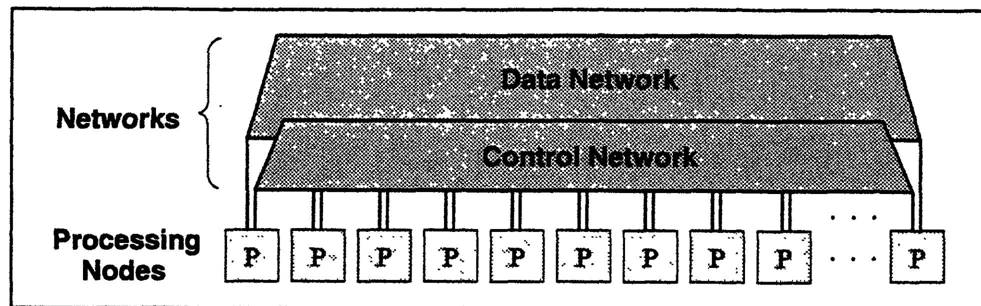


Figure 1. The CM-5 system: Processing nodes, plus Data and Control Networks.

The CM-5 contains a large number of *processing nodes*, which perform the arithmetic computations involved in a CM-5 program. The processing nodes are linked together by two internal communications networks, the *Data Network* and the *Control Network*. (See Figure 1.) The two CM-5 networks are similar in design; both are scalable, high-speed data communications networks. However, the two networks have distinct intents and purposes. 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 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.

Copyright © 1994 Thinking Machines Corporation

The Control Network

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

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

For the Curious: The Diagnostic Network

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

1.1.2 Processing Nodes

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

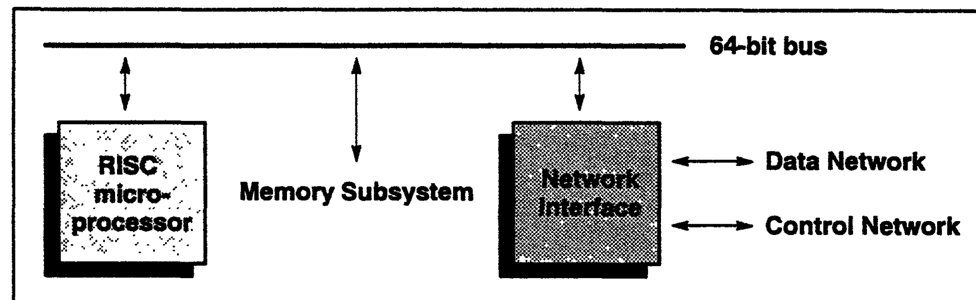


Figure 2. The components of a typical processing node.

Copyright © 1994 Thinking Machines Corporation

For the Curious: In the current implementation, the microprocessor is a SPARC chip; it executes both user and operating system (OS) code. The memory subsystem consists of DRAM memory controlled either by a single memory controller or by a set of four vector units (if your CM-5 has the vector unit option installed).

1.1.3 Partitions and Partition Managers

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

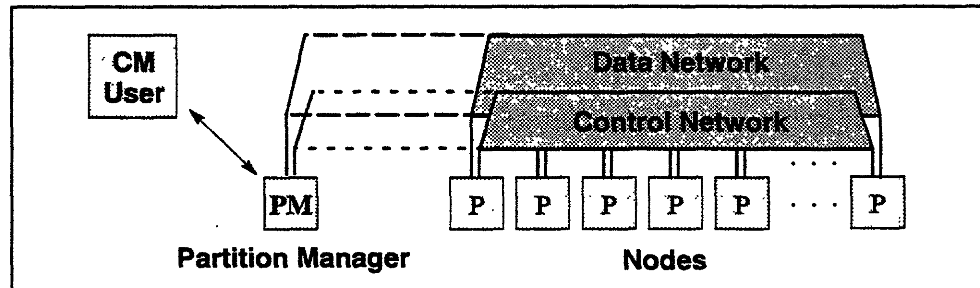


Figure 3. A partition of nodes and its partition manager.

The partition manager (PM) contains a RISC CPU and connecting hardware that allows the PM to interact with other computers and with users on terminals. Thus, the PM is the "gateway" by which a programmer gains access to the processing nodes of the CM-5 and instructs the CM-5 to execute a program.

The PM is attached to the Data and Control Networks, and can communicate with its partition of 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.

Copyright © 1994 Thinking Machines Corporation

1.1.4 Programming Models

User Programming Model

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

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 7.)

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

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

OS Programming Model

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

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

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

Copyright © 1994 Thinking Machines Corporation

1.2 The NI Chip

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

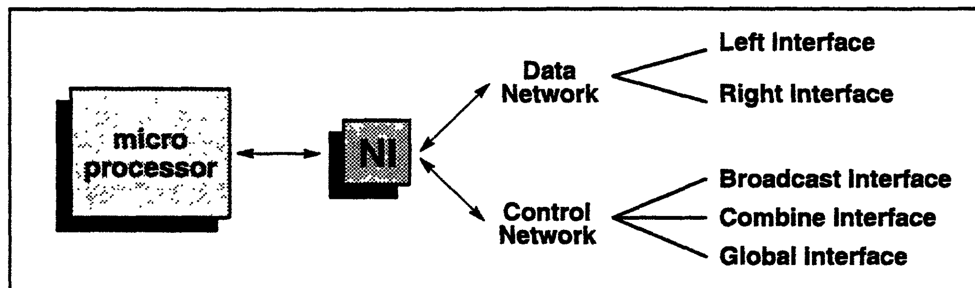


Figure 4. NI provides access to Data Network and Control Network.

When the microprocessor directs the NI to send a message via one of the networks, the NI handles the dispatching of the message, and collects any replies from the networks. The NI uses send FIFOs (queues) to hold outgoing messages until they can be sent, and receive FIFOs to hold incoming messages until the microprocessor can read 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. Thus, from a programmer's point of view the NI appears as a region of processor memory with some unique properties.

The microprocessor can either directly use the registers of the NI chip to send and receive messages, or it can use indirect methods, such as having the NI signal an interrupt whenever a message arrives. (Interrupts can also be "broadcast" from one NI chip to all other NIs in a partition.) Control of the NI is therefore based on register operations, interrupts, and (in extreme cases) NI Resets, which are described later in this chapter.

Copyright © 1994 Thinking Machines Corporation

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

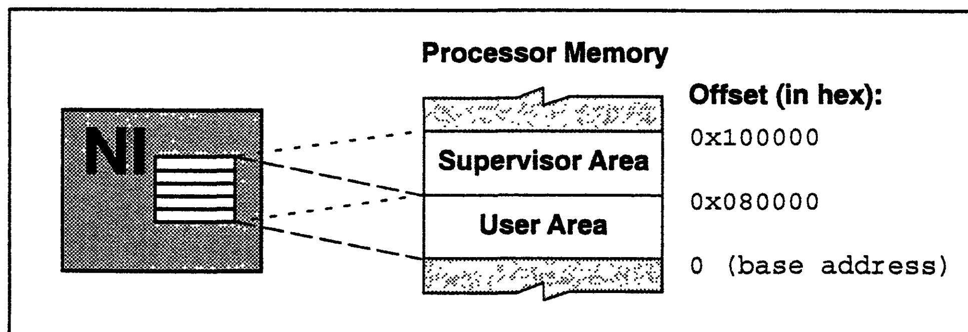


Figure 5. The NI registers are mapped into user and supervisor memory areas.

The user area occupies 512 Kbytes of virtual memory, starting at the base address of the NI memory region (see Section 1.3.1). The supervisor area occupies the 512 Kbytes immediately following the user area.

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

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

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

Copyright © 1994 Thinking Machines Corporation

1.3.1 For the Curious: The NI Base Address

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

The NI region's *virtual* base address, on the other hand, depends on the way the operating system sets up the virtual memory map. The operating system is free to map the NI memory regions to any virtual memory location, so long as both user and supervisor areas remain contiguous and user programs are prevented from accessing the supervisor area.

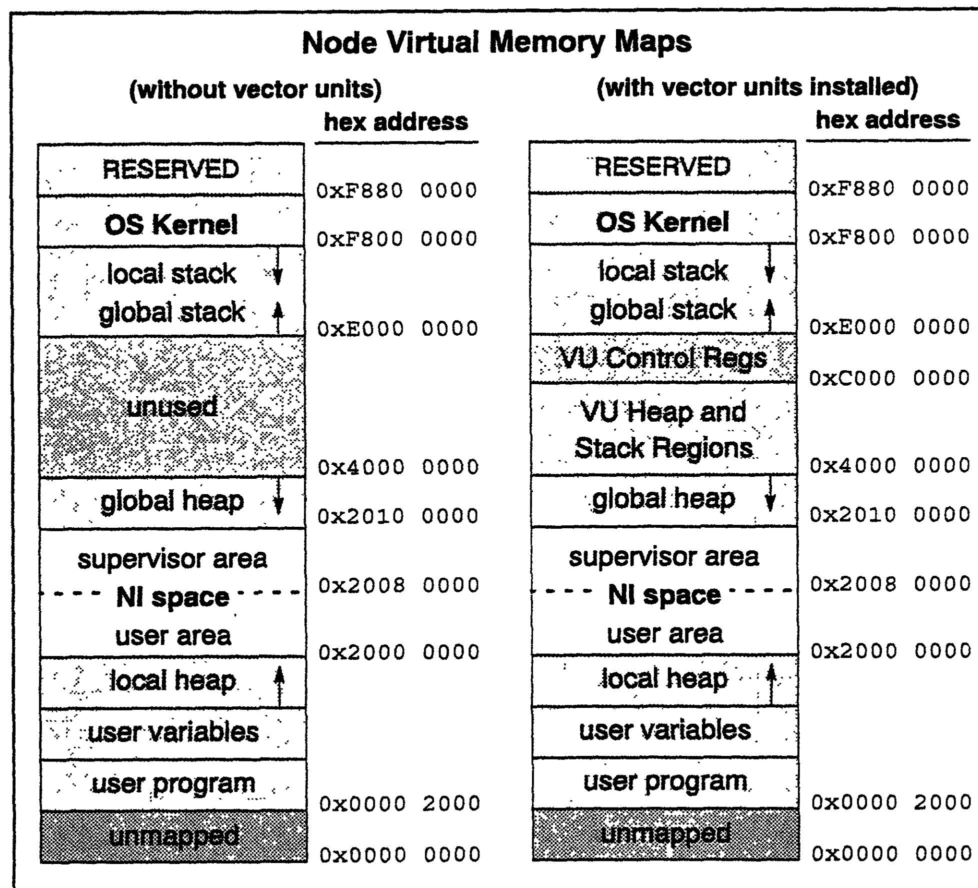


Figure 6. Sample virtual memory maps showing location of NI memory region.

Copyright © 1994 Thinking Machines Corporation

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

1.3.2 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 2.3.2.)

Performance Note: In terms of cycles, reading and writing NI registers is midway between reading the registers of the microprocessor and reading a value directly from processor memory (that is, *not* from cache memory). See Section 8.1.1 for details on the time taken to read and write NI registers.

Copyright © 1994 Thinking Machines Corporation

Important: Some registers are less than 32 bits long, even though they occupy a 32-bit memory location. When such a register is read, the value of the unused bits is undefined. However, when writing to the register, the unused bits should be written with either the same value that was last read from them, or with zeros. The effect of violating this restriction is not defined, but in some cases serious failures can result. (In at least one case, failing to zero out the unused bits causes your partition of nodes to crash. See Section 8.2.2.)

1.3.3 NI Register and Field Names

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

ni_*interface_purpose*

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

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

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

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

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

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

Copyright © 1994 Thinking Machines Corporation

1.3.4 NI Register and Field Programming Constants

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

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

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

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

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

Finding the Constant You Need

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

Register Constants

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

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

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

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

Copyright © 1994 Thinking Machines Corporation

Field Constants

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

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

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

NI Base Address Constant

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

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

1.3.5 C Macros Useful for Writing NI Code

You can write NI code using any programming method that allows you to read and write memory addresses. However, the examples in this manual 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 C.

Copyright © 1994 Thinking Machines Corporation

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 7 for more information.)

1.4 Interrupts

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

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

The five interrupt classes can be briefly summarized as follows:

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

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

Copyright © 1994 Thinking Machines Corporation

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

1.5 NI Reset

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

1.6 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 7.

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 7 to see how NI programs are structured.

Chapters 5 and 6 describe features of the NI that are primarily of interest to systems programmers (things such as interrupts and other OS-related operations).

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

Copyright © 1994 Thinking Machines Corporation

1.7 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 NI registers. When you use NI features in the manner described here, you should encounter no problems other than an 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!

Copyright © 1994 Thinking Machines Corporation

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. These register interfaces typically have a number of features in common. This chapter presents a "generic" network interface that describes these common features. With one exception (the global interface), all network interfaces conform to the model described here — individual variations for each network interface are discuessed in subsequent chapters.

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

2.1 Network Interface Registers

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

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

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

Copyright © 1994 Thinking Machines Corporation

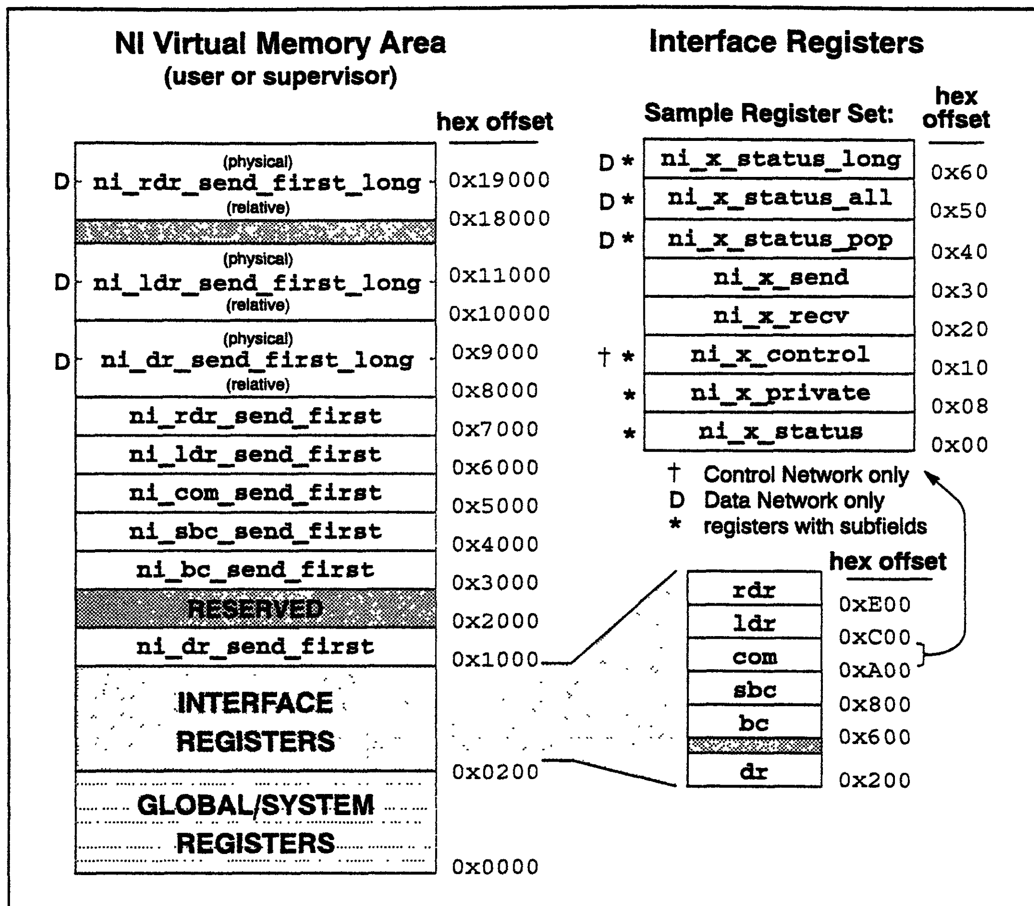


Figure 7. NI registers associated with each interface.

2.2 Network Messages

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

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

Copyright © 1994 Thinking Machines Corporation

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

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

2.2.1 Performance Note — Using Doubleword Operations

You can use doubleword (64-bit) operations to read and write FIFO registers. A doubleword read or write has exactly the same effect as the corresponding pair of single-word (32-bit) reads or writes, but the doubleword operation is usually more efficient. (See Section 8.1.2.) From here on, where this manual refers to a "value" of a message, you should understand this as referring to either a single- or doubleword value. Any network-specific restrictions that prevent the use of doubleword operations are noted in the descriptions of the networks themselves.

2.3 Sending a Message

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

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

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

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

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

Copyright © 1994 Thinking Machines Corporation

2.3.1 Message Discarding

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

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

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

Performance Note: You can use message discarding to your advantage and thereby make your code more efficient. Rather than check the **`send_ok`** flag after writing each word of a message to the send FIFO, you can simply check the flag once, after the entire message has been written. (For more information, see Section 8.1.3.)

2.3.2 Auxiliary Information

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

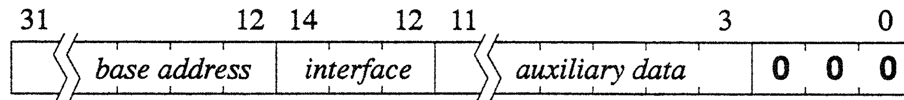
The **`send_first`** register for each network interface is actually mapped onto a block of memory locations. Writing a value to any one of these locations has the effect of writing that value to the **`send_first`** register, but the actual memory location that you use implicitly supplies the auxiliary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

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

Copyright © 1994 Thinking Machines Corporation

2.3.3 Calculating ni_*interface*_send_first Addresses

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



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

The interface numbering is as follows:

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

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

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

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

Send First Address Constants

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

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

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

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

Copyright © 1994 Thinking Machines Corporation

The following *interface_number* constants are defined:

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

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

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

2.3.4 C Macros for Writing a Message

If you are programming in C, there are macros that you can use to automatically calculate the appropriate addresses for 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*)

These 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 doubleword 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`** macros so that you can use doubleword register operations to make your code more efficient. (See Section 8.1.2 for more information.) For the most part, however, this manual focuses on singleword operations for clarity.

Copyright © 1994 Thinking Machines Corporation

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.

2.4 Receiving a Message

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

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

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

2.4.1 C Macros for Reading a Message

Just as there are C macros for writing network messages, there are macros for reading them: three network-specific 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.

Copyright © 1994 Thinking Machines Corporation

2.4.2 Detecting Arrival of a Message

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

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

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

2.4.3 Simulating the Arrival of a Message

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

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

Implementation Restriction: Currently, writing to the `recv` register does not work. The workaround for this restriction is for the node involved to send a message to itself — this message will wind up at the end of the receive FIFO, as if it had been written directly to the `recv` register.

Copyright © 1994 Thinking Machines Corporation

2.5 The Status Register

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

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

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

2.5.1 The "Send OK" Flag

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

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

Copyright © 1994 Thinking Machines Corporation

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

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

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

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

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

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

2.5.4 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.)

For each network, there is a C macro that returns the **status** register's value:

```
int value = CMNA_interface_status()
```


Copyright © 1994 Thinking Machines Corporation

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.

Note: A change in the broadcast interfaces requires the use of a different macro to access the **rec_length_left** field. See Section 4.1.6 for more information.

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);
```

2.6 Abstaining from an Interface — The Control Register

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

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

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

Copyright © 1994 Thinking Machines Corporation

2.6.1 Effect of Abstain Flags

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

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

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

2.6.2 Combine Interface Abstain Flags

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

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

For more information, see Section 4.2.10.

2.6.3 Reading and Writing the Abstain Flag

To read and write the the abstain flag of a network, you can use these C 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.

Copyright © 1994 Thinking Machines Corporation

2.6.4 Use the Abstain Flags Safely

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

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

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

This restriction generally requires that you use one of the interfaces (for example, the global interface) to synchronize the nodes and halt the operations of another interface while you change that interface's abstain flags. For this reason, most NI programmers set the abstain flags once, at the beginning of a program or routine, and then leave them set that way until the program or routine finishes executing, changing the flags within the routine only where absolutely necessary.

2.6.5 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.

Copyright © 1994 Thinking Machines Corporation

2.7 The Private Register

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

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

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

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

2.7.1 Message Receipt Interrupts — The Rec Interrupt Enable Flag

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

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

The Green interrupts that can be signaled are:

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

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

Copyright © 1994 Thinking Machines Corporation

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

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

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

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

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

2.7.3 Grabbing the Receive FIFO Registers — The Rec Stop Flag

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

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

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

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

Copyright © 1994 Thinking Machines Corporation

2.7.4 Blocking Unsent Broadcast Messages — The Send Stop Flag

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

2.7.5 Detecting a Full Receive FIFO — The Receive Full Flag

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

2.8 Using a Generic Network Interface

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

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

Copyright © 1994 Thinking Machines Corporation

2.9 From the Generic to the Specific

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

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

Copyright © 1994 Thinking Machines Corporation

Chapter 3

The Data Network

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

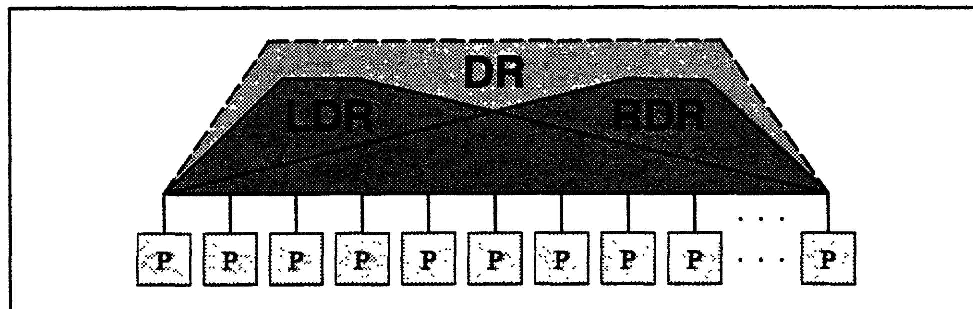


Figure 8. The three interfaces of the Data Network: DR, LDR, and RDR.

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

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

Copyright © 1994 Thinking Machines Corporation

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 network (DR). Each network interface is 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.
- The DR interface cannot be used to receive any messages, and 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 Data Network interface, the following registers are used:

`ni_`*dinterface*`_send_first`	Used to send the first value of a message.
`ni_`*dinterface*`_send_first_long`	Used for first value of long message.
`ni_`*dinterface*`_send`	Used to send the rest of the message.
`ni_`*dinterface*`_recv`	Used to receive a message.
`ni_`*dinterface*`_status`	Status register.
`ni_`*dinterface*`_status_long`	Status register for long messages.
`ni_`*dinterface*`_status_all`	Alternate status register.
`ni_`*dinterface*`_status_pop`	Status reg, also receives messages.
`ni_`*dinterface*`_private`	Supervisor control register.

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

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

Copyright © 1994 Thinking Machines Corporation

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

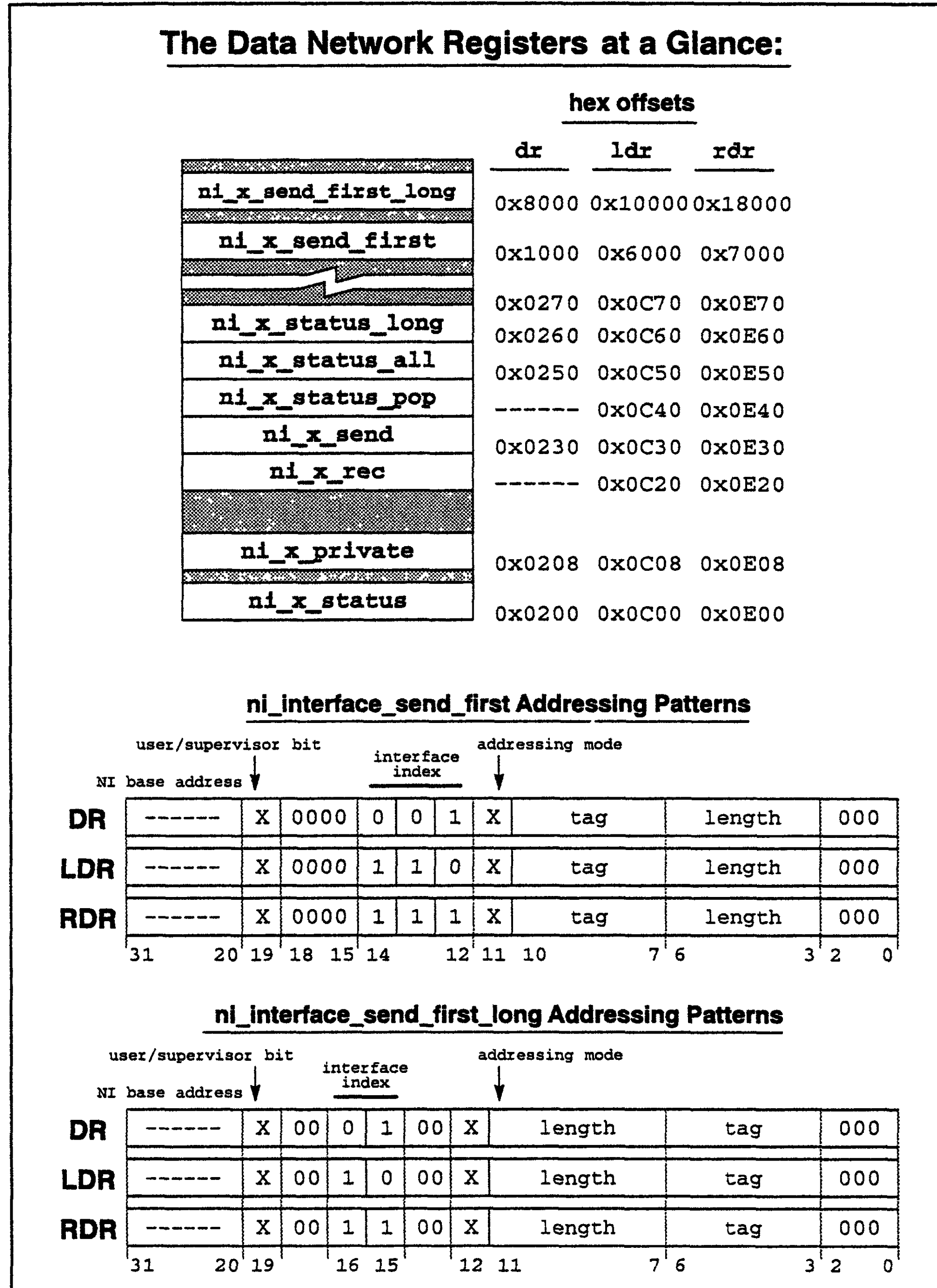


Figure 9. NI registers associated with each of the Data Network interfaces.

Copyright © 1994 Thinking Machines Corporation

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

`ni_longest_dr_message`	Length limit on Data Network messages.
`ni_hodgepodge`	Register with "hodgepodge" of flags:
`ni_msg_too_long_ie`	Message too long interrupt enable.
`ni_user_tag_mask`	User/supervisor tag reservation register.
`ni_rec_interrupt_mask`	Contains tag value interrupt flags.
`ni_user_rec_interrupt_mask`	Contains tag value interrupt flags.
`ni_dr_message_count`	Contains current message count.
`ni_count_mask`	Contains tag-count enable flags.

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

3.2 Data Network Messages

The Data Network is essentially asynchronous in operation — nodes can send and receive messages freely, so long as enough nodes are receiving messages so that the network does not become clogged (see Section 3.9). 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.)

3.2.1 Short and Long Data Network Messages

Each of the three Data Network interfaces can send messages of two types: short and long. A *short* message is sent as described in Chapter 2, and has a length limit of 5 words. A *long* message is sent via an alternate register interface, and has a length limit of 18 words. The long message interface is intended for messages that consist primarily of large quantities of data.

Implementation Note: The long message feature of the Data Network is an addition as of Version 2.2 of the NI chip. The short message type is actually the same Data Network message format used in previous NI versions, and is retained in Version 2.2 for software compatibility reasons.

Copyright © 1994 Thinking Machines Corporation

3.2.2 Long Data Network Message Interrupt

The NI register `ni_longest_dr_message` overrides the default length limits for long messages — trying to send a message longer than the value in `ni_longest_dr_message` signals a Yellow interrupt (`message too long`). This is intended to provide compatibility in CMs that contain NI chips of different versions. The flag `ni_msg_too_long_ie` in the `ni_hodgepodge` register controls this interrupt feature. If `ni_msg_too_long_ie` is 1, the `message too long` interrupt is signaled. If `ni_msg_too_long_ie` is 0, no interrupt is signaled. In either case, however, a Bus Error is signaled.

3.2.3 Data Network Message-Sending Conventions

Data Network messages are atomic; individual messages are not sent through the network until all the words of each message have been written into the send FIFO, and arrival of each message is not reported until all the words of the message have arrived in the receive FIFO. The component words of a single Data Network message are always received in the same order as they were sent. However, if you use multiple Data Network messages as "packets" to send long messages from one node to another, the order in which the packets arrive is not guaranteed to be the same as the order in which they were sent.

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

3.3 Data Network Addressing

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

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

Copyright © 1994 Thinking Machines Corporation

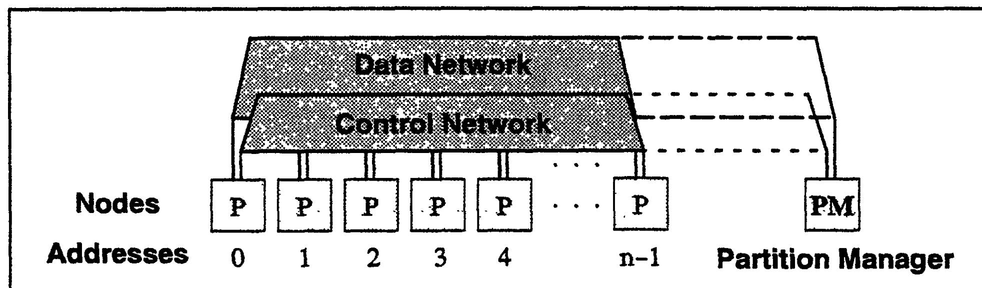


Figure 10. Relative addressing of nodes in a partition.

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 C 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.

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

3.3.1 Physical and Relative Addressing Modes

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

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

When a message is sent in relative mode, the address word is treated as a relative address, and is translated into a physical address based on the current partitioning arrangement. This translation is performed automatically by the NI hardware, using a *chunk table*, described in Section 6.3. The translation also includes automatic error checking to make certain that the supplied address is a legal relative address for the current partition. Messages that contain illegal relative addresses

Copyright © 1994 Thinking Machines Corporation

are not sent through the network; instead, the sending NI signals a Yellow interrupt (`bad relative address`).

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

3.4 Sending and Receiving Messages

The Data Network message format is the same for all three interfaces (and for short and long messages alike). The first word of the message is a 20-bit destination address. The remaining words form the content of the message, which must be no longer than the length limit allowed by the message type in use.

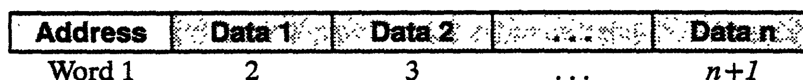


Figure 11. Data Network message format

For short messages, the data length limit is currently 5 words, and is given by the constant `MAX_ROUTER_MSG_WORDS`. For long messages the limit is 18 words. (The `ni_longest_dr_message` register value, if less, overrides these limits.)

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

Important: The address word of the message *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.

Copyright © 1994 Thinking Machines Corporation

3.4.1 Sending Short Messages

The protocol for sending a short message is as described 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.

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

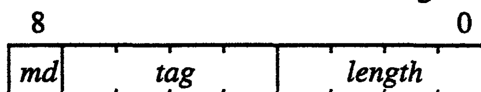
CMNA_*dinterface*`_send_first`(*tag*, *length*, *value*)
CMNA_*dinterface*`_send_first_double`(*tag*, *length*, *value*)
CMNA_*dinterface*`_send_word`(*value*)
CMNA_*dinterface*`_send_float`(*value*)
CMNA_*dinterface*`_send_double`(*value*)

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.

Auxiliary Information for Short Messages

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



where

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

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

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

Copyright © 1994 Thinking Machines Corporation

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

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

The *md* flag is 0 for a message with a relative destination address, and 1 for a message with a physical destination address. The following constants can be used to specify the *md* flag value:

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

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

The *tag* can be any value from 0 to 7 inclusive for user messages, or from 0 to 15 for supervisor messages. Message tags are described in more detail in Section 3.5.4 below. The *length* field can have any value from 1 up to **MAX_ROUTER_MSG_WORDS**.

3.4.2 Sending Long Messages

The protocol for sending a long message is the same that for short messages, except that the first word of the message must be written to a special register:

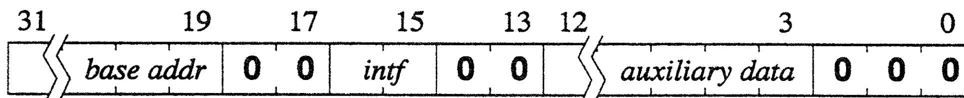
ni_*dinterface*_**send_first_long** Used for first value of long message.

and for each *dinterface* there are corresponding **send_first_long** macros:

