Chapter 12

# CMAML: Active Messages

This chapter describes the user-visible functions within the CMMD active message layer (CMAML). CMAML is the protocol-less transport layer upon which the higher-level CMMD functions are built. You can use these functions to perform specialized low-level communication and to define custom network protocols.

CMAML is an internal layer of the CMMD software. It is subject to change in future releases. CMAML is made available for advanced programmers and library developers who may require functionality or performance not otherwise provided by the higher-level CMMD functions.

Active messages were developed by von Eicken et al. at UC Berkeley.* CMAML represents an independent implementation of the same ideas. Berkeley CMAM and the CMAML functions presented here are not interchangeable. Their interfaces, functionality, and performance characteristics differ.

## 12.1 CMAML

CMAML provides transport functions for three types of message transfer:

- *Short active messages* consist of one *packet* (see Appendix B) five words in length. Their purpose is to invoke a function upon the receiving node and to supply arguments to be passed to the function upon invocation.

* von Eicken, Thorsten, David E. Culler, Seth Copen Goldstein, and Klaus Eric Schauser, "Active Messages: A Mechanism for Integrated Communication and Computation," *Proceedings of the 19th International Symposium on Computer Architecture* (Gold Coast, Australia: ACM Press, May 1992).


*Copyright © 1995 Thinking Machines Corporation*

- *Block data transfers*, as their name implies, transfer a block of data from one or more source nodes to a single receiving node. The transfer may move data from serial memory to serial memory or from parallel memory to parallel memory. When the transport operation completes, the source and/or destination node may invoke a callback function. (See Chapter 14.)

- *Long active messages* consist of one packet, between 1 and 18 words in length. They can be used both to invoke handler functions on the receiving node and to transport small amounts of data in a rapid manner, without using the CMMD protocol. *Long active messages can be used on the CM-5E only*.

In support of these functions, CMAML also provides functions that allow polling of the networks, manipulating interrupt state, and preserving register contents.

CMAML functions require some understanding and care in their use. Given such understanding and care, however, these functions enable users to construct a wide variety of additional communication and memory exchange functions, such as remote memory store and fetch, send-to-queue, store-with-op, and compare-and-swap.

**PLEASE NOTE**

1. As with most transport layers, CMAML functions assume that some higher layer of software is providing any protocol needed, and that (for example) receiving nodes know what to do with any data sent to them. Users must ensure that their applications provide such a protocol.

2. The request functions (`CMAML_request_xxx` and `CMAML_request_xxx_host`) are the only CMAML functions that can be called with interrupts enabled. All other CMAML functions require that interrupts be disabled and polling used instead.

3. The CMMD interrupt handler saves only the SPARC IU registers, not the SPARC FP or VU data/control registers. If your handler or callback function alters


Version 3.3, April 1995
Copyright © 1995 Thinking Machines Corporation

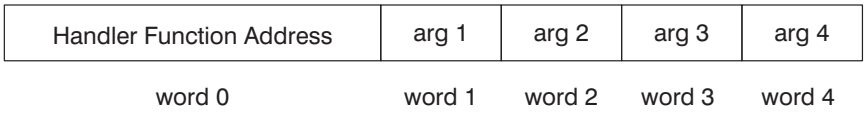
the state of these registers, you may be required to save and restore their state yourself. (See Sections 14.1.1 and 14.2.)

This chapter discusses short and long active messages. Chapter 13 discusses block data transfers, and Chapter 14 discusses handler and callback functions.

## 12.2 Active Messages

A short active message is a single Data Network packet consisting of five words. The first word (word 0) is the address of a *handler function* on the destination node. The remaining *data words* are arguments to be passed to the function. (SPARC ABI calling conventions apply.)

The CMAML implementation for short active messages on the CM-5 and CM-5E uses a five-word format:




Long active messages, available for use on the CM-5E only, are similar but can contain from 1 to 18 words. Again, word 0 contains the address of the handler function on the receiving node, while the subsequent data words are to be used by the handler function.

