CMMD 3.3 Release Notes*

Part I Overview

1 Introduction

The main difference between CMMD Version 3.3 and earlier releases is that Version 3.3 supports the CM-5E. Although Version 3.2 will run on a CM-5E, only Version 3.3 takes advantage of the CM-5E's features. CMMD 3.3 running on a CM-5E yields significant performance improvements as well as enhanced features.

The CMMD library provides facilities for programming the CM-5 and CM-5E in a MIMD style. It supports the following operations:

- sending and receiving messages between nodes
- global operations: scan, reduce, broadcast, concatenate, synchronization
- timing functions
- node-level I/O (both independent and cooperative)
- active messages and rport operations
- I/O functions that support Scalable Disk Arrays (SDA)

2 Hardware and Software Requirements

2.1 Hardware Required

CMMD Version 3.3 runs on both the CM-5 and the CM-5E.

---

* These release notes are based on the Version 3.3 Beta 2 release notes. Substantive changes appear in the following sections: 4.1 (modified), 5 (new), 8 (replaced), 9 (replaced), and 10 (new).

Copyright © 1995 Thinking Machines Corporation

2.2 Software Required

CMMD Version 3.3 requires CMOST Version 7.4 Beta 1 or higher.

CMMD can be called from programs written in C, C++, C*, Fortran 77, or CM Fortran. Specifically,

- CM Fortran Version 2.1 Final and higher are supported. C* Version 7.1 and higher are supported.
- Sun F77 versions 2.0 and earlier are supported. To debug Version 2.0, `pndbx` Version 1.2-final-patch6 (or higher) or Prism Version 2.0 (or higher) is required.
- All Sun bundled C compilers are supported. Versions of `acc` (unbundled Sun C) prior to and including Version 2.0 are supported. To debug Version 2.0, `pndbx` Version 1.2-final-patch6 or higher is required.
- GNU C Version 2.3.3 is supported.
- The Sun CFront compiler Version 1.0 and the GNU G++ compiler Version 2.3.3 and earlier (both C++ compilers) are supported for compilation and linking. Neither `pndbx` nor Prism supports C++ debugging.

C and C* programs using CMMD must include the include file `<cm/cmmd.h>`. Other standard include files, such as `<stdio.h>`, `<fcntl.h>`, and `<sys/types.h>`, may also be needed, depending on the particular program.

Fortran programs must include `cm/cmmd_fort.h`, in addition to whatever files they would normally include.

3 Compiling and Linking

Applications should be recompiled and relinked when moving from earlier versions of CMMD to 3.3.

Version 3.3, April 1995
Copyright © 1995 Thinking Machines Corporation

4 New Features

4.1 New Functionality

Features added in CMMD 3.3 since CMMD 3.2:

- Long Active Messages (for the CM-5E only; see Part II, Section 8, for detailed information)
- New parallel I/O scratch space management functions (see Part II, Section 9, for detailed information)
- New sample programs

4.2 Performance Enhancements

- Roughly a factor-of-two performance improvement in message passing (for the CM-5E)

5 Limitations and Restrictions

Code written before CMMD Version 3.3 that mixes CMNA and CMMD/CMAML (i.e., code that allocates Data Network Hardware Tags and that calls `CMAML_get_[l|r]dr`) *will not work on a CM-5E without modification*. It will continue to work on a CM-5. Details on this, including instructions as to how to modify existing code, appear below (see Section 10).

NOTE: This warning does not apply to most users, since most users do not program at this level.

6 Documentation for This Release

Hardcopy documentation includes these release notes and the following user manuals:

- *CMMD User's Guide,* Version 3.3
- *CMMD Reference Manual,* Version 3.3

Copyright © 1995 Thinking Machines Corporation

- *CMMD for C Quick Reference Guide,* Version 3.3 and
  CMMD for Fortran Quick Reference Guide, Version 3.3

The following on-line documentation is available:

- CMview versions of *CMMD User's Guide* and *CMMD Reference Manual* for Version 3.3
- ASCII release notes and manuals, accessible through Prism.
- Release notes in ASCII form, typically installed in

  ```
  /usr/doc/cmmd-3.3*releasenotes
  ```

- PostScript versions of all documents (except quick reference guides), which are intended for printing. These PostScript documents are typically installed in

  ```
  /usr/doc/cmmd/refman-3.3*/
  /usr/doc/cmmd/usersguide-3.3*/
  ```

- UNIX man pages are in

  ```
  /usr/man
  ```

- Directories of sample programs are in

  ```
  /usr/examples/cmmd/
  ```

7 Errors, Feedback, and Assistance

7.1 Bug Update Files

Information about known and fixed bugs is provided on-line; see the file `/usr/doc/cmmd-3.3*bugupdate`.

Contact your system administrator if you do not find it there; the pathname may be different at your site.

Version 3.3, April 1995
Copyright © 1995 Thinking Machines Corporation

7.2 Request for Feedback

Thinking Machines Customer Support encourages customers to report errors in Connection Machine system and software 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 © 1995 Thinking Machines Corporation

Part II: Detailed Information about New and Changed Features

8 Long Active Messages

(**NOTE:** The following material also appears as Section 12.3 of the *CMMD Reference Manual* for Version 3.3.)

Long active messages are designed to take advantage of the 18-word packet length on the CM-5E. Three varieties exist:

- VA-class messages are essentially identical to short active messages, except that they allow from 0 to 17 arguments to be passed to the handler function. (VA is a mnemonic for Variable number of Arguments.)
- VAPOP-class messages allow from 0 to 17 data arguments, similar to VA-class messages, but the handler functions are not written to expect a sequence of immediate arguments. Rather, they expect (and are passed) two special opaque objects ("magic cookies"), which allow them to "pop" the data directly from the network into memory, thereby bypassing a redundant copy to and from the process stack. (VAPOP is a mnemonic for Variable number of Arguments to POP.)
- ARRPOP-class messages provide a pointer to an array in source memory. The array is transferred directly into the network for transmission, and is transferred directly from the network into memory by the handler at the receiving end. This allows the user code to bypass a redundant copy on the sending end as well as the receiving end. (ARRPOP is a mnemonic for ARRay to POP.)

To enable the latter two classes of functions to remove data from the network, three sets of accessor functions are provided:

- `CMAML_pop` and `CMAML_popd` remove all the data contained in the active message and store it at a specified address.
- The `CMAML_poptofirst` functions (`CMAML_poptofirst[d|_ra|_ra_d|_rla|_rla_d]`) remove all the data sent by the active message and store it at the address specified by the first data word. The first word must of course be a locally valid address; otherwise a fatal error may occur.

Copyright © 1995 Thinking Machines Corporation

- `CMAML_popn` and `CMAML_popnd` remove *n* words of data from the network and store it at a specified location.

  `CMAML_n_to_pop` is used to tell `CMAML_popn[d]` how many words of data are available to be removed from the network. (**NOTE:** If a handler pops less than the total number of words in a message, a fatal error will result. Every handler *must* completely drain the network of all the data that came with it before returning. This is a hard, fast rule that, for reasons of speed, is *not enforced* except in the form of fatal, difficult-to-debug errors.)

CMAML also supplies two informational functions that tell whether or not you can use long active messages on your current system, and, if you can, the maximum number of words the message can contain (currently 17 on the CM-5E):

- `CMAML_pn_long_am_nwords`
  `CMAML_cp_long_am_words`

8.1 VA-Class Active Messages

VA-class active message functions allow up to 17 immediate arguments for use by the handler, plus a preceding argument telling how many arguments are being passed. The name VA is a reminder that these functions accept a **V**ariable number of **A**rguments.

Handlers for this class of function are written to accept a fixed number of arguments. (Typically, they should accept all the arguments that the active message function sends.) Handlers expecting four or fewer arguments may be shared between short active messages and VA-class active messages.

In all other particulars, VA-class active message functions are identical to the short active message functions. They are the safest class of long active messages, and the easiest to use. As might be expected, they are also the least efficient for many applications.

8.2 VAPOP-Class Active Messages

VAPOP-class long active messages specify a **V**ariable number of immediate **A**rguments, all of which are **POP**ped by the handler function directly from the network into destination memory. The sender's syntax for these func-

Copyright © 1995 Thinking Machines Corporation

tions is identical to that of the VA-class active message functions. The semantics, however, are quite different.

The difference can be seen most clearly by looking at the handler functions. Handlers for VAPOP-class active functions must be written so that they accept exactly two arguments, with fixed meanings: *fifo* and *status*. Neither of these "magic cookies" are among the 0 to 17 arguments that constitute the active message itself. Instead, they are supplied by internal CMAML software. The user neither sets nor reads them, nor do they take up any space within the transmitted message. Their sole purpose is to be passed to other functions.

To deal with the message, the handler must call an accessor function (one of the `CMAML_pop_xxx` functions listed above), to which it passes *fifo* and *status*. Using the *fifo* and *status* information, the accessor function removes the data arguments sent by the VAPOP-class function from the network and stores the data in an appropriate location in memory.

Efficiencies and Dangers: For many applications, handlers for VAPOP-class active message functions are inherently more efficient than handlers for VA-class functions, since they can avoid making a redundant copy of data to the stack on the destination nodes. However, they are more dangerous to use than VA-class functions. Both the efficiency and the danger come from the fact that the handler function itself has full responsibility for removing all data transmitted by the function from the network, *and for ensuring that it removes precisely the amount of data present, no more and no less, before returning.*

8.3 ARRPOP-Class Active Messages

ARRPOP-class long active message functions allow a source node to write an ARRay directly into the network. The handler function on the destination node then POPs the array from the network directly into destination memory. Thus, ARRPOP-class functions provide direct memory-to-memory transmission.