```
CMNA_dinterface_send_first_long(tag, length, value)
CMNA_dinterface_send_first_double_long(tag, length, value)
```

Send First Long Address Format

The **send_first_long** address for a Data Network message is a 32-bit value of the form



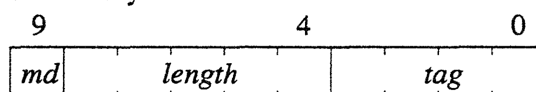
Copyright © 1994 Thinking Machines Corporation

where *intf* is the interface number (an integer from 0 to 3 representing the Data Network interface being used), and *auxiliary data* is the auxiliary information. The following *intf* values are defined:

0 - Not used
1 - DR network interface
2 - LDR network interface
3 - RDR network interface

Auxiliary Information for Long Messages

The format of the auxiliary information is



where

md is the addressing mode (0 = relative, 1 = physical).

length is the length of the message in words, excluding address word.

tag is the 4-bit tag value.

Aside from size and position, these three fields are the same as those defined above for the auxiliary information of a short message.

3.4.3 Receiving Messages

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

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

Both long and short messages are received as described in Chapter 2, by reading successive words of the message from the `recv` register. (Messages can also be received via the `ni_`*dinterface*`_status_pop` register. See Section 3.5.3.)

To receive a message from the LDR or RDR, use the network-specific reading operations described in Section 2.4.1:

```
value = CMNA_dinterface_receive_word();
value = CMNA_dinterface_receive_float();
value = CMNA_dinterface_receive_double();
```


Copyright © 1994 Thinking Machines Corporation

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.

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

3.5 The Status Registers

3.5.1 The Standard Status Registers

Each of the Data Network interfaces has two main status registers, one each for short and long messages, which contain the subfields shown below:

`ni_`*dinterface*`_status_long`	Status register for long messages.
`ni_`*dinterface*`_status`	Status register for regular messages.

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

The only difference between the `status` and `status_long` registers is that in the `status_long` register the `send_space`, `rec_length`, and `rec_length_left` fields are five bits long instead of four to accommodate the extra length of long messages.

Each of these fields has the same value in both the `status` and `status_long` registers, except where the value exceeds 15. In this case, the `status_long` field contains the correct value, while the `status` field is always 15. If the supervisor writes a value to any of the four-bit `status` fields, the corresponding

Copyright © 1994 Thinking Machines Corporation

five-bit **status_long** field is automatically updated to the same value, with a 0 for the most significant bit. The reverse is also true, with the **status** field being set to 15 as described above if the **status_long** value exceeds 15.

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

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

The **send_ok**, **send_space**, **rec_ok**, **rec_length**, and **rec_length_left** subfields are as described in Chapter 2. The **dr_rec_tag** field is described in Section 3.5.4 below, the **dr_{send,rec}_state** fields in Section 3.5.5, and the **ni_router_done_complete** flag is described in Section 3.5.6.

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

3.5.2 The "Status All" Alternate Status Register

Each Data Network interface also has an alternate status register, which gathers information about all three Data Network interfaces into a single word value:

ni_*dinterface*_status_all	Alternate status register.
ni_*firstintf*_rec_ok	Flag, indicates receipt of message.
ni_*secondintf*_rec_ok	Flag, receipt of other interface message.
ni_*dinterface*_send_ok	Flag, send OK flag of interface.
ni_*firstintf*_rec_tag	Field, tag value of LDR message.
ni_*secondintf*_rec_tag	Field, tag value of RDR message.
ni_*firstintf*_rec_length_long	Field, total length of LDR message.
ni_*secondintf*_rec_length_long	Field, total length of RDR message.
ni_*dinterface*_send_space	Field, space left in DR send FIFO.
ni_*firstintf*_rec_all_fall_down	Flag, indicates All Fall Down message.
ni_*secondintf*_rec_all_fall_down	Flag, indicates All Fall Down message.
ni_router_done_complete	Flag, indicates empty send FIFOs.

Note: Currently, there are no predefined C access routines for the **status_all** register; you must use the predefined register address constants.

Copyright © 1994 Thinking Machines Corporation

In the field names listed above, the *firstintf* and *secondintf* portions of the names are different for each network interface:

Interface	*dinterface*	*firstintf*	*secondintf*
DR	`dr`	`ldr`	`rdr`
LDR	`ldr`	`ldr`	`rdr`
RDR	`rdr`	`rdr`	`ldr`

In general, *firstintf* is the same as *dinterface*, while *secondintf* is the opposite interface in the pair of LDR and RDR. This is so that when a program is using the two half networks, the status values for the "current" and "opposite" halves of the network can be obtained from the same positions in the `status_all` register, regardless of the interface (LDR or RDR) that is in use.

The flag and field values in the `status_all` register are copied from the appropriate Data Network `status` registers, with the exception of the `rec_all_fall_down` flags that are taken from the `ni_`*dinterface*`_private` register (see Section 3.6). At all times, the value of the `status_all` register mirrors the current values available from the individual `status` registers.

3.5.3 The "Status Pop" Register

The `status_all` register also has a convenient doubleword alias:

`ni_`*dinterface*`_status_pop`	Status register, also receives messages.

The `status_pop` register is identical to the `status_all` register, except that the `status_pop` register can only be read with a doubleword operation.

When this is done, the first word of the result is the current value of the `status_all` register. The second word of the result is a value popped from the appropriate *dinterface* receive FIFO, if a value is available.

Thus, a single doubleword read of the `status_pop` register can be used to check whether a value is available for reading from the network interface, and also to get the value if there is one.

Note: The `status_pop` feature is defined only for the LDR and RDR interfaces, since it is not possible to read a value from the DR interface.

Also, there is currently no `NI_`*dinterface*`_STATUS_POP_A` register constant; use the offset value (x40) shown on the NI memory map.

Copyright © 1994 Thinking Machines Corporation

3.5.4 Message Tags

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

Tag values are not mandatory. You can, for instance, simply supply a tag value of 0 for all Data Network messages.

Tag values are primarily used for

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

To get the **rec_tag** field, use the macro

```
RECEIVE_TAG(status)
```

User/Supervisor Tag Reservation

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.

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

ni_user_tag_mask	User/supervisor tag reservation register.

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

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

Tag Fields and Interrupts

Tag values can be used to trigger interrupts; when a message with an interrupting tag value becomes available for reading in the receive FIFO, the NI signals an

Copyright © 1994 Thinking Machines Corporation

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

The following registers and register flags are used to determine which tag values cause interrupts, and how they are signaled:

`ni_rec_interrupt_mask`	Register, Supervisor tag interrupt flags.
`ni_user_rec_interrupt_mask`	Register, User tag interrupt flags.
`ni_hodgepodge`	Register containing "hodgepodge" of flags:
`ni_ldr_rec_tag_ie`	LDR supervisor tag interrupt enable.
`ni_rdr_rec_tag_ie`	RDR supervisor tag interrupt enable.
`ni_ldr_user_rec_tag_ie`	LDR user tag interrupt enable.
`ni_rdr_user_rec_tag_ie`	RDR user tag interrupt enable.

The `interrupt_mask` registers each contain 16 flags, one for each tag value. If the *n*th bit of either register is 1, it indicates that an arriving message with a tag value of *n* should signal an interrupt. However, the "supervisor" tag value register `ni_rec_interrupt_mask` has overriding contol over which tag values signal interrupts. The `ni_user_rec_interrupt_mask` register is dependent on the value of `ni_rec_interrupt_mask`; only if the *nth* bit of the `ni_rec_interrupt_mask` is set to 0, will a 1 in the corresponding bit of `ni_user_rec_interrupt_mask` cause an interrupt to be signaled.

The interrupt enable flags in the `ni_hodgepodge` register enable and disable the interface-specific supervisor and user interrupts (see Figure 12).

When a message with a tag value of *n* arrives at the LDR interface of the Data Network, the *n*th flag bit of `ni_rec_interrupt_mask` is checked. If the flag is 1, then a Green interrupt (`ldr rec tag`) is signaled. If the flag is 0, the *n*th flag bit of the `ni_user_rec_interrupt_mask` register is checked. If this flag is 1, then a Green interrupt (`ldr user rec tag`) is signaled. If the flag is 0, then no LDR interrupt is signaled. A similar method is used to determine whether to signal the `rdr rec tag` and `rdr user rec tag` interrupts when a message arrives via the RDR interface.

In all cases, if the *n*th bit of `ni_rec_interrupt_mask` is 1, the arrival of a Data Network message with tag value *n* by either interface (LDR or RDR) always signals a Green interrupt (`dr rec tag`).

Copyright © 1994 Thinking Machines Corporation

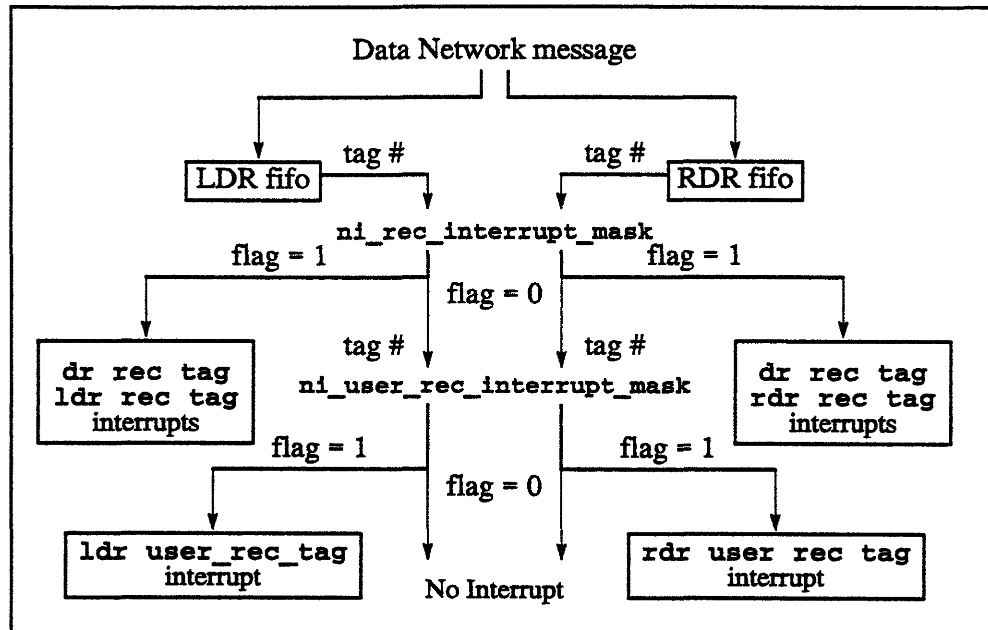


Figure 12. Tag value interrupt paths for Data Network messages

The **ni_user_rec_interrupt_mask** register is both readable and writable by user programs, but the interrupt enable flags **ni_ldr_user_rec_ie** and **ni_rdr_user_rec_ie** are writable only by supervisor programs. The intent of this is to allow the supervisor to use the user interrupt enable flags as "permission" bits — by setting either of the two **user_rec_ie** flags, the supervisor grants to user programs the ability to turn interrupts on and off for all tags not already reserved for supervisor interrupts. This avoids the need for a supervisor call whenever a user program wants to enable or disable user interrupts.

The **ni_rec_interrupt_mask** is also used to inhibit user access to supervisor messages. If the *n*th bit of the **ni_rec_interrupt_mask** register is 1, then if a Data Network message with a tag value of *n* arrives (say via the LDR interface) the message is effectively invisible and inaccessible to user programs. Specifically, the **rec_ok** flag will be 0 when read by the user, and an attempt by the user to read from the receive FIFO will fail, as though the FIFO were empty. When the supervisor attempts to read the message, however, the **rec_ok** flag will have the correct value, and reading from the receive FIFO will receive the message as usual.

Copyright © 1994 Thinking Machines Corporation

Using CMOST Commands to Set Up NI Interrupt Handlers

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 the **CMOS_signal** function, you must **#include** the file **cm/cm_signal.h**. For more information on **CMOS_signal**, see the UNIX manual page for the function. (This is included as Appendix E to this document.)

Tag Fields and the Message-Counting Registers

Tag fields also allow system software to automatically maintain a count of messages sent and received by the NI. This is a key part of the network-done feature of the Control Network (see Section 4.2.9). It allows the NI to determine quickly when the Data Network is clear of user messages. Two registers are used to control this message-counting feature:

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

Copyright © 1994 Thinking Machines Corporation

Message Count Disabling

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

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

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

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

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

Negative Message Count Interrupts

If the sum of the `message_count` registers for all nodes becomes negative, it means that either a message was lost in transit or was counted incorrectly. If the global `message_count` sum is negative when a Data Network operation is attempted, a Yellow interrupt (`dr count negative`) is signaled. (See Section B.3.4 in Appendix B.)

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

Copyright © 1994 Thinking Machines Corporation

3.5.5 The Send and Receive State Fields

The DR interface is mutually exclusive with the LDR and RDR interfaces. It is an error to try to write a message to the DR send FIFO while there is a partially completed message in either the LDR or RDR send FIFOs. Likewise, having a partially completed message in the DR send FIFO makes it an error to try to send a message via the LDR or RDR FIFOs. In either case, the status registers and FIFOs of the excluded interface(s) are invalidated.

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

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

There is also a corresponding **`ni_dr_rec_state`** field that you can use to determine which receive interfaces are in use. (However, because the DR interface cannot be used to receive messages, this field is not as useful as **`ni_dr_send_state`**.) The value of the **`ni_dr_rec_state`** field is again an integer from 0 to 2:

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

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 accessible only from the **`ni_dr_status`** register).

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

Copyright © 1994 Thinking Machines Corporation

3.5.6 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.

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);
```

As noted above, the message-counting register `ni_dr_message_count` also plays a part in the network-done feature. For more information on network-done messages, see Section 4.2.9.

3.6 The Private Register

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

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

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

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

Copyright © 1994 Thinking Machines Corporation

3.7 All Fall Down Mode

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

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

3.7.1 Triggering All Fall Down Mode

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

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

3.7.2 Detecting All Fall Down Mode Messages

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

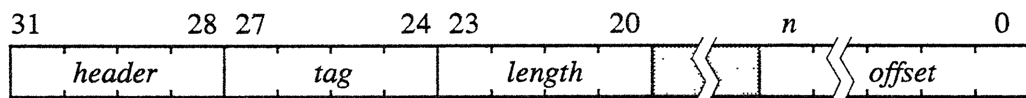
You can also have the NI trigger an interrupt when an All Fall Down message becomes available in the receive FIFO (either by arriving at an empty FIFO, or by being brought forward after a preceding message has been read out). If the interrupt enable flag **ni_all_fall_down_ie** is set, the arrival of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

Copyright © 1994 Thinking Machines Corporation

3.7.3 Resending All Fall Down Mode Messages

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

The All Fall Down address word has the following format:



where

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

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

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

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

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

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

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

Copyright © 1994 Thinking Machines Corporation

3.8 Interrupt Enable Flags

There are two interrupt enable flags in the `ni_`*dinterface*`_private` register.

The `ni_sfifo_goes_empty_ie` flag controls whether a Green interrupt (`send FIFO empty`) is signaled when any Data Network send FIFO goes empty. The `ni_rdone_complete_ie` flag controls whether an Orange interrupt (`router done complete`) is signaled when a network-done operation completes.

3.9 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.

The Data Network has a large capacity for messages from nodes, but the sheer number of nodes connected to it can overwhelm it if the nodes send messages into the network without attempting to receive them. Your code should be biased toward removing messages from the network rather than adding them. Your code should also provide fair opportunities for both receiving and sending, where "fair" means the ratio between the two 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.

Copyright © 1994 Thinking Machines Corporation

3.10 Examples

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

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

Sending and Receiving a Message

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

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

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

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


Copyright © 1994 Thinking Machines Corporation

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 functions above 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();
```


Copyright © 1994 Thinking Machines Corporation

```
            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);
```


Copyright © 1994 Thinking Machines Corporation

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_ROUT-
ER_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]);
```


Copyright © 1994 Thinking Machines Corporation

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);
}
```


Copyright © 1994 Thinking Machines Corporation

Chapter 4

The Control Network

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

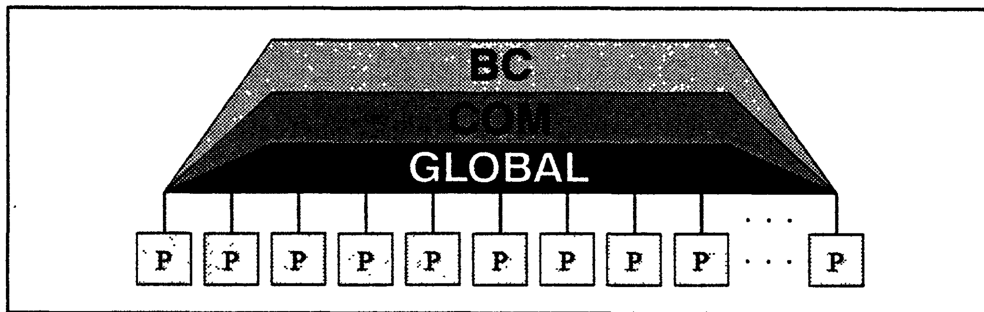


Figure 13. The three interfaces of the Control Network: BC, COM, and global.

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.

Copyright © 1994 Thinking Machines Corporation

4.1 The Broadcast Interface

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

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

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

4.1.1 Broadcast Register Interfaces

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

The following NI registers form the broadcast interface:

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

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

`bc` - user broadcast interface   `sbc` - supervisor broadcast interface

The purpose and use of each of these registers is described in the sections below. Figure 14 contains a memory map showing the relative locations of these registers in the user and supervisor areas.

Copyright © 1994 Thinking Machines Corporation

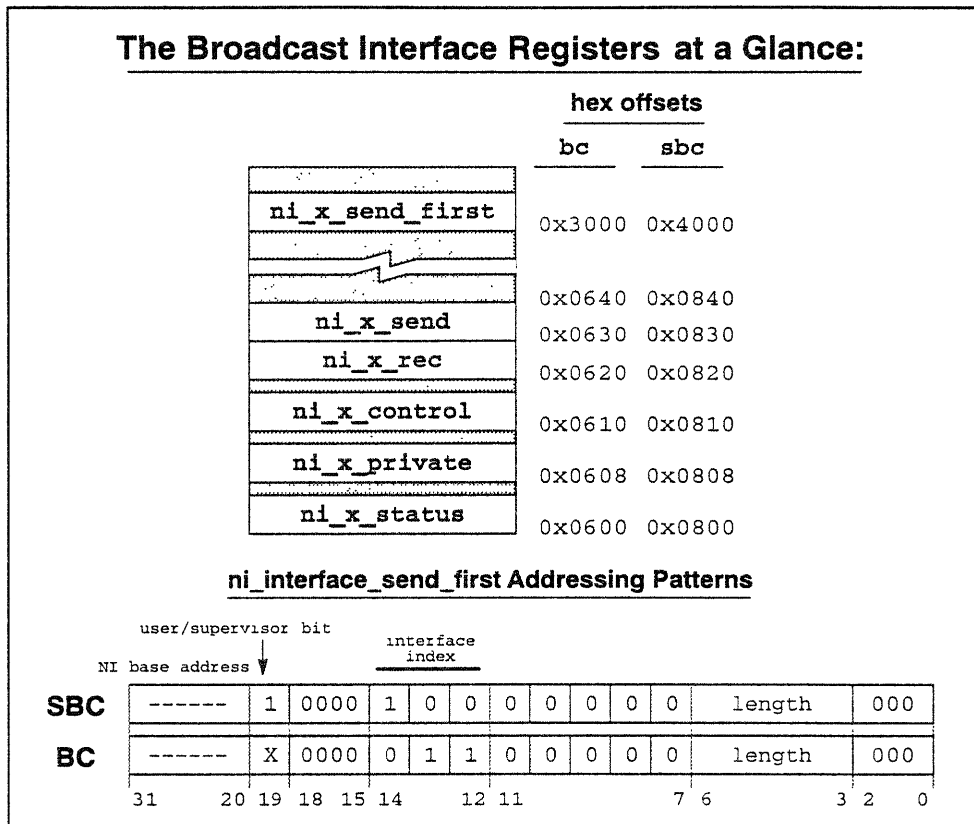


Figure 14. NI registers associated with each of the broadcast interfaces.

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

Copyright © 1994 Thinking Machines Corporation

Broadcast messages are atomic with respect to sending; a broadcast message is not transmitted until all its component words have been written to the send FIFO. Broadcast messages are not atomic in transit, however. A multiword message may be split in transit into two or more smaller messages. In addition, 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 (that is, when it examines the `status` register to determine the length of message that is available), the length of the message is fixed, and a new "message" is formed behind it in the FIFO from any words that arrive while the first message is being read out.

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

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

Implementation Note: Each broadcast interface's `private` register includes a supervisor flag, `ni_send_enable`, which controls whether broadcast sending is enabled via that interface. (See Section 4.1.8 for a description of these flags.)

4.1.3 Sending Broadcast Messages

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

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

Copyright © 1994 Thinking Machines Corporation

The following FIFO registers are used to send messages:

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

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

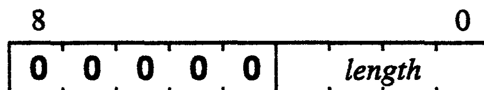
`CMNA_bc_send_first(`*length*`,` *value*`)`
`CMNA_bc_send_first_double(`*length*`,` *value*`)`

`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.

4.1.4 Auxiliary Information

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



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

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

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

4.1.5 Receiving Broadcast Messages

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

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

Copyright © 1994 Thinking Machines Corporation

To receive a message from the broadcast interface, use the network-specific reading operations described in Chapter 2:

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

4.1.6 The Broadcast Status Register

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

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

The meanings of these 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 2.5.4).

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

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

Note: As described in Section I.4 in the Appendixes, the bit length of the **length_left** field has changed; to access this field, you should use the macro **BC_RECEIVE_LENGTH**(*status-value*) for both the BC and SBC interfaces.

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

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

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

Copyright © 1994 Thinking Machines Corporation

4.1.7 Abstaining from the Broadcast Interface

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

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

The address constant for the abstain register is **bc_control_reg**. You can use the macros described in Section 2.6.3 to read and write the abstain flag:

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

4.1.8 The Broadcast Private Register

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

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

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

The Send Enable Flag

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

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

Copyright © 1994 Thinking Machines Corporation

Implementation Note: The CMOST operating system sets the **`send_enable`** flag for the broadcast interface (but not the supervisor interface) to 0 by default. This flag must be set to 1 to permit broadcasting of messages. To turn on this flag, you can use the following C macro call; this call must be made prior to any broadcast interface operations:

```
CMNA_participate_in(NI_BC_SEND_ENABLE);
```

4.1.9 Broadcast Interface Examples

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

Sending and Receiving a Message

These functions send and receive messages via the broadcast interface. The message is assumed to be composed of *length* words of data starting at the memory 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()));
}

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);
 }
```


Copyright © 1994 Thinking Machines Corporation

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.

These are the supported operations:

- 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 8.2.6.)

Copyright © 1994 Thinking Machines Corporation

4.2.1 The Combine Register Interface

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

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

The purpose and use of each of these registers is described in the sections below. Figure 15 contains a memory map showing the relative locations of these registers in the user and supervisor areas.

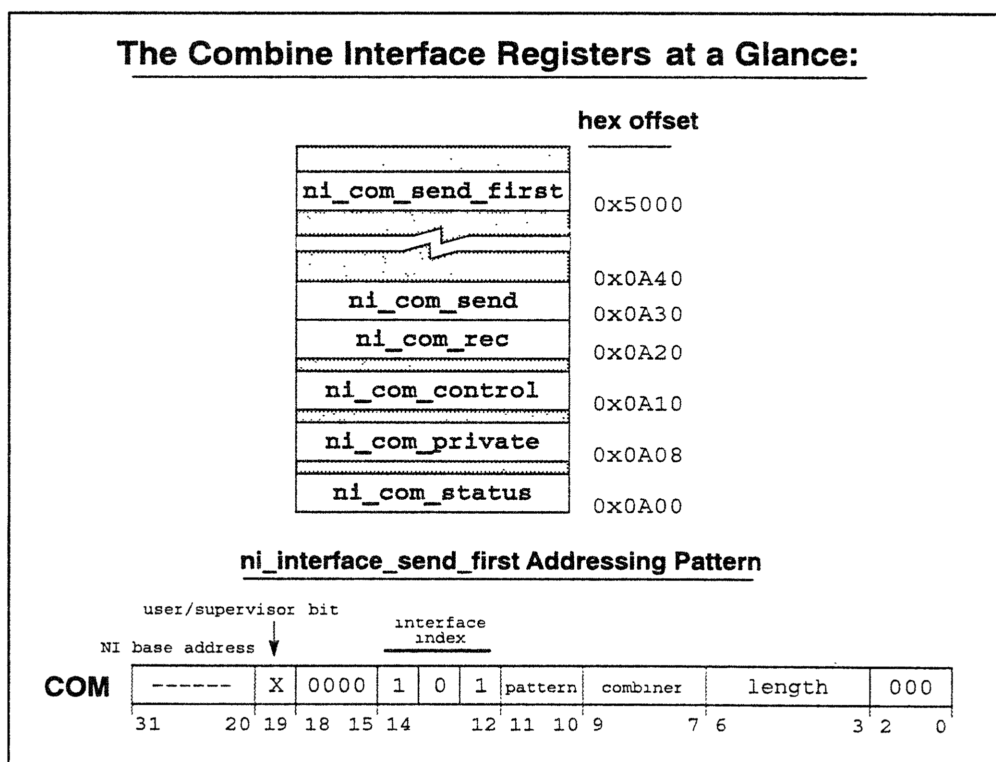


Figure 15. NI registers associated with the combine interface.

Copyright © 1994 Thinking Machines Corporation

4.2.2 Combine Messages

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

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

The order of combine messages is strictly preserved in transit. With the exception of the network-done operation, which uses a different mechanism, the results of combine operations are delivered into the receive FIFO in the same order the 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 8.1.2.

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 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.

Copyright © 1994 Thinking Machines Corporation

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*)

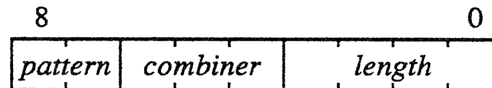
For the **send_first** macros, the *length* argument is the length of the message in words, and *value* is the first value of the message. The *combiner* and *pattern* arguments are described in the sections below, covering each of the possible combine operations.

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

4.2.4 Auxiliary Information

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

The auxiliary data field of the message has the form



where

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

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

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

Copyright © 1994 Thinking Machines Corporation

To construct a send_first address, add the following values:

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

4.2.5 Legal Combiner and Pattern Values

For scans and reductions, these are the legal *pattern* and *combiner* values:

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

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

A *pattern* value of 0, together with a *combiner* value of 5, specifies a network-done operation, described later in this chapter. The *combiner* values 6 and 7 are not currently used.