Receipt of an active message automatically invokes the handler function. The interpretation of the data words is under the control of the handler function. When the handler function returns, the node either returns to whichever function was polling the network or resumes its interrupted activity. (See Figure 17.)


Copyright © 1995 Thinking Machines Corporation

Active messages are never buffered on arrival, except to the small extent to which the Data Network itself natually buffers message traffic. When an active message is received (whether by polling or by interrupt), its handler is invoked immediately. Figure 17 illustrates the typical sequence of activities performed by a program that uses active messages.

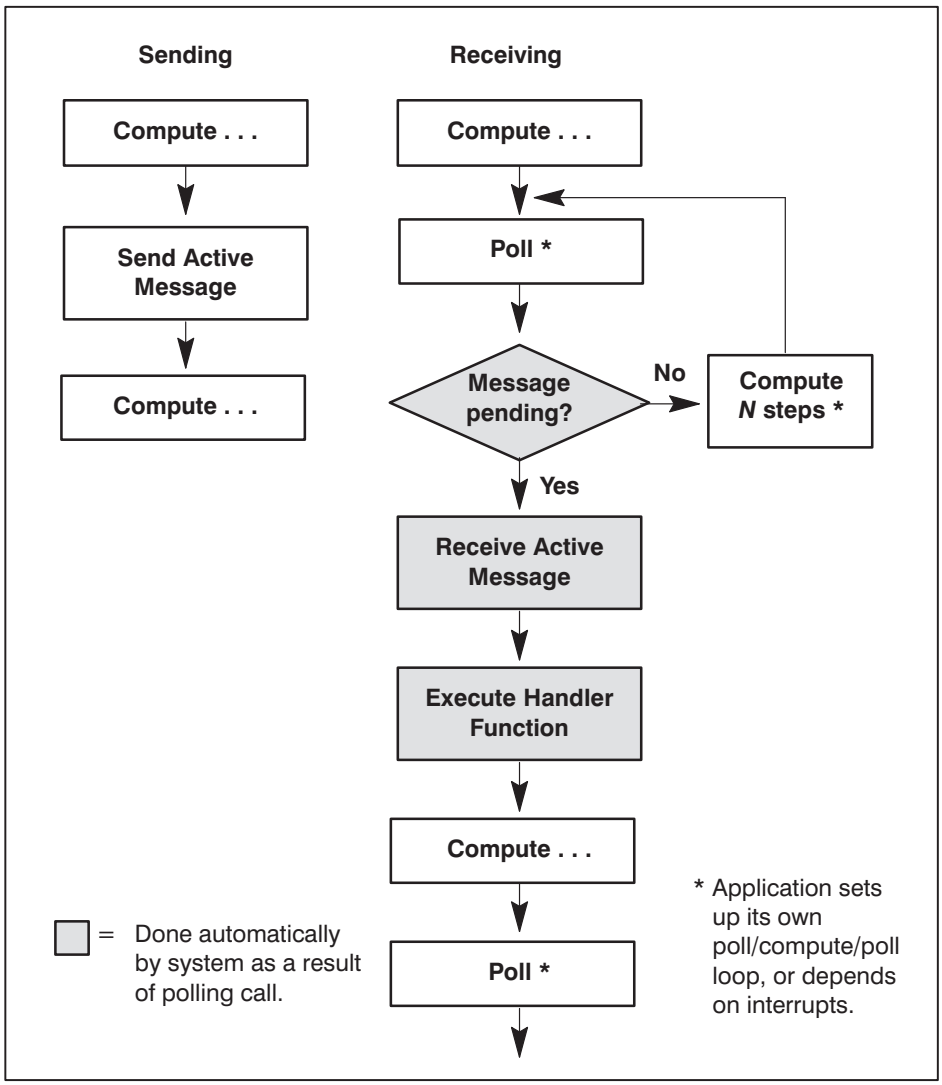



**Figure 17. Active Messages.**



Copyright © 1995 Thinking Machines Corporation

### 12.2.1 Requests, Replies, and RPCs

Three types of active messages exist: Requests, Replies, and Remote Procedure Calls (RPCs). They differ in their use of the Data Network and in the context in which they can be used.