The syntax of ARRPOP-class functions allows the sending node to specify a handler function, up to two immediate "auxiliary" data arguments, a pointer to an array of data, and a length argument specifying the length of the array. When the ARRPOP function is invoked, data from the array (plus the auxiliary arguments, if any) are written directly to the network. When the handler executes on the receiving node, it calls one or more accessor

Copyright © 1995 Thinking Machines Corporation

functions to remove the data from the network and write it directly into memory. Thus, any redundant copying is avoided.

If auxiliary arguments are supplied, the data they contain are prepended to that contained in the pointed-to array.

Handlers for ARRPOP-class functions are written in the same manner as those for VAPOP-class functions, and they use the same accessor functions to remove data from the network.

Efficiencies and Dangers: The ARRPOP-class functions avoid a redundant copy on *both* the sending and receiving nodes. This makes them the most efficient of the long active message functions for many applications, both in speed and in efficient memory use. The cost of this efficiency is the danger inherent in giving the user-written handler functions full responsibility for ensuring that they remove only the correct amount of data from the network. (**NOTE:** The danger lies, as with the VAPOP class, on the receiving end.)

Programmers writing ARRPOP-class functions should also recognize that doubleword- or singleword-aligned pointers are handled much more efficiently than those that are byte- or halfword-aligned. This is because the transport mechanisms that underlie active messages are word-based. Thus, the number of bytes of data available to a handler is always a multiple of four. Doubleword alignment is preferred whenever possible; it is the fastest.

To underscore the efficiency of word- and doubleword alignment, all arguments pertaining to length or number of arguments use *words* as their unit of measurement, even though pointers can be byte-aligned. This differs from the analogous arguments for most other functions, such as `CMMD_send_block`, which use bytes as their units.

PLEASE NOTE

When using VAPOP or ARRPOP functions, it is absolutely critical to ensure that the handlers always pop exactly the right number of words from the network. Popping the wrong number of words from the network can result in bus errors, partition crashes, or programs that are virtually undebuggable.

Copyright © 1995 Thinking Machines Corporation

8.4 Active Message Function Dictionary

CMAML provides six functions for sending short (5-word) active messages on the CM-5 and CM-5E:

```
CMAML_request           CMAML_request_tohost
CMAML_reply             CMAML_reply_tohost
CMAML_rpc               CMAML_rpc_tohost
```

CMAML provides similar groups of long active messages, for use on the CM-5E only:

```
CMAML_request_va        CMAML_request_va_tohost
CMAML_reply_va          CMAML_reply_va_tohost
CMAML_rpc_va            CMAML_rpc_va_tohost

CMAML_request_vapop     CMAML_request_vapop_tohost
CMAML_reply_vapop       CMAML_reply_vapop_tohost
CMAML_rpc_vapop         CMAML_rpc_vapop_tohost

CMAML_request_arrpop0   CMAML_request_arrpop0_tohost
CMAML_reply_arrpop0     CMAML_reply_arrpop0_tohost
CMAML_rpc_arrpop0       CMAML_rpc_arrpop0_tohost

CMAML_request_arrpop1   CMAML_request_arrpop1_tohost
CMAML_reply_arrpop1     CMAML_reply_arrpop1_tohost
CMAML_rpc_arrpop1       CMAML_rpc_arrpop1_tohost

CMAML_request_arrpop2   CMAML_request_arrpop2_tohost
CMAML_reply_arrpop2     CMAML_reply_arrpop2_tohost
CMAML_rpc_arrpop2       CMAML_rpc_arrpop2_tohost
```

Reference pages for the long active message functions follow.

Version 3.3, April 1995
Copyright © 1995 Thinking Machines Corporation

VA-Class Long Active Messages

VA-class long active messages send a VA-class long active message (a packet containing a handler followed by 0−17 data words). Can be used on the CM-5E only.

SYNOPSIS

C Syntax

```
void CMAML_request_va
void CMAML_reply_va
void CMAML_rpc_va
    (int dest_node, void (*handler)(),
     int n_data_args[, int data1, ..., int dataN])

void CMAML_request_va_tohost
void CMAML_reply_va_tohost
void CMAML_rpc_va_tohost
    (void (*handler)(),
     int n_data_args[, int data1, ..., int dataN])
```

Fortran Syntax

```
SUBROUTINE CMAML_request_va
SUBROUTINE CMAML_reply_va
SUBROUTINE CMAML_rpc_va
          (dest_node, handler, n_data_args[, data1, ..., dataN])
    INTEGER dest_node
    EXTERNAL handler
    INTEGER n_data_args[, data1, ..., dataN]

SUBROUTINE CMAML_request_va_tohost
SUBROUTINE CMAML_reply_va_tohost
SUBROUTINE CMAML_rpc_va_tohost
          (handler, n_data_args[, data1, ..., dataN])
    EXTERNAL handler
    INTEGER n_data_args[, data1, ..., dataN]
```

Copyright © 1995 Thinking Machines Corporation

ARGUMENTS

dest_node An integer specifying the destination node for the message.

handler Address of a handler function on the destination node.

n_data_args Number of arguments this function provides for use by the handler; hence, the number of words of data being sent to the receiving node.

data1, ..., dataN
Arguments (that is, words of data) to be passed to the handler function. From 0 to 17 such arguments may be provided for VA-class long active messages.

DESCRIPTION

The VA class is the simplest form of long active message functions. Except for the number of words of data they can send as arguments (anything between 0 and 17, inclusive), they are functionally identical to the corresponding short active message functions. The name VA is a reminder that these functions accept a **V**ariable number of **A**rguments.

In all other particulars, the VA class of active message functions are identical to the short active message functions. They represent the safest class of long active messages, and the easiest to use; as might be expected, they are also the least efficient for many applications.

Each of these functions sends an active message to the destination node specified by *dest_node* with the specified *handler* function and the supplied *data* words:

`CMAML_request_va` sends a Request active message.
`CMAML_reply_va` sends a Reply active message.
`CMAML_rpc_va` sends an RPC active message.

`CMAML_request_va_tohost` sends a Request active message to the host.
`CMAML_reply_va_tohost` sends a Reply active message to the host.
`CMAML_rpc_va_tohost` sends an RPC active message to the host.

VA-class active messages can be used only on the CM-5E. Invoking a VA-class function on the CM-5 produces a fatal error, as does invoking a VA `_tohost` function on the host. An application can call `CMAML_pn_long_am_words` to determine whether or not the partition on which it is running can handle long active messages.

Version 3.3, April 1995
Copyright © 1995 Thinking Machines Corporation

Handlers

VA-class active message functions allow up to 17 *immediate* arguments for use by the handler, plus a preceding argument telling how many arguments are being passed. The name VA is a reminder that these functions accept a **V**ariable number of **A**rguments. Handlers for this class of function are written to accept a fixed number of arguments. Thus, handlers expecting four or fewer arguments may be shared between short active messages and VA-class active messages.

Implementation Note: Short active messages always send five words through the network (the handler plus four data words), though some compilers allow the sending function to be called with fewer than four data arguments. In such a case, the remaining words in the packet contain garbage. VA-class active messages, in contrast, always send only the number of words specified by the `n_args` argument (plus one word for the handler).

It is not necessarily an error for the handler to be written to accept either more or fewer arguments than are passed by a given invocation of a VA-class active message function. If you send ten arguments, but your handler accepts only seven, then the last three arguments are ignored. Conversely, if you invoke a function with `n_args` equal to 10, but your handler expects 12 arguments, then the last two arguments seen by the handler will be garbage.

Requests, Replies, and RPCs

Programmers should remember the distinctions between Request, Reply, and RPC functions:

Request function:

- may be called only from user code (not from within any handlers)
- can be sent with interrupts enabled
- can send only Reply active messages from within its handler (cannot invoke any other type of communications function)
- block until complete

Reply function:

- may be called only from within the handler of a Request active message
- are not allowed to invoke any communication functions whatsoever in their handlers
- check only the Reply interface for incoming messages

Copyright © 1995 Thinking Machines Corporation

- block until complete

Note that it is not necessary to stipulate that Replies be invoked with interrupts disabled, for it is already stated that they may be invoked only inside a handler — where interrupts are *always* disabled. Note also the implication that enabling interrupts inside a handler is a serious, possibly fatal, error.

RPC function:

- may be called from either user code or a handler
- must be called with interrupts disabled
- block when called from user code; do not (and must not) block when called from within a handler
- may call nonblocking communication functions from within their handlers
- when called from within a handler, might not actually send its message upon the call, but might queue the message for later execution. The user is guaranteed only that the message will be sent before control returns to top-level user code.

Interrupts

We emphasize that `CMAML_request` functions may be called with interrupts enabled. All other active message functions must be called with interrupts disabled.

_tohost Functions

The `_tohost` functions are for use in host/node programming. The short versions may be called on the nodes or the host. The long versions may be called only on the nodes; attempting to call them on the host will produce a fatal error. (**NOTE:** Neither short nor long `_tohost` functions should ever be called in a hostless program.)

Warning: For the `_tohost` functions, the handler argument is the address of a function on the *host* (the partition manager). Whereas the address of a function on any one of the nodes is the same across all the nodes, the corresponding address of the function on the host may be entirely different, and can vary from run to run. You must therefore obtain the correct handler address from the host *each time* your program executes (for example, by exchanging a message using `CMMD_send_block` and `CMMD_receive_block`), and *before* sending any `_tohost` active messages.

Version 3.3, April 1995
Copyright © 1995 Thinking Machines Corporation

VAPOP-Class Long Active Messages

Send a VAPOP-class active message (a handler, followed by 0 to 17 words of data, which the handler must pop from the network into destination memory); usable on the CM-5E only.