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

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

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

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

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

Copyright © 1994 Thinking Machines Corporation

4.2.6 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 4.2.9).

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 Chapter 2:

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

4.2.7 The Combine Status Register

The combine status register contains the following subfields:

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

The **send_ok**, **send_space**, **send_empty**, **rec_ok**, **rec_length**, and **rec_length_left** subfields are as described in Chapter 2, and you can obtain the values of these sub-fields by using the generic field extractors described in that chapter. The remaining flag, **com_scan_overflow**, is described in Section 4.2.8.

Use this C macro to get the value of the combine status register:

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


Copyright © 1994 Thinking Machines Corporation

4.2.8 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. A scan or reduction message is from 1 to 5 words in length, representing a value to be combined.

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

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

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

The legal *combiner* and *pattern* values for scans and reductions 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 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.

Copyright © 1994 Thinking Machines Corporation

Scanning with Segments

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

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

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

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

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

Note: It is an error to change the value of the **`scan_start`** flag while the combine send FIFO is not empty. (For example, you can't toggle the **`scan_start`** flag in the middle of a series of pipelined combine operations.)

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 multiword message, the value of **`ni_scan_start`** when the first value is written applies to the entire message. Altering the flag after the first value is written has no effect on the message.

Addition Scan Overflow

Addition scans on large values can cause arithmetic overflow in some nodes. The **`ni_com_scan_overflow`** flag in the **`status`** register indicates whether the current scan result has suffered arithmetic overflow.

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

Copyright © 1994 Thinking Machines Corporation

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

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

4.2.9 Network-Done Messages

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

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

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

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.

Copyright © 1994 Thinking Machines Corporation

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 C 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 a Bus Error.

How Network-Done Works...

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

The network-done operation makes use of the `ni_dr_message_count` register of the Data Network to determine when the Data Network is clear. As described in Section 3.5.4, each node increments this register when it sends a message, and decrements the register when it receives a message. (Not counting, of course, messages for which counting is disabled by a 0 flag in `ni_count_mask`.)

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

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

Copyright © 1994 Thinking Machines Corporation

...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 4.2.10 for a discussion of the combine interface's abstain flags.)

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

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

4.2.10 Abstaining from the Combine Interface

The combine interface has two abstain flags that you can use to cause the NI to abstain from combine interface transactions.

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

Setting the `ni_rec_abstain` flag to 1 causes the NI to discard any arriving combine interface messages, and allows any messages sent by other nodes to complete without the participation of the abstaining node.

In 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 (with the exception of reduction operations — see below).

Copyright © 1994 Thinking Machines Corporation

You can use the abstain flag macros described in Section 2.6.3 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);
```

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

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

The Reduction Receive Abstain Flag

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

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

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.

Copyright © 1994 Thinking Machines Corporation

4.2.11 The Combine Private Register

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

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

The `rec_ok_ie`, `lock`, `rec_stop`, and `rec_full` subfields are as described in Chapter 2. The `ni_com_scan_overflow_ie` flag is described in Section 4.2.8. The remaining fields are described in the sections below.

Empty Receive FIFO Interrupt

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

Clearing the Combine Send FIFO

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

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

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

Copyright © 1994 Thinking Machines Corporation

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

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

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

Specifically:

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

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

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

Copyright © 1994 Thinking Machines Corporation

4.2.12 Combine Interface Examples

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

Sending and Receiving a Combine Message

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

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

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

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


Copyright © 1994 Thinking Machines Corporation

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 );
```


Copyright © 1994 Thinking Machines Corporation

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);
```


Copyright © 1994 Thinking Machines Corporation

4.3 The Global Interface

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

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

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

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

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

The Global Interface Registers at a Glance:

Register	hex offset
ni_sync_global_send	0x00C0
ni_hodgepodge	0x00B8
ni_async_sup_global	0x00B0
ni_async_global	0x00A8
ni_sync_global_abstain	0x0098
ni_sync_global	0x0090

Figure 16. NI registers associated with the global interface.

Copyright © 1994 Thinking Machines Corporation

4.3.1 The Three Global Register Interfaces

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

Synchronous global interface:

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

Aynchronous global interface:

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

Supervisor aynchronous global interface:

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

The purpose and use of these registers is described in the sections below, and Figure 16 contains a memory map showing their relative locations in NI memory.

4.3.2 The Synchronous Global Interface

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

The following registers and flags form the synchronous global interface:

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

Copyright © 1994 Thinking Machines Corporation

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's operations:

ni_sync_global_abstain	Status register, contains global abstain flag.

When the **ni_sync_global_abstain** flag is set to 1, synchronous global messages complete without the node's participation (as if the node has sent the message with its **ni_sync_global_send** flag set to 0). 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: As with all abstain flags, **ni_sync_global_abstain** should be changed only when there is no global message pending. A Bus Error is signaled if the abstain flag is modified when the **ni_sync_global_complete** flag is 0. Also, a Bus Error is signaled if the **ni_sync_global_send** register is written while the abstain flag is 1.

Copyright © 1994 Thinking Machines Corporation

Synchronous Global Receive Interrupt

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

Supervisor Operations for the Synchronous Global Interface

The supervisor can write a new value into the `ni_sync_global_rec` flag when the flag `ni_sync_global_complete` is set to 1. If `ni_sync_global_rec` is written when `ni_sync_global_complete` is 0, a Bus Error (`bad memory access`) is signaled.

Implementation Note: Even when `ni_sync_global_rec_ie` is 1, the supervisor's writing a 1 to `ni_sync_global_rec` does not signal the corresponding Green interrupt (`sync global rec`).

The supervisor can take control of the synchronous global interface (for example, during a context-switch) as follows. Each node in the partition to be context-switched should save the values of the `sync_global_complete`, `sync_global_rec` and `sync_global_send` flags. All nodes for which `ni_sync_global_complete` is 1 should write a 1 to the `ni_sync_global_send` flag, thus completing any pending operation.

To restore the state of the synchronous global interface, all nodes restore the value of the `sync_global_send` flag by writing the saved value back into it. When the resulting global operation completes (`ni_sync_global_complete` becomes 1), all nodes restore the saved value of the `ni_sync_global_rec` flag. All nodes with a saved value of 0 for `ni_sync_global_complete` write the `ni_sync_global_send` flag again to restart the interrupted global operation. Control can then be handed back to user code.

4.3.3 The Asynchronous Global Interface

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

Copyright © 1994 Thinking Machines Corporation

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

The following registers and flags form the asynchronous global interface:

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

Sending and Receiving Messages

Because the asynchronous global interface operates continuously, 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();
```


Copyright © 1994 Thinking Machines Corporation

Asynchronous Global Receive Interrupt

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

4.3.4 The Supervisor Asynchronous Global Interface

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

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

The following register and flags form the supervisor asynchronous interface:

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

Sending and Receiving Messages

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

Supervisor Asynchronous Global Receive Interrupt

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

Copyright © 1994 Thinking Machines Corporation

4.3.5 Global Interface Examples

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

Using the Synchronous Global 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());
}
```


Copyright © 1994 Thinking Machines Corporation

Chapter 5

NI Interrupts

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

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

Interrupts are signaled by one of two different methods:

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

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

5.1 Interrupt Classes

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

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

Copyright © 1994 Thinking Machines Corporation

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

The possible Red interrupts are:

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

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

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

  The possible Orange interrupts are:

`timer interrupt`	NI timer reached `interrupt_now`.
`rdone complete`	Router done complete interrupt.
`bc interrupt orange`	Orange broadcast interrupt.

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

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

  The possible Yellow interrupts are:

`dr count negative`	Negative DR message count.
`bc or com collision`	Conflict in broadcast/combine ops.
`com abstain changed`	Flag changed while interface in use.
`bad relative address`	Address outside partition, etc.
`bad memory access`	Bus Error signaled as interrupt.
`message too long`	Data Network message too long.
`bc interrupt yellow`	Yellow broadcast interrupt.

Copyright © 1994 Thinking Machines Corporation

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

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

  The possible Green interrupts are:

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

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

  There is basically one flavor of Bus Error:

`bad memory access`	Meaningless or illegal reference.

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

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

Copyright © 1994 Thinking Machines Corporation

5.1.1 Disabling Bus Errors

Some Mbus devices do not respond well to the NI signaling a bus error. In order to allow the NI to be used in systems that include such devices, the NI can optionally "signal" a bus error as a Yellow interrupt (`bad_memory_access`). This feature is controlled by a flag in the `ni_hodgepodge` register, `ni_disable_bus_error`. This flag is turned off by default, and by an NI reset, to provide backward compatibility.

5.2 Interrupt Pathways

The four colored interrupts (Red, Orange, Yellow, and Green) result from a number of different causes. Figure 17 shows the pathways followed by the various types of interrupts on their way to the microprocessor. These pathways are described in detail in the sections below.

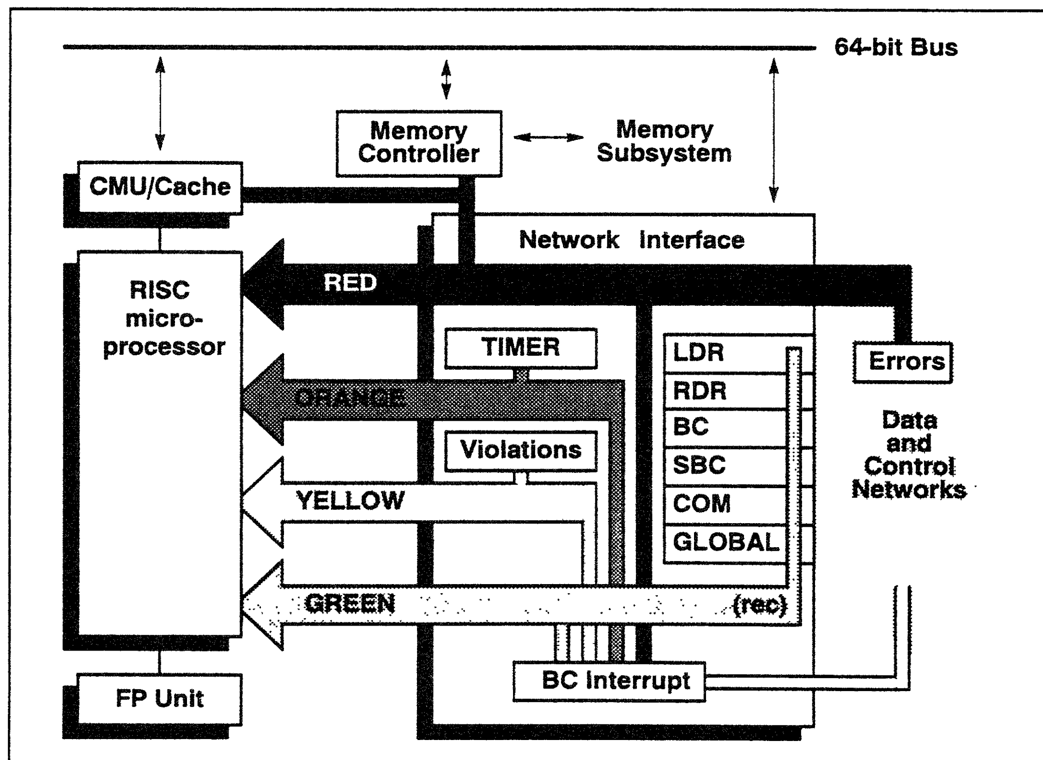


Figure 17. The possible pathways for colored interrupts.

Copyright © 1994 Thinking Machines Corporation

5.2.1 Red Interrupts

The Red interrupts are of two varieties:

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

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

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

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

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

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

- MC errors are caused by either a fault in the MC itself (usually fatal), or (if the CM-5 has the vector unit option installed) by an error signaled from one or more of the vector units. In either case, it is necessary to examine the state of the appropriate hardware to determine the actual cause of the interrupt.

- CMU errors are only caused by bad memory writes (typically memory writes to illegal addresses) and are always fatal. CMU errors are asynchronous, so that the error is not signaled until some time after the offending write instruction.

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

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

One is caused by NI chip problems:

`internal fault` — NI chip fault.

Copyright © 1994 Thinking Machines Corporation

And one can be signaled by software:

`bc interrupt red` — Red broadcast interrupt.

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

5.2.2 Orange Interrupts

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

`timer interrupt` — Timer alarm interrupt.

One is caused by the completion of a Data Network network-done operation:

`rdone complete` — Network-done-complete interrupt.

And the remaining interrupt can be signaled by software:

`bc interrupt orange` — Orange broadcast interrupt.

5.2.3 Yellow Interrupts

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

`com abstain changed` — Illegal abstain flag change.
`bc or com collision` — Multiple message collision.
`bad relative address` — Illegal DR destination.
`dr count negative` — Negative DR message count.
`bad memory access` — Bus Error signaled as interrupt.
`message too long` — Data Network message too long.

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

`bc interrupt yellow` — Yellow broadcast interrupt.

Copyright © 1994 Thinking Machines Corporation

5.2.4 Green Interrupts

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

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

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

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

`scan overflow`	— Combine interface add-scan overflow.

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

`dr rec tag`	— DR message arrived with interrupting tag.
`ldr rec tag`	— LDR message arrived with interrupting tag.
`rdr rec tag`	— RDR message arrived with interrupting tag.
`ldr user rec tag`	— LDR message with user interrupt tag received.
`rdr user rec tag`	— RDR message with user interrupt tag received.
`dr rec all fall down`	— DR All Fall Down mode message received.
`com rec empty`	— Combine receive FIFO empty.
`dperr`	— Vector unit error.
`sfifo empty`	— Data Network send FIFO empty.

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

`bc interrupt green`	— Green broadcast interrupt.

Copyright © 1994 Thinking Machines Corporation

5.3 The Interrupt Cause and Clear Registers

There are six NI registers that you can use to determine which interrupt(s) have been signaled, to clear the interrupts once you have finished handling them, and to force interrupts to be signaled when necessary:

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

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

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

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

Interrupts can be triggered artificially by writing to either of the **ni_interrupt_set** or **ni_interrupt_set_green** registers. The value written to the register should contain one bits in locations corresponding to the interrupts that are to be signaled. In the case of an interrupt with an enable bit, the interrupt can be signaled even if the interrupt is currently disabled. It is not possible to read the **ni_interrupt_clear**, **ni_interrupt_clear_green**, **ni_interrupt_set** or **ni_interrupt_set_green** registers.

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

Copyright © 1994 Thinking Machines Corporation

5.4 Interrupt Levels

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

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

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

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

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

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

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

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

Copyright © 1994 Thinking Machines Corporation

5.5 Broadcast Interrupts

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

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

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

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

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

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

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

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

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

Copyright © 1994 Thinking Machines Corporation

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

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

5.6 Recovering from Interrupts

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

Copyright © 1994 Thinking Machines Corporation

Chapter 6

Other NI Interfaces and Features

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

6.1 The "Hodgepodge" Register

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

ni_hodgepodge	Register with "hodgepodge" of flags:
ni_sync_global_rec_ie	Sync global receive interrupt enable.
ni_global_rec_ie	Asynch global receive intrpt. enable.
ni_supervisor_global_rec_ie	Supervisor asynch. rec. intrpt. enable.
ni_interrupt_send_ok	Broadcast interrupt send ok flag.
ni_interrupt_rec_enable	Broadcast interrupt receive enable.
ni_flush_complete	Combine flush complete flag.
ni_timer_ie	NI timer interrupt enable flag.
ni_configuration_complete	Configuration complete flag.
ni_cn_stop_send	Control Network disable flag.
ni_disable_bus_error	Bus Error disable flag.
ni_ldr_rec_tag_ie	LDR supervisor tag interrupt enable.
ni_rdr_rec_tag_ie	RDR supervisor tag interrupt enable.
ni_ldr_user_rec_tag_ie	LDR user tag interrupt enable.
ni_rdr_user_rec_tag_ie	RDR user tag interrupt enable.
ni_msg_too_long_ie	Message too long interrupt enable.

Copyright © 1994 Thinking Machines Corporation

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

6.2 Node Address Registers

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

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

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

6.3 NI Chunk Table and Address Translation

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

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

6.3.1 Node Address Translation

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

For the Curious: A side effect of the use of the chunk table is that it implicitly divides the Data Network up into "partitions" of nodes. That is, there is no hard-

Copyright © 1994 Thinking Machines Corporation

ware restriction preventing a Data Network message from traveling between partitions; it is the chunk tables that determine whether a relative address is legal for a given partition of nodes.

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

First, the relative address is compared with the **ni_partition_size** register, to determine whether it is legal for the current partition. (If the relative address is greater than or equal to **ni_partition_size**, the address is guaranteed not to correspond to a node in the current partition, and an error is signaled.)

Next, the relative address is split into two parts (see Figure 18).

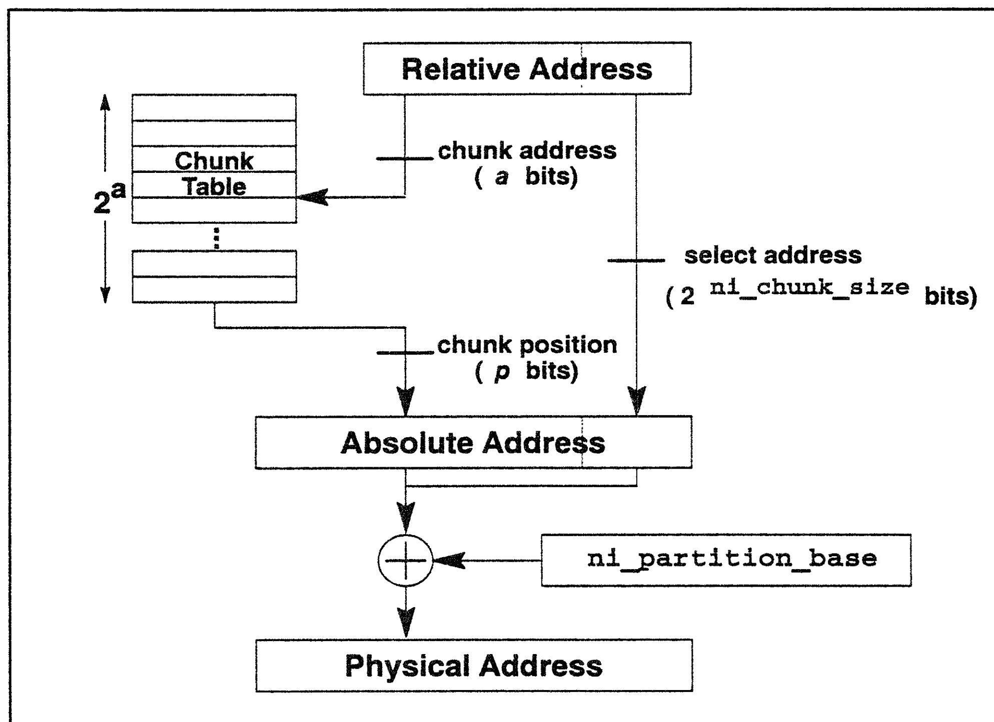


Figure 18. Translation from relative addresses to physical addresses.

The two parts of the address are:

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

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

Copyright © 1994 Thinking Machines Corporation

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

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

6.3.2 Chunk Sizes and Address Allocation

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

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

ni_chunk_size	Size of chunks referenced by the chunk table.

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

The number of bits in a select address is $2^{\texttt{ni_chunk_size}}$. As a result, the number of physical addresses in a chunk is $4^{\texttt{ni_chunk_size}}$, and this means that the number of possible relative addresses (in other words, the number of accessible nodes) is $2^a * 4^{\texttt{ni_chunk_size}}$. This also means that the total physical address space accessible through the chunk table is $2^p * 4^{\texttt{ni_chunk_size}}$. Thus, the accessible physical address space is always 2^{p-a} times the size of the relative address space. This extra "unused" space between chunks is used to isolate regions of broken or missing hardware. (See Figure 19.)

Copyright © 1994 Thinking Machines Corporation

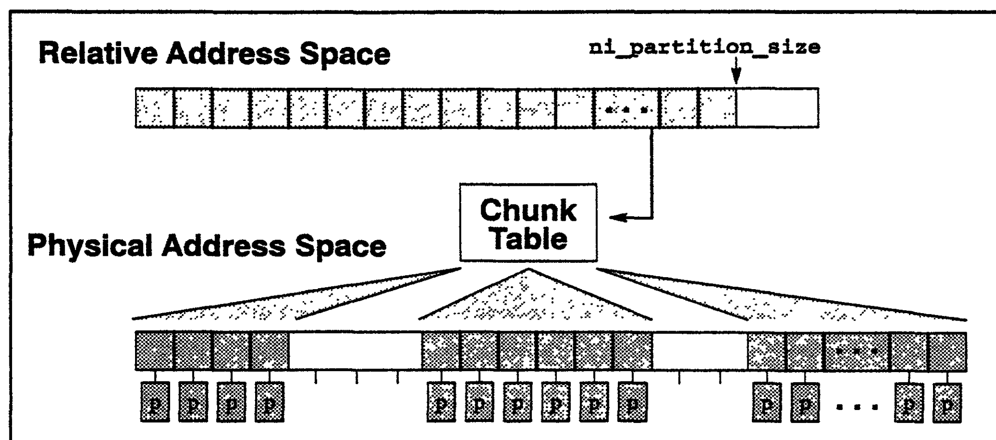


Figure 19. The chunk table is used to map contiguous relative addresses onto discontiguous physical addresses.

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

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

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

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

Copyright © 1994 Thinking Machines Corporation

6.3.3 Modifying the Chunk Table

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

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

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

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

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

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

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

6.4 Combine Interface Flush

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

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

Copyright © 1994 Thinking Machines Corporation

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

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

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

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

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

6.5 The NI Timer

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

The following registers and flags form the timer interface:

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

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

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

Copyright © 1994 Thinking Machines Corporation

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

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

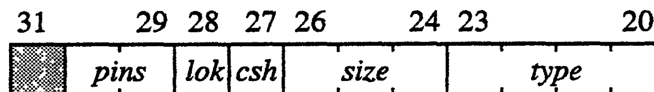
6.6 The Bad Address Register

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

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

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

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



where

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

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

Copyright © 1994 Thinking Machines Corporation

6.7 NI Partition Configuration

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

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

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

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

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

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

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

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

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

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

Copyright © 1994 Thinking Machines Corporation

6.8 Disabling the Control Network

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

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

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

The **stop_send** flag is generally used only during an NI Reset (see Section 6.10) when it is necessary to totally disable the Control Network. When the **stop_send** flag is 1, the Control Network is disabled. When the **stop_send** flag is set to 0, normal network operations resume.

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

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

6.9 NI Serial Number

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

ni_serial_number	Version serial number of NI chip.

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

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

Copyright © 1994 Thinking Machines Corporation

6.10 NI Reset

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

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

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

- **ni_disable_bus_error** is negated.
- **ni_longest_dr_message** is set to a value of 5.
- All abstain and lock flags are set to 1, thus isolating the NI from all networks. These flags are:

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

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

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

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

Copyright © 1994 Thinking Machines Corporation

Chapter 7

Writing NI Programs

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

7.1 Transferring Data between Nodes and the PM

As described in Section 3.3, 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:

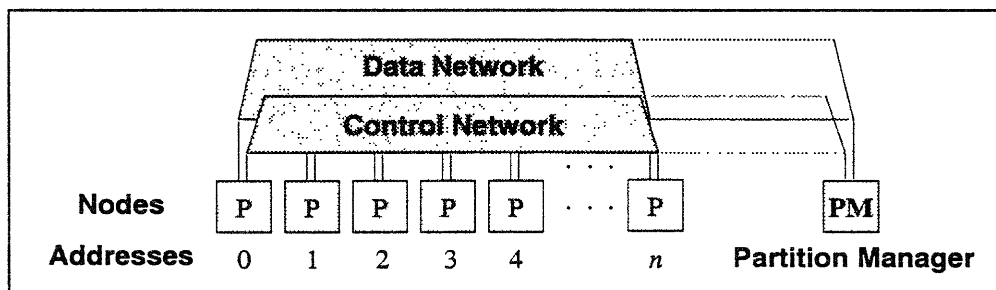


Figure 20. The partition manager stands apart from the partition it manages.

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

Copyright © 1994 Thinking Machines Corporation

7.1.1 Sending Messages from the PM to Nodes

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

For example:

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

Each of the nodes should perform two broadcast 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.

Copyright © 1994 Thinking Machines Corporation

7.1.2 Sending Messages from Nodes to the PM

Sending a message from a node to the PM is almost as straightforward, but involves two 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 7.2, below.)

Next, the PM broadcasts a message containing the address of a processing node, as in the **PM_send_to_NODE** example above. The nodes respond by signaling a combine 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 7.2.)

Copyright © 1994 Thinking Machines Corporation

7.1.3 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.

7.1.4 For the Curious: Using the Data Network

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

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

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

```
int *source, length, tag
CMNA_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 7.1.2.

Sending messages to and from the PM via the Data Network is primarily useful when 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.

Copyright © 1994 Thinking Machines Corporation

7.2 Setting the Abstain Flags

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

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

With these factors in mind, here is 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);
}
```


Copyright © 1994 Thinking Machines Corporation

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!)

7.3 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 4.1.8, 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.

Copyright © 1994 Thinking Machines Corporation

7.4 NI Program Structure

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

An NI program consists of three files:

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

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

7.4.1 The cmna.h Header File

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

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

This header file contains **`#include`** directives that load the other files needed to define the NI programming tools described in this manual. **Note:** If you plan to call **`CMOS_signal()`** (see Section 3.5.4), you must also **`#include`** the header file **`<cmsys/cm_signal.h>`**.)

7.4.2 Partition Manager Code

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

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

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

Copyright © 1994 Thinking Machines Corporation

7.4.3 Node Code

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

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

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

The Node's "Main" Routine

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

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

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

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

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

Copyright © 1994 Thinking Machines Corporation

7.4.4 Interface Code

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

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

```
void node_program();
```

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

This is the reason that 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.

7.5 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);
```


Copyright © 1994 Thinking Machines Corporation

```
    /*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);
```


Copyright © 1994 Thinking Machines Corporation

```
/* 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();
```


Copyright © 1994 Thinking Machines Corporation

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);
}
```


Copyright © 1994 Thinking Machines Corporation

```
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();
}
```


Copyright © 1994 Thinking Machines Corporation

7.6 Compiling and Executing an NI Program

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

To compile an NI program you must:

- preprocess the interface file by calling **sp-pe-stubs**
- compile the resulting file, as well as the PM and node routine files
- link the three object files together with the CM linking program **cmld**

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

First, preprocess the interface code file:

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

Next, compile the three code files:

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

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

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

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

Copyright © 1994 Thinking Machines Corporation

7.6.1 A Simple Compiling Script

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

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

```
#! /usr/bin/csh -e -f
# 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
```


Copyright © 1994 Thinking Machines Corporation

7.6.2 Compiling and Running the Program

Note: The following examples assume that you are currently logged in to 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.
```

7.6.3 On-Line Code Examples

As of Version 7.1.3 of the CM system software, there are on-line 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.

Copyright © 1994 Thinking Machines Corporation

Chapter 8

NI Programming Issues

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

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

8.1 Performance Hints

8.1.1 NI Register Operation Times

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

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

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

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.

Copyright © 1994 Thinking Machines Corporation

8.1.2 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 singleword values, but involves only one register operation. Likewise, reading a doubleword from a register is the same as reading two singlewords.

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 doubleword 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;
```


Copyright © 1994 Thinking Machines Corporation

```
        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 */
    }
```


Copyright © 1994 Thinking Machines Corporation

8.1.3 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.

8.1.4 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. As noted in Section 2.6.5, 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.

Copyright © 1994 Thinking Machines Corporation

8.2 Potential Programming Traps and Snares

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

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

8.2.1 Address Calculation on the Partition Manager

On any of the processing nodes, the `NI_BASE` address (the base address of the NI register region) is constant, and furthermore is set to a value that is zero in all the low-order bits used for the send-first auxiliary data fields. Thus, it is possible on the nodes to logically IOR the auxiliary data with the base address to get the final result.

On the partition manager, however, the base address of the NI register region can be any arbitrary address assigned by the operating system. Hence, on the partition manager you *must* use arithmetic addition when combining auxiliary data with the base address of a send-first register.

As a point of style, it is simplest to just use addition in all cases, as this will work on either the PM or the nodes. (This is the usage shown in this document.)

8.2.2 Pay Attention to Data Network Addresses

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

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

Implementation Note: Currently, there is an additional restriction on the most significant (19th) bit of the address — it too must be 0, or an error will result.

Copyright © 1994 Thinking Machines Corporation

8.2.3 "Middle" Data Network Interface Restrictions

Because of design limitations, it is not possible to receive messages via the "middle" Data Network interface. This is a permanent restriction — the corresponding **recv** registers in fact do not exist in the 2.2 version of the NI.

8.2.4 Make Sure Doubleword Data Is Doubleword-Aligned

C Programmers: This is also mentioned in the performance section above, but it doesn't hurt 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.

8.2.5 Order Is Important in Combine Messages

As noted in Section 4.2.8, for scan messages longer than one word, the order in which the words of the message are written depends on the combine operation:

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

8.2.6 Broadcast and Combine Interface Conflicts

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

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

Copyright © 1994 Thinking Machines Corporation

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 broadcast messages from backing up in the network 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);
```

8.2.7 Broadcast Enabling

As noted in Section 4.1.8, each broadcast interface has a **send_enable** flag. These flags are set to 0 by default in the CMOST operating system, and must be set to 1 before broadcasts are used. The CMOST system call to set these flags is:

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

8.2.8 Combine Interface Pipelining Restriction

As noted in Section 8.1.2, pipelined combine operations cannot be started using doubleword operations. However, you *can* use doubleword reads with pipelined operations, and doubleword writes *are* permitted for non-pipelined combine operations.