**NOTE:** The Data Network has two independent interfaces. They are sometimes called the "left" and "right" interfaces. Here they are called the "Request" and "Reply" interfaces.

A *Request* message:

- can only be sent from the main thread of control, never from a handler.
- may be sent with interrupts enabled.
- cannot have a handler that calls any type of active message except a Reply.
- always blocks until complete.
- uses the request interface to the Data Network to transmit data.
- interleaves attempts to send with attempts to receive data until the transmission succeeds. (This helps prevent the network from getting clogged because of only being "filled" and never "emptied.") When attempting to receive, it polls both interfaces alternately.

A *Reply* message:

- can be sent only from handler functions, never from the main thread of control.
- must be sent with interrupts disabled.
- cannot call *any* communication functions from within its handler.
- always blocks until complete.
- uses the reply interface to the Data Network to transmit data. Alternates attempts to send with attempts to receive; polls only the reply interface when attempting to receive.

A *Remote Procedure Call (RPC)* message:

- can be sent either from a node's main thread of control or from a handler function.


Copyright © 1995 Thinking Machines Corporation

- must be sent with interrupts disabled.
- blocks except when called from an active message handler, in which case it is queued for later execution.
- alternates send requests between the request and reply interfaces; alternates polling attempts on both interfaces.
- may have a handler function that calls other RPCs, Replies, or any other *nonblocking* communication function.

Request and Reply messages are often used for purposes such as implementing node-to-node memory fetch operations, where one node sends a Request for data, and the other returns the data as its Reply. In fact, it is common for the handler function of a Request message to send a Reply immediately, either to the node that sent the Request or to another node.

RPC messages are most commonly used to send active messages in cases where the handler function does not reply, or causes only an easily manageable amount of communication activity by its reply.

For examples of these functions at work, see the sample programs in Section 12.5.

**PLEASE NOTE**

Since the main point of CMAML is speed, it does very little error checking. In particular, most of the rules listed above are *not* enforced explicitly by CMAML. Therefore, generally speaking, failure to observe them will not produce a clear error message, but will rather produce bugs that are very subtle and difficult to fix.

### 12.2.2 Processing the Data Stream

In a typical application, active messages rarely arrive in isolation, but rather in the middle of some stream of data.


Version 3.3, April 1995
Copyright © 1995 Thinking Machines Corporation

Referring to the sample data stream from Appendix B (and reproduced here, for convenience):