SYNOPSIS

C Syntax

```
void CMAML_request_vapop
void CMAML_reply_vapop
void CMAML_rpc_vapop
    (int dest_node, void (*handler)(),
     int n_data_args[, int data1, ..., int dataN])

void CMAML_request_vapop_tohost
void CMAML_reply_vapop_tohost
void CMAML_rpc_vapop_tohost
    (void (*handler)(),
     int n_data_args[, int data1, ..., int dataN])
```

Fortran Syntax

```
SUBROUTINE CMAML_request_vapop
SUBROUTINE CMAML_reply_vapop
SUBROUTINE CMAML_rpc_vapop
          (dest_node, handler, n_data_args[, data1, ..., dataN])
    INTEGER dest_node
    EXTERNAL handler
    INTEGER n_data_args[, data1, ..., dataN]

SUBROUTINE CMAML_request_vapop_tohost
SUBROUTINE CMAML_reply_vapop_tohost
SUBROUTINE CMAML_rpc_vapop_tohost
          (handler, n_data_args[, data1, ..., dataN])
    EXTERNAL handler
    INTEGER n_data_args[, data1, ..., dataN]
```

ARGUMENTS

dest_node	An integer specifying the destination node for the message.

Copyright © 1995 Thinking Machines Corporation

handler Address of a handler function on the destination node.

n_data_args Number of arguments this VAPOP function provides for use by the handler; hence, the number of words of data being sent to the receiving node.

data1, ..., dataN
Data accessible by the handler function. For a VAPOP function, they contain data that the handler must "pop" from the network. From 0 to 17 words of data may be provided, one word per argument.

DESCRIPTION

VAPOP-class long active messages specify a Variable number of immediate Arguments, all but the first two of which are POPped by the handler function directly from the network into destination memory. They may be viewed as combining the function of active messages (in that they invoke handler functions) with the function of data transmission (in that they transfer up to 17 words of data from the source node directly into destination memory). The usual triad of types is available, plus `_tohost` counterparts.

`CMAML_request_vapop` sends a Request active message.
`CMAML_reply_vapop` sends a Reply active message.
`CMAML_rpc_vapop` sends an RPC active message.

`CMAML_request_vapop_tohost` sends a Request active message to the host.
`CMAML_reply_vapop_tohost` sends a Reply active message to the host.
`CMAML_rpc_vapop_tohost` sends an RPC active message to the host.

The VAPOP- and ARRPOP-class functions differ from other active message functions in that the handlers they invoke have full responsibility for removing (or POPping) data from the network and placing it in destination memory. They use the network accessor functions, as explained below, to perform this task.

VAPOP-class active messages can be used only on the CM-5E. Attempting to use them on the CM-5 causes a fatal error. The functions `CMAML_[pn|cp]_long_am_nwords` (see below) are used to determine whether it is permissible to call them.

Version 3.3, April 1995
Copyright © 1995 Thinking Machines Corporation

Handlers

Whereas the lengths of the formal parameter lists of VA-class handlers may differ from handler to handler, VAPOP-class handlers all look essentially the same: they have two formal parameters, traditionally called *fifo* and *status*. The precise names are unimportant; they could just as well be called *cookie1* and *cookie2*, for they are simply placeholders for opaque objects or "magic cookies" provided by CMAML. Their values are neither created nor read by the user. Instead, they are generated "behind the scenes" and passed to the handler when it is invoked. The handler in turn passes them to special *accessor functions* (such as `CMAML_popd`), which use them as "keys" to remove the data from the network and store it *directly to memory*. These accessor functions are described in detail below.

Efficiencies and Dangers: For many applications, handlers for VAPOP active message functions are inherently more efficient than handlers for VA functions, since they avoid making a redundant copy of data to the stack on the destination node. However, they are more dangerous to use than VA functions. Both the efficiency and the danger come from the fact that the handler function itself has full responsibility for removing all data transmitted by the function from the network, *and for ensuring that it removes the correct amount.*

PLEASE NOTE: When using VAPOP or ARRPOP functions, it is absolutely critical to ensure that the handlers always pop exactly the right amount of data from the network. If they fail to do so, bus errors, partitions crashes, or virtually undebuggable programs may result. In general, this is a danger only if the program contains handlers that call `CMAML_popn[d]`; all other accessor functions ("pop" functions) *automatically* remove exactly the right amount of data.

Requests, Replies, and RPCs

Programmers should remember the distinctions between Request, Reply, and RPC functions:

Request function:

- may be called only from user code (not from within any handlers)
- can be sent with interrupts enabled
- can send only Reply active messages from within its handler (cannot invoke any other type of communications function)
- block until complete

Copyright © 1995 Thinking Machines Corporation

Reply function:

- may be called only from within the handler of a Request active message
- are not allowed to invoke any communication functions whatsoever in their handlers
- check only the Reply interface for incoming messages
- block until complete

Note that it is not necessary to stipulate that Replies be invoked with interrupts disabled, for it is already stated that they may be invoked only inside a handler — where interrupts are *always* disabled. Note also the implication that enabling interrupts inside a handler is a serious, possibly fatal, error.

RPC function:

- may be called from either user code or a handler
- must be called with interrupts disabled
- block when called from user code; do not (and must not) block when called from within a handler
- may call nonblocking communication functions from within their handlers
- when called from within a handler, might not actually send its message upon the call, but might queue the message for later execution. The user is guaranteed only that the message will be sent before control returns to top-level user code.

Interrupts

We emphasize that `CMAML_request` functions may be called with interrupts enabled. All other active message functions must be called with interrupts disabled.

_tohost Functions

The `_tohost` functions are for use in host/node programming. The short versions may be called on the nodes or the host. The long versions may be called only on the nodes; attempting to call them on the host will produce a fatal error. (**NOTE:** Neither short nor long `_tohost` functions should ever be called in a hostless program.)

Warning: For the `_tohost` functions, the handler argument is the address of a function on the *host* (the partition manager). Whereas the address of a function on

Version 3.3, April 1995
Copyright © 1995 Thinking Machines Corporation

any one of the nodes is the same across all the nodes, the corresponding address of the function on the host may be entirely different, and can vary from run to run. You must therefore obtain the correct handler address from the host *each time* your program executes (for example, by exchanging a message using `CMMD_send_block` and `CMMD_receive_block`), and *before* sending any `_tohost` active messages.

Copyright © 1995 Thinking Machines Corporation

ARRPOP-Class Long Active Messages

The source node writes an ARRay directly from source memory into the network, to be POPped directly into destination memory by the handler function; usable on the CM-5E only.

SYNOPSIS

C Syntax

```
void CMAML_request_arrpop0
void CMAML_reply_arrpop0
void CMAML_rpc_arrpop0
    (int dest_node, void (*handler)(), int n_words, void *data)

void CMAML_request_arrpop1
void CMAML_reply_arrpop1
void CMAML_rpc_arrpop1
    (int dest_node, void (*handler)(), int arg0, int n_words,
     void *data)

void CMAML_request_arrpop2
void CMAML_reply_arrpop2
void CMAML_rpc_arrpop2
    (int dest_node, void (*handler)(), int arg0, int arg1,
     int n_words, void *data)

void CMAML_request_arrpop0_tohost
void CMAML_reply_arrpop0_tohost
void CMAML_rpc_arrpop0_tohost
    (void (*handler)(), int n_words, void *data)

void CMAML_request_arrpop1_tohost
void CMAML_reply_arrpop1_tohost
void CMAML_rpc_arrpop1_tohost
    (void (*handler)(), int arg0, int n_words, void *data)

void CMAML_request_arrpop2_tohost
void CMAML_reply_arrpop2_tohost
void CMAML_rpc_arrpop2_tohost
    (void (*handler)(), int arg0, int arg1, int n_words,
     void *data)
```


Version 3.3, April 1995
Copyright © 1995 Thinking Machines Corporation

Fortran Syntax

```
SUBROUTINE CMAML_request_arrpop0
SUBROUTINE CMAML_reply_arrpop0
SUBROUTINE CMAML_rpc_arrpop0
          (dest_node, handler, n_words, data)
    INTEGER dest_node
    EXTERNAL handler
    INTEGER n_words
    <ARRAY> data

SUBROUTINE CMAML_request_arrpop1
SUBROUTINE CMAML_reply_arrpop1
SUBROUTINE CMAML_rpc_arrpop1
          (dest_node, handler, arg1, n_words, data)
    INTEGER dest_node,
    EXTERNAL handler
    INTEGER arg1, n_words
    <ARRAY> data


SUBROUTINE CMAML_request_arrpop2
SUBROUTINE CMAML_reply_arrpop2
SUBROUTINE CMAML_rpc_arrpop2
          (dest_node, handler, arg1, arg2, n_words, data)
    INTEGER dest_node
    EXTERNAL handler
    INTEGER arg1, arg2, n_words
    <ARRAY> data

SUBROUTINE CMAML_request_arrpop0_tohost
SUBROUTINE CMAML_reply_arrpop0_tohost
SUBROUTINE CMAML_rpc_arrpop0_tohost
          (handler, n_words, data)
    EXTERNAL handler
    INTEGER n_words
    <ARRAY> data

SUBROUTINE CMAML_request_arrpop1_tohost
SUBROUTINE CMAML_reply_arrpop1_tohost
SUBROUTINE CMAML_rpc_arrpop1_tohost
          (handler, n_words, data)
    EXTERNAL handler
    INTEGER arg1, n_words
    <ARRAY> data
```


Copyright © 1995 Thinking Machines Corporation

```
SUBROUTINE CMAML_request_arrpop2_tohost
SUBROUTINE CMAML_reply_arrpop2_tohost
SUBROUTINE CMAML_rpc_arrpop2_tohost
          (handler, n_words, data)
    EXTERNAL handler
    INTEGER arg1, arg2, n_words
    <ARRAY> data
```

ARGUMENTS

dest_node	An integer specifying the destination node for the message.
handler	Address of a handler function on the destination node.
arg0, arg1	Immediate arguments to be prepended to the array data.
n_words	Number of words of data that are to be written into the network from the source location beginning at *data* and popped from the network by the handler into destination memory. For `arrpop0` functions, the maximum value of *n_words* is 17. For `arrpop1` functions, the maximum value is 16. For `arrpop2` functions, the maximum value is 15.
data	Starting location of an area of memory on the sending node. Starting at this location, *n_words* of data will be copied by the function directly into the network, then retrieved from the network and stored into destination memory by the handler. For best performance, the data must be doubleword-aligned; word alignment gives poorer performance than doubleword alignment, but better performance than byte alignment.