8.2.9 Restriction on Scan Segment Start Flag

As noted in Section 4.2.8, it is an error to change the state of the **ni_scan_start** register while the combine send FIFO is not empty.

Copyright © 1994 Thinking Machines Corporation

8.2.10 Be Careful When Altering Abstain Flags

As mentioned in Section 2.6.5, some programming systems use the abstain flags for their own purposes. When you alter the abstain flags, you must save the original settings and restore them before handing control back to these systems. Failing to do so can cause user or OS code to signal errors that are hard to trace.

8.2.11 Simulating Receipt of Messages

As noted in Section 3.4.3, a hardware defect in the NI chip does not allow `recv` registers to be written by the supervisor. The workaround is for a node to send a message into the network using its own address as the destination. Assuming the network is clear (as it is, for example, during context switches) this causes the message to be delivered to the front of the node's receive queue.

8.2.12 Message Too Long Interrupt Restriction

Currently, the `message too long` interrupt, described in Section B.3.7, does not work properly. The bus error still occurs, however. There is at present no workaround for this restriction.

8.2.13 All Fall Down Restriction

All Fall Down messages sometimes don't set the `all_fall_down` bit in the `private` register The workaround for this restriction is to check that `rec_len_left` is greater than `rec_len`.

8.2.14 Send/Receive and FIFO Locking Restrictions

It is an error to send a message while in the middle of receiving one — a bug in the NI causes the receive status information to get written to the send status register. Also, it is an error to lock the send/receive FIFOs while in the middle of receiving a message — doing so can cause the remainder of the message to be lost. In general, it is best to lock and unlock the FIFOs only when both the send and receive FIFOs are clear.

Copyright © 1994 Thinking Machines Corporation

Appendixes

Copyright © 1994 Thinking Machines Corporation

Appendix A

NI Registers, Fields, and Constants

This appendix lists the registers and fields of the NI chip, as well as the constants used to locate them. To use these constants, either include the header file `cmna.h` (see Section 1.3.4), or the appropriate CMNA header file (see Appendix H).

Note: The notation "**2.2**" indicates new registers/fields in Version 2.2 of the NI.

A.1 NI Registers

For each register the following information is provided:

- the name of the register
- the hex offset of the register from the user or supervisor base address
- the size of the register in bits, and its memory length in words
- the read/write permissions of the register for both user and supervisor

Register Constants

Note: With the exception of the `send_first` and `send_first_long` registers (which are described in Section A.3 and A.4 below), the names of the constants used to access NI registers are derived from the names of the registers themselves by uppercasing the register name and adding the suffix "`_A`". Each register constant provides the absolute address of the register, in either the user or supervisor memory area, depending on which header file (`cmna.h` or `cmna_sup.h`) has been included.

Copyright © 1994 Thinking Machines Corporation

A.1.1 Global and System Registers

				Permissions:		
Register Name:	**Address:**	**Size:**	**Len:**	**Super:**	**User:**	
ni_interrupt_cause	0x0000	15	1	R/W	None	
ni_interrupt_cause_green	0x0008	14	1	R/W	None	
ni_interrupt_level	0x0010	32	1	R/W	None	
ni_physical_self	0x0018	20	1	R/W	None	
ni_partition_base	0x0020	20	1	R/W	None	
ni_partition_size	0x0028	20	1	R/W	None	
ni_chunk_table_address	0x0030	6	1	R/W	None	
ni_chunk_table_data	0x0038	8	1	R/W	None	
ni_chunk_size	0x0040	3	1	R/W	None	
ni_dr_message_count	0x0048	32	1	R/W	None	
ni_count_mask	0x0050	16	1	R/W	None	
ni_rec_interrupt_mask	0x0058	16	1	R/W	None	
ni_user_tag_mask	0x0060	16	1	R/W	None	
ni_time	0x0070	32	1	R/W	R	
ni_configuration	0x0078	5	1	R/W	None	
ni_interrupt_send	0x0080	5	1	R/W	None	
ni_serial_number	0x0088	32	1	R	None	
ni_sync_global	0x0090	2	1	R	R	
ni_sync_global_abstain	0x0098	1	1	R/W	R/W	
ni_com_flush_send	0x00A0	1	1	W	None	
ni_async_global	0x00A8	2	1	R/W	R/W	
ni_async_sup_global	0x00B0	2	1	R/W	None	
ni_hodgepodge	0x00B8	6	1	R/W	None	
ni_sync_global_send	0x00C0	1	1	R/W	R/W	
ni_interrupt_clear	0x00C8	15	1	W	None	
ni_interrupt_clear_green	0x00D0	14	1	W	None	
ni_interrupt_now	0x00D8	32	1	R/W	None	
ni_scan_start	0x00E0	1	1	R/W	R/W	
ni_bad_address	0x00E8	32	1	R/W	None	
ni_longest_dr_message	0x0160	5	1	R/W	R	2.2
ni_user_rec_interrupt_mask	0x0168	16	1	R/W	R/W	2.2
ni_interrupt_set	0x0190	20	1	W	None	2.2
ni_interrupt_set_green	0x0198	19	1	W	None	2.2

Copyright © 1994 Thinking Machines Corporation

A.1.2 Network Interface Registers

Combined Data Network Interface (DR)

Register Name:	Address:	Size:	Len:	Permissions: Super:	Permissions: User:	
`ni_dr_status`	0x0200	24	1	R/W	R	
`ni_dr_private`	0x0208	10	1	R/W	None	
`ni_dr_send`	0x0230	32	16	W	W	
`ni_dr_status_all`	0x0250	32	1	R	R	2.2
`ni_dr_status_long`	0x0260	32	1	R	R	2.2
`ni_dr_send_first` (block)	0x1000	32	2	W	W	
`ni_dr_send_first_long`	0x8000	32	2	W	W	2.2

Left Data Network Interface (LDR)

Register Name:	Address:	Size:	Len:	Permissions: Super:	Permissions: User:	
`ni_ldr_status`	0x0c00	32	1	R/W	R	
`ni_ldr_private`	0x0c08	24	1	R/W	None	
`ni_ldr_recv`	0x0c20	32	16	R/W	R	
`ni_ldr_send`	0x0c30	32	16	W	W	
`ni_ldr_status_pop`	0x0c40	32	2	R	R	2.2
`ni_ldr_status_all`	0x0c50	32	1	R	R	2.2
`ni_ldr_status_long`	0x0c60	32	1	R	R	2.2
`ni_ldr_send_first` (block)	0x6000	32	2	W	W	
`ni_ldr_send_first_long`	0x10000	32	2	W	W	2.2

Right Data Network Interface (RDR)

Register Name:	Address:	Size:	Len:	Permissions: Super:	Permissions: User:	
`ni_rdr_status`	0x0e00	32	1	R/W	R	
`ni_rdr_private`	0x0e08	24	1	R/W	None	
`ni_rdr_recv`	0x0e20	32	16	R/W	R	
`ni_rdr_send`	0x0e30	32	16	W	W	
`ni_rdr_status_pop`	0x0e40	32	2	R	R	2.2
`ni_rdr_status_all`	0x0e50	32	1	R	R	2.2
`ni_rdr_status_long`	0x0e60	32	1	R	R	2.2
`ni_rdr_send_first` (block)	0x7000	32	2	W	W	
`ni_rdr_send_first_long`	0x18000	32	2	W	W	2.2

Copyright © 1994 Thinking Machines Corporation

Broadcast Interface (BC)

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

Supervisor Broadcast Interface (SBC)

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

Combine Interface (COM)

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

Copyright © 1994 Thinking Machines Corporation

A.2 NI Message Length Limit Constants

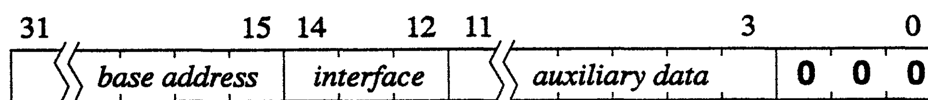
The following constants give the message length limits of the network interfaces:

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

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

A.3 Send First Register Addresses

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



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

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

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

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

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

Copyright © 1994 Thinking Machines Corporation

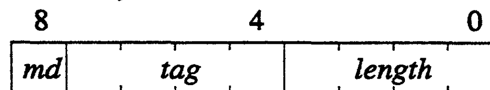
The following *interface_number* constants are defined:

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

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

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

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



where

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

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

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

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

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

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

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

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

Copyright © 1994 Thinking Machines Corporation

Broadcast (BC/SBC) Auxiliary Data Fields

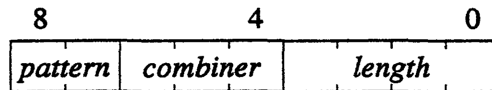
The format of the auxiliary data of a broadcast message is:

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

NI_BC_SEND_AUXILIARY_LENGTH_P	The *length* field offset (0).

Combine Auxiliary Data Fields

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



where

pattern is a two-bit value selecting the order in which values are combined

combiner is a three-bit value selecting the combine operation performed

length is the length of the message in words

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

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

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

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

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

SCAN_FORWARD	Forward scan pattern (2).
SCAN_BACKWARD	Backward scan pattern (1).

Copyright © 1994 Thinking Machines Corporation

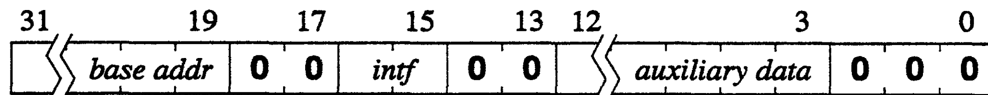
`SCAN_REDUCE`	Reduction scan pattern (3).
`SCAN_ROUTER_DONE`	Network-done operation (0).

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).

A.4 Send First Long (Data Network) Register Addresses

The `send_first_long` address for a Data Network message is a 32-bit value of the form

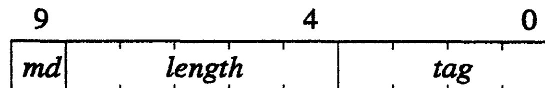


where *intf* is the interface number (an integer from 0 to 3 representing the Data Network interface being used), and *auxiliary data* is the auxiliary information. (The *base address* portion is the base address of the NI memory area, user or supervisor.)

The following *intf* values are defined:

0 - Not used	2 - LDR network interface
1 - DR network interface	3 - RDR network interface

The format of the auxiliary information is:



where

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

Copyright © 1994 Thinking Machines Corporation

A.5 NI Fields

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

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

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

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

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

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

A.5.1 Combined Data Network (DR) Fields

The ni_dr_status Register

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

Copyright © 1994 Thinking Machines Corporation

The ni_dr_status_long Register

Field Name:	Constant:	Pos:	Len:	Permissions: Super:	User:	
ni_send_space	NI_SEND_SPACE_LONG_P	0	5	R	R	2.2
ni_rec_ok	NI_REC_OK_LONG_P	5	1	R	R	2.2
ni_send_ok	NI_SEND_OK_LONG_P	6	1	R	R	2.2
ni_router_done_complete	NI_ROUTER_DONE_COMPLETE_LONG_P	7	1	R	R	2.2
ni_rec_length_left	NI_REC_LENGTH_LEFT_LONG_P	8	5	R/W	R	2.2
ni_rec_length	NI_REC_LENGTH_LONG_P	13	5	R/W	R	2.2
ni_dr_rec_tag	NI_DR_REC_TAG_LONG_P	18	4	R/W	R	2.2
ni_dr_send_state	NI_DR_SEND_STATE_LONG_P	24	2	R	R	2.2
ni_dr_rec_state	NI_DR_REC_STATE_LONG_P	26	2	R	R	2.2

The ni_dr_status_{all/pop} Registers

Field Name:	Constant:	Pos:	Len:	Permissions: Super:	User:	
ni_ldr_rec_ok		0	1	R	R	2.2
ni_rdr_rec_ok		1	1	R	R	2.2
ni_dr_send_ok		2	1	R	R	2.2
ni_ldr_rec_tag		3	4	R	R	2.2
ni_rdr_rec_tag		7	4	R	R	2.2
ni_ldr_rec_length_long		11	5	R	R	2.2
ni_rdr_rec_length_long		16	5	R	R	2.2
ni_dr_send_space		21	5	R	R	2.2
ni_ldr_rec_all_fall_down		26	1	R	R	2.2
ni_rdr_rec_all_fall_down		27	1	R	R	2.2
ni_router_done_complete		31	1	R	R	2.2

The ni_dr_private Register

Field Name:	Constant:	Pos:	Len:	Permissions: Super:	User:	
ni_rec_ok_ie	NI_REC_OK_IE_P	0	1	R/W	None	
ni_lock	NI_LOCK_P	1	1	R/W	None	
ni_rec_stop	NI_REC_STOP_P	2	1	R/W	None	
ni_rec_full	NI_REC_FULL_P	3	1	R	None	
ni_dr_rec_all_fall_down	NI_DR_REC_ALL_FALL_DOWN_P	5	1	R/W	None	
ni_all_fall_down_ie	NI_ALL_FALL_DOWN_IE_P	6	1	R/W	None	
ni_all_fall_down_enable	NI_ALL_FALL_DOWN_ENABLE_P	7	1	R/W	None	
ni_sfifo_goes_empty_ie	NI_SFIFO_GOES_EMPTY_IE_P	8	1	R/W	None	2.2
ni_rdone_complete_ie	NI_RDONE_COMPLETE_IE_P	9	1	R/W	None	2.2

Copyright © 1994 Thinking Machines Corporation

A.5.2 Left Data Network Interface (LDR) Fields

The ni_ldr_status Register

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

The ni_ldr_status_long Register

Field Name:	Constant:	Pos:	Len:	Permissions: Super:	User:	
ni_send_space	NI_SEND_SPACE_LONG_P	0	5	R	R	2.2
ni_rec_ok	NI_REC_OK_LONG_P	5	1	R	R	2.2
ni_send_ok	NI_SEND_OK_LONG_P	6	1	R	R	2.2
ni_rec_length_left	NI_REC_LENGTH_LEFT_LONG_P	8	5	R/W	R	2.2
ni_rec_length	NI_REC_LENGTH_LONG_P	13	5	R/W	R	2.2
ni_dr_rec_tag	NI_DR_REC_TAG_LONG_P	18	4	R/W	R	2.2

The ni_ldr_status_{all/pop} Registers

Field Name:	Constant:	Pos:	Len:	Permissions: Super:	User:	
ni_ldr_rec_ok		0	1	R	R	2.2
ni_rdr_rec_ok		1	1	R	R	2.2
ni_ldr_send_ok		2	1	R	R	2.2
ni_ldr_rec_tag		3	4	R	R	2.2
ni_rdr_rec_tag		7	4	R	R	2.2
ni_ldr_rec_length_long		11	5	R	R	2.2
ni_rdr_rec_length_long		16	5	R	R	2.2
ni_ldr_send_space		21	5	R	R	2.2
ni_ldr_rec_all_fall_down		26	1	R	R	2.2
ni_rdr_rec_all_fall_down		27	1	R	R	2.2
ni_router_done_complete		31	1	R	R	2.2

Copyright © 1994 Thinking Machines Corporation

The ni_ldr_private Register

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

A.5.3 Right Data Network Interface (RDR) Fields

The ni_rdr_status Register

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

The ni_rdr_status_long Register

Field Name:	Constant:	Pos:	Len:	Permissions: Super:	User:	
ni_send_space	NI_SEND_SPACE_LONG_P ...	0	5	R	R	2.2
ni_rec_ok	NI_REC_OK_LONG_P	5	1	R	R	2.2
ni_send_ok	NI_SEND_OK_LONG_P	6	1	R	R	2.2
ni_rec_length_left	NI_REC_LENGTH_LEFT_LONG_P ...	8	5	R/W	R	2.2
ni_rec_length	NI_REC_LENGTH_LONG_P ...	13	5	R/W	R	2.2
ni_dr_rec_tag	NI_DR_REC_TAG_LONG_P ...	18	4	R/W	R	2.2

Copyright © 1994 Thinking Machines Corporation

The ni_rdr_status_{all/pop} Registers

Field Name:	Constant:	Pos:	Len:	Permissions: Super:	User:	
ni_rdr_rec_ok		0	1	R	R	2.2
ni_ldr_rec_ok		1	1	R	R	2.2
ni_rdr_send_ok		2	1	R	R	2.2
ni_rdr_rec_tag		3	4	R	R	2.2
ni_ldr_rec_tag		7	4	R	R	2.2
ni_rdr_rec_length_long		11	5	R	R	2.2
ni_ldr_rec_length_long		16	5	R	R	2.2
ni_rdr_send_space		21	5	R	R	2.2
ni_rdr_rec_all_fall_down		26	1	R	R	2.2
ni_ldr_rec_all_fall_down		27	1	R	R	2.2
ni_router_done_complete		31	1	R	R	2.2

The ni_rdr_private Register

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

A.5.4 Broadcast Interface (BC) Fields

The ni_bc_status Register

Field Name:	Constant:	Pos:	Len:	Permissions: Super:	User:	
ni_send_space	NI_SEND_SPACE_P	0	4	R	R	
ni_rec_ok	NI_REC_OK_P	4	1	R	R	
ni_send_ok	NI_SEND_OK_P	5	1	R	R	
ni_send_empty	NI_SEND_EMPTY_P	6	1	R	R	
ni_rec_length_left	NI_REC_LENGTH_LEFT_P (NI_BC_REC_LENGTH_LEFT_LONG_L)	7	7	R	R	2.2

Copyright © 1994 Thinking Machines Corporation

The ni_bc_private Register

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

The ni_bc_control Register

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

A.5.5 Supervisor Broadcast Interface (SBC) Fields

The ni_sbc_status Register

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

The ni_sbc_private Register

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

Copyright © 1994 Thinking Machines Corporation

The ni_sbc_control Register

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

A.5.6 Combine Interface (COM) Fields

The ni_com_status Register

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

The ni_com_private Register

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

Copyright © 1994 Thinking Machines Corporation

The ni_com_control Register

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

A.5.7 Global Interface Fields

The ni_sync_global Register

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

The ni_async_global Register

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

The ni_async_sup_global Register

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

A.5.8 Interrupt Register Fields

Note: The position ("`_P`") constants for these flags are as described above. The length for all flags (1) is given by the single constant `NI_INTERRUPT_L`. To locate the flags in the `interrupt_clear/set` registers, use the constants defined for the `interrupt_cause` registers — the flag positions are the same.

Copyright © 1994 Thinking Machines Corporation

The ni_interrupt_cause Register

Flag Name:	Pos:	Len:	Permissions: Super:	User:	
ni_cause_internal_fault	0	1	R/W	None	
ni_cause_mc_error	1	1	R/W	None	
ni_cause_cmu_error	2	1	R/W	None	
ni_cause_bc_interrupt_red	3	1	R/W	None	
ni_cause_cn_checksum_error	4	1	R/W	None	
ni_cause_cn_hard_error	5	1	R/W	None	
ni_cause_dr_checksum_error	6	1	R/W	None	
ni_cause_timer_interrupt	7	1	R/W	None	
ni_cause_bc_interrupt_orange	8	1	R/W	None	
ni_cause_bc_interrupt_yellow	9	1	R/W	None	
ni_cause_bc_or_com_collision	10	1	R/W	None	
ni_cause_com_abstain_changed	11	1	R/W	None	
ni_cause_dr_count_negative	12	1	R/W	None	
ni_cause_bad_relative_address	13	1	R/W	None	
ni_cause_bad_memory_access	14	1	R/W	None	
ni_cause_message_too_long	15	1	R/W	None	**2.2**
ni_cause_rdone_complete	16	1	R/W	None	**2.2**

The ni_interrupt_cause_green Register

Flag Name:	Pos:	Len:	Permissions: Super:	User:	
ni_cause_bc_interrupt_green	0	1	R/W	None	
ni_cause_scan_overflow	1	1	R/W	None	
ni_cause_bc_rec_ok	2	1	R/W	None	
ni_cause_sbc_rec_ok	3	1	R/W	None	
ni_cause_com_rec_ok	4	1	R/W	None	
ni_cause_com_rec_empty	5	1	R/W	None	
ni_cause_sync_global_rec	6	1	R/W	None	
ni_cause_global_rec	7	1	R/W	None	
ni_cause_supervisor_global_rec	8	1	R/W	None	
ni_cause_dr_rec_ok	9	1	R/W	None	
ni_cause_ldr_rec_ok	10	1	R/W	None	
ni_cause_rdr_rec_ok	11	1	R/W	None	
ni_cause_dr_rec_tag	12	1	R/W	None	
ni_cause_dr_rec_all_fall_down	13	1	R/W	None	
ni_cause_ldr_rec_tag	14	1	R/W	None	**2.2**
ni_cause_rdr_rec_tag	15	1	R/W	None	**2.2**
ni_cause_ldr_user_rec_tag	16	1	R/W	None	**2.2**
ni_cause_rdr_user_rec_tag	17	1	R/W	None	**2.2**
ni_cause_sfifo_empty	18	1	R/W	None	**2.2**
ni_cause_dperr	19	1	R/W	None	**2.2**

Copyright © 1994 Thinking Machines Corporation

The ni_interrupt_{clear,set} Registers

Field Name:	Pos:	Len:	Permissions: Super:	User:	
ni_{clear,set}_internal_fault	0	1	W	None	
ni_{clear,set}_mc_error	1	1	W	None	
ni_{clear,set}_cmu_error	2	1	W	None	
ni_{clear,set}_bc_interrupt_red	3	1	W	None	
ni_{clear,set}_cn_checksum_error	4	1	W	None	
ni_{clear,set}_cn_hard_error	5	1	W	None	
ni_{clear,set}_dr_checksum_error	6	1	W	None	
ni_{clear,set}_timer_interrupt	7	1	W	None	
ni_{clear,set}_bc_interrupt_orange	8	1	W	None	
ni_{clear,set}_bc_interrupt_yellow	9	1	W	None	
ni_{clear,set}_bc_or_com_collision	10	1	W	None	
ni_{clear,set}_com_abstain_changed	11	1	W	None	
ni_{clear,set}_dr_count_negative	12	1	W	None	
ni_{clear,set}_bad_relative_address	13	1	W	None	
ni_{clear,set}_bad_memory_access	14	1	W	None	
ni_{clear,set}_message_too_long	15	1	W	None	2.2
ni_{clear,set}_rdone_complete	16	1	W	None	2.2

The ni_interrupt_{clear,set}_green Registers

Field Name:	Pos:	Len:	Permissions: Super:	User:	
ni_{clear,set}_bc_interrupt_green	0	1	W	None	
ni_{clear,set}_scan_overflow	1	1	W	None	
ni_{clear,set}_bc_rec_ok	2	1	W	None	
ni_{clear,set}_sbc_rec_ok	3	1	W	None	
ni_{clear,set}_com_rec_ok	4	1	W	None	
ni_{clear,set}_com_rec_empty	5	1	W	None	
ni_{clear,set}_sync_global_rec	6	1	W	None	
ni_{clear,set}_global_rec	7	1	W	None	
ni_{clear,set}_supervisor_global_rec	8	1	W	None	
ni_{clear,set}_dr_rec_ok	9	1	W	None	
ni_{clear,set}_ldr_rec_ok	10	1	W	None	
ni_{clear,set}_rdr_rec_ok	11	1	W	None	
ni_{clear,set}_dr_rec_tag	12	1	W	None	
ni_{clear,set}_dr_rec_all_fall_down	13	1	W	None	
ni_{clear,set}_ldr_rec_tag	14	1	W	None	2.2
ni_{clear,set}_rdr_rec_tag	15	1	W	None	2.2
ni_{clear,set}_ldr_user_rec_tag	16	1	W	None	2.2
ni_{clear,set}_rdr_user_rec_tag	17	1	W	None	2.2
ni_{clear,set}_sfifo_empty	18	1	W	None	2.2
ni_{clear,set}_dperr	19	1	W	None	2.2

Copyright © 1994 Thinking Machines Corporation

A.5.9 Other Register Fields and Constants

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

The ni_interrupt_level Register

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

The ni_hodgepodge Register

Field Name:	Pos:	Len:	Permissions: Super:	User:	
`ni_global_rec_ie`	0	1	R/W	None	
`ni_supervisor_global_rec_ie`	1	1	R/W	None	
`ni_flush_complete`	2	1	R	None	
`ni_interrupt_send_ok`	3	1	R	None	
`ni_configuration_complete`	4	1	R	None	
`ni_interrupt_rec_enable`	5	1	R/W	None	
`ni_sync_global_rec_ie`	6	1	R/W	None	
`ni_timer_ie`	7	1	R/W	None	
`ni_cn_stop_send`	8	1	R/W	None	
`ni_disable_bus_error`	9	1	R/W	None	**2.2**
`ni_ldr_rec_tag_ie`	10	1	R/W	None	**2.2**
`ni_rdr_rec_tag_ie`	11	1	R/W	None	**2.2**
`ni_ldr_user_rec_tag_ie`	12	1	R/W	None	**2.2**
`ni_rdr_user_rec_tag_ie`	13	1	R/W	None	**2.2**
`ni_msg_too_long_ie`	14	1	R/W	None	**2.2**

Copyright © 1994 Thinking Machines Corporation

The ni_bad_address Register

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

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

Copyright © 1994 Thinking Machines Corporation

Appendix B

NI Interrupts

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

For each interrupt, the following information is provided:

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

Note: It is possible for the supervisor to trigger an interrupt artificially, by setting the appropriate flag in one of the registers `ni_interrupt_cause/set` or `ni_interrupt_cause/set_green`. Since this can be done for any interrupt, it is not documented under the triggering events for each interrupt.

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

Copyright © 1994 Thinking Machines Corporation

B.1 Red Interrupts

Red interrupts indicate a failure of the hardware, such as checksum violations and message format errors. They occur at unpredictable times relative to the instruction stream and are usually irrecoverable. Determining the precise cause of a Red interrupt may require the use of the Diagnostic Network.

The cause, clear, and set flags listed for each interrupt are found in the registers:

```
ni_interrupt_cause
ni_interrupt_clear
ni_interrupt_set
```

B.1.1 Internal Fault .... Red Interrupt

Flags: `ni_cause/clear/set_internal_fault`

Cause: A fault has been detected in the NI chip.

Effect: The effects are undefined and irrecoverable.

Handling: No software-serviceable parts inside. Please report this fault to your applications engineer or systems manager for correction.

B.1.2 CN Checksum Error, DR Checksum Error .... Red Interrupt

Flags: `ni_cause/clear/set_cn_checksum_error`
`ni_cause/clear/set_dr_checksum_error`

Cause: A message with a bad checksum value was received from either the Control Network or Data Network. This interrupt is signaled as soon as the bad checksum value is received by the NI.

Effect: None. The received message(s) may still be read. However, they will almost certainly contain an error in either data or address.

Handling: This interrupt indicates that a network chip (or the NI chip itself) has failed. The failed chip must be tracked down with the Diagnostic Network. Please report this fault to your applications engineer or system manager for correction.

Copyright © 1994 Thinking Machines Corporation

B.1.3 CN Hard Error .... Red Interrupt

Flags: `ni_cause/clear/set_cn_hard_error`

Cause: A hardware error occurred in the Control Network.

Effect: The effects are undefined and irrecoverable.

Handling: This interrupt indicates one of two things: either a hardware problem in the Control Network, which must be located by use of the Diagnostic Network; or a serious software problem (specifically, a double trap forcing a processor (IU) reset). Please report this fault to your applications engineer or system manager for correction.

B.1.4 MC Error, CMU Error .... Red Interrupt

Flags: `ni_cause/clear/set_mc_error`
`ni_cause/clear/set_cmu_error`

Cause: An interrupt is being signaled by either the memory controller, or by the CMU (cache and memory management unit). These two kinds of external interrupt are signaled to the microprocessor by way of the NI chip.

Effect: None, aside from the interrupt itself.

Handling: These interrupts continue to be signaled until they are cleared on the memory controller or CMU.

Note: Unlike most NI interrupts, these two interrupts are not cleared by writing the corresponding `ni_interrupt_clear` flag. Instead, a flag on the memory controller or CMU must be reset.

Nevertheless, it *is* legal to write a 1 to the `ni_interrupt_clear` flags for these interrupts. While this has no effect, it is permitted so that you can write uniform interrupt handler code.

Copyright © 1994 Thinking Machines Corporation

B.1.5 BC Interrupt Red .... Red Interrupt

Flags: `ni_cause/clear/set_bc_interrupt_red`

Cause: The NI received a Red broadcast interrupt, and the broadcast interrupt enable flag `ni_interrupt_rec_enable` was set to 1.

Effect: None, aside from the interrupt itself.

Handling: This is a software-signaled interrupt. Your interrupt handler should detect and handle this interrupt as appropriate for your program.

B.2 Orange Interrupts

Orange interrupts indicate that the attention of the operating system is required, as in timer interrupts and broadcast interrupt messages. They occur at unpredictable times relative to the instruction stream and do not destroy any information that might be needed to determine the cause of the interrupt.

The cause, clear, and set flags listed for each interrupt are found in the registers:

```
ni_interrupt_cause
ni_interrupt_clear
ni_interrupt_set
```

B.2.1 Timer Interrupt .... Orange Interrupt

Flags: `ni_cause/clear/set_timer_interrupt`

Cause: The `ni_time` register is equal to the `ni_interrupt_now` register, and the timer interrupt flag `ni_timer_ie` flag is 1.

Effect: None, aside from the interrupt itself.

Handling: This interrupt is software-controlled, and should be handled by your interrupt handler.

Copyright © 1994 Thinking Machines Corporation

B.2.2 Network Done Complete .... Orange Interrupt

Flags: `ni_cause/clear/set_rdone_complete`

Cause: The `ni_rdone_complete_ie` flag is true, and a network-done operation has just completed (that is, the flag `ni_router_done_complete` flag has been set).

Effect: None, aside from the interrupt itself.

Handling: This interrupt is software-controlled, and should be handled by your interrupt handler. It is intended to allow system code to do operations such as setting the `ni_dr_message_count` register to zero at the end of a network-done operation.