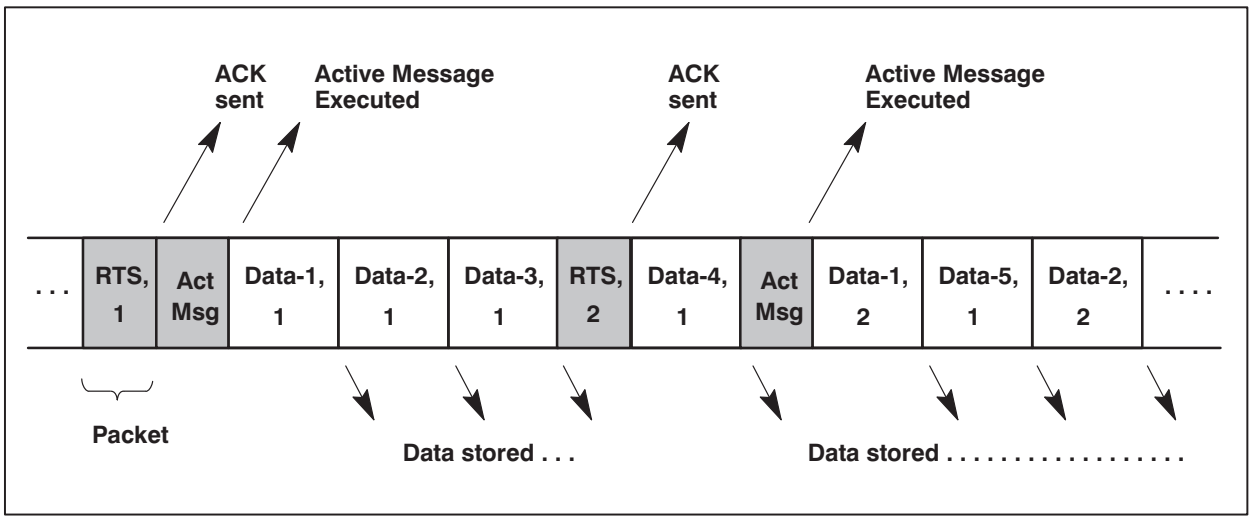



**Figure 18. A sample data stream.**

Recall that, once the CMMD/CMAML internal polling routine is invoked, it proceeds to receive and process incoming packets until the node's network interface is empty.

Notice the different actions the receiving node takes while dealing with a data stream:

- When it receives a data packet, it deposits it immediately into memory.
- When it receives an active message, it immediately invokes its handler function.

  If a handler function invokes a communication function (that is, if it attempts to send data into the network), the communication may either occur immediately or be queued for later execution, when the polling function has no more packets to handle.
- If a data packet terminates an asynchronous send or receive, the node may invoke a callback function (see Chapter 14). It may do so immediately, or it may queue the callback function for later execution, when the polling function finds no more packets in the network.


Copyright © 1995 Thinking Machines Corporation

If the callback function itself invokes a communication function, that function may similarly execute immediately or be queued for later execution.

These issues are discussed in more detail in Chapter 14. The reader is encouraged to read that chapter.

The order in which events occur depends partly on the order in which packets occur in the data stream, and partly as determined by the CMAML priority system. These priorities determine when a node should pause in its basic task of pulling data from the network in order to invoke and execute some new (callback or handler) function. In particular, they determine when the node can pause in its task of receiving data in order to send some new data into the network. The priorities are as follows:

| | |
|---|---|
| AM | Active message Request and Reply functions (highest priority) |
| RPC | Active message RPC functions |
| SEND | Scopies and pcopies |
| CALLBACK | Callback functions |
| USER | User code (lowest priority) |

When a CMAML function executes, it does whatever polling it does with the *current priority* set to its own value. It can be suspended by a function with a higher priority, but not by a function with lower priority.

Assume, for instance, that an RPC active message is being sent. The sending function (`CMAML_rpc`) sets the current priority to RPC, and begins sending (and receiving) through the network. If an active message is received during this process, and if its handler tries to send a new RPC message, that message cannot take priority over the original RPC message. Instead, it must be queued until the original RPC call succeeds in sending its message. On the other hand, if the handler tries to send a Reply message, the Reply message is sent immediately, because it has higher priority than the RPC.


Version 3.3, April 1995
Copyright © 1995 Thinking Machines Corporation

### 12.2.3 Why Are There Different Active Message Types?

The reason for multiple types of active messages, together with stringent restrictions as to the context in which they can be used and the network interfaces they affect, has to do with the mechanics of message transmission and retrieval. Let's look at this issue in somewhat more detail. (Note that this topic is discussed in still more detail in the paper by von Eicken et al., but we reproduce enough of their reasoning to motivate the Request/Reply/RPC triad.)

When the handler function of an active message sends another active message, the process of attempting to send the message causes either or both of the network interfaces to be checked for incoming messages. If yet another active message is received at this point, its handler function is invoked immediately. If nothing were done to restrict this chain of automatic invocations of handler functions, then sending a single active message could lead to a potentially unbounded series of uncompleted attempts to send a message, resulting eventually in stack overflow.

To prevent this worst-case scenario from occurring, CMAML imposes restrictions on Requests and Replies. A Request message can be sent only from a node's main thread of control, never from within a Reply handler, while a Reply message can be sent only from within a Request handler. A Reply never checks the Request interface and can never call any communications functions. Thus, sending a Reply message can cause at most one other Reply message to be read and invoked. This ensures that