DESCRIPTION

The ARRPOP class of long active message functions allow a source node to write an ARRay directly from source memory into the network. The handler function on the destination node then POPs the array from the network directly into destination memory. Thus, ARRPOP-class functions represent direct memory-to-memory transmissions.

The syntax of ARRPOP-class functions allows the sending node to specify a handler function on the receiving node, supply up to two data arguments for the

Version 3.3, April 1995
Copyright © 1995 Thinking Machines Corporation

function, and then specify (via *pointer* and *length* arguments) an array of data that will be included in the transmission for the receiving node's handler to pop from the network. When the function executes on the sending node, the pointed-at data are written directly from memory into the network. On the receiving node, the data are popped, by the handler, directly from the network into memory. Thus, all redundant copying is avoided.

If the sending function is one of the `arrpop1` types, then the auxiliary argument *arg0* is prepended to the array data before it is sent. The first word extracted from the network by the handler is *arg0*, followed by the array words 0 through *n_words*−1. Similarly, if the sending function is one of the `arrpop2` types, then both auxiliary arguments *arg0* and *arg1* are prepended to the array data before it is sent. The first two words extracted from the network by the handler are *arg0* and *arg1*, in that order, followed by the array words 0 through *n_words*−1.

Handlers for ARRPOP functions are written with the same calling sequence as those for VAPOP functions, and they use the same accessor functions.

Efficiencies and Dangers: The ARRPOP-class functions avoid a redundant copy on both the sending and receiving nodes. This makes them the most efficient of the long active message functions for many applications, both in speed and in efficient memory use. The cost of this efficiency is the danger inherent in giving the user-written handler functions full responsibility for determining exactly how much data to remove from the network and for removing it.

Programmers writing ARRPOP-class functions should also recognize that doubleword- or singleword-aligned pointers are handled much more efficiently than those that are byte- or halfword-aligned. (As usual, doubleword alignment is most efficient.) This happens because the transport mechanisms that underlie all active messages are word-based. Thus, the number of bytes of data available to a handler is always a multiple of four.

To underscore the efficiency of word- and double-word alignment, all arguments pertaining to length or number of arguments use *words* as their unit of measurement. This differs from similar arguments for most other functions, which use bytes as their units.

PLEASE NOTE: When using VAPOP or ARRPOP functions, it is absolutely critical to ensure that the handlers always pop exactly the right number of words from the network. Popping the wrong number of words from the network can result in bus errors, partition crashes, or virtually undebuggable programs.

Like other long active message functions, the ARRPOP-class functions can be used only on the CM-5E. Attempting to use them on the CM-5 causes a fatal error.

Copyright © 1995 Thinking Machines Corporation

As usual, the form of active message varies with the function:

`CMAML_request_arrpop`*N* sends a Request active message.
`CMAML_reply_arrpop`*N* sends a Reply active message.
`CMAML_rpc_arrpop`*N* sends an RPC active message.

`CMAML_request_arrpop`*N*`_tohost` sends a Request active message to the host.
`CMAML_reply_arrpop`*N*`_tohost` sends a Reply active message to the host.
`CMAML_rpc_arrpop`*N*`_tohost` sends an RPC active message to the host.

Requests, Replies, and RPCs

Programmers should remember the distinctions between Request, Reply, and RPC functions:

Request function:

- may be called only from user code (not from within any handlers)
- can be sent with interrupts enabled
- can send only Reply active messages from within its handler (cannot invoke any other type of communications function)
- block until complete

Reply function:

- may be called only from within the handler of a Request active message
- are not allowed to invoke any communication functions whatsoever in their handlers
- check only the Reply interface for incoming messages
- block until complete

Note that it is not necessary to stipulate that Replies be invoked with interrupts disabled, for it is already stated that they may be invoked only inside a handler — where interrupts are *always* disabled. Note also the implication that enabling interrupts inside a handler is a serious, possibly fatal, error.

RPC function:

- may be called from either user code or a handler
- must be called with interrupts disabled

Version 3.3, April 1995
Copyright © 1995 Thinking Machines Corporation

- block when called from user code; do not (and must not) block when called from within a handler
- may call nonblocking communication functions from within their handlers
- when called from within a handler, might not actually send its message upon the call, but might queue the message for later execution. The user is guaranteed only that the message will be sent before control returns to top-level user code.

Interrupts

We emphasize that `CMAML_request` functions may be called with interrupts enabled. All other active message functions must be called with interrupts disabled.

_tohost Functions

The `_tohost` functions are for use in host/node programming. The short versions may be called on the nodes or the host. The long versions may be called only on the nodes; attempting to call them on the host will produce a fatal error. (**NOTE:** Neither short nor long `_tohost` functions should ever be called in a hostless program.)

Warning: For the `_tohost` functions, the handler argument is the address of a function on the *host* (the partition manager). Whereas the address of a function on any one of the nodes is the same across all the nodes, the corresponding address of the function on the host may be entirely different, and can vary from run to run. You must therefore obtain the correct handler address from the host *each time* your program executes (for example, by exchanging a message using `CMMD_send_block` and `CMMD_receive_block`), and *before* sending any `_tohost` active messages.

Copyright © 1995 Thinking Machines Corporation

8.5 Active Messages: Popping Data

VAPOP- and ARRPOP-class handlers are responsible for removing data from the network and placing it in the memory of the receiving node. They accomplish this by invoking accessor functions, and providing those functions with two "magic cookies" that have been passed as arguments by the VAPOP or ARRPOP function call. The accessor functions are

`CMAML_pop` `CMAML_popd`	Remove the data waiting in the network and store it at a specified address

`CMAML_poptofirst`
`CMAML_poptofirstd`
`CMAML_poptofirst_ra`
`CMAML_poptofirst_ra_d`
`CMAML_poptofirst_rla`
`CMAML_poptofirst_rla_d`
	Remove the data waiting in the network and store it at the address specified by the first data word

`CMAML_popn` `CMAML_popnd`	Remove *n* words of data from the network and store it at the specified address

The recommended functions from this list are `CMAML_popd` and `CMAML_poptofirstxxx`. They expect doubleword-aligned addresses, and are thus highly efficient. Moreover, they allow the system to make the critical decision as to how many words of data to read from the network, and thus always pop the correct amount of data. (But note the WARNING, below.)

In contrast, `CMAML_popn` and `CMAML_popnd` require that the handler function know how much data to remove from the network and remove exactly the right amount. Using these functions incorrectly can result in fatal errors that are hard to debug. These functions are useful when you want to scatter the data from a single message among several addresses; they should be avoided otherwise.

WARNING: Bus errors caused by providing a bad *n* to `CMAML_popn` or `CMAML_popnd` will not necessarily occur in the same call to which the bad *n* is passed. For example, if a call to `CMAML_popn` reads too many words from the network, a subsequent "correct" call to `CMAML_popd` may fail, with a bus error, for lack of sufficient data. Such delayed bus errors are difficult to debug.

Reference pages for these functions follow.

Copyright © 1995 Thinking Machines Corporation

CMAML_pop
CMAML_popd

Remove data from the network and store it into node memory.

SYNOPSIS

C Syntax

```
int CMAML_pop (int fifo, int status, void *dest)

int CMAML_popd(int fifo, int status, double *dest)
```

Fortran Syntax

```
INTEGER FUNCTION CMAML_pop(fifo, status, dest)
    INTEGER fifo, status
    <ARRAY> dest

INTEGER FUNCTION CMAML_popd(fifo, status, dest)
    INTEGER fifo, status,
    REAL*8 dest(*)
```

ARGUMENTS

fifo — The first of the two "magic cookie" arguments that all VAPOP and ARRPOP handlers accept. The value of *fifo* is generated by system software and passed automatically to VAPOP and ARPOP handlers when they are invoked. The handler in turn passes it as a first argument to one of the *pop* functions, which use it as one half of a "key" to transfer data from the network into user memory.

status — The second of the two "magic cookie" arguments that all VAPOP and ARRPOP handlers accept. Like *fifo*, it is never generated or inspected by the user, but is simply passed on to one of the functions being described. It forms the other half of the "key" to the network.

Copyright © 1995 Thinking Machines Corporation

dest The starting location in memory to which the data words are to be copied. For the popd, poptofirstd, and popnd functions, *dest* must be doubleword-aligned. The other functions accept arbitrary alignments, though word alignment or double alignment improves their performance. In all cases, *dest* must point to a valid region of memory with room for the entire message (that is, 4 x *n_words* bytes).

NOTE: It is possible for a user program to contain, textually, not only definitions of short active message or VA-class handlers, but calls to them as well. This is because such handlers are written to accept arguments that could in priniciple be user-generated. Thus they are "ordinary" functions that may be called either locally, in text, or remotely, through an active message. In contrast, the text of a program may contain *definitions* of VAPOP and ARRPOP handlers, but no actual *calls*. This is because the "magic cookie" arguments that these handlers expect can be generated only by non−user-visible CMAML system software.

RETURN VALUES

These two functions return the number of words popped from the network.