B.2.3 BC Interrupt Orange .... Orange Interrupt

Flags: `ni_cause/clear/set_bc_interrupt_orange`

Cause: The NI received a Orange broadcast interrupt, and the broadcast interrupt enable flag `ni_interrupt_rec_enable` was set to 1.

Effect: None, aside from the interrupt itself.

Handling: This is a software-signaled interrupt. Your interrupt handler should detect and handle this interrupt as appropriate for your program.

B.3 Yellow Interrupts

Yellow interrupts indicate a software error. They are usually irrecoverable, as they indicate that your program is doing something illegal. Sufficient information is retained in the NI to permit isolation of the cause of the interrupt, but it is not always possible to recover all information relating to the cause. Yellow interrupts are associated with particular instructions, but are not signaled at the exact point of the error, because of the loose NI/microprocessor coupling. The cause, clear, and set flags listed for each interrupt are found in the registers:

```
ni_interrupt_cause
ni_interrupt_clear
ni_interrupt_set
```


Copyright © 1994 Thinking Machines Corporation

B.3.1 BC Interrupt Yellow .... Yellow Interrupt

Flags: `ni_cause/clear/set_bc_interrupt_yellow`

Cause: The NI received a Yellow broadcast interrupt, and the broadcast interrupt enable flag `ni_interrupt_rec_enable` was set to 1.

Effect: None, aside from the interrupt itself.

Handling: This is a software-signaled interrupt. Your interrupt handler should detect and handle this interrupt as appropriate for your program.

B.3.2 Bad Memory Access .... Yellow Interrupt

Flags: `ni_cause/clear/set_bad_memory_access`

Cause: The NI would have signaled a Bus Error, but the flag `ni_disable_bus_error` was set to 1.

Effect: Same as described for Bus Errors in Section B.5.

Handling: Examine the `ni_bad_address` register to determine what memory transaction caused the error.

B.3.3 COM Abstain Changed .... Yellow Interrupt

Flags: `ni_cause/clear/set_com_abstain_changed`

Cause: The `ni_com_abstain` or `ni_reduce_rec_abstain` flags were changed while the combiner send FIFO was not empty.

Effect: The attempted change does not occur. Whether execution is allowed to continue depends on the interrupt handler.

Handling: Your interrupt handler should decide whether to signal this as an error, or to recover from it quietly, perhaps displaying a warning message.

Copyright © 1994 Thinking Machines Corporation

B.3.4 DR Count Negative .... Yellow Interrupt

Flags: `ni_cause/clear/set_dr_count_negative`

Cause: The combined value of all `ni_dr_message_count` registers in the Data Network has become negative, indicating a mismatch in the sending and/or receiving of Data Network messages.

Effect: None, but this interrupt is signaled repeatedly until the situation is corrected.

Handling: This may occur either when a failure in a Data Network or NI chip causes the annihilation of a message, or when an OS error causes a countable Data Network message to be sent out of its partition. This interrupt may also occur if two or more nodes in a paritition do not agree on which Data Network message tags are to be counted (that is, their `ni_count_mask` registers are not equal).

To restore the Data Network to a proper state, make sure that the partition is empty of Data Network messages, and then set all the `ni_dr_message_count` registers in the partition to 0.

Note: It may be that by the time the interrupt is signaled, the values of one or more of the `ni_dr_message_count` registers will have changed. This may make it difficult to locate the error, since the sum of the `ni_dr_message_count` registers may be positive by the time the interrupt is signaled.

B.3.5 BC or COM Collision .... Yellow Interrupt

Flags: `ni_cause/clear/set_bc_or_com_collision`

Cause: Three separate conditions cause this interrupt:

- Two NIs attempted to broadcast at the same time.
- Two different combine operations signaled at the same time.
- Two NIs simultaneously attempted a broadcast interrupt.

Effect: No combining or broadcast operations can proceed while the `ni_cause_bc_or_com_collision` flag is set. If the error was colliding broadcast interrupts, the broadcast is not signaled.

Copyright © 1994 Thinking Machines Corporation

Handling: If the error was colliding combine messages, the messages are still in the combine send FIFO. The supervisor should take control of this FIFO and read out the messages to determine where the collision occurred. If the error was colliding broadcast messages, the `ni_bc_send_empty` (or `ni_sbc_send_empty`) flags will be set to 0 in the contending processors. If the error was colliding broadcast interrupts, the `ni_interrupt_send_ok` will be 0 in the processors that sent the colliding broadcast interrupts.

The proper way to handle this interrupt is to set all the combine stop flags, then set the FIFO lock flags, then read out any remaining data values from the combine send FIFO.

Note: When the `ni_clear_bc_or_com_collision` flag is written, all messages in the broadcast and supervisor broadcast send FIFOs disappear, and the `ni_interrupt_send_ok` flag is set to 1. None of the other FIFOs, either send or receive, are affected.

B.3.6 Bad Relative Address .... Yellow Interrupt

Flags: `ni_cause/clear/set_bad_relative_address`

Cause: An attempt was made to send a Data Network message with a relative address that is illegal for the current partition.

Effect: The message with the bad address is discarded and the appropriate `ni_`*interface*`_send_ok` flag is set to 0, indicating that the attempt to send the message failed.

Handling: Your interrupt handler should decide whether to signal this as an error, or to recover from it quietly, perhaps displaying a warning message.

B.3.7 Message Too Long .... Yellow Interrupt

Flags: `ni_cause/clear/set_message_too_long`

Cause: An attempt was made to send a Data Network message with a length greater than is allowed for the interface in use. For each of the three `send_first_long` interfaces, this is the value of the

Copyright © 1994 Thinking Machines Corporation

ni_longest_dr_message register. For the **send_first** register interfaces this is either the **ni_longest_dr_message** value or five words, whichever is less.

Effect: The message with the bad address is discarded and the appropriate **ni_***interface***_send_ok** flag is set to 0, indicating that the attempt to send the message failed. A bus error is also signaled.

Handling: Your interrupt handler should decide whether to signal this as an error, or to recover from it quietly, perhaps displaying a warning message.

B.4 Green Interrupts

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

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

The cause, clear, and set flags listed for each interrupt are found in the registers:

```
ni_interrupt_cause
ni_interrupt_clear
ni_interrupt_set
```

B.4.1 BC Interrupt Green .... Green Interrupt

Flags: **ni_cause/clear/set_bc_interrupt_green**

Cause: The NI received a Green broadcast interrupt, and the broadcast interrupt enable flag **ni_interrupt_rec_enable** was set to 1.

Effect: None, aside from the interrupt itself.

Handling: This is a software-signaled interrupt. Your interrupt handler should detect and handle this interrupt as appropriate for your program.

Copyright © 1994 Thinking Machines Corporation

B.4.2 DR Receive Tag .... Green Interrupt

Flags: `ni_cause/clear/set_dr_rec_tag`

Cause: A message arrived at the front of a Data Network receive FIFO that has an interrupting tag (a tag corresponding to a set flag in the register `ni_rec_interrupt_mask`).

Effect: None, aside from the interrupt itself.

Handling: This interrupt is software-controlled, and should be handled by your interrupt handler.

B.4.3 DR Receive All Fall Down .... Green Interrupt

Flags: `ni_cause/clear/set_dr_rec_all_fall_down`

Cause: An All Fall Down mode message arrived at the front of a Data Network receive FIFO, while `ni_all_fall_down_ie` is 1.

Effect: The first word read from the FIFO is the All Fall Down mode address word, which is used to determine the correct destination address for the message. The `rec_length` field contains the original length of the message (that is, *not counting* the address word), while the `rec_length_left` field contains the total length of the message *counting* the address word.

Handling: Your handler should receive and store the message in such a way that it can later be resent to its correct destination.

B.4.4 Interface (DR, BC, COM, etc.) Receive OK ... Green Interrupt

Flags:

```
ni_cause/clear/set_bc_rec_ok
ni_cause/clear/set_sbc_rec_ok
ni_cause/clear/set_com_rec_ok
ni_cause/clear/set_dr_rec_ok
ni_cause/clear/set_ldr_rec_ok
ni_cause/clear/set_rdr_rec_ok
```


Copyright © 1994 Thinking Machines Corporation

Cause: A new message became available from the receive FIFO of one of the interfaces while the corresponding `ni_`*interface*`_rec_ok_ie` flag was set to 1.

Effect: While enabled, each of these interrupts is signaled once for each arriving message in the appropriate interface's receive FIFO.

Handling: This interrupt is software-controlled, and should be handled by your interrupt handler. (Typically, your handler reads the interrupting message from the FIFO, but you can decide to do otherwise.)

B.4.5 Global Rec (Sync, Global, or Supervisor) .... Green Interrupt

Flags:
`ni_cause/clear/set_sync_global_rec`
`ni_cause/clear/set_global_rec`
`ni_cause/clear/set_supervisor_global_rec`

Cause: One of the following events happened:

A synchronous global operation completed with a result of 1, and the `ni_sync_global_rec_ie` flag is 1.

The asynchronous global receive flag `ni_global_rec` changed from 0 to 1, and the `ni_global_rec_ie` flag is 1.

The supervisor asynchronous receive flag `ni_supervisor_global_rec` changed from 0 to 1, and the `ni_supervisor_global_rec_ie` flag is 1.

Effect: None, aside from the interrupts themselves.

Handling: These interrupts are software-controlled, and should be handled by your interrupt handler.

B.4.6 Com Receive Empty .... Green Interrupt

Flags: `ni_cause/clear/set_com_rec_empty`

Cause: The combine receive FIFO became empty while the empty receive FIFO interrupt flag `ni_com_rec_empty_ie` is 1.

Copyright © 1994 Thinking Machines Corporation

Effect: None, aside from the interrupt itself.

Handling: This interrupt is software-controlled, and should be handled by your interrupt handler.

B.4.7 Scan Overflow .... Green Interrupt

Flags: `ni_cause/clear/set_scan_overflow`

Cause: The first word of a scan or reduce message that suffered arithmetic overflow was read from the combine receive FIFO, and the `ni_scan_overflow_ie` interrupt enable flag is 1. This can only happen if the message combiner is a signed or unsigned addition.

Effect: None. The arrived message may be read normally.

Handling: Your interrupt handler should decide whether to signal this as an error, or to recover from it quietly, perhaps displaying a warning message.

B.4.8 DP Error (Vector Unit Error) .... Green Interrupt

Flags: `ni_cause/clear/set_dperr`

Cause: An interrupt has been signaled by the node's memory controller (the vector units in CMs so equipped). These interrupts are sent to the PE by way of the NI.

Effect: This interrupt will continue to be signaled until it is cleared both on the memory controller and in the NI.

Handling: This interrupt is introduced in Version 2.2 so that the vector units, integrated into the memory controller chips, can signal green interrupts. Both the NI `ni_interrupt_clear_green` flag and the corresponding flag on the memory controller (or VU) must be written to clear this interrupt.

Copyright © 1994 Thinking Machines Corporation

B.4.9 Send FIFO Empty (Data Network Only) ...... Green Interrupt

Flags: `ni_cause/clear/set_sfifo_empty`

Cause: The `ni_sfifo_empty_ie` flag is set, and a send FIFO in one of the Data Network interfaces (DR, LDR, RDR) has become empty.

Effect: None. The arrived message may be read normally.

Handling: This interrupt is intended as an aid in sending Data Network messages; in particular, the supervisor can wait until this condition occurs before sending messages, rather than attempting several failed sends when the Data Network is congested.

B.4.10 LDR/RDR Tag, LDR/RDR User Tag .... Green Interrupt

Flags:
`ni_cause/clear/set_ldr_tag`
`ni_cause/clear/set_rdr_tag`
`ni_cause/clear/set_ldr_user_tag`
`ni_cause/clear/set_rdr_user_tag`

Cause: A message arrives at the front of the left (or right) Data Network receive FIFO, having a tag that corresponds to a 1 bit in the register `ni_rec_interrupt_mask` (for `ldr/rdr_tag` interrupts) or in `ni_user_rec_interrupt_mask` (for the `ldr/rdr_user_tag` interrupts).

For the `user_tag` interrupts, not only must the appropriate tag mask be set in the `ni_user_rec_interrupt_mask`, but the same bit must be *cleared* in the `ni_rec_interrupt_mask` register.

Effect: None. The arrived message may be read normally.

Handling: These interrupts are intended as an aid in receiving Data Network messages. Your interrupt handler should determine the appropriate action to take to receive the tagged message that signaled the interrupt.

Copyright © 1994 Thinking Machines Corporation

B.5 Bus Errors

Bus Errors indicate that a bus transaction cannot be completed, as in an attempt to read a virtual address that does not correspond to a register, or to write a message that doesn't conform to protocol. Bus Errors are signaled asynchronously and are usually irrecoverable. Bus Errors are distinct from segmentation violation errors, which result from attempting to read an unmapped virtual address, and are signaled synchronously with the offending instruction.

The cause and clear flags listed for each interrupt are found in these registers:
`ni_interrupt_cause` `ni_interrupt_clear`

B.5.1 Bad Memory Access .... Bus Error

Flags: `ni_cause/clear/set_bad_memory_access`

Cause: Bus Errors are signaled for number of reasons, including:

- attempting to read a read-protected address
- attempting to write a write-protected address
- attempting to read or write a value that does not fit in a register
- attempting to read or write an address that is not a register

Note: If the flag `ni_disable_bus_error` is set, Bus Errors are signaled as a Yellow Interrupt (see Section B.3.2 above).

Some specific examples of Bus Error causes are:

Bus Errors caused by reads or writes:

- reading or writing a supervisor-only register from the user area
- reading the `ni_`*interface*`_rec` register of an empty receive FIFO
- attempting to read a doubleword from a FIFO that has only a word left, or attempting to use a doubleword operation to write a singleword message
- writing the `send_first` register of a network interface while there is an incomplete message pending in the send FIFO
- writing the `send` register of a network interface without having first written a value to the corresponding `send_first` register

Copyright © 1994 Thinking Machines Corporation

- writing a message to any of the Data Network's **send_first** registers with a length value that is greater than either five words or the value of the register **ni_longest_dr_message**, whichever is less.

- writing a message to any of the Data Network's **send_first_long** registers with a length value that is greater than the value of the register **ni_longest_dr_message**.

Bus Errors caused by sending a message:

- attempting to send a message longer than the entire send FIFO

- attempting to send a message via a network interface for which the corresponding abstain flag is set

- attempting to send a user message with a supervisor-reserved tag

- attempting to send or receive a message through an excluded Data Network interface

- attempting to send a combine message with an illegal combiner or pattern value

- attempting to send a network-done message with a length greater than 1, or attempting to send any network-done message while the **ni_network_done** flag is 0 or the **ni_com_abstain** flag is 1

- attempting to send a synchronous global message or to change the **ni_sync_global_abstain** flag while the **ni_sync_global_complete** flag is 0

Bus Errors caused by other operations:

- attempting to start a flush operation while the **ni_flush_complete** flag is 0

- attempting to start a configuration operation while the **ni_configuration_complete** flag is 0

- attempting to send a broadcast interrupt while the **ni_interrupt_send_ok** flag is 0

- attempting to write a value to the **ni_**_interface_**_rec** register when the receive FIFO is full.

Copyright © 1994 Thinking Machines Corporation

Effect: The address, size, and type of the offending memory transaction is stored in the `ni_bad_address` register.

Any data written by the offending transaction is lost. Any side effects that would have been triggered by the offending transaction (such as the initiation of a synchronous global operation) do not occur. In particular, an attempted doubleword read from a receiving FIFO containing only one word will not result in popping the word.

Handling: Examine the `ni_bad_address` register to determine what memory transaction caused the error.

Copyright © 1994 Thinking Machines Corporation

Appendix C

Programming Tools

This appendix describes the important C macros and constants defined by the CMNA software layer (that is, those relating to the NI chip itself).

C.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 fields of the **ni**_*interface*_**status** register (but *not* the **status_all** register) for any *interface* that has such a register:

Field Name:	Macros 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_rec_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);
```


Copyright © 1994 Thinking Machines Corporation

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.

C.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:
ni_dr_send_first	CMNA_dr_send_first{_long}(*tag*, *length*, *value*)
	CMNA_dr_send_first_double{_long}(*tag*, *length*, *value*)
ni_ldr_send_first	CMNA_ldr_send_first{_long}(*tag*, *length*, *value*)
	CMNA_ldr_send_first_double{_long}(*tag*, *length*, *value*)
ni_rdr_send_first	CMNA_rdr_send_first{_long}(*tag*, *length*, *value*)
	CMNA_rdr_send_first_double{_long}(*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 C.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();

Copyright © 1994 Thinking Machines Corporation

Register Name:	**Macros Used to Access Register:**
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 these 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 these 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)

Copyright © 1994 Thinking Machines Corporation

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
```

C.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:
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 this macro:

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


Copyright © 1994 Thinking Machines Corporation

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`	`BC_RECEIVE_LENGTH(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 C.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
```


Copyright © 1994 Thinking Machines Corporation

C.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:
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 C.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();

Copyright © 1994 Thinking Machines Corporation

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)

Copyright © 1994 Thinking Machines Corporation

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*);

C.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*);

Copyright © 1994 Thinking Machines Corporation

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()`

Copyright © 1994 Thinking Machines Corporation

Appendix D

Predefined Low-Level NI Constants

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

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

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

Length Constant:   NI_SEND_FIRST_L (32)

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

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


Copyright © 1994 Thinking Machines Corporation

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

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

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

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

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


Copyright © 1994 Thinking Machines Corporation

```
=== Status Register ===
NI_DR_STATUS_A            (NI_BASE | 0x0200)
NI_DR_STATUS_ALL_A        (NI_BASE + 0x0250) /* 2.2 */
NI_DR_STATUS_LONG_A       (NI_BASE + 0x0260) /* 2.2 */
NI_LDR_STATUS_A           (NI_BASE | 0x0c00)
NI_LDR_STATUS_ALL_A       (NI_BASE + 0x0c50) /* 2.2 */
NI_LDR_STATUS_LONG_A      (NI_BASE + 0x0c60) /* 2.2 */
NI_RDR_STATUS_A           (NI_BASE | 0x0e00)
NI_RDR_STATUS_ALL_A       (NI_BASE + 0x0e50) /* 2.2 */
NI_RDR_STATUS_LONG_A      (NI_BASE + 0x0e60) /* 2.2 */
NI_XDR_STATUS_L (19)     NI_STATUS_LONG_L (28) /* 2.2 */

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

NI_COM_STATUS_A           (NI_BASE | 0x0a00)
NI_COM_STATUS_L (21)
NI_STATUS_L (25)

Field Constants:
NI_SEND_SPACE_P (0)            NI_SEND_SPACE_L (4)
NI_REC_OK_P (4)                NI_REC_OK_L (1)
NI_SEND_OK_P (5)               NI_SEND_OK_L (1)
NI_ROUTER_DONE_COMPLETE_P(6) NI_ROUTER_DONE_COMPLETE_L (1)
NI_SEND_EMPTY_P (6)            NI_SEND_EMPTY_L (1)
NI_REC_LENGTH_LEFT_P (7)       NI_REC_LENGTH_LEFT_L (4)
    NI_BC_REC_LENGTH_LEFT_LONG_L (7) /* 2.2 */
NI_REC_LENGTH_P (11)           NI_REC_LENGTH_L (4)
NI_DR_REC_TAG_P (15)           NI_DR_REC_TAG_L (4)
NI_COM_SCAN_OVERFLOW_P (20)    NI_COM_SCAN_OVERFLOW_L (1)
NI_DR_SEND_STATE_P (21)        NI_DR_SEND_STATE_L (2)
NI_DR_REC_STATE_P (23)         NI_DR_REC_STATE_L (2)
/* 2.2 */
NI_SEND_SPACE_LONG_P (0)       NI_SEND_SPACE_LONG_L (5)
NI_REC_OK_LONG_P (5)           NI_REC_OK_LONG_L (1)
NI_SEND_OK_LONG_P (6)          NI_SEND_OK_LONG_L (1)
NI_ROUTER_DONE_COMPLETE_LONG_P (7)
NI_ROUTER_DONE_COMPLETE_LONG_L (1)
NI_REC_LENGTH_LEFT_LONG_P (8)  NI_REC_LENGTH_LEFT_LONG_L (5)
NI_REC_LENGTH_LONG_P (13)      NI_REC_LENGTH_LONG_L (5)
NI_DR_REC_TAG_LONG_P (18)      NI_DR_REC_TAG_LONG_L (4)
NI_DR_SEND_STATE_LONG_P (24) NI_DR_SEND_STATE_LONG_L (2)
NI_DR_REC_STATE_LONG_P (26)  NI_DR_REC_STATE_LONG_L (2)
```


Copyright © 1994 Thinking Machines Corporation

```
=== Control Registers ===
NI_BC_CONTROL_A          (NI_BASE | 0x0610)
NI_SBC_CONTROL_A         (NI_BASE | 0x0810)
NI_BC_CONTROL_L (1)
NI_COM_CONTROL_A         (NI_BASE | 0x0a10)
NI_COM_CONTROL_L (2)
NI_CONTROL_L (2)

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

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

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

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

NI_COM_PRIVATE_A         (NI_BASE | 0x0a08)
NI_COM_PRIVATE_L (18)

NI_PRIVATE_L (18)

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


Copyright © 1994 Thinking Machines Corporation

```
=== Global and System Registers ===
NI_INTERRUPT_CAUSE_A          (NI_BASE | 0x0000)
NI_INTERRUPT_SET_A            (NI_BASE + 0x0190) /* 2.2 */
NI_CAUSE_INTERNAL_FAULT_P (0)
NI_CAUSE_MC_ERROR_P (1)
NI_CAUSE_CMU_ERROR_P (2)
NI_CAUSE_BC_INTERRUPT_RED_P (3)
NI_CAUSE_CN_CHECKSUM_ERROR_P (4)
NI_CAUSE_CN_HARD_ERROR_P (5)
NI_CAUSE_DR_CHECKSUM_ERROR_P (6)
NI_CAUSE_TIMER_INTERRUPT_P (7)
NI_CAUSE_BC_INTERRUPT_ORANGE_P (8)
NI_CAUSE_BC_INTERRUPT_YELLOW_P (9)
NI_CAUSE_BC_OR_COM_COLLISION_P (10)
NI_CAUSE_COM_ABSTAIN_CHANGED_P (11)
NI_CAUSE_DR_COUNT_NEGATIVE_P (12)
NI_CAUSE_BAD_RELATIVE_ADDRESS_P (13)
NI_CAUSE_BAD_MEMORY_ACCESS_P (14)
NI_CAUSE_MESSAGE_TOO_LONG_P (15) /* 2.2 */
NI_CAUSE_RDONE_COMPLETE_P (16) /* 2.2 */
NI_INTERRUPT_L (1)

NI_INTERRUPT_CAUSE_GREEN_A          (NI_BASE | 0x0008)
NI_INTERRUPT_SET_GREEN_A            (NI_BASE + 0x0198) /* 2.2 */
NI_CAUSE_BC_INTERRUPT_GREEN_P (0)
NI_CAUSE_SCAN_OVERFLOW_P (1)
NI_CAUSE_BC_REC_OK_P (2)
NI_CAUSE_SBC_REC_OK_P (3)
NI_CAUSE_COM_REC_OK_P (4)
NI_CAUSE_COM_REC_EMPTY_P (5)
NI_CAUSE_SYNC_GLOBAL_REC_P (6)
NI_CAUSE_GLOBAL_REC_P (7)
NI_CAUSE_SUPERVISOR_GLOBAL_REC_P (8)
NI_CAUSE_DR_REC_OK_P (9)
NI_CAUSE_LDR_REC_OK_P (10)
NI_CAUSE_RDR_REC_OK_P (11)
NI_CAUSE_DR_REC_TAG_P (12)
NI_CAUSE_DR_REC_ALL_FALL_DOWN_P (13)
NI_CAUSE_LDR_REC_TAG_P (14) /* 2.2 */
NI_CAUSE_RDR_REC_TAG_P (15) /* 2.2 */
NI_CAUSE_LDR_USER_REC_TAG_P (16) /* 2.2 */
NI_CAUSE_RDR_USER_REC_TAG_P (17) /* 2.2 */
NI_CAUSE_SFIFO_EMPTY (18) /* 2.2 */
NI_CAUSE_DPERR (19) /* 2.2 */
NI_INTERRUPT_L (1)
```


Copyright © 1994 Thinking Machines Corporation

```
NI_INTERRUPT_LEVEL_A              (NI_BASE | 0x0010)
NI_INTERRUPT_LEVEL_L (32)
NI_INTERRUPT_LEVEL_COLOR_L (8)

NI_LONGEST_DR_MESSAGE_A      (NI_BASE + 0x0160) /* 2.2 */
NI_USER_REC_INTERRUPT_MASK_A (NI_BASE + 0x0168) /* 2.2 */

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

NI_CHUNK_TABLE_ADDRESS_A          (NI_BASE | 0x0030)
NI_CHUNK_TABLE_ADDRESS_L (6)

NI_CHUNK_TABLE_DATA_A             (NI_BASE | 0x0038)
NI_CHUNK_TABLE_DATA_L (8)

NI_CHUNK_SIZE_A                   (NI_BASE | 0x0040)
NI_CHUNK_SIZE_L (3)

NI_DR_MESSAGE_COUNT_A             (NI_BASE | 0x0048)
NI_DR_MESSAGE_COUNT_L (32)

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

NI_TIME_A                         (NI_BASE | 0x0070)
NI_TIME_L (32)

NI_CONFIGURATION_A                (NI_BASE | 0x0078)
NI_CONFIGURATION_L (5)

NI_INTERRUPT_SEND_A               (NI_BASE | 0x0080)
NI_INTERRUPT_SEND_L (5)

NI_SERIAL_NUMBER_A                (NI_BASE | 0x0088)
NI_SERIAL_NUMBER_L (32)

NI_SYNC_GLOBAL_A                  (NI_BASE | 0x0090)
NI_SYNC_GLOBAL_REC_P (0)
NI_SYNC_GLOBAL_REC_L (1)
NI_SYNC_GLOBAL_COMPLETE_P(1)
```


Copyright © 1994 Thinking Machines Corporation

```
NI_SYNC_GLOBAL_COMPLETE_L (1)
NI_SYNC_GLOBAL_L (2)

NI_SYNC_GLOBAL_ABSTAIN_A           (NI_BASE | 0x0098)
NI_SYNC_GLOBAL_ABSTAIN_L (1)

NI_COM_FLUSH_SEND_A                (NI_BASE | 0x00a0)
NI_FLUSH_SEND_L (1)

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

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

NI_HODGEPODGE_A                    (NI_BASE | 0x00b8)
NI_GLOBAL_REC_IE_P (0)
NI_GLOBAL_REC_IE_L (1)
NI_SUPERVISOR_GLOBAL_REC_IE_P (1)
NI_SUPERVISOR_GLOBAL_REC_IE_L (1)
NI_FLUSH_COMPLETE_P (2)
NI_FLUSH_COMPLETE_L (1)
NI_INTERRUPT_SEND_OK_P (3)
NI_INTERRUPT_SEND_OK_L (1)
NI_CONFIGURATION_COMPLETE_P (4)
NI_CONFIGURATION_COMPLETE_L (1)
NI_INTERRUPT_REC_ENABLE_P (5)
NI_INTERRUPT_REC_ENABLE_L (1)
NI_SYNC_GLOBAL_REC_IE_P (6)
NI_SYNC_GLOBAL_REC_IE_L (1)
NI_TIMER_IE_P (7)
NI_TIMER_IE_L (1)
NI_CN_STOP_SEND_P (8)
NI_CN_STOP_SEND_L (1)
NI_DISABLE_BUS_ERROR_P (9) /* 2.2 */
NI_DISABLE_BUS_ERROR_L (1) /* 2.2 */
NI_LDR_REC_TAG_IE_P (10) /* 2.2 */
NI_LDR_REC_TAG_IE_L (1) /* 2.2 */
NI_RDR_REC_TAG_IE_P (11) /* 2.2 */
NI_RDR_REC_TAG_IE_L (1) /* 2.2 */
```


Copyright © 1994 Thinking Machines Corporation

```
NI_LDR_USER_REC_TAG_IE_P (12) /* 2.2 */
NI_LDR_USER_REC_TAG_IE_L (1) /* 2.2 */
NI_RDR_USER_REC_TAG_IE_P (13) /* 2.2 */
NI_RDR_USER_REC_TAG_IE_L (1) /* 2.2 */
NI_MSG_TOO_LONG_IE_P (14) /* 2.2 */
NI_MSG_TOO_LONG_IE_L (1) /* 2.2 */
NI_HODGEPODGE_L (15)

NI_SYNC_GLOBAL_SEND_A              (NI_BASE | 0x00C0)
NI_SYNC_GLOBAL_SEND_L (1)

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

NI_INTERRUPT_NOW_A                 (NI_BASE | 0x00d8)
NI_INTERRUPT_NOW_L (32)

NI_SCAN_START_A                    (NI_BASE | 0x00e0)
NI_SCAN_START_L (1)

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


Copyright © 1994 Thinking Machines Corporation

Appendix E

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 Network 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) that 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.

Copyright © 1994 Thinking Machines Corporation

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.)

Copyright © 1994 Thinking Machines Corporation

Appendix F

NI Accessor Examples

Here are some examples of macros that C programmers can use to access the registers and fields of the NI. In most cases, these macros take as arguments the register and field constants defined previously in this manual.

F.1 Reading and Writing Registers

The simplest NI register operations involve reading and writing the value of a register, typically with one of three types of values: unsigned, float, and double. The macros below provide a simple register reading/writing interface.