DESCRIPTION

CMAML_pop and CMAML_popd are used only by VAPOP or ARRPOP handlers, to remove data from the network and store them into the receiver's memory. They take two "magic cookie" arguments and use information suppied by those arguments to pop the entire message sent by the VAPOP or ARRPOP function into a region of memory beginning at location *dest* on the local (receiving) node. For CMAML_popd, *dest* must be doubleword-aligned.

Users neither set nor access the values for the *status* and *fifo* arguments. Instead, they simply write handlers that expect *status* and *fifo* as arguments and that then pass these arguments to the accessor functions they invoke. The system then supplies the value directly to the accessor functions, which use it for such purposes as determining the length of the message and its location in the network.

The *dest* argument is set by the user. It must point to a valid region of memory containing as many words as are in the message (one word per argument, four bytes per word). For CMAML_popd, the location must be doubleword-aligned. If it is not, a fatal error results.

Version 3.3, April 1995
Copyright © 1995 Thinking Machines Corporation

`CMAML_popd` is far more efficient than `CMAML_pop`. Its use is therefore recommended when possible. Please note that its alignment restriction applies only to the destination buffer. The sender need not have sent doubleword-aligned data, and the number of words in the message does not need to be a multiple of two.

Both `CMAML_pop` and `CMAML_popd` are guaranteed to remove the right number of words of data from the network. **NOTE:** It is *possible* for `CMAML_pop` or `CMAML_popd` to generate an error that *appears* as though they are reading the wrong amount of data. However, this can only be due to an improper previous use of `CMAML_popn[d]` (or improper user `CMNA` programming).

Copyright © 1995 Thinking Machines Corporation

CMAML_poptofirst, CMAML_poptofirstd, CMAML_poptofirst_ra, CMAML_poptofirst_ra_d, CMAML_poptofirst_rla, CMAML_poptofirst_rla_d

Remove data from the network and store it into node memory, at the address specified by the first data word.

SYNOPSIS

C Syntax

```
int CMAML_poptofirst(int fifo, int status)
int CMAML_poptofirstd(int fifo, int status)
int CMAML_poptofirst_ra(int fifo, int status)
int CMAML_poptofirstd_ra_d(int fifo, int status)
int CMAML_poptofirst_rla(int fifo, int status)
int CMAML_poptofirstd_rla_d(int fifo, int status)
```

Fortran Syntax

```
INTEGER FUNCTION CMAML_poptofirst(fifo, status)
    INTEGER fifo, status

INTEGER FUNCTION CMAML_poptofirstd(fifo, status)
    INTEGER fifo, status
```

ARGUMENTS

fifo — The first of the two "magic cookie" arguments that all VAPOP and ARRPOP handlers accept. The value of *fifo* is generated by system software and passed automatically to VAPOP and ARPOP handlers when they are invoked. The handler in turn passes it as a first argument to one of the *pop* functions, which use it as one half of a "key" to transfer data from the network into user memory.

Version 3.3, April 1995
Copyright © 1995 Thinking Machines Corporation

status The second of the two "magic cookie" arguments that all VAPOP and ARRPOP handlers accept. Like *fifo*, it is never generated or inspected by the user, but is simply passed on to one of the functions being described. It forms the other half of the "key" to the network.

RETURN VALUES

These two functions return the number of words popped from the network.

These functions differ only in their return values:

- `CMAML_poptofirst[d]` returns the number of words popped from the network.
- `CMAML_poptofirst_ra[_d]` returns the address into which the data was written
- `CMAML_poptofirst_rla[_d]` returns both length and address.

In the last case, length is an actual C-language return value, while the address is written into a pointer that is passed as an argument.

DESCRIPTION

`CMAML_poptofirst` and `CMAML_poptofirstd` use the information supplied by the *fifo* and *status* arguments to pop the entire message sent by a VAPOP or ARRPOP function into a region of memory beginning at the location specified by the first data argument passed to the VAPOP or ARRPOP function's handler function.

Users neither set nor access the values for the *status* and *fifo* arguments. Instead, they simply write handlers that expect *status* and *fifo* as arguments and that then pass these arguments to the accessor functions they invoke. The system supplies the correct values directly to the handler functions upon handler invocation.

In some applications, the first data word sent in the active message may actually be an address indicating a destination buffer. It must point to a valid region of memory containing at least as many words as are in the message (one word per argument, four bytes per word). For `CMAML_poptofirst_[ra|rla]_d`, the location must be doubleword-aligned. If it is not, a fatal error results.

Copyright © 1995 Thinking Machines Corporation

`CMAML_poptofirst_[ra|rla]_d` demand doubleword-aligned buffers. Since these are more efficient than word-aligned or byte-aligned buffers, `CMAML_poptofirst_[ra|rla]_d` are far more efficient than `CMAML_poptofirst_[ra|rla]`. Their use is therefore recommended when possible.

Please note that the alignment restriction for `CMAML_poptofirst_[ra|rla]_d` applies only to the destination buffer. The sender's data need not be doubleword-aligned, and the number of words in the message may be odd.

Version 3.3, April 1995
Copyright © 1995 Thinking Machines Corporation

CMAML_popn
CMAML_popnd

Remove *n* words of data from the network and store it into node memory. These are the most dangerous functions in the CMAML library.

SYNOPSIS

C Syntax

```
void CMAML_popn(int fifo, void *dest, int n_words)

void CMAML_popnd(int fifo, double *dest, int n_words)
```

Fortran Syntax

```
SUBROUTINE CMAML_popn(fifo, dest, n_words)
    INTEGER fifo,
    <ARRAY> dest
    INTEGER n_words

SUBROUTINE CMAML_popnd(fifo, dest, n_words)
    INTEGER fifo
    REAL*8 dest
    INTEGER n_words
```

ARGUMENTS

fifo — The first of the two "magic cookie" arguments that all VAPOP and ARRPOP handlers accept. The value of *fifo* is generated by system software and passed automatically to VAPOP and ARPOP handlers when they are invoked. The handler in turn passes it as a first argument to one of the *pop* functions, which use it as one half of a "key" to transfer data from the network into user memory.

dest — The starting location in memory to which the data words are to be copied. For the popnd function, *dest* must be doubleword-aligned. The popn function accepts arbitrary alignments. In all cases, *dest* must point to a valid region of memory with room for the entire message (that is, 4 x *n_words* bytes).

Copyright © 1995 Thinking Machines Corporation

n_words Number of words of data that are to be popped from the network by the handler into destination memory. The number can be obtained by a call to CMAML_n_to_pop.

RETURN VALUES

CMAML_popn and CMAML_popnd return no values.

DESCRIPTION

CMAML_popn and CMAML_popnd are used by VAPOP or ARRPOP handler functions to pop *n* words of data from the network and store them in the memory location starting at *dest*. Note that the number of words the handler removes from the network before it returns *must* equal the number sent by the VAPOP or ARRPOP function whose handler is executing; otherwise, immediate or delayed bus errors, or even partition crashes, will result. The data may be removed in one call to CMAML_popn or CMAML_popnd or in several calls, but the correct amount must ultimately be removed.

The functions use the "magic cookie" *fifo* argument provided by the handler function invoked by the VAPOP or ARRPOP function call. They do not use the *status* "magic cookie."

The popn[d] functions should use the CMAML_n_to_pop function to tell them how many words of data are in the network waiting to be popped. This is the only safe way to determine the value of *n_words*, or at least to check that the provided value is not disastrously wrong. (See the sample handler provided below.)

These functions are slower than the equivalent CMAML_pop functions. They are also far more dangerous, as they may lead to virtually undebuggable bus errors, or even to partition crashes. Their purpose is to allow the data being sent by a *single* long active message to be scattered among different locations.

The *dest* argument must point to a valid region of memory containing as many words as are to be stored (one word per argument, four bytes per word). For CMAML_popnd, the location must be doubleword-aligned. If it is not, a fatal error may result. Note that CMAML_popnd is far more efficient than CMAML_popn.

Note that the following two calls are functionally equivalent:

```
CMAML_popn(fifo, dest, CMAML_n_to_pop());
CMAML_pop(fifo, status, dest);
```


Version 3.3, April 1995
Copyright © 1995 Thinking Machines Corporation

`CMAML_pop` may therefore appear to be simply a convenience function; however, it is in fact faster than `CMAML_popn`. Therefore the first of the two forms above should never be written.

SAMPLE PROGRAM

The following program illustrates the use of `CMAML_arrpop1` and `CMAML_popn`.

```
#include <stdio.h>
#include <cm/cmmd.h>

/* This is a transpose program: it copies a 16x16 matrix in pn0
 * to a 16x16 matrix in pn1, interchanging rows and columns in
 * the process.
 *
 *            It turns this:           into this:
 *                             a e i m            a b c d
 *                             b f j n            e f g h
 *                             c g k o            i j k l
 *                             d h l p            m n o p,
 *
 * (except that the matrices are 16x16 instead of 4x4).

 * The matrices are represented as one-dimensional arrays of length
 * 256, where the 16 contiguous 16-words blocks are considered columns.
 * Therefore the communication pattern looks like this:
 *
 *       col: 1                2                3                4...
 * Source:   $XXXXXXXXXXXXXXXX$XXXXXXXXXXXXXXXX$XXXXXXXXXXXXXXXX$XXXX.....
 *            | \_ \_  . . .   /  | \_  . . .
 *            |   \_ \__     _/   |   \_
 *            |     \_  \__ /     |     \_
 *            |       \_  /\____  |       \_           etc....
 *            |         \/      \_|___      \_
 *            |        _/ \_      |   \_____  \_
 *            |     __/     \_    |         \_  \_
 *            |    /          \   |           \_  \
 *           \|/ \/_ ...      _\|\|/ ...       _\|\|/ ...
 * Dest:     $XXXXXXXXXXXXXXXX$XXXXXXXXXXXXXXXX$XXXXXXXXXXXXXXXX$XXX-.....
 *       col: 1                2                3                4...
 *
```


Copyright © 1995 Thinking Machines Corporation

```
 *
 */

int src[256];
int dst[256];

volatile int columns_received = 0;

void transpose_handler(int fifo, int status)
{
  int i, j; /* local row, column */

  /* This handler receives an entire 16-word column from the source
   * matrix and lays it down as a *row* in the dest matrix.
   */

  /* Since we know what is going on here, we know that all messages
   * should be of length 17.
   */
  if( CMAML_n_to_pop(status) != 17 )
    CMMD_error("Error. Unexpected message length 7, should be 17.",
         CMAML_n_to_pop(status));

  /* The first word of the message is the column it came from,
   * i.e., its new row (this was j in the sending node):
   */
  CMAML_popn(fifo, (char *)(&i), 1);

  /* Lay down the remaining data with a stride of 16:
   */
  for( j=0; j<16; j++ )
    CMAML_popn(fifo, (char *)(dst+i+(16*j)), 1);

  /* Bump the global counter:
   */
  columns_received++;
}

void main(int argc, char *argv[])
{
  int i, j;

  /* As a general rule, disable interrupts when
   * writing in CMAML.  (If you can relax this constraint
   * later, then fine.)
```


Version 3.3, April 1995
Copyright © 1995 Thinking Machines Corporation

```
 */
CMAML_disable_interrupts();

/* We'll be printing stuff from individual nodes...
 */
CMMD_fset_io_mode(stdout, CMMD_independent);

/* SENDING NODE:
 *
 * Initialize the source matrix with a very human-readable pattern:
 */
if( CMMD_self_address() == 0 ) {

  /* Rows are "i", columns are "j": (i,j)
   */
  printf("Source matrix:\n");

  /* Fill in the matrix one column at a time, with an
   * integer that looks like "i j" when printed as decimal:
   */
  for( j=0; j<16; j++ )   /* <-col */
    for( i=0; i<16; i++ ) /* <-row */
        src[(j*16)+i] = (((i+1)*100)+(j+1));

  /* Print out the src matrix, one row at a time:
   */
  for( i=0; i<16; i++ ) { /* <-row */
    for( j=0; j<16; j++ ) /* <-col */
       printf("%04d ", src[(j*16)+i]);
    printf("\n");
  }

  /* Send all the columns to pn1:
   */
  for( j=0; j<16; j++ ) /* <-col */
    CMAML_rpc_arrpop1(1,                    /* <-- dest pn     */
                      transpose_handler,    /* <-- handler     */
                      j,                    /* <-- column      */
                      16,                   /* <-- data length */
                      (char *)(src+(j*16)) /* <-- source seg. */

}

/* RECEIVING NODE:
 */
```


Copyright © 1995 Thinking Machines Corporation

```
  if( CMMD_self_address() == 1 ) {
    while( columns_received < 16 )
      CMAML_poll();

    /* Print out the dst matrix, one row at a time:
     */
    printf("Dest matrix:\n");

    for( i=0; i<16; i++ ) { /* <-row */
      for( j=0; j<16; j++ ) /* <-col */
          printf("%04d ", dst[(j*16)+i]);
      printf("\n");
    }
  }
}
```

WARNING

It is a peculiarity of the CM-5 system that the error of attempting to read too many words from the Data Network is often manifested as a bus error. In general, the bus error may not show up immediately. If data from a second message is waiting in the network, the `popn` function may read some of that data. The second message then either finds itself short of data or reads data from an incorrect message from the network, i.e., data belonging to the message behind it in line. The chain of corruption continues until some luckless function comes up short of data words and signals the bus error.

Errors of the type described above are notoriously difficult to find and fix. Because of the degree of trouble they can cause, we strongly urge that you use the `pop` or `poptofirst` functions rather than the `popn` functions, except where you need the specific functionality provided by `popn[d]`.

Version 3.3, April 1995
Copyright © 1995 Thinking Machines Corporation

8.6 Active Messages: Informational Functions

CMAML provides an informational function to inform `CMAML_popn[d]` as to how many words of data have been placed in the network by the long active message that invoked the handler function. Thus, the return value of this function identifies the total amount of data to be removed from the network.

CMAML also provides two functions that return the maximum number of words a long active message can contain. By checking the return value for zero or nonzero, an application can find out whether the node (or host) hardware in this partition is capable of sending or receiving a long active message, or whether a long active message call will instead result in a fatal error.

The functions are

```
CMAML_n_to_pop
CMAML_pn_long_am_nwords
CMAML_cp_long_am_nwords
```

Their reference pages follow.

Copyright © 1995 Thinking Machines Corporation

CMAML_n_to_pop

Tells how many words of data the active message provided; this is the total number of words that must be removed from the network by the handler function.

SYNOPSIS

C Syntax

```
int CMAML_n_to_pop(int status)
```

Fortran Syntax

```
INTEGER FUNCTION CMAML_n_to_pop(status)
    INTEGER status
```

ARGUMENTS

status — The second of two "magic cookie" arguments that all VAPOP and ARRPOP handlers accept. The value of *status* is generated by system software and passed automatically to VAPOP and ARRPOP handlers when they are invoked. The handler in turn passes it either as the first argument to one of the pop functions previously described, or as the sole argument to CMAML_n_to_pop.

RETURN VALUES

CMAML_n_to_pop returns an integer equal to the number of data words associated with a given VAPOP or ARRPOP handler. A few examples follow.

If the user sends an active message from node 0 to node 17 using the following call:

```
CMAML_rpc_arrpop1(17, foo, x, 12, data);
```

then when foo is invoked as a handler on node 17, a call to CMAML_n_to_pop inside foo will return 13 (12 words of *data*, plus the auxiliary word *x*).

If the user calls

```
CMAML_request_vapop(17, foo, 6, a, b, c, d, e, f);
```


Version 3.3, April 1995
Copyright © 1995 Thinking Machines Corporation

then when `foo` is invoked as a handler on node 17, a call to `CMAML_n_to_pop` inside `foo` will return 6.

If the sender calls

```
CMAML_reply_arrpop0(17, foo, 17, data);
```

then when `foo` is invoked as a handler on node 17, a call to `CMAML_n_to_pop` inside `foo` will return 17.

NOTE: It is never necessary to call `CMAML_n_to_pop` more than once in a given handler, since its return value depends only on *`status`*, and *`status`* does not change within the dynamic scope of a given handler. *In particular, the value returned by* `CMAML_n_to_pop` *does not automatically decrease as words are popped from the network.* It is the user's responsibility to call `CMAML_n_to_pop` once inside a handler, and to keep track of how many words are still waiting to be removed at each instant. Failure to do so will in general result in a fatal error.

Copyright © 1995 Thinking Machines Corporation

CMAML_pn_long_am_nwords
CMAML_cp_long_am_nwords

Tell whether long active messages can be sent to nodes (or host) on this partition, and, if so, the maximum number of words such messages can contain.

SYNOPSIS

C Syntax

```
int CMAML_pn_long_am_nwords(void)
int CMAML_cp_long_am_nwords(void)
```

Fortran Syntax

```
INTEGER FUNCTION CMAML_pn_long_am_nwords()
INTEGER FUNCTION CMAML_cp_long_am_nwords()
```

RETURNS

`CMAML_pn_long_am_nwords` returns the maximum number of data words the calling processor (node or host) can send to any node using the long active message function. On the CM-5E, this number is 17.

`CMAML_cp_long_am_nwords` returns the maximum number of data words the calling processor (host or node) can send to the host using the long active message function.

A return value of 0 indicates that an attempt by the caller to send a long active message will cause a fatal error.

DESCRIPTION

`CMAML_pn_long_am_nwords` returns the maximum number of data words this processor can send to another node using the long active message function.

When called on the host, this function returns the maximum number of data words that can be contained in a long active message from the host to any node. When called on a node, it returns the maximum for messages from one node to any other node. If long active messges are not callable from the calling processor (host or

Version 3.3, April 1995
Copyright © 1995 Thinking Machines Corporation

node) because of an inappropriate hardware configuration, then the function returns 0. Attempting to execute a long active message in these cases causes a fatal error. Therefore, this function serves as the query function for long active message callability.

`CMAML_cp_long_am_nwords` does the same for messages to be sent to the host. When called on a node, it returns the maximum number of data words that can be contained in a message from a node to the host. When called on the host, it returns the maximum number of data words that can be contained in a long active message from any node to the host. (Remember, however, that the host can never send long active messages to itself.) It too returns 0 if the hardware configuration on the calling processor does not allow the sending of long active messages.

On the CM-5E, the maximum number of data words that a long active message can contain is normally 17. There may be some anomalous cases, however. This function can be used to check for them.

Copyright © 1995 Thinking Machines Corporation

9 Managing I/O Rearrange Space

(**NOTE:** The following material also appears as Section 15.6.1 of the *CMMD Reference Manual* for Version 3.3.)

Prior to CMMD Version 3.3 there was a two-gigabyte limit on global I/O using the SDA or DV. The reason was that CMMD I/O makes use of a temporary, non-user-visible "staging area" called the *rearrange area* that is used by OS routines which themselves have an inviolable two-gigabyte limit. Even though the amount of memory on individual processing nodes does not approach this limit, a large group of nodes can easily exceed it. For example, if 256 nodes each attempt to write sixteen megabytes in `sync_sequential` mode, then the total size is four gigabytes. Thus, even though each node may have called its I/O routine with parameters that are "locally reasonable," the underlying two-gigabyte limit is nevertheless exceeded.

In CMMD Version 3.3 all global I/O routines implicitly loop over a rearrange area whose size is always less than or equal to two gigabytes. If the total I/O size is greater, then the operation is transparently broken down into a sequence of smaller operations.