```
#define ni_register(type,reg)      *((type *) (reg))
#define ni_read_reg(reg)        ni_register(unsigned, reg)
#define ni_read_reg_f(reg)      ni_register(float, reg)
#define ni_read_reg_d(reg)      ni_register(double, reg)

#define ni_set_register(type,reg,value)
              ni_register(type, reg) = ((type) (value))
#define ni_write_reg(reg)
                   ni_set_register(unsigned, reg, value)
#define ni_write_reg_f(reg)
                   ni_set_register(float, reg, value)
#define ni_write_reg_d(reg)
                   ni_set_register(double, reg, value)
```

In these examples the *reg* argument is the address constant of the appropriate register, and the *value* argument is the word, float, or double to be written.

Copyright © 1994 Thinking Machines Corporation

F.2 Reading and Writing Subfields

Often, you'll want to read or write the value of a register subfield. Here's a set of macros that efficiently extract a field from a register. (Note that the *field* argument in these examples is the name of the field constant *without* the `_P` or `_L` suffixes — these are added automatically by the macros themselves.)

```
/* mask for values that will fit into the given field */
#define ni_mask_field_values (field_length) \
    (~(~0 << field_length))

/* mask that extracts a field from the register */
#define ni_mask_field (position, length) \
    (ni_mask_field_values (length) << position)

/* right-shift register value, mask out the field */
#define ni_get_field(register_val, pos, len) \
    ((register_val >> pos) & ni_mask_field_values(len))

#define ni_read_field(register, pos, len) \
    ni_get_field(ni_read_reg(register), pos, len)
```

And here's a set of macros that efficiently modify the value of a register field:

```
/* mask that is ANDed with register to change field */
#define ni_new_value_mask(pos, len, new_value) \
    ~((new_value ^ ni_mask_field_values(len)) << pos)

/* Logical AND register with mask that changes field */
#define ni_set_field(reg_val, pos, len, new_value) \
  (reg_val & ni_new_value_mask(pos, len, new_value))

#define ni_write_field(reg, pos, len, new_value) \
  ni_write_reg(register, \
    ni_set_field(ni_read_reg(reg), pos, len, new_value))
```

You may also want to simply set or clear an arbitrary set of register bits:

```
#define ni_set_bits_in_register(reg, bitmask) \
  ni_write_reg(reg, ni_read_reg(reg) | (bitmask))

#define ni_clear_bits_in_register(reg, bitmask)\
  ni_write_reg(reg, ni_read_reg(reg) & ~(bitmask))
```


Copyright © 1994 Thinking Machines Corporation

F.3 Constructing Send-First Addresses

The only other major set of programming tools that you might need are macros that construct a **send_first** address for a given interface. For example:

```
#define ni_send_first_a(interface,auxiliary_data) \
  ((unsigned *) (  NI_BASE + \
                  (interface      << SF_FIFO_OFFSET | \
                   auxiliary_data << AUXILIARY_START_P)))

#define ni_send_first(interface,auxiliary_data,value) \
  ni_write_reg(ni_send_first_a(interface,auxiliary_data), \
               value)
```

Data Network Send-First Macros

Here's a set of macros that constructs the **send_first** addresses for the three Data Network interfaces:

```
#define ni_xdr_auxiliary_data(mode,tag,length) \
    ( mode    << NI_DR_SEND_AUXILIARY_ADDRESS_MODE_P | \
      tag     << NI_DR_SEND_AUXILIARY_TAG_P | \
      length  << NI_DR_SEND_AUXILIARY_LENGTH_P )

#define ni_dr_send_first(mode, tag, length, value) \
  ni_send_first(DATA_ROUTER_FIFO, \
                ni_xdr_auxiliary_data(mode,tag,length), \
                value)

#define ni_ldr_send_first(mode, tag, length, value) \
  ni_send_first(LEFT_DR_FIFO, \
                ni_xdr_auxiliary_data(mode,tag,length), \
                value)

#define ni_rdr_send_first(mode, tag, length, value) \
  ni_send_first(RIGHT_DR_FIFO, \
                ni_xdr_auxiliary_data(mode,tag,length), \
                value)
```


Copyright © 1994 Thinking Machines Corporation

Broadcast Interface Send-First Macros

Here's a set of macros that constructs the **send_first** addresses for the two broadcast interfaces:

```
#define ni_xbc_auxiliary_data(length) \
    ( length  << NI_BC_SEND_AUXILIARY_LENGTH_P )

#define ni_bc_send_first(length, value) \
  ni_send_first(USER_BC_FIFO, \
                ni_xbc_auxiliary_data(length), \
                value)

#define ni_sbc_send_first(length, value) \
  ni_send_first(SUPERVISOR_BC_FIFO, \
                ni_xbc_auxiliary_data(length), \
                value)
```

Combine Interface Send-First Macros

Finally, here's a set of macros that constructs the **send_first** addresses for the combine interface:

```
#define ni_com_auxiliary_data(pattern,combiner,length) \
    ( pattern  << NI_COM_SEND_AUXILIARY_PATTERN_P | \
      combiner << NI_COM_SEND_AUXILIARY_COMBINER_P | \
      length   << NI_COM_SEND_AUXILIARY_LENGTH_P )

#define ni_bc_send_first(pattern,combiner,length,value)\
  ni_send_first(COMBINE_FIFO, \
                ni_com_auxiliary_data(pattern,combiner,\
                                      length) \
                value)
```


Copyright © 1994 Thinking Machines Corporation

Appendix G

Sample NI Programs

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

As of Version 7.1.3 of the CM system software, CMOST, 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.

G.1 Data Network Test

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

- sending/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

Copyright © 1994 Thinking Machines Corporation

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);
```


Copyright © 1994 Thinking Machines Corporation

```
    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);
}
```


Copyright © 1994 Thinking Machines Corporation

```
/* 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;
   }
 }
}
```


Copyright © 1994 Thinking Machines Corporation

```
/* 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);
}
```


Copyright © 1994 Thinking Machines Corporation

```
/* 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;
  }
```


Copyright © 1994 Thinking Machines Corporation

```
    /* 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();
}
```


Copyright © 1994 Thinking Machines Corporation

G.2 Data Network Doubleword Messages Test

This program demonstrates the use of doubleword 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();
}
```


Copyright © 1994 Thinking Machines Corporation

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++;
```


Copyright © 1994 Thinking Machines Corporation

```
        *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;
```


Copyright © 1994 Thinking Machines Corporation

```
  /* 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();
}
```

G.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"
```


Copyright © 1994 Thinking Machines Corporation

```
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()));
}
```


Copyright © 1994 Thinking Machines Corporation

```
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();
    }
```


Copyright © 1994 Thinking Machines Corporation

G.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 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 */
    /* 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);
```


Copyright © 1994 Thinking Machines Corporation

```
    /* 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()));
}
```


Copyright © 1994 Thinking Machines Corporation

```
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);
```


Copyright © 1994 Thinking Machines Corporation

```
/* 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();
```


Copyright © 1994 Thinking Machines Corporation

```
/* 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();
}
```

G.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);
```


Copyright © 1994 Thinking Machines Corporation

```
    /* 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();
    }
```


Copyright © 1994 Thinking Machines Corporation

Appendix H

CMNA Header Files

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

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

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

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

H.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.

Copyright © 1994 Thinking Machines Corporation

H.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:

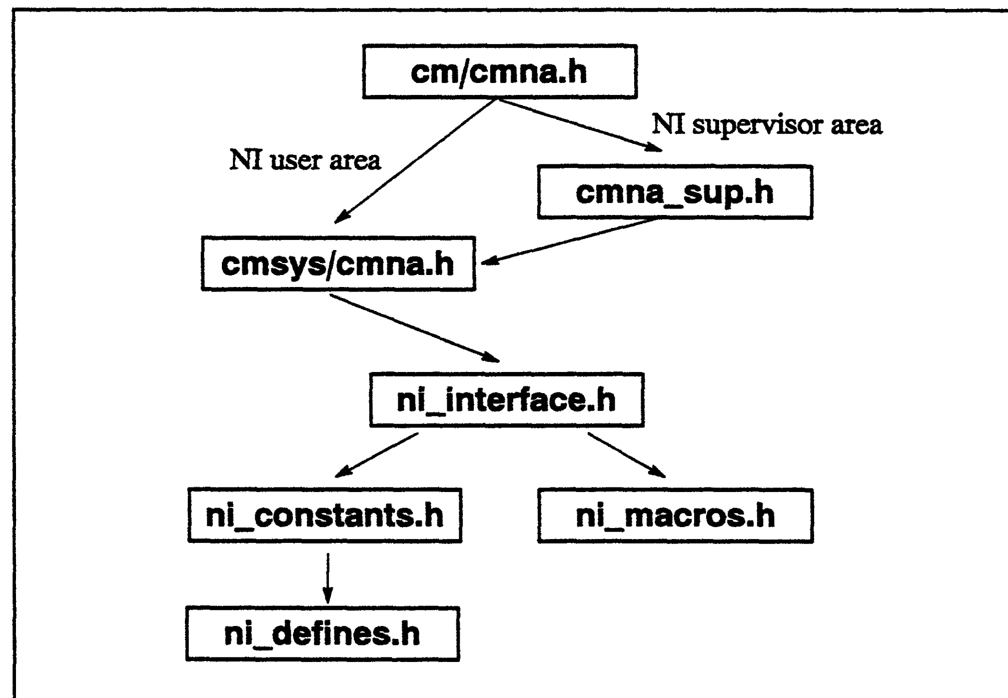


Figure 21. Relationship between CMNA and NI header files.

Copyright © 1994 Thinking Machines Corporation

H.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`).

H.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.)

H.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.

Copyright © 1994 Thinking Machines Corporation

H.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.)

H.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.

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

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

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

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

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

Copyright © 1994 Thinking Machines Corporation

H.3 CMNA Functions

Below are listed the basic functions provided by the CMNA library aside from those that directly access the NI chip (which are described elsewhere in this manual).

Important: The functions defined here are designed for usability, not performance. In actual production applications, you will want to write your own routines to obtain the highest communications performance possible. Use the routines described below as examples of how you *might* write an NI accessor function, not as hard and fast examples of how such a function *should* be written.

H.3.1 CMNA Version

```
CMNA_version()
Returns: char *
```

H.3.2 Activity Functions

```
CMNA_abstain_from(activity)
CMNA_participate_in(activity)
CMNA_sup_abstain_from(activity)
CMNA_sup_participate_in(activity)
int activity;
Return: int

/* valid activity to participate in or abstain from
participating in: */

#define  NI_REDUCE_RECEIVE   (1)
#define  NI_BC_RECEIVE       (2)
#define  NI_COMBINE          (4)
#define  NI_SYNC_GLOBAL      (8)
#define  NI_SBC_RECEIVE      (16)
#define  NI_BC_SEND_ENABLE   (32)
```


Copyright © 1994 Thinking Machines Corporation

H.3.3 DR Interface Functions

```
CMNA_dr_msg_to_receive()
Returns: int

CMNA_dr_send_fifo_amount(dest_proc,source_base,
                         word_length,tag)
unsigned int dest_proc;
void *source_base;
int word_length;
unsigned int tag;
Returns: int

CMNA_dr_send_fifo_amount_physical(dest_proc,source_base,
                                  word_length,tag)
void *source_base;
int word_length;
unsigned dest_proc;
unsigned tag;
Returns: int

CMNA_dr_send_msg(dest_proc,source_base,word_length,tag)
unsigned int dest_proc;
void *source_base;
int word_length;
unsigned int tag;

CMNA_dr_send_msg_physical(dest_proc,source_base,
                          word_length,tag)
void *source_base;
int word_length;
unsigned dest_proc;
unsigned tag;

CMNA_dr_status()
Returns: unsigned
```

H.3.4 LDR Interface Functions

```
CMNA_ldr_receive(base)
void *base;
Returns: int
```


Copyright © 1994 Thinking Machines Corporation

```
CMNA_ldr_receive_msg(base,word_length)
void *base;
int word_length;

CMNA_ldr_send_fifo_amount(dest_proc,source_base,
                          word_length,tag)
unsigned int dest_proc;
void *source_base;
int word_length;
unsigned int tag;
Returns: int

CMNA_ldr_send_fifo_amount_physical(dest_proc,source_base,
                                   word_length,tag)
void *source_base;
int word_length;
unsigned dest_proc;
unsigned tag;
Returns: int

CMNA_ldr_send_msg(dest_proc,source_base,word_length,tag)
unsigned int dest_proc;
void *source_base;
int word_length;
unsigned int tag;

CMNA_ldr_send_msg_physical(dest_proc,source_base,
                           word_length,tag)
void *source_base;
int word_length;
unsigned dest_proc;
unsigned tag;
```

H.3.5 RDR Interface Functions

```
CMNA_rdr_receive(base)
void *base;
Returns: int

CMNA_rdr_receive_msg(base,word_length)
void *base;
int word_length;
```


Copyright © 1994 Thinking Machines Corporation

```
CMNA_rdr_send_fifo_amount(dest_proc,source_base,
                          word_length,tag)
unsigned int dest_proc;
void *source_base;
int word_length;
unsigned int tag;
Returns: int
```

```
CMNA_rdr_send_fifo_amount_physical(dest_proc,source_base,
                                   word_length,tag)
void *source_base;
int word_length;
unsigned dest_proc;
unsigned tag;
Returns: int
```

```
CMNA_rdr_send_msg(dest_proc,source_base,word_length,tag)
unsigned int dest_proc;
void *source_base;
int word_length;
unsigned int tag;
```

```
CMNA_rdr_send_msg_physical(dest_proc,source_base,
                           word_length,tag)
void *source_base;
int word_length;
unsigned dest_proc;
unsigned tag;
```

H.3.6 BC Interface Functions

```
CMNA_bc_read_double()
Returns: double
```

```
CMNA_bc_read_float()
Returns: float
```

```
CMNA_bc_read_int()
Returns: int
```

```
CMNA_bc_read_uint()
Returns: unsigned
```

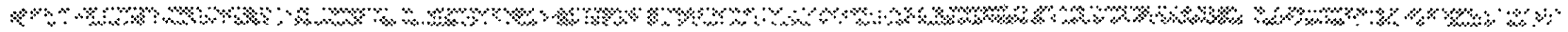

Copyright © 1994 Thinking Machines Corporation

```
CMNA_bc_receive(msg,length)
void *msg;
unsigned int length;

CMNA_bc_receive_participation()
Returns: int

CMNA_bc_send_and_receive_msg(msg,result,length)
void *msg;
void *result;
int length;

CMNA_bc_send_fifo_amount(msg,length)
void *msg;
int length;

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

CMNA_bc_wait_for_receive_ok()
Returns: int

CMNA_bc_write_double(data)
double data;

CMNA_bc_write_float(data)
float data;

CMNA_bc_write_int(data)
int data;

CMNA_bc_write_uint(data)
unsigned int data;
```

H.3.7 SBC Interface Functions

```
CMNA_sbc_double(data)
double data;

CMNA_sbc_float(data)
double data;
```


Copyright © 1994 Thinking Machines Corporation



```
CMNA_sbc_int(data)
int data;

CMNA_sbc_receive(msg,length)
void *msg;
unsigned length;

CMNA_sbc_send(msg,length)
void *msg;
int length;

CMNA_sbc_send_msg(msg,length)
void *msg;
int length;
Returns: int

CMNA_sbc_uint(data)
unsigned int data;

CMNA_sbc_wait_for_receive_ok()
Returns: int

CMNA_sup_dr_send_packet_to_scalar
              (source_base,word_length,tag)
void *source_base;
int word_length;
unsigned tag;
Returns: int

CMNA_sup_ldr_send_packet_to_scalar
              (source_base,word_length,tag)
void *source_base;
int word_length;
unsigned tag;
Returns: int

CMNA_sup_rdr_send_packet_to_scalar
              (source_base,word_length,tag)
void *source_base;
int word_length;
unsigned tag;
Returns: int
```


Copyright © 1994 Thinking Machines Corporation

H.3.8 COM Interface Functions

```
CMNA_com(combiner,pattern,data,length,result)
int combiner;
int pattern;
void *data;
int length;
void *result;

CMNA_com_participation()
Returns: int

CMNA_com_receive(result)
unsigned int *result;

CMNA_com_send(combiner,pattern,data,word_length)
int combiner;
int pattern;
void *data;
int word_length;

CMNA_reduce_rec_participation()
Returns: int
```

H.3.9 Global Interface Functions

```
CMNA_global_async(value)
unsigned int value;
Returns: int

CMNA_global_sync(value)
unsigned int value;
Returns: int

CMNA_global_sync_read()
Returns: int

CMNA_global_sync_read_when_ready()
Returns: int

CMNA_global_sync_participation()
Returns: int
```


Copyright © 1994 Thinking Machines Corporation

Appendix I

NI Chip Version 2.2 Changes

This appendix presents a summary of the additions and changes made to the NI chip as of the most recent chip version (2.2), and indicates where they are described in the main body of this manual.

I.1 Long Data Network Messages

The Data Network now has the capability to send *long messages*. These messages, sent by a special register interface, have a length in data bytes that is much longer than the limit imposed on Data Network messages in Version 1.0. (Currently, the long message length limit is 18 words of data.)

Several new Data Network registers are introduced to support this feature:

```
ni_dr/ldr/rdr_send_first_long
ni_dr/ldr/rdr_status_long
ni_longest_dr_message
```

The long message register interface is described in detail in Chapter 3. Section 3.4.2 in particular describes how to send long messages.

Copyright © 1994 Thinking Machines Corporation

I.2 New Data Network Status Interface

Registers have been added to allow more convenient access to the status information of the Data Network's message FIFOs, and to allow "popping" of messages from the Data Network receive FIFOs at the same time:

```
ni_dr/ldr/rdr_status_all
ni_ldr/rdr_status_pop
```

These registers are described in Section 3.5.

I.3 New Data Network Tag Interrupt Interface

The mechanism for detecting and signaling interrupts based on the tag values of Data Network messages has been changed to allow more precise control over the selection of user and supervisor tag values. The new mechanism is described in detail in Section 3.5.4.

I.4 Non-Compatible Change to Broadcast Interface

The `ni_rec_length_left` field in the `ni_bc_status` and `ni_sbc_status` registers has been expanded from 4 bits to 7 bits in length to handle the change in the maximum broadcast message length. This means that software written for earlier versions of the NI chip may not execute properly; if only the first four bits of this field are extracted, the software cannot determine whether the value thus obtained is correct.

The basic fix for this problem is to have your code extract 7 bits rather than 4 for this field. If your code uses the predefined NI constants, you should substitute the new length constant `NI_BC_REC_LENGTH_LEFT_LONG_L` for all references to the `rec_length_left` field.

Copyright © 1994 Thinking Machines Corporation

I.5 New Interrupts

A number of new interrupts have been added as of Version 2.1:

`rdone complete`	(Orange) — Completion of network-done operation
`message too long`	(Yellow) — Data Network message length error
`dperr`	(Green) — Error signaled by CM-5 vector units
`sfifo empty`	(Green) — Data Network send fifo empty
`ldr rec tag`	(Green) — LDR supervisor message tag interrupt
`rdr rec tag`	(Green) — RDR supervisor message tag interrupt
`ldr user rec tag`	(Green) — LDR supervisor message tag interrupt
`rdr user rec tag`	(Green) — RDR supervisor message tag interrupt

These interrupts are described in Chapter 5, and in Appendix B.

I.6 New Data Network Interrupt Enable Flags

The following flags have been added to the Data Network private registers to allow enabling and disabling of the corresponding interrupts:

```
ni_sfifo_goes_empty_ie
ni_rdone_complete_ie
```

These flags are described in Section 3.8.

I.7 New Bus Error Conditions

The following bus error conditions now exist, in connection with the long message feature of the Data Network:

- writing a message to any of the Data Network's `send_first` registers with a length value that is greater than either five words or the value of the register `ni_longest_dr_message`, whichever is less

- writing a message to any of the Data Network's `send_first_long` registers with a length value that is greater than the value of the register `ni_longest_dr_message`

Copyright © 1994 Thinking Machines Corporation

I.8 Disabling Bus Errors

The flag `ni_disable_bus_error` in the `hodgepodge` register, when set, causes the NI to signal bus errors as a yellow interrupt, `bad_memory_access`. (See Section 5.1.1.)

I.9 Manually Triggering Interrupts

Interrupts can be triggered artificially by writing to the new registers `ni_interrupt_set` and `ni_interrupt_set_green` (See Section 5.3).

I.10 Global Interface Context-Switching

A supervisor method for context-switching and then restoring the state of the synchronous global interface is described in Section 4.3.2.

I.11 New Hodgepodge Register Fields

The following fields have been added to the "hodgepodge" register to support various new NI features:

`ni_disable_bus_error`	(See Section 5.1.1.)
`ni_ldr_rec_tag_ie`	(See Section 3.5.4.)
`ni_rdr_rec_tag_ie`	"
`ni_ldr_user_rec_tag_ie`	"
`ni_rdr_user_rec_tag_ie`	"
`ni_msg_too_long_ie`	(See Section 3.2.2.)

Copyright © 1994 Thinking Machines Corporation

Index

Copyright © 1994 Thinking Machines Corporation

Programming Tools Index

This index lists the register names and fields, programming constants, functions, and macros referred to within this document. Bold page numbers indicate a defining reference or important description.

A

ADD_SCAN
- combine combiner constant, **75**, **152**
- combiner constant, 77, 186

ASSERT_ROUTER_DONE
- combine combiner constant, **75**, **152**
- combiner constant, 79, 186

AUXILIARY_START_P, send-first field offset constant, **21**, **149**

B

bad memory access
- bus error, **97**, **178**
- Yellow interrupt, 96, **98**, 100, 170

bad relative address, Yellow interrupt, **41**, 96, 100, 172

bc interrupt green, Green interrupt, 97, 101, **104**, 173

bc interrupt orange, Orange interrupt, 96, 100, **104**, 169

bc interrupt red, Red interrupt, 96, 100, **104**, 168

bc interrupt yellow, Yellow interrupt, 96, 100, **104**, 170

bc or com collision, Yellow interrupt, **73**, 96, 100, **104**, 171

bc rec ok, Green interrupt, **30**, 174

bc_control_reg, constant, **69**, 185

C

CMNA_bc_receive_type(), macro, **68**, 184

CMNA_bc_send_first(), macro, **67**, 184

CMNA_bc_send_first_double(), macro, **67**, 184

CMNA_bc_send_type(), macro, **67**, 184

CMNA_bc_status(), macro, **68**, 184

CMNA_com_receive_type(), macro, **76**, 186

CMNA_com_send_first(), macro, **74**, 186

CMNA_com_send_first_double(), macro, **74**, 186

CMNA_com_send_type(), macro, **74**, 186

CMNA_com_status(), macro, **76**, 187

CMNA_*dinterface*_receive_type(), macro, **44**, 182

CMNA_*dinterface*_send_first(), macro, **42**, 182

CMNA_*dinterface*_send_first_double, macro, **42**, 182

CMNA_*dinterface*_send_first_double_long, macro, **43**

CMNA_*dinterface*_send_first_long(), macro, **43**

CMNA_*dinterface*_send_*type*, macro, **42**, 182

CMNA_*dinterface*_status(), macro, **46**, 183

CMNA_dr_send_status(), macro, **46**, 183

Copyright © 1994 Thinking Machines Corporation

CMNA_global_async_read(), macro, **92**, 189
CMNA_global_sync_complete(), macro, **90**, 188
CMNA_global_sync_rec(), macro, **90**, 188
CMNA_*interface*_receive_*type*, macro, **23**
CMNA_*interface*_send_first(), macro, **22**
CMNA_*interface*_send_first_double, macro, **22**
CMNA_interface_send_packet_to_scalar(), system function, 122
CMNA_*interface*_send_*type()*, macro, **23**
CMNA_*interface*_status(), macro, **26**
CMNA_ldr_status(), macro, **46**, 183
CMNA_or_global_async_bit(), macro, **92**, 189
CMNA_or_global_sync_bit(), macro, **90**, 188
CMNA_participate_in()
- system call, **141**
- system function, **70**, 124

CMNA_partition_size, variable, **40**, 181
CMNA_rdr_status(), macro, **46**, 183
CMNA_read_abstain_flag(), macro, **28**, 181
CMNA_segment_start(), macro, **78**, 187
CMNA_self_address, variable, **40**, 181
CMNA_set_segment_start(), macro, **78**, 187
CMNA_write_abstain_flag(), macro, **28**, 181
CMOS_signal(), system call, **51**, 199
cmu error, Red interrupt, 96, **99**, 167
cn checksum error, Red interrupt, 96, **99**, 166
cn hard error, Red interrupt, 96, **99**, 167
com abstain changed, Yellow interrupt, **82**, 96, 100, 170
com rec empty, Green interrupt, **83**, 97, 101, 175
com rec ok, Green interrupt, **30**, 97, 101, 174
com_control_reg, constant, **82**, 188
COMBINE_FIFO, interface number constant, **22**, **150**
COMBINE_OVERFLOW(), macro, **79**, 187

D

DATA_ROUTER_FIFO, interface number constant, **22**, **150**
dp error, Green interrupt, 176
dperr, Green interrupt, 97, 101
dr checksum error, Red interrupt, 96, **99**, 166
dr count negative, Yellow interrupt, **52**, 96, 100, 171
dr rec all fall down, Green interrupt, **55**, 97, 101, 174
dr rec ok, Green interrupt, **30**, 97, 101, 174
dr rec tag, Green interrupt, **49**, 97, 101, 174
DR_RECEIVE_STATE(), macro, **53**, 183
DR_ROUTER_DONE(), macro, **54**, 80, 183
DR_SEND_STATE(), macro, **53**, 183

G

global rec, Green interrupt, **93**, 97, 101, 175

I

internal fault, Red interrupt, 96, **99**, 166

L

ldr rec ok, Green interrupt, **30**, 97, 101, 174
ldr rec tag, Green interrupt, 97
ldr tag, Green interrupt, 177
ldr user rec ok, Green interrupt, 101
ldr user rec tag, Green interrupt, 97

Copyright © 1994 Thinking Machines Corporation

ldr user tag, Green interrupt, 177
LEFT_DR_FIFO, interface number constant, 22, **150**

M

MAX_BROADCAST_MSG_WORDS, constant, **66**, 67, **149**, 185
MAX_COMBINE_MSG_WORDS, constant, **73**, **149**, 187
MAX_ROUTER_MSG_WORDS, constant, **41**, 43, **149**, 184
MAX_SBC_MSG_WORDS, constant, **66**, 67, **149**
MAX_SCAN
- combine combiner constant, **75**, **152**
- combiner constant, 77, 186

mc error, Red interrupt, 96, **99**, 167
message too long, Yellow interrupt, 96, 100, 172

N

ni_all_fall_down_enable, flag, 54, **55**, **154**
ni_all_fall_down_ie, flag, 54, **55**, **154**
ni_async_global, register, 89, **92**, **146**, **160**, 189
ni_async_sup_global, register, 89, **93**, **146**, **160**
ni_bad_address, register, **114**, **146**, **164**
ni_bad_address_low, field, **114**, **164**
ni_bad_address_type, field, **114**, **164**
NI_BASE, constant, **12**, **21**, **149**
ni_bc_..., register. *See* ni_*binterface*_...
ni_bc_control, register, **148**, **158**, 185
ni_bc_private, register, **148**, **158**
ni_bc_recv, register, **148**, 184
ni_bc_send, register, **148**, 184
NI_BC_SEND_AUXILIARY_LENGTH_B field offset, **67**, **151**
ni_bc_send_first, register, **148**, 184
ni_bc_status, register, **148**, **157**, 185
ni_*binterface*_control, register, 64, **69**
ni_*binterface*_private, register, 64, **69**
ni_*binterface*_recv, register, 64, **67**
ni_*binterface*_send, register, 64, **67**
ni_*binterface*_send_first, register, 64, **67**
ni_*binterface*_status, register, 64, **68**
ni_cause_bad_memory_access, flag, **161**
ni_cause_bad_relative_address, flag, **161**
ni_cause_bc_interrupt_green, flag, **161**
ni_cause_bc_interrupt_orange, flag, **161**
ni_cause_bc_interrupt_red, flag, **161**
ni_cause_bc_interrupt_yellow, flag, **161**
ni_cause_bc_or_com_collision, flag, **161**
ni_cause_bc_rec_ok, flag, **161**
ni_cause_cmu_error, flag, **161**
ni_cause_cn_checksum_error, flag, **161**
ni_cause_cn_hard_error, flag, **161**
ni_cause_com_abstain_changed, flag, **161**
ni_cause_com_rec_empty, flag, **161**
ni_cause_com_rec_ok, flag, **161**
ni_cause_dperr, flag, **161**
ni_cause_dr_checksum_error, flag, **161**
ni_cause_dr_count_negative, flag, **161**
ni_cause_dr_rec_all_fall_down, flag, **161**
ni_cause_dr_rec_ok, flag, **161**
ni_cause_dr_rec_tag, flag, **161**
ni_cause_global_rec, flag, **161**
ni_cause_internal_fault, flag, **161**
ni_cause_ldr_rec_ok, flag, **161**
ni_cause_ldr_rec_tag, flag, **161**
ni_cause_ldr_user_rec_tag, flag, **161**
ni_cause_mc_error, flag, **161**
ni_cause_message_too_long, flag, **161**
ni_cause_rdone_complete, flag, **161**
ni_cause_rdr_rec_ok, flag, **161**
ni_cause_rdr_rec_tag, flag, **161**
ni_cause_rdr_user_rec_tag, flag, **161**
ni_cause_sbc_rec_ok, flag, **161**
ni_cause_scan_overflow, flag, **161**
ni_cause_sfifo_empty, flag, **161**

Copyright © 1994 Thinking Machines Corporation

ni_cause_supervisor_global_rec, flag, **161**
ni_cause_sync_global_rec, flag, **161**
ni_cause_timer_interrupt, flag, **161**
ni_chunk_size, register, **110**, **146**
ni_chunk_table_address, register, **112**, **146**
ni_chunk_table_data, register, **112**, **146**
ni_clear_bad_memory_access, flag, **162**
ni_clear_bad_relative_address, flag, **162**
ni_clear_bc_interrupt_green, flag, **162**
ni_clear_bc_interrupt_orange, flag, **162**
ni_clear_bc_interrupt_red, flag, **162**
ni_clear_bc_interrupt_yellow, flag, **162**
ni_clear_bc_or_com_collision, flag, **162**
ni_clear_bc_rec_ok, flag, **162**
ni_clear_cmu_error, flag, **162**
ni_clear_cn_checksum_error, flag, **162**
ni_clear_cn_hard_error, flag, **162**
ni_clear_com_abstain_changed, flag, **162**
ni_clear_com_rec_empty, flag, **162**
ni_clear_com_rec_ok, flag, **162**
ni_clear_dperr, flag, **162**
ni_clear_dr_checksum_error, flag, **162**
ni_clear_dr_count_negative, flag, **162**
ni_clear_dr_rec_all_fall_down, flag, **162**
ni_clear_dr_rec_ok, flag, **162**
ni_clear_dr_rec_tag, flag, **162**
ni_clear_global_rec, flag, **162**
ni_clear_internal_fault, flag, **162**
ni_clear_ldr_rec_ok, flag, **162**
ni_clear_ldr_rec_tag, flag, **162**
ni_clear_ldr_user_rec_tag, flag, **162**
ni_clear_mc_error, flag, **162**
ni_clear_message_too_long, flag, **162**
ni_clear_rdone_complete, flag, **162**
ni_clear_rdr_rec_ok, flag, **162**
ni_clear_rdr_rec_tag, flag, **162**
ni_clear_rdr_user_rec_tag, flag, **162**
ni_clear_sbc_rec_ok, flag, **162**
ni_clear_scan_overflow, flag, **162**
ni_clear_sfifo_empty, flag, **162**
ni_clear_supervisor_global_rec, flag, **162**
ni_clear_sync_global_rec, flag, **162**
ni_clear_timer_interrupt, flag, **162**
ni_cn_stop_send, flag, 107, **116**, **163**
ni_rec_abstain, flag
  of a network, **27**, 28
  of broadcast interface, **69**
  of combine interface, **81**
ni_com_control, register, 72, **81**, **148**, **160**, 188
ni_com_flush_send, register, **113**, **146**
ni_com_private, register, 72, **83**, **148**, **159**
ni_com_rec_empty_ie, flag, **83**
  in **ni_com_private** register, **159**
ni_com_recv, register, 72, **76**, **148**, 186
ni_com_scan_overflow, flag, 76, **78**
  in **ni_com_status** register, **159**
  of combine interface, 187
ni_com_scan_overflow_ie, flag, **79**, 83
  in **ni_com_private** register, **159**
ni_com_send, register, 72, **73**, 83, **148**, 186
NI_COM_SEND_AUXILIARY_COMBINER_P, field offset, **74**, **151**
NI_COM_SEND_AUXILIARY_LENGTH_P, field offset, **74**, **151**
NI_COM_SEND_AUXILIARY_PATTERN_P, field offset, **74**, **151**
ni_com_send_combiner, field, 83, **84**
  in **ni_com_private** register, **159**
ni_com_send_first, register, 72, **73**, **148**, 186
ni_com_send_length, field, 83, **84**
  in **ni_com_private** register, **159**
ni_com_send_pattern, field, 83, **84**
  in **ni_com_private** register, **159**
ni_com_send_start, flag, 83, **84**
  in **ni_com_private** register, **159**
ni_com_status, register, 72, **76**, 78, **148**, **159**, 187
ni_configuration, register, **115**, **146**

NI Version 2.2 (CM-5E), June 1994
Copyright © 1994 Thinking Machines Corporation

ni_configuration_complete, flag, 107, **115**, **163**
ni_count_mask, register, 38, **51**, 80, **146**
ni_*dinterface*_private, register, 36, **54**
ni_*dinterface*_rec_tag, field, 183
ni_*dinterface*_recv, register, 36, **44**, 182
ni_*dinterface*_send, register, 36, **42**, 182
ni_*dinterface*_send_first, register, 36, **42**, 182
ni_*dinterface*_send_first_long, register, 36, 43
ni_*dinterface*_status, register, 36, **45**, 80
ni_*dinterface*_status(), register, 183
ni_*dinterface*_status_all, register, 36, 46
ni_*dinterface*_status_long, register, 36
ni_*dinterface*_status_pop, register, 36, 47
ni_disable_bus_error, flag, 98, 107, **163**
ni_dr_.... *See* ni_*dinterface*_...
ni_dr_message_count, register, 38, **51**, 54, 80, **146**
ni_dr_private, register, **147**, **154**
ni_dr_rec_all_fall_down, flag, 54, **55**
  in ni_dr_private register, **154**
  in ni_ldr_private register, **156**
  in ni_rdr_private register, **157**
ni_dr_rec_state, field, 45, **53**, **153**, **154**
ni_dr_rec_tag, field, **45**
  in ni_dr_status register, **153**
  in ni_ldr_status register, **155**
  in ni_ldr_status_long register, **155**
  in ni_rdr_status register, **156**
  in ni_rdr_status_long register, **156**
  in ni_dr_status_long register, **154**
ni_dr_send, register, **147**
NI_DR_SEND_AUXILIARY_ADDRESS_MODE_P, offset constant, **42**, **150**
NI_DR_SEND_AUXILIARY_LENGTH_P, offset constant, **42**, **150**
NI_DR_SEND_AUXILIARY_TAG_P, offset constant, **42**, **150**
ni_dr_send_first, register, **147**
ni_dr_send_first_long, register, **147**
ni_dr_send_ok, flag, in ni_dr_status_all/pop register, **46**, **154**
ni_dr_send_space, field, in ni_dr_status_all/pop register, **46**, **154**
ni_dr_send_state, field, 45, **53**, **153**, **154**
ni_dr_status, register, **147**, **153**
ni_dr_status_all, register, **147**
ni_dr_status_all/pop, register, **154**
ni_dr_status_long, register, **147**, **154**
ni_flush_complete, flag, 107, **113**, **163**
ni_global_rec, flag, **92**, **160**, 189
ni_global_rec_ie, flag, 92, **93**, 107, **163**
ni_global_send, flag, **92**, **160**, 189
ni_hodgepodge, register, **38**, **49**, **98**, **107**, **146**, **163**
  and asynchronous global interface, **89**
  and supervisor asynch global interface, **89**
  and synchronous global interface, **89**
  asynch global rec interrupt enable flag, 92, **93**
  broadcast interrupt flags, **104**
  configuration flag, **115**
  flush complete flag, **113**
  NI timer interrupt enable flag, **113**
  send stop flag, **116**
  supervisor rec interrupt enable flag, **93**
  synch global rec interrupt enable flag, 89, **91**
ni_*interface*_control, register, 17, **25**, **27**
ni_*interface*_private, register, 17, **30**
ni_*interface_purpose*, register naming format, **10**
ni_*interface*_recv, register, 17, **23**
ni_*interface*_send, register, 17, **19**
ni_*interface*_send_first, register, 17, **19**, **149**
ni_*interface*_send_first_long, register, **43**, **152**
ni_*interface*_status, register, 17
ni_interrupt_cause, register, **102**, **146**, **161**
ni_interrupt_cause_green, register, **102**, **146**, **161**
ni_interrupt_clear, register, **102**, **146**, **162**

Copyright © 1994 Thinking Machines Corporation

ni_interrupt_clear_green, register, **102**, **146**, **162**
ni_interrupt_level, register, **103**, **146**, **163**
ni_interrupt_level_green, field, **103**, **163**
ni_interrupt_level_orange, field, **103**, **163**
ni_interrupt_level_red, field, **103**, **163**
ni_interrupt_level_yellow, field, **103**, **163**
ni_interrupt_now, register, **113**, **146**
ni_interrupt_rec_enable, flag, **104**, 107, **163**
ni_interrupt_send, register, **104**, **146**
ni_interrupt_send_ok, flag, **104**, 107, **163**
ni_interrupt_set, register, **146**, **162**
ni_interrupt_set_green, register, **146**, **162**
ni_ldr_.... *See* ni_*dinterface_...*
ni_ldr_private, register, **147**, **156**
ni_ldr_rec_all_fall_down, flag
  in ni_dr_status_all/pop register, **46**, **154**
  in ni_ldr_status_all/pop register, **155**
  in ni_rdr_status_all/pop register, **157**
ni_ldr_rec_length, field
  in ni_dr_status_all/pop register, **154**
  in ni_ldr_status_all/pop register, **155**
  in ni_rdr_status_all/pop register, **157**
ni_ldr_rec_length_long, field, in ni_dr_status_all/pop register, **46**
ni_ldr_rec_ok, flag
  in ni_dr_status_all/pop register, **46**, **154**
  in ni_ldr_status_all/pop register, **155**
  in ni_rdr_status_all/pop register, **157**
ni_ldr_rec_tag, field
  in ni_dr_status_all/pop register, **46**, **154**
  in ni_ldr_status_all/pop register, **155**
  in ni_rdr_status_all/pop register, **157**
ni_ldr_rec_tag_ie, flag, 49, 107, **163**
ni_ldr_recv, register, **147**
ni_ldr_send, register, **147**
ni_ldr_send_first, register, **147**
ni_ldr_send_first_long, register, **147**
ni_ldr_send_ok, flag, in ni_ldr_status_all/pop register, **155**
ni_ldr_send_space, field, in ni_ldr_status_all/pop register, **155**
ni_ldr_status, register, **147**, **155**
ni_ldr_status_all, register, **147**
ni_ldr_status_all/pop, register, **155**
ni_ldr_status_long, register, **147**, **155**
ni_ldr_status_pop, register, **147**
ni_ldr_user_rec_tag_ie, flag, 49, 107, **163**
ni_lock, flag
  in ni_bc_private register, **158**
  in ni_com_private register, **159**
  in ni_dr_private register, **154**
  in ni_ldr_private register, **156**
  in ni_rdr_private register, **157**
  in ni_sbc_private register, **158**
  of a network, 30, **31**
  of broadcast interface, **69**
  of combine interface, **83**
  of Data Networks, **54**
ni_longest_dr_message, register, 38, **39**, **146**
ni_message_too_long_ie, flag, 38, 107
ni_msg_too_long_ie, flag, **163**
ni_partition_base, register, 108, **110**, **146**
ni_partition_size, register, 108, **109**, **146**
ni_physical_self, register, **108**, **146**

NI Version 2.2 (CM-5E), June 1994
Copyright © 1994 Thinking Machines Corporation

ni_rdone_complete_ie, flag, 54, 57, **154**
ni_rdr_.... *See* **ni_***dinterface_*...
ni_rdr_private, register, **147**, **157**
ni_rdr_rec_all_fall_down, flag
in **ni_dr_status_all/pop** register, **46**, **154**
in **ni_ldr_status_all/pop** register, **155**
in **ni_rdr_status_all/pop** register, **157**
ni_rdr_rec_length, field
in **ni_dr_status_all/pop** register, **154**
in **ni_ldr_status_all/pop** register, **155**
in **ni_rdr_status_all/pop** register, **157**
ni_rdr_rec_length_long, field, in **ni_dr_status_all/pop** register, **46**
ni_rdr_rec_ok, flag
in **ni_dr_status_all/pop** register, **46**, **154**
in **ni_ldr_status_all/pop** register, **155**
in **ni_rdr_status_all/pop** register, **157**
ni_rdr_rec_tag, field
in **ni_dr_status_all/pop** register, **46**, **154**
in **ni_ldr_status_all/pop** register, **155**
in **ni_rdr_status_all/pop** register, **157**
ni_rdr_rec_tag_ie, flag, 49, 107, **163**
ni_rdr_recv, register, **147**
ni_rdr_send, register, **147**
ni_rdr_send_first, register, **147**
ni_rdr_send_first_long, register, **147**
ni_rdr_send_ok, flag, in **ni_rdr_status_all/pop** register, **157**
ni_rdr_send_space, field, in **ni_rdr_status_all/pop** register, **157**
ni_rdr_status, register, **147**, **156**
ni_rdr_status_all, register, **147**
ni_rdr_status_all/pop, register, **157**
ni_rdr_status_long, register, **147**, **156**
ni_rdr_status_pop, register, **147**
ni_rdr_user_rec_tag_ie, flag, 49, 107, **163**
ni_rec_abstain, flag
in **ni_bc_control** register, **158**
in **ni_com_control** register, **160**
in **ni_sbc_control** register, **159**
of broadcast interface, 185
of combine interface, 188
ni_rec_full, flag
in **ni_bc_private** register, **158**
in **ni_com_private** register, **159**
in **ni_dr_private** register, **154**
in **ni_ldr_private** register, **156**
in **ni_rdr_private** register, **157**
in **ni_sbc_private** register, **158**
of a network, 30, **32**
of broadcast interface, **69**
of combine interface, **83**
of Data Networks, **54**
ni_rec_interrupt_mask, register, 38, **49**, **146**
ni_rec_length, field
in **ni_com_status** register, **159**
in **ni_dr_status** register, **153**
in **ni_dr_status_long** register, **154**
in **ni_ldr_status** register, **155**
in **ni_ldr_status_long** register, **155**
in **ni_rdr_status** register, **156**
in **ni_rdr_status_long** register, **156**
of a network, 25, **26**
of a network interface, 181
of combine interface, **76**, 187
of Data Networks, **45**, 183
ni_rec_length_left
field
in **ni_bc_status** register, **157**, **238**
in **ni_com_status** register, **159**
in **ni_dr_status** register, **153**
in **ni_dr_status_long** register, **154**
in **ni_ldr_status** register, **155**
in **ni_ldr_status_long** register, **155**

Copyright © 1994 Thinking Machines Corporation

in ni_rdr_status register, **156**
in ni_rdr_status_long register, **156**
in ni_sbc_status register, **158**
in ni_sbc_status register, 238
of a network, 25, **26**
of broadcast interface, **68**, 68, **185**
of combine interface, **76**, 187
of Data Networks, **45**, 183
flag, of a network interface, 181
ni_rec_ok, flag
in ni_bc_status register, **157**
in ni_com_status register, **159**
in ni_dr_status register, **153**
in ni_dr_status_long register, **154**
in ni_ldr_status register, **155**
in ni_ldr_status_long register, **155**
in ni_rdr_status register, **156**
in ni_rdr_status_long register, **156**
in ni_sbc_status register, **158**
of a network, 25, **26**
of a network interface, 181
of broadcast interface, **68**, 185
of combine interface, **76**, 187
of Data Networks, **45**, 183
ni_rec_ok_ie, flag
in ni_bc_private register, **158**
in ni_dr_private register, **154**
in ni_ldr_private register, **156**
in ni_rdr_private register, **157**
in ni_sbc_private register, **158**
in ni_com_private register, **159**
of a network, **30**, 30
of broadcast interface, **69**
of combine interface, **83**
of Data Networks, **54**
ni_rec_state, field, of Data Networks, 183
ni_rec_stop, flag
in ni_bc_private register, **158**
in ni_com_private register, **159**
in ni_dr_private register, **154**
in ni_sbc_private register, **158**
of a network, 30, **31**
of combine interface, **83**
of Data Networks, **54**
ni_rec_tag, field, of Data Networks, 183
ni_reduce_rec_abstain, flag, **81**, **160**, 188
of combine interface, 27, 28
ni_router_done_complete, flag, 45, **46**, **54**, **76**, 80, **153**, **154**
in ni_ldr_status_all/pop register, **155**
in ni_rdr_status_all/pop register, **157**
of Data Networks, 183
ni_sbc_..., register. *See* ni_*binterface*_...
ni_sbc_control, register, **148**, **159**
ni_sbc_private, register, **148**, **158**
ni_sbc_recv, register, **148**
ni_sbc_send, register, **148**
ni_sbc_send_first, register, **148**
ni_sbc_status, register, **148**, **158**
ni_scan_start, register, 72, **78**, **146**, 187
ni_send_empty, flag
in ni_bc_status register, **157**
in ni_com_status register, **159**
in ni_sbc_status register, **158**
of a network, 25
of a network interface, 181
of broadcast interface, **68**, 185
of combine interface, **76**, 187
ni_send_enable, flag
in ni_bc_private register, **158**
in ni_sbc_private register, **158**
of broadcast interface, **69**, 69
ni_send_ok, flag
for Data Networks, **45**
in ni_bc_status register, **157**
in ni_com_status register, **159**
in ni_dr_status register, **153**
in ni_dr_status_long register, **154**
in ni_ldr_status register, **155**
in ni_ldr_status_long register, **155**
in ni_rdr_status register, **156**
in ni_rdr_status_long register, **156**
in ni_sbc_status register, **158**
of a network, **25**, 25
of a network interface, 181
of broadcast interface, **68**, 185

Copyright © 1994 Thinking Machines Corporation

of combine interface, **76**, 187
of Data Networks, 183
`ni_send_space`, field
in `ni_bc_status` register, **157**
in `ni_com_status` register, **159**
in `ni_dr_status` register, **153**
in `ni_dr_status_long` register, **154**
in `ni_ldr_status` register, **155**
in `ni_ldr_status_long` register, **155**
in `ni_rdr_status` register, **156**
in `ni_rdr_status_long` register, **156**
in `ni_sbc_status` register, **158**
of a network, **25**, 26
of a network interface, 181
of broadcast interface, **68**, 185
of combine interface, **76**, 187
of Data Networks, **45**, 183
`ni_send_state`, field, of Data Networks, 183
`ni_send_stop`, flag, of broadcast interface, 30, **32**, **69**
`ni_serial_number`, register, **116**, **146**
`ni_set_bad_memory_access`, flag, **162**
`ni_set_bad_relative_address`, flag, **162**
`ni_set_bc_interrupt_green`, flag, **162**
`ni_set_bc_interrupt_orange`, flag, **162**
`ni_set_bc_interrupt_red`, flag, **162**
`ni_set_bc_interrupt_yellow`, flag, **162**
`ni_set_bc_or_com_collision`, flag, **162**
`ni_set_bc_rec_ok`, flag, **162**
`ni_set_cmu_error`, flag, **162**
`ni_set_cn_checksum_error`, flag, **162**
`ni_set_cn_hard_error`, flag, **162**
`ni_set_com_abstain_changed`, flag, **162**
`ni_set_com_rec_empty`, flag, **162**
`ni_set_com_rec_ok`, flag, **162**
`ni_set_dperr`, flag, **162**
`ni_set_dr_checksum_error`, flag, **162**
`ni_set_dr_count_negative`, flag, **162**
`ni_set_dr_rec_all_fall_down`, flag, **162**
`ni_set_dr_rec_ok`, flag, **162**
`ni_set_dr_rec_tag`, flag, **162**
`ni_set_global_rec`, flag, **162**
`ni_set_internal_fault`, flag, **162**
`ni_set_ldr_rec_ok`, flag, **162**
`ni_set_ldr_rec_tag`, flag, **162**
`ni_set_ldr_user_rec_tag`, flag, **162**
`ni_set_mc_error`, flag, **162**
`ni_set_message_too_long`, flag, **162**
`ni_set_rdone_complete`, flag, **162**
`ni_set_rdr_rec_ok`, flag, **162**
`ni_set_rdr_rec_tag`, flag, **162**
`ni_set_rdr_user_rec_tag`, flag, **162**
`ni_set_sbc_rec_ok`, flag, **162**
`ni_set_scan_overflow`, flag, **162**
`ni_set_sfifo_empty`, flag, **162**
`ni_set_supervisor_global_rec`, flag, **162**
`ni_set_sync_global_rec`, flag, **162**
`ni_set_timer_interrupt`, flag, **162**
`ni_sfifo_goes_empty_ie`, flag, 54, 57, **154**
`ni_supervisor_global_rec`, flag, **93**, **160**
`ni_supervisor_global_rec_ie`, flag, **93**, 107, **163**
`ni_supervisor_global_send`, flag, **93**, **160**
`ni_sync_global`, register, **89**, 89, **146**, **160**, 188
`ni_sync_global_abstain`, register, 89, **90**, **146**, 188
`ni_sync_global_complete`, flag, 89, **90**, **160**, 188
`ni_sync_global_rec`, flag, 89, **90**, **160**, 188
`ni_sync_global_rec_ie`, flag, 89, **91**, 107, **163**
`ni_sync_global_send`, register, 89, **90**, **146**, 188
`ni_time`, register, **113**, **146**
`ni_timer_ie`, flag, 107, **113**, **163**
`ni_user_rec_interrupt_mask`, register, 38, 49, **146**
`ni_user_tag_mask`, register, 38, **48**, **146**

Copyright © 1994 Thinking Machines Corporation

O

OR_SCAN
 combine combiner constant, **75**, **152**
 combiner constant, 77, 186

P

PHYSICAL, flag value constant, **43**, **150**

R

rdone complete, Orange interrupt, 96, 100
rdr rec ok, Green interrupt, **30**, 97, 101, 174
rdr rec tag, Green interrupt, 97
rdr tag, Green interrupt, 177
rdr user rec ok, Green interrupt, 101
rdr user rec tag, Green interrupt, 97
rdr user tag, Green interrupt, 177
RECEIVE_LENGTH(), macro, **27**, 181
RECEIVE_LENGTH_LEFT(), macro, **27**, 181
RECEIVE_OK(), macro, **27**, 181
RECEIVE_TAG(), macro, **48**, 183
RELATIVE, flag value constant, **43**, **150**
RIGHT_DR_FIFO, interface number constant, **22**, **150**
router done complete, Orange interrupt, **57**, **169**

S

sbc rec ok, Green interrupt, **30**, 97, 101, 174
scan overflow, Green interrupt, **79**, 97, 101, 176
SCAN_BACKWARD
 combine pattern constant, **75**, **151**
 pattern constant, 77, 186
SCAN_FORWARD
 combine pattern constant, **75**, **151**
 pattern constant, 77, 186
SCAN_REDUCE
 combine pattern constant, **75**, **152**
 pattern constant, 77, 186
SCAN_ROUTER_DONE
 combine pattern constant, **75**, **152**
 pattern constant, 79, 186
send fifo empty, Green interrupt, 57, 177
SEND_EMPTY(), macro, **27**, 181
SEND_OK(), macro, **27**, 181
SEND_SPACE(), macro, **27**, 181
SF_FIFO_OFFSET, send-first field offset constant, **21**, **149**
sfifo empty, Green interrupt, 97, 101
sp-pe-stubs, preprocessor, **127**
supervisor global rec, Green interrupt, **93**, 97, 101, 175
SUPERVISOR_BC_FIFO, interface number constant, **22**, **150**
sync global rec, Green interrupt, **91**, 97, 101, 175
sync_global_abstain_reg, constant, **90**, 188

T

timer interrupt, Orange interrupt, 96, 100, **114**, 168

U

UADD_SCAN
 combine combiner constant, **75**, **152**
 combiner constant, 77, 186
USER_BC_FIFO, interface number constant, **22**, **150**

V

VU error, Green interrupt, 176

X

XOR_SCAN
 combine combiner constant, **75**, **152**
 combiner constant, 77, 186

Copyright © 1994 Thinking Machines Corporation

Concepts Index

This index lists the essential concepts referred to within this document. Bold page numbers indicate a defining reference or important description.

A

absolute address, in chunk table translations, **110**
abstain flag, **27**
  effect of, **28**
  function to set values of, **123**
  in control registers, **9**
  of broadcast interface, **69**
  of combine interface, 28, **81**
    for reduction operations, **28**
  of global interface, **90**
  using efficiently, **138**
  using safely, **29**
abstaining
  from a network interface, **27**
  from a synchronous global message, **90**
  from broadcast interface, **69**
  from combine interface, **81**
addition (signed), combine operation, **75**
addition (unsigned), combine operation, **75**
addition scan overflow, **78**
address (node) registers, **108**
address translation, and NI chunk table, **108**
addresses
  calculating `send_first`, **21**
  calculating `send_first_long`, **43**
  of registers, **145**
    programming constants, **11**
addressing
  of nodes, **39**, *119*, **139**
  of partition manager, **119**
  of registers, programming constants, **11**
  physical. *See* physical addressing
  relative. *See* relative addressing
alignment of doubleword data, **140**
"All Fall Down interrupt enable" flag, 54, **55**
"All Fall Down message" flag, 54, **55**
All Fall Down Mode, **55**
  address word format, **56**
  detecting, **55**
  resending, **56**
  triggering, **55**
"All Fall Down mode enable" flag, 54, **55**
alternate status register, of Data Networks, 36, **46**
asynch global receive interrupt, **93**
"asynch global receive interrupt enable" flag, of asynchronous global interface, 92, **93**, 107
asynch supervisor global receive interrupt, **93**
"asynch supervisor global receive" flag, of supervisor asynch global interface, **93**

Copyright © 1994 Thinking Machines Corporation

"asynch supervisor global send" flag, of supervisor asynch global interface, **93**
"asynch supervisor global" register, of supervisor asynch global interface, 89, **93**
"asynch supervisor receive interrupt enable", of supervisor asynch global interface, **93**, 107
"asynch global receive" flag, of asynchronous global interface, **92**
"asynch global send" flag, of asynchronous global interface, **92**
"asynch global" register, of asynchronous global interface, 89, **92**
asynchronous interface, of global interface, 88, **89**
auxiliary information, **20**
  for broadcast messages, **67**
  for combine messages, **74**
  for Data Network messages, **41**, **42**
  of a network message, **18**

B

backward scan, combine pattern, **75**
"bad address low" field, **114**
"bad address type" field, **114**
"bad address" register, **114**
base address, of NI memory region, **8**
  programming constant, **12**, **21**
broadcast enabling, **69**
  CMOST operation for, **141**
broadcast interface, 3, 63, **64**
  abstaining from, **69**
  auxiliary information, **67**
  broadcast interrupt interface, **104**
  conflicts with combine interface, **140**
  enabling, **69**
    CMOST operation for, **141**
  message format, **66**
  message ordering, **66**
  messages, **65**
  receiving, **67**
  registers, **64**
  sending, **66**
  supervisor broadcast interface, **64**
  user broadcast interface, **64**
"broadcast interrupt receive enable" flag, **104**, 107
"broadcast interrupt send ok" flag, **104**, 107
"broadcast interrupt send" register, **104**
broadcast interrupts. *See* interrupts, broadcast