Some of the most useful aspects of this mechanism have been exposed to the user. There are three new routines:

```
int CMMD_set_rearrange_size(int fd, int size);
int CMMD_get_rearrange_size(int fd);
int CMMD_max_rearrange_size(int fd);
```

`CMMD_set_rearrange_size` allows the user to specify the size of the rearrange area, on a per-node, per-file descriptor basis. *This number must be a nonnegative integer multiple of 64*. `CMMD_get_rearrange_size` allows the user to query a file descriptor for the same quantity. `CMMD_max_rearrange_size` returns an integer equal to the largest allowable second argument to `CMMD_set_rearrange_size`.

By default, CMMD will break down an I/O operation into the shortest possible sequence, using the largest possible staging area per iteration (up to approximately two gigabytes). However, if a program has very tight memory constraints, the user may wish to set the staging area size to a smaller value, thereby reducing the degree to which CMMD I/O encroaches upon the rest of the program's memory. (Of course, a performance penalty is incurred whenever this is done, due to the overhead of the looping.)

Version 3.3, April 1995
Copyright © 1995 Thinking Machines Corporation

For example, if the user has a 512-node partition, and wishes to write out a file consisting of sixteen megabytes per node, then the total I/O size (eight gigabytes) will cause CMMD to transparently break down the operation into four two-gigabyte operations, each requiring four megabytes of staging area per node. However, if the user cannot allow CMMD to take up that much space, then he or she can call `CMMD_set_rearrange_size` on that file descriptor with a second argument of (for example) one-half megabyte. This means that CMMD will transparently break down the operation into 32 separate writes, each using one-half megabyte of staging area per node.

`CMMD_get_rearrange_size` and `CMMD_max_rearrange_size` may be called individually and asynchronously on any node. They will always return the same value on every node.

`CMMD_set_rearrange_size`, in contrast, is a synchronous function (like `CMMD_reduce_xxx`). It must be called by all nodes simultaneously, and they must all pass the same arguments. *It will generate a negative error return code if* ***any*** *of the following restrictions are violated:*

- *`size`* must be nonnegative
- *`size`* must be an integer multiple of 64
- *`size`* must be less than or equal to the current value returned by `CMMD_max_rearrange_size(`*`fd`*`)`
- All nodes must eventually call `CMMD_set_rearrange_size`
- All nodes must pass the same value of *`fd`*
- All nodes must pass the same value of *`size`*

It is *strongly* recommended that the user always check for a negative return value and take appropriate action.

For all files, the default value of `CMMD_get_rearrange_size` is zero. This is a *special value* or "shorthand" for telling CMMD to use the largest possible staging area. Therefore the following two calls

```
CMMD_set_rearrange_size(fd, 0);
```

and

```
CMMD_set_rearrange_size(fd, CMMD_max_rearrange_size(fd));
```

are always functionally equivalent.

Copyright © 1995 Thinking Machines Corporation

CMMD_set_rearrange_size

Allows the user to control the amount of "scratch space" used by a particular file description during parallel I/O.

SYNOPSIS

C Syntax

```
int CMMD_set_rearrange_size(int fd, int size)
```

Fortran Syntax

```
INTEGER FUNCTION CMMD_set_rearrange_size(fd, size)
    INTEGER fd, size
```

ARGUMENT

fd	A file descriptor for a global file, intended to be used for `sync_sequential` I/O to the SDA or the DV.
size	The size (on a per-node basis) of the internal staging area or "scratch space" that CMMD is allowed to use for parallel I/O.

DESCRIPTION

This synchronous function allows the user to control the amount of "scratch space" used by a particular file during parallel I/O.

I/O to the SDA or DV makes use of a chunk of non-user-visible scratch space called the *rearrange area*. The size of the rearrange area (the *rearrange size*) is interpreted on a per-node basis, meaning that the total amount of scratch space that CMMD actually uses on a given I/O is equal to the per-node rearrange size times the partition size.

CMMD keeps track of the rearrange size on a per-file descriptor basis. The rearrange size used by a particular file descriptor *fd* may be changed using `CMMD_set_rearrange_size`. The default value for every file descriptor is zero. This does *not* mean that no rearrange area is used. Rather, it is a special value, or "shorthand," that tells CMMD to use the maximum possible amount of rearrange

Copyright © 1995 Thinking Machines Corporation

area. By default, CMMD chooses a per-node value whose product with `CMMD_partition_size` is approximately equal to two gigabytes.

`fd` may be any valid global file descriptor. *`size`* must be a nonnegative integer multiple of 64, less than or equal to the current value returned by `CMMD_max_rearrange_size(`*`fd`*`)` (see the man page for `CMMD_max_rearrange_size`). All nodes must call `CMMD_set_rearrange_size` with the same arguments.

SEE ALSO

`CMMD_get_rearrange_size`
`CMMD_max_rearrange_size`

Copyright © 1995 Thinking Machines Corporation

CMMD_get_rearrange_size

Returns the size of the per-node I/O rearrange area.

SYNOPSIS

C Syntax

```
int CMMD_get_rearrange_size(int fd)
```

Fortran Syntax

```
INTEGER FUNCTION CMMD_get_rearrange_size(fd)
    INTEGER fd
```

ARGUMENT

fd	A file descriptor for a global file, intended to be used for `sync_sequential` I/O to the SDA or the DV.

RETURNS

A nonnegative integer multiple of 64, equal to the size of the per-node I/O rearrange area for the given file descriptor, in bytes.

DESCRIPTION

(See `CMMD_set_rearrange_size`.)

This function returns the last value used as a second argument to `CMMD_set_rearrange_size`, which must always be a nonnegative integer multiple of 64. If `CMMD_set_rearrange_size` has never been called, then it returns zero. `CMMD_get_rearrange_size` is asynchronous, meaning that it may be called independently on any node, at any time.

Version 3.3, April 1995
Copyright © 1995 Thinking Machines Corporation

SEE ALSO

```
CMMD_set_rearrange_size
CMMD_max_rearrange_size
```

Copyright © 1995 Thinking Machines Corporation

CMMD_max_rearrange_size

Returns the largest size allowable for the per-node rearrange area.

SYNOPSIS

C Syntax

```
int CMMD_max_rearrange_size(int fd)
```

Fortran Syntax

```
INTEGER FUNCTION CMMD_max_rearrange_size(fd)
    INTEGER fd
```

ARGUMENT

fd	A file descriptor for a global file, intended to be used for `sync_sequential` I/O to the SDA or the DV.

RETURNS

A positive integer multiple of 64, equal to the greatest number allowable as a second argument to `CMMD_set_rearrange_size`.

DESCRIPTION

(See `CMMD_set_rearrange_size`.)

The per-node rearrange area is always limited such that its product with the partition size is approximately equal to two gigabytes. `CMMD_max_rearrange_size` returns this limit. Since passing zero as a second argument to `CMMD_set_rearrange_size` is a shorthand way of telling CMMD to "use the maximum possible rearrange size," the two calls

```
CMMD_set_rearrange_size(fd, 0);
```

and

```
CMMD_set_rearrange_size(fd, CMMD_max_rearrange_size(fd));
```


Copyright © 1995 Thinking Machines Corporation

are functionally equivalent. The first form is merely a convenient shorthand for the second. *Notice that* `CMMD_max_rearrange_size` *will never return zero.*

SEE ALSO

`CMMD_set_rearrange_size`
`CMMD_gax_rearrange_size`

Copyright © 1995 Thinking Machines Corporation

10 Extending CMAML to Use Hardware Tags

This section is for programmers who mix CMNA and CMAML/CMMD (i.e., who make use of Appendix A of the *CMMD Reference Manual* for Version 3.3). This section of the release notes may be ignored by all other programmers.

10.1 Processing Messages

Some aspects of CMNA programming are different between the CM-5 and the CM-5E. In particular, *CMNA code written for the CM-5 that works in conjunction with CMMD will not work on a CM-5E*. It must be modified.

(**NOTE:** The following material also appears as Appendix A of the *CMMD Reference Manual* for Version 3.3.)

This section describes several internal CMAML functions and mechanisms. The information is provided for programmers interested in experimentation and research.

The functions and mechanisms described in this appendix allow programmers to write custom network code with C and CMNA (Connection Machine Network Accessors), or directly with SPARC assembler, yet have that code co-exist with CMMD. Specifically, CMAML programmers may allocate hardware tags and write customized network code that directly accesses the NI (Network Interface).

NOTE: CMNA is a low-level macro set used for programming the NI chip. It is *not* intended for general use. CMNA is documented in *Programming the NI*, Version 7.1 (for the CM-5) and the *NI Programmer's Handbook*, NI Version 2.2 (for the CM-5E). These manuals are available on request from Thinking Machines Corporation.

This appendix does *not* explain how to program the NI, program in CMNA, etc. The scope of this appendix is limited strictly to

- allocating and registering hardware tags
- registering hardware tag handler functions
- receiving and dispatching packets

Copyright © 1995 Thinking Machines Corporation

10.2 Hardware Tag Allocation and Registration

CMAML provides functions for *allocating* (i.e., obtaining exclusive ownership of) Data Network message tags on a node. The hardware tag registration functions help guarantee that different software modules will not simultaneously use identical tags. It is the programmer's responsibility to maintain global consistency.

Whenever a message arrives, its tag field must be tested. If the field matches a known tag, allocated to the application, then the application is free to directly eject the message from the network and process it. The application may also call a CMAML function to invoke a tag handler function previously installed by the application for this message type. If the tag field represents an unknown tag, the program must call one of a small number of packet-dispatching functions.

If a tag allocated by the application arrives when CMMD is reading packets from the network, CMMD will call the tag handler installed by the application.

10.3 Hardware Tag Handler Functions

After allocating a hardware tag, programmers are responsible for *registering* a handler function for that tag with CMMD. A tag handler function is registered for each DR FIFO. These handler functions may be any SPARC ABI-compliant function or leaf routine, provided that they at least read the entire current message from the appropriate DR FIFO. Further interpretation of the message content is left to the handler.

CMAML maintains a mask of allocated hardware tags. The arrival of a message whose tag is marked as allocated may optionally interrupt the node, and cause an interrupt handler to be invoked. Otherwise, arriving messages will pile up first in the NI FIFOs and then into the Data Network itself. When interrupts are disabled, code is required to periodically poll the Data Network for pending messages.

Copyright © 1995 Thinking Machines Corporation

10.4 CMAML Extension Functions

10.4.1 Allocating and Freeing Tags

The function

```
int CMAML_allocate_tag()
```

attempts to allocate a hardware tag for the caller. It returns a tag from 0 to 7, inclusive, if successful; it returns −1 if no tags were available.

Currently, CMOST allows eight hardware tags to be allocated by user-level software. In its current implementation, CMMD allocates the lowest 3 or 4 of these tags for its own internal use before invoking the user's application; it will allocate one further tag the first time the application performs parallel I/O.

The function

```
int CMAML_allocate_this_tag(int i)
```

attempts to allocate the specified hardware tag. Returns `TRUE` if the tag was previously unallocated, and `FALSE` if the tag was already allocated.

The function

```
int CMAML_free_tag(int tag)
```

returns `TRUE` if the tag was successfully freed, `FALSE` if the tag was already free.

Once an application frees a tag, it cannot use that tag again until it reallocates the tag. Attempting to use a freed tag will produce unpredictable results.

10.4.2 Registering Hardware Tag Handlers

```
int CMAML_set_ldr_handler(int tag, void *handler)
int CMAML_set_rdr_handler(int tag, void *handler)
```

These functions register the specified hardware tag handler functions for the LDR and RDR FIFOs, respectively, for the specified tag. (The LDR and the RDR are the two "halves" of the Data Network. They are used selectively by CMAML, as explained in Section 12.2.1 of the *CMMD Reference Manual* for Version 3.3.

Copyright © 1995 Thinking Machines Corporation

No protection whatsoever is provided to prevent users from overwriting any of the default CMMD tag handlers. After this call, any arriving LDR (or RDR) packets with the specified tag encountered by CMMD communiction functions will automatically call the specified external handler function.

PLEASE NOTE

On the CM-5, the *handler* argument to `CMAML_set_[l|r]dr_handler` must point to a function containing *no more than 64 SPARC assembly instructions.* If you are writing your handlers directly in assembly code, then you can easily count the number of instructions to ensure that they do not exceed this limit. If your handlers are compiled, then you should disassemble their code and count the instructions. If the length of your handler exceeds 64 instructions, then there is a simple workaround: simply install a "stub" that does nothing but call your handler. (Such a stub will not contain more than a handful of instructions.)

Failure to observe this restriction will result in a fatal error.

Note that on the CM-5E there is no such limit. Hardware tag handlers can be arbitrarily long.

10.4.3 Processing Messages

Some aspects of CMNA programming are different between the CM-5 and the CM-5E. In particular, *CMNA code (that works in conjunction with CMMD) written for the CM-5 will not work on a CM-5E. It must be modified to use the special CM-5E functions described below*.

The relevant CMAML functions are

(for the CM-5):

```
void CMAML_get_ldr(int ni_base_addr, int ldr_fifo_status)
void CMAML_get_rdr(int ni_base_addr, int rdr_fifo_status)
```

(for the CM-5E):

```
void CMAML_get_ldr_first(int ni_base_addr,
                         int ldr_fifo_status_all,
                         int first_word)
```


Copyright © 1995 Thinking Machines Corporation

```
void CMAML_get_rdr_first(int ni_base_addr,
                         int rdr_fifo_status_all,
                         int first_word)
```

Once a tag has been allocated and its handler set, the `CMAML_get_[l|r]dr[_first]` functions may be called to process any message arriving via the specified Data Network interface.

The functions `CMAML_get_[l|r]dr` must be used on the CM-5 only. A fatal error will occur if they are used on a CM-5E. The functions assume that

- `ni_base_addr` contains the NI base address.
- `[l|r]dr_fifo_status` contains the current *NI-[L/R]DR_status* word, as obtained by calling `CMNA_[l|r]dr_status`. (You may bypass CMNA if you wish, and write directly in assembly code; however, this requires a detailed knowledge of the CMNA header files.)
- the `RECEIVE_OK` bit in the status word is set, i.e., there *is* a waiting message.

The functions `CMAML_get_[l|r]dr_first` must be used on the CM-5E only. A fatal error will occur if they are used on a CM-5. These functions assume that

- `ni_base_addr` contains the NI base address.
- `[l|r]dr_fifo_status_all` contains the current *NI-[L/R]DR-status-all* word, as obtained (for example) by calling `CMNA_mni_[l|r]dr_status_all_pop`. (You may bypass CMNA if you wish, and write directly in assembly code; however, this requires a detailed knowledge of the CMNA header files.) The *status-all* word is new to the CM-5E. It is a slightly different implementation of the CM-5's *status* word. The suffix "pop" means that if there is a message waiting, the function "pops" it from the NI. *You need not, and **must** not, ever pop the first word of a message by yourself. Rather, you should rely on the fact that on the CM-5E, the first word of every message is automatically popped as part and parcel of reading the* status-all *word.*
- The `RECEIVE_OK` bit in the status-all word is set, i.e., there *is* a waiting message. On the CM-5E there is no ready-made macro analogous to the `RECEIVE_OK` that exists for the CM-5. On the CM-5E, the

Version 3.3, April 1995
Copyright © 1995 Thinking Machines Corporation

RECEIVE_OK bit happens to be the least significant bit, so you should include the following definition in your code:

```
#define RECEIVE_OK_CM5E(status_all) ((status_all)&0x1)
```

and use RECEIVE_OK_CM5E on the CM-5E wherever you would use RECEIVE_OK on the CM-5.

- The *first_word* argument contains the first word of the message, if there is a waiting message. Note that if there is a waiting message, the function CMNA_mni_[l|r]dr_status_all_pop will already have read it from the NI, at the same time that it read the status-all word. (This is one of the ways in which the CM-5E is more efficient than the CM-5.)

The CMNA functions CMNA_mni_[l|r]dr_status_all_pop return a double-precision floating-point value. The number is meaningless as a floating-point quantity. The first 32 bits of it represent the status-all word, and the second 32 bits represent the first word of the waiting message. Therefore, the standard idiom for writing code to handle CMAML packets in CMNA looks like this:

```
{
  int statall;
  double statall_and_first;

  statall_and_first = CMNA_mni_ldr_status_all_pop();
  statall = ((int *)&statall_and_first)[0];
  if ( RECEIVE_OK_CM5E( statall ) )
    CMAML_get_ldr_first(NI_BASE, statall,
                        ((int *)&statall_and_first)[1]);
      ...
      ...
      ...
}
```


Copyright © 1995 Thinking Machines Corporation

which is slightly more complicated than the corresponding idiom for the CM-5, which looks like this:

```
{
  int stat;

  stat = CMNA_ldr_status();
  if ( RECEIVE_OK( stat ) )
    CMAML_get_ldr(NI_BASE, stat);
      ...
      ...
      ...
}
```

What follows is a typical CMNA code fragment, written in the two different styles.

First, for the CM-5:

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

stat = CMNA_ldr_status();

if ( RECEIVE_OK( stat ) ) {

  if( USER_TAG( stat ) == my_tag ) {

    w1   = CMNA_ldr_receive();
    dw23 = CMNA_ldr_receive_double();
    dw45 = CMNA_ldr_receive_double();

    <process data accordingly>

    } else { /* not my tag */
      CMAML_get_ldr(NI_BASE, stat);
  }
}
```


Copyright © 1995 Thinking Machines Corporation

Next, for the CM-5E (notice that in order to access the hardware tag, you must introduce the macro `USER_TAG_CM5E`, which does not currently exist in CMNA):

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

#define USER_TAG_CM5E (statall) (((statall) & \
                        CMAML_NI_HWTAG_MASK_ALL)>>3)
  ...
  ...
  ...
statall_and_first =
  CMNA_mni_ldr_status_all_pop();
statall = ((int *)&statall_and_first)[0];
w1      = ((int *)&statall_and_first)[1];

if ( RECEIVE_OK_CM5E( statall ) ) {

  if( USER_TAG_CM5E( statall ) == my_tag ) {

     dw23  = CMNA_ldr_receive_double();
     dw45  = CMNA_ldr_receive_double();

     <process data accordingly>

  } else {  /* not my tag */
    CMAML_get_ldr_first(NI_BASE, statall, w1);
  }
}
```

These examples show that you do not need to call the `CMAML_get_[l|r]dr[_first]` functions to process your own packets (i.e., packets that you recognize), but that you *must* do so for packets that you do not recognize (i.e., packets without your private hardware tags). Of course, you could just as well install your handlers using `CMAML_set_[l|r]dr_handler`, and process them using `CMAML_get_[l|r]dr[_first]`.

10.4.4 Using Hardware Tag Handlers

Note that hardware tag handlers on Cypress platforms must be no longer than 64 machine instructions. If this restriction is violated, the results are unpredictable and most likely will result in a segmentation fault or a bus error. If you wish to register a hardware tag handler that is longer than 64

Copyright © 1995 Thinking Machines Corporation

instructions, you can register a short stub handler that simply calls the desired handler. This restriction does not apply on SPARC superscalar platforms such as the CM-5E.

Also note that the interface to hardware tag handlers is subject to change without notice from one release to another. Programmers coding at this level should be prepared to modify their code in future releases.

Version 3.3, April 1995
Copyright © 1995 Thinking Machines Corporation