broadcast messages, user and supervisor, **64**
"Bus Error disable" flag, **107**
Bus Errors, **97**, **178**
  and bad address register, **114**
  on abstain flag change during global message, **90**
  on bad memory access, **97**, **178**
  on broadcast interrupt error, **104**
  on broadcasting with sending disabled, **69**
  on combine flush error, **113**
  on configuration error, **115**
  on excessively long messages, **19**
  on improper message format, **19**
  on network-done message error, **80**
  on reading from empty rec FIFO, **26**
  on reading/writing undefined addresses, **7**
  on sending with abstain flag set, **28**, **90**
  on user access to supervisor features, **7**
  on user sending message with supervisor tag, **48**
  on user sending physical mode message, **43**

C

casting register constants, for C coding, **11**
chunk address, **109**
chunk position, **110**
"chunk size" register, **110**
chunk sizes, **110**
chunk table, 40, **108**
  modifying, **112**
  size of chunks, **110**
"chunk table address" register, **112**

Copyright © 1994 Thinking Machines Corporation

"chunk table data" register, **112**
clearing combine send FIFO, **83**
`cm_signal.h`, header file, **51**
CM-5, **2**
  networks, **2**
  operating system, **5**
  partition manager, **4**
  partitions, **4**
  processing nodes, **3**
  programs, **5**
CMMD, software interface, 1
CMNA
  (CM Network Accessors), **225**
  header files, **226**
`cmna.h`, header file, **11**, 13, **125**, **225**
code
  for nodes, **5**
  for PM, **5**
"combine add-scan overflow" flag, 76, **78**
combine flush, **112**
"combine flush complete" flag, 107, **113**
"combine flush" register, **113**
combine interface, 3, 63, **71**
  abstaining from, **81**
  auxiliary information, **74**
  conflicts with broadcast interface, **140**
  flushing, **112**
  message format, **73**
  message ordering, **73**
  messages, **73**
  network-done messages, **79**
  parallel prefix. *See* scanning
  pipelining, **73**
  receiving, **76**
  reduction messages, **77**
  registers, **72**
  scan overflow, **78**
  scanning, **77**
  sending, **73**
  status register, **76**
  word order in scans, **77**, 140
combine messages, word order in, **140**
combine patterns
  addition (signed), **75**
  addition (unsigned), **75**
  backward scan, **75**
  exclusive OR, **75**
  forward scan, **75**
  inclusive OR, **75**
  maximum, **75**
  network-done, **75**
  reduction, **75**
combiner field, combine interface, legal values, **75**
"combiner value" supervisor field, of combine interface, 83, **84**
combiner values, for combine messages, **77**
communications networks. *See* CM-5 networks; networks
compiling NI programs. *See* programs
configuration, partition, **115**
"configuration complete" register, 107, **115**
"configuration" register, **115**
conflicts, between broadcast and combine interfaces, **140**
Connection Machine CM-5 Technical Summary, xix
constants
  NI base address, **12**, **21**
  programming, **11**
  register, address, **11**
  register field, position and length, **12**
Control Network, **2**, **3**, **63**
  See also broadcast interface; combine interface; global interface
  disabling, **116**
"Control Network disable" flag, 107, **116**
control register, register type, **9**
"control" register
  of a network interface, 17, **27**
  of broadcast interface, 64, **69**
  of combine interface, 72, **81**
"count mask" register, 38, **51**, 80
"current" message, in receive FIFO, **25**

D

Data Network (DR), **2**, 2, **35**
  addressing. *See* addressing

Copyright © 1994 Thinking Machines Corporation

All Fall Down mode, **55**
address word format, **56**
detecting, **55**
resending, **56**
triggering, **55**
auxiliary information, **41, 42**
chunk table, **108**
interactions between interfaces, **36**
length field, **42**
message format, **41**
message mode bit, **42**
message modes, physical and relative, **40**
message ordering, **39**
message tags, **48**
messages, **38**
auxiliary information, **42**
length field, **42**
mode bit, **42**
tag field, **42**
receiving, **44**
registers, **36**
send FIFO, registers, **42**
sending, **42, 43**
tag value of messages, **42**
Data Network interfaces
Data Network (DR), **36**
left interface (LDR), 2, **36**
registers, **36**
See also Data Network
right interface (RDR), **36**
detecting arrival of messages, **24**
Diagnostic Network, **3**
disabling the Control Network, **116**
discarded messages, **20**
and `send_ok` flag, **25**
using efficiently, **138**
doubleword data, alignment, **140**
doubleword operations, for reading/writing registers, **19**
doubleword operators, **22**, 136
"DR length limit" register, 38, **39**
"DR network done" flag, 45, 46, **54, 76**
"DR receive state" field, 45, **53**
"DR send state" field, **45**, 45, **46, 53**

E

examples, on-line, **134**
exclusive OR, combine operation, **75**
executing NI programs. *See* programs

F

fields, register
See also register fields
position and length constants, **12**
fields and flags, status. *See* status register, fields and flags
"flush complete" flag, 107, **113**
"flush" register, of combine interface, **113**
flushing, the combine interface, **112**
format of messages, 18, **19**
for asynchronous global interface, **92**
for broadcast interface, **66**
for combine interface, **73**
for Data Network, **41**
for supervisor asynch global interface, **93**
for synchronous global interface, **90**
forward scan, combine pattern, **75**

G

generic network interface, **17**
using effectively, **32**
getting value of status register, **26**
See also status registers
"global abstain" register, of synchronous global interface, 89, **90**
global interface, 3, 63, **88**
asynchronous interface, **91**
supervisor asynch interface, **93**
"global receive" register, of synchronous global interface, **89**, 89
"global send" register, of synchronous global interface, 89, **90**
Green broadcast interrupt, **104**
Green interrupt, **97, 101, 173**
Green broadcast interrupt, 97, 101, **104, 173**
on add scan overflow, **79**, 97, 101, **176**

Copyright © 1994 Thinking Machines Corporation

on All Fall Down message receipt, **55**, 97, 101, **174**
on DP (vector unit) error, **176**
on empty combine receive FIFO, **83**, 97, 101, **175**
on empty Data Network send FIFO, **57**, 97, 101, **177**
on interrupting DR message tag, **49**, 97, 101, **174**
on LDR/RDR tag, **177**
on LDR/RDR user tag, **177**
on message receipt, **30**, **91**, **93**, 97, 101, **174**, **175**
on vector unit error, 97, 101, **176**
"Green interrupt clear" register, **102**
"Green interrupt level" field, **103**

H

header files
`cm_signal.h`, **51**
`cmna.h`, **11**, **225**
"hodgepodge" register, **38**, **49**, **107**
and asynchronous global interface, **89**
and supervisor asynch global interface, **89**
and synchronous global interface, **89**
broadcast interrupt flags, **104**
configuration flag, **115**
flush complete flag, **113**
global receive interrupt enable flag, 92, **93**
NI timer interrupt enable flag, **113**
send stop flag, **116**
supervisor receive interrupt enable flag, **93**
sync global receive interrupt enable flag, 89, **91**

I

inclusive OR, combine operation, **75**
interface, register
of asynchronous global interface, **91**
of broadcast interface, **64**
of combine interface, **72**
of Data Networks, **36**
of global interface, **89**
of supervisor asynch global interface, **93**
of synchronous global interface, **89**
interface code file. *See* programs
"interrupt cause" register, **102**
"interrupt clear" register, **102**
"interrupt level" register, **103**
"interrupt now" register, **113**
interrupts, 13, **95**, **165**
and tag fields, **48**
broadcast, **104**
Bus Errors, **178**
on bad memory access, **178**
Bus Errors, **97**
and bad address register, **114**
on abstain flag change during global message, **90**
on bad memory access, **97**
on broadcast interrupt error, **104**
on broadcasting with sending disabled, **69**
on combine flush error, **113**
on configuration error, **115**
on excessively long messages, **19**
on improper message format, **19**
on network-done message error, **80**
on reading from empty receive FIFO, **26**
on reading/writing undefined addresses, **7**
on sending with abstain flag set, **28**, **90**
on user access of supervisor features, **7**
on user sending message with supervisor tag, **48**
on user sending physical mode message, **43**
cause and clear registers, **102**
classes, 13, **95**
detecting and clearing, **102**
Green, **97**, **101**, **173**
on add scan overflow, **79**

Copyright © 1994 Thinking Machines Corporation

on All Fall Down message receipt, **55**
on broadcast interrupt, **104**
on empty receive FIFO, **83**
on interrupting DR message tag, **49**
on message receipt, **30**, **91**, **93**
interrupt levels, **103**
Orange, **96**, **100**, **168**
on broadcast interrupt, **104**
on NI timer interrupt, **114**
pathways, **98**
recovery, **98**, **105**
Red, **95**, **99**, **166**
off-chip faults, **99**
on broadcast interrupt, **104**
on-chip faults, **99**
using to retrieve Data Network messages, **48**
Yellow, **96**, **100**, **169**
on bad relative address, **41**
on broadcast interrupt, **104**
on broadcast/combine collision, **73**
on broadcast/combine conflict, **104**
on Bus Error signaled as interrupt, **98**
on combine/abstain flag error, **82**
on negative message count, **52**
IOR, combine operation, **75**

L

"LDR supervisor tag interrupt enable" flag, **49**, 107
"LDR user tag interrupt enable" flag, **49**, 107
left Data Network interface (LDR), 2, **35**
length limit
of network interface FIFOs, **19**
on broadcast interface messages, **66**
length of message
remaining words, **26**
total (as received), **26**
"lock" flag
of a network interface, 30, **31**
of broadcast interface, **69**
of combine interface, **83**
of Data Network interfaces, **54**

M

mapping, relative to physical addresses, **111**
maximum, combine operation, **75**
memory maps
network interface registers, **18**
node virtual memory, **9**
of broadcast interface registers, **65**
of combine interface registers, **72**
of Data Network registers, **37**
of global interface registers, **88**
memory subsystem, of nodes, **3**
"message count" register, 38, **51**, 54, 80
message counting, **51**
in network-done operations, **80**
message format
asynchronous global interface, **92**
broadcast interface, **66**
combine interface, **73**
Data Network, **41**
supervisor asynch global interface, **93**
synchronous global interface, **90**
message ordering, broadcast interface, **66**
message tags, **48**
user/supervisor, **48**
"Message too long interrupt enable" flag, 38, 107
messages
broadcast interface, **65**
combine interface, **73**
word order, **140**
Data Network, **38**
detecting arrival of, **24**
discarded, **20**
and send_ok flag, **25**
format, **18**
for asynchronous global interface, **92**
for broadcast interface, **66**
for combine interface, **73**
for Data Network, **41**
for supervisor asynch global interface, **93**
for synchronous global interface, **90**
from nodes to PM, **121**
from PM to nodes, **120**

Copyright © 1994 Thinking Machines Corporation

global interface, **88**
length field, for Data Network, **42**
mode bit, for Data Network, **42**
modes, (for Data Network), **40**
network, **18**
receipt order, for Data Network, **39**
receiving, **23**
microprocessor, of processing node, **3**
"middle" Data Network interface, **2**
"middle" Data Network interface restrictions, **140**

N

"network done" flag
See also "DR network done" flag
of Data Network, (network-done operation), **80**
Network Interface (NI), **2**, **6**
base address, **8**
constant, **12**, **21**
chip, 2, **6**
interrupts, 13, **95**, **165**
memory region, occupied by registers, **7**
memory regions, physical and virtual, **8**
operation times, **135**
performance hints, **135**
register names, **10**
register types, **9**
registers, **6**
Reset, 14, **117**
serial number, **116**
supervisor area, **7**
timer, **113**
user area, **7**
network interfaces, interactions between, **140**
network-done interrupt enable flag, 54
network-done
combine interface operation, 71, **79**
combine operation, **75**
message format, **79**
network-done messages, (via combine interface), **79**
networks, **2**
common features, **17**
conflicts between. *See* broadcast network, conflicts; combine network, conflicts
interface, registers, **17**
interface numbering, **21**
interfaces, generic, **17**
messages, **18**
NI. *See* Network Interface (NI)
NI programs. *See* programs
NI Reset, **117**
"NI timer enable" flag, 107, **113**
node, program, **5**
node program. *See* programs
nodes. *See* processing nodes

O

off-chip faults, (Red interrupts), **99**
on-line code examples, **134**
on-chip faults, (Red interrupts), **99**
operating system. *See* CM-5 operating system
operation times, of NI, **135**
OR, combine operation, **75**
See also XOR, combine operation
Orange broadcast interrupt, **104**
Orange interrupt, **96**, **100**, **168**
network-done complete, **57**, **169**
NI timer interrupt, 96, 100, **114**, **168**
Orange broadcast interrupt, 96, 100, **104**, **169**
router-done complete. *See* Orange interrupt, network-done complete
"Orange interrupt level" field, **103**
order of words, in scan messages, **77**
overflow, in addition scans, **78**

P

parallel prefix, combine interface operation. *See* scanning

Copyright © 1994 Thinking Machines Corporation

partition
  See also partitions
  size of, variable, **40**
"partition base address" register, 108, **110**
partition configuration, **115**
"partition configuration" register, **115**
partition manager (PM), **4**
  address of, **40**, 119
  code, **5**
  exchanging data with nodes, **119**
  program. *See* programs
"partition size" register, **109**
partitioning, by system administrator, **4**
partitions, **4**
  configuration, **115**
  defined by the NI chunk table, **108**
  relative addressing within, (for Data Network), **40**
  size, **4**
pattern field, combine interface, legal values, **75**
pattern values, for combine messages, **77**
performance hints, **135**
physical addressing
  See also addressing
  translation from relative addressing, **109**
physical base address, of NI memory region, **8**
"physical self address" register, **108**
pipelining combine operations, **73**
PM program. *See* programs
"private" register, **30**
  of a network interface, 17, 24, **30**
  of broadcast interface, 64, **69**
  of combine interface, 72, **83**
  of Data Network interface, 36, **54**
processing node program. *See* programs
processing nodes, **2**, **3**
  address registers, **108**
  address translation, **108**
  addresses of, **39**
    registers, **108**
  addressing. *See* addressing
  exchanging data with PM, **119**
  internal structure, **3**
programming models, user and OS, **5**
programs
  compiling and executing, **132**
  interface code file, **127**
  NI, **5**, 12
  node code file, **126**
  PM and node, 4
  PM code file, **125**
  structure of, **125**
protocol
  See also messages, format
  for sending messages, **19**

Q

FIFO register
  of a network interface. *See* receive FIFO register; send FIFO registers
  register type, **9**

R

"RDR supervisor tag interrupt enable" register, **49**, 107
"RDR user tag interrupt enable" flag, **49**, 107
reading a message, **23**
reading registers, using doubleword operators, **136**
reading status registers, **26**
"receive abstain" flag
  for broadcast interface, **69**
  of a network, **27**, 28
  of combine interface, **81**
  of global interface, **90**
"receive FIFO empty interrupt enable" flag, of combine interface, **83**
"receive FIFO full" flag
  of a network, 30, **32**
  of broadcast interface, **69**
  of combine interface, **83**
  of Data Networks, **54**
"receive ok interrupt enable" flag
  of a network, **30**, 30
  of broadcast interface, **69**

Copyright © 1994 Thinking Machines Corporation

of combine interface, **83**
of Data Networks, **54**
"receive interrupt mask" register, 38, **49**
"receive length left" field
of a network, 25, **26**
of broadcast interface, **68**, 68
of combine interface, **76**
of Data Networks, **45**
"receive length" field
of a network, 25, **26**
of combine interface, **76**
of Data Networks, **45**, **46**
"receive ok" flag
of a network, **24**, 25, **26**
of broadcast interface, **68**
of combine interface, **76**
of Data Networks, **45**, **46**
receive FIFO
network register for, **23**
of a network, **9**, 18, 23
receive FIFO register, of a network, **23**
"receive state" field, of Data Network, 45, **53**
"receive stop" flag, of a network, **31**
"receive" register
of a network, 17, **23**
of broadcast interface, 64, **67**
of combine interface, 72, **76**
of Data Networks, 36, **44**
receiving
a broadcast interface message, **67**
a combine interface message, **76**
a Data Network message, **44**
a global interface message, **92**
a network message, 18, **23**
a network-done message, **80**
a reduction-scan message, **77**
a scan message, **77**
a synchronous global message, **90**
an asynch supervisor global message, **93**
an asynchronous global message, **92**
Red broadcast interrupt, **104**
Red interrupt, **95**, **99**, **166**
off-chip faults, **99**
on cache/MMU error, 96, **99**, **167**
on Control Network checksum failure, 96, **99**, **166**
on Control Network hardware failure, 96, **99**, **167**
on Data Network checksum failure, 96, **99**, **166**
on memory controller error, 96, **99**, **167**
on NI chip fault, 96, **99**, **166**
on-chip faults, **99**
Red broadcast interrupt, 96, 100, **104**, **168**
"Red interrupt level" field, **103**
reduction
combine interface operation, 71, **77**
See also scanning
combine pattern, **75**
"reduction abstain" flag, of combine interface, **28**, **81**
reduction messages, (via combine interface), **77**
register constants, **11**
casting, for C coding, **11**
register fields
names, **10**
programming constants, **11**
register interface
of asynchronous global interface, **91**
of broadcast interface, **64**
of combine interface, **72**
of Data Networks, **36**
of global interface, **89**
of supervisor asynch global interface, **93**
of synchronous global interface, **89**
register naming format, `ni_`*interface_purpose*, **10**
register types, **9**
register
address constants, **11**
doubleword operators, **136**
names, **10**
NI, **6**
status, **25**
relative addressing
See also addressing
translation to physical addressing, **109**

Copyright © 1994 Thinking Machines Corporation

Reset, NI, 14, **117**
right Data Network interface (RDR), 2, **35**
RISC microprocessor, of processing node, **3**
router, **35**
  See also Data Network
router done complete
  Orange broadcast interrupt, 96
  Orange interrupt, 100
"router done" flag. *See* "DR network done" flag
router-done. *See* network done
running NI programs. *See* programs

S

scan overflow, in addition scans, **78**
"scan overflow interrupt enable" flag, of combine interface, **79**, 83
"scan start" register, of combine interface, 72, **78**
scanning
  addition scan overflow, **78**
  combine interface operation, 71, **77**
scanning with segments, **78**
segmented scanning, **78**
select address, for chunk table addressing, **109**
"self address", of a processing node, **40**
"send combiner value" supervisor field, of combine interface, 83, **84**
"send empty" flag
  of a network, 25, **26**
  of broadcast interface, **68**
  of combine interface, **76**
send FIFO empty interrupt enable flag, 54
"send FIFO enable" flag, of broadcast interface, **69**, 69
"send length" supervisor field, of combine interface, 83, **84**
"send ok" flag
  and discarded messages, **25**
  of a network, **25**, 25
  of broadcast interface, **68**
  of combine interface, **76**
  of Data Networks, **45**, **46**
"send pattern" supervisor field, of combine interface, 83, **84**
send FIFO
  network registers for, **19**
  of a network, **9**, 18, 19
"send space" field
  of a network, 25, **26**
  of broadcast interface, **68**
  of combine interface, **76**
  of Data Networks, **45**, **46**
"send start" supervisor field, of combine interface, 83, **84**
"send state" field, of Data Network, 45, **53**
"send stop" flag, of broadcast interface, 30, **32**
"send" register
  of a network, 17, **19**
  of broadcast interface, 64, **67**, 67
  of combine interface, 72, **73**
    using to clear the send FIFO, **83**
  of Data Networks, 36, **42**
`send_first` addresses
  calculating, **21**
  constants, **21**
`send_first_long` addresses, calculating, **43**
"send-first long" register, of Data Networks, 36, **43**
"send-first" register
  of a network, 17, **19**
  of broadcast interface, 64, **67**, 67
  of combine interface, 72, **73**
  of Data Networks, 36, **42**
sending
  a broadcast interface message, **66**
  a combine interface message, **73**
  a Data Network message, **42**, **43**
    message modes, **40**
  a global interface message, **92**
  a network message, 18, **19**
  a network-done message, **79**
  a reduction-scan message, **77**
  a scan message, **77**

Copyright © 1994 Thinking Machines Corporation

a synchronous global message, **90**
an asynch supervisor global message, **93**
an asynchronous global message, **92**
sending messages from nodes to PM, **121**
sending messages from PM to nodes, **120**
serial number (of NI), register, **116**
simulating arrival of a message, **24**, 142
status pop register, of Data Networks, 36, **47**
status register for long messages, of Data Networks, 36, **45**
status register, alternate, of Data Networks, 36, **46**
status register
fields and flags, **25**
of a network interface, 17, **25**
of broadcast interface, 64, **68**
of combine interface, 72, **76**
of Data Networks, 36, **45**, 80
register type, **9**
status registers
accessor macro, **26**
reading, **26**
"stop send" flag, 107, **116**
"stop" flag
of a network, **30**
of broadcast interface, **69**
of combine interface, **83**
of Data Networks, **54**
supervisor area, of NI memory region, **7**
supervisor asynchronous global interface, of global interface, 88, **89**
"supervisor asynchronous global" register, of supervisor asynch global interface, 89, **93**
supervisor broadcast interface, **64**
See also broadcast interface
supervisor message tags, **48**
supervisor operations, **7**
clearing combine send FIFO, **83**
clearing interface send FIFO, **31**
grabbing control of receive and status registers, **31**
reserving Data Network message tags, **48**
simulating arrival of a message, **24**, 142
triggering All Fall Down mode in DR, **55**
"synch global receive interrupt enable" flag, of synchronous global interface, 89, **91**, 107
"synchronous global completion" flag, of synchronous global interface, 89, **90**
synchronous global receive interrupt, **91**
"synchronous global receive" flag, of synchronous global interface, 89, **90**
synchronous interface, of global interface, 88, **89**

T

tag fields
and interrupts, **48**
and message counting, **51**
of Data Network messages, **48**
tag value, of Data Network message, **42**
timer, NI. *See* Network Interface timer
timer (NI), register, **113**
"timer enable" flag, **113**
timing, of NI operations, **135**
total length of message, **26**

U

user area, of NI memory region, **7**
user broadcast interface, **64**
See also broadcast network
user message tags, **48**
user programming model, **5**
"user receive interrupt mask" register, 38, 49
"user tag mask" register, 38, **48**

V

value, of a message, (single- or doubleword), **19**
virtual base address, of NI memory regions, **8**
VU Programmer's Handbook, xix

Copyright © 1994 Thinking Machines Corporation

W

writing a message, **22**
writing a value to recieve register, to simulate arrival of message, **24**
writing registers, using doubleword operators, **136**

X

XOR, combine operation, **75**

Y

Yellow broadcast interrupt, **104**
Yellow interrupt, **96, 100, 169**
  Bad memory (Bus Error) interrupt, **170**
  bad memory (Bus Error) interrupt, 96, 100
  Data Network message too long, **39**, 96, 100, 172
  on bad relative address, **41**, 100
  on broadcast/combine conflict, **73**, 96, 100, **104**, 171
  on combine abstain flag error, **82**, 96, 100, 170
  on illegal relative address, 96, 172
  on negative DR message count, **52**, 96, 100, **171**
  Yellow broadcast interrupt, 96, 100, **104, 170**
"Yellow interrupt level" field, **103**

Copyright © 1994 Thinking Machines Corporation

NI Memory Map

Copyright © 1994 Thinking Machines Corporation

Node Virtual Memory Map
(with or without VUs installed)

Region	hex address
RESERVED	
	0xF880 0000
OS Kernel	
	0xF800 0000
local stack ↓ / global stack ↑	
	0xE000 0000
VU Control Regs	
	0xC000 0000
VU Heap and Stack Regions	
	0x4000 0000
global heap ↓	
	0x2010 0000
NI space: supervisor area	
	0x2008 0000
NI space: user area	
	0x2000 0000
local heap ↑	
user variables	
user program	
	0x0000 2000
unmapped	
	0x0000 0000

NI Virtual Memory Area
(user or supervisor)

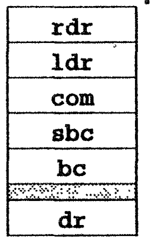
Register	hex offset
ni_rdr_send_first_long (physical)	
	0x19000 [2.2]
ni_rdr_send_first_long (relative)	
	0x18000 [2.2]
(shaded)	
ni_ldr_send_first_long (physical)	
	0x11000 [2.2]
ni_ldr_send_first_long (relative)	
	0x10000 [2.2]
ni_dr_send_first_long (physical)	
	0x9000 [2.2]
ni_dr_send_first_long (relative)	
	0x8000 [2.2]
ni_rdr_send_first	0x7000
ni_ldr_send_first	0x6000
ni_com_send_first	0x5000
ni_sbc_send_first	0x4000
ni_bc_send_first	0x3000
RESERVED	0x2000
ni_dr_send_first	0x1000
INTERFACE REGISTERS	0x0200
GLOBAL/SYSTEM REGISTERS	0x0000

Interface Registers

Sample Register Set:

	Register	hex offset
D *	ni_x_status_long	0x60 [2.2]
D *	ni_x_status_all	0x50 [2.2]
D *	ni_x_status_pop	0x40 [2.2]
	ni_x_send	0x30
	ni_x_recv	0x20
† *	ni_x_control	0x10
*	ni_x_private	0x08
*	ni_x_status	0x00

† Control Network only
D Data Network only



Interface	hex offset
rdr	0xE00
ldr	0xC00
com	0xA00
sbc	0x800
bc	0x600
(shaded)	
dr	0x200

Global & System Registers

	Register	hex offset
	RESERVED	0x1A0
*	ni_interrupt_set_green	0x198 [2.2]
*	ni_interrupt_set	0x190 [2.2]
	ni_user_rec_interrupt_mask	0x168 [2.2]
	ni_longest_dr_message	0x160 [2.2]
*	ni_bad_address	0x0E8
	ni_scan_start	0x0E0
	ni_interrupt_now	0x0D8
*	ni_interrupt_clear_green	0x0D0
*	ni_interrupt_clear	0x0C8
	ni_sync_global_send	0x0C0
*	ni_hodgepodge	0x0B8
*	ni_async_sup_global	0x0B0
*	ni_async_global	0x0A8
	ni_com_flush_send	0x0A0
	ni_sync_global_abstain	0x098
*	ni_sync_global	0x090
	ni_serial_number	0x088
	ni_interrupt_send	0x080
	ni_configuration	0x078
	ni_time	0x070
	(shaded)	0x068
	ni_user_tag_mask	0x060
	ni_rec_interrupt_mask	0x058
	ni_count_mask	0x050
	ni_dr_message_count	0x048
	ni_chunk_size	0x040
	ni_chunk_table_data	0x038
	ni_chunk_table_address	0x030
	ni_partition_size	0x028
	ni_partition_base	0x020
	ni_physical_self	0x018
*	ni_interrupt_level	0x010
*	ni_interrupt_cause_green	0x008
*	ni_interrupt_cause	0x000

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

2.2 Indicates new feature in 2.2

ni_interface_send_first_long Addressing Patterns

NI base address (bits 31–20); user/supervisor bit (bit 19); interface index (1) (bits 16–15); addressing mode (bit 12)

	31–20	19	18	17	16	15	14	13	12	11–7	6–3	2	1	0
{L,R}DR	------	X	0	0	n	n	0	0	X	length	tag	0	0	0

(1) DR = 01, LDR = 10, RDR = 11

ni_interface_send_first Addressing Patterns

NI base address (bits 31–20); user/supervisor bit (bit 19); interface index (2) (bits 14–12); addressing mode (bit 11)

	31–20	19	18	17	16	15	14	13	12	11	10	9	8	7	6–3	2	1	0
{L,R}DR	------	X	0	0	0	0	n	n	n	X	tag				length	0	0	0
{S}BC	------	x	0	0	0	0	n	n	n	0	0	0	0	0	length	0	0	0
COM	------	X	0	0	0	0	1	0	1	pattern	combiner				length	0	0	0

(2) DR = 001, LDR = 110, RDR = 111, BC = 011, SBC = 100

Thinking Machines Corporation
Confidential and Proprietary
©1994

Version: 2.2

Register: ni_interface_status{_long}2.2

31							network done / send empty	send ok	rec ok	0
UNUSED	rec state	send state	ovf	rec tag	rec length	rec len left				send space

Field Name:	status Pos	status Size	status_long 2.2 Pos	status_long Size	DR	L/RDR	S/BC	COM
ni_send_space	0	4	0	5	√	√	√	√
ni_rec_ok	4	1	5	1	√	√	√	√
ni_send_ok	5	1	6	1	√	√	√	√
ni_router_done_complete	6	1	7	1	√			
ni_send_empty	6	1	7	1			√	√
ni_rec_length_left	7	4 *2.2	8	5	√	√	√ *	√
ni_rec_length	11	4	13	5	√	√		√
ni_dr_rec_tag	15	4	18	4	√	√		
ni_com_scan_overflow	20	1	22	1				√
ni_dr_send_state	21	2	23	2	√			
ni_dr_rec_state	23	2	25	2	√			

* 7 bits for BC interface 2.2

Register: ni_dr/ldr/rdr_status_all 2.2

31										send Ok		0
router done		rec AFD			rec len long		rec tag				rec ok	
d		x	y	send space	x	y	x	r	d	x	y	

Field Name:	Pos:	Size:
ni_ldr/ldr/rdr_rec_ok	0	1
ni_rdr/rdr/ldr_rec_ok	1	1
ni_dr/ldr/rdr_send_ok	2	1
ni_ldr/ldr/rdr_rec_tag	3	4
ni_rdr/rdr/ldr_rec_tag	7	4
ni_ldr/ldr/rdr_rec_length_long	11	5
ni_rdr/rdr/ldr_rec_length_long	16	5
ni_dr/ldr/rdr_send_space	21	5
ni_ldr/ldr/rdr_rec_all_fall_down	26	1
ni_rdr/rdr/ldr_rec_all_fall_down	27	1
ni_router_done_complete	31	1

Thinking Machines Corporation
Confidential and Proprietary
©1994

Version: 2.2

Register: ni_interface_private

Field Name:	Pos:	Size:	DR	L/RDR	S/BC	COM
ni_rec_ok_ie	0	1	√	√	√	√
ni_lock	1	1	√	√	√	√
ni_rec_stop	2	1	√			√
ni_send_stop	2	1			√	
ni_rec_full	3	1	√	√	√	√
ni_send_enable	4	1			√	
ni_com_scan_overflow_ie	4	1				√
ni_dr_rec_all_fall_down	5	1	√	√		
ni_com_rec_empty_ie	5	1				√
ni_all_fall_down_ie	6	1	√			
ni_all_fall_down_enable	7	1	√			
ni_com_send_length	8	4				√
ni_com_send_combiner	12	3				√
ni_com_send_pattern	15	2				√
ni_com_send_start	17	1				√
ni_sfifo_goes_empty_ie	18 2.2	1	√			
ni_rdone_complete_ie	19 2.2	1	√			

Register: ni_interface_control

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

Registers: ni_interface_cause/clear/set, ni_user_rec_interrupt_mask

(same bit positions, all flags)

Field Name:	Pos:
ni_c/c/s_internal_fault	0
ni_c/c/s_mc_error	1
ni_c/c/s_cmu_error	2
ni_c/c/s_bc_interrupt_red	3
ni_c/c/s_cn_checksum_error	4
ni_c/c/s_cn_hard_error	5
ni_c/c/s_dr_checksum_error	6
ni_c/c/s_timer_interrupt	7
ni_c/c/s_bc_interrupt_orange	8
ni_c/c/s_bc_interrupt_yellow	9
ni_c/c/s_bc_or_com_collision	10
ni_c/c/s_com_abstain_changed	11
ni_c/c/s_dr_count_negative	12
ni_c/c/s_bad_relative_address	13
ni_c/c/s_bad_memory_access	14
ni_c/c/s_message_too_long	15 2.2
ni_c/c/s_rdone_complete	16 2.2

Registers: ni_interrupt_cause/clear/set_green

(same bit positions, all flags)

Field Name:	Pos:
ni_c/c/s_bc_interrupt_green	0
ni_c/c/s_scan_overflow	1
ni_c/c/s_bc_rec_ok	2
ni_c/c/s_sbc_rec_ok	3
ni_c/c/s_com_rec_ok	4
ni_c/c/s_com_rec_empty	5
ni_c/c/s_sync_global_rec	6
ni_c/c/s_global_rec	7
ni_c/c/s_supervisor_global_rec	8
ni_c/c/s_dr_rec_ok	9
ni_c/c/s_ldr_rec_ok	10
ni_c/c/s_rdr_rec_ok	11
ni_c/c/s_dr_rec_tag	12
ni_c/c/s_dr_rec_all_fall_down	13
ni_c/c/s_ldr_rec_tag	14 2.2
ni_c/c/s_rdr_rec_tag	15 2.2
ni_c/c/s_ldr_user_rec_tag	16 2.2
ni_c/c/s_rdr_user_rec_tag	17 2.2
ni_c/c/s_dperr	18 2.2
ni_c/c/s_sfifo_empty	19 2.2

Register: ni_hodgepodge

Field Name:	Pos:	Size:
ni_global_rec_ie	0	1
ni_supervisor_global_rec_ie	1	1
ni_flush_complete	2	1
ni_interrupt_send_ok	3	1
ni_configuration_complete	4	1
ni_interrupt_rec_enable	5	1
ni_sync_global_rec_ie	6	1
ni_timer_ie	7	1
ni_cn_stop_send	8	1
ni_disable_bus_error	9 2.2	1
ni_ldr_rec_tag_ie	10 2.2	1
ni_rdr_rec_tag_ie	11 2.2	1
ni_ldr_user_rec_tag_ie	12 2.2	1
ni_rdr_user_rec_tag_ie	13 2.2	1
ni_msg_too_long_ie	14 2.2	1

Register: ni_bad_address

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

Register: ni_sync_global

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

Register: ni_async_global

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

Register: ni_async_sup_global

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

Register: ni_interrupt_level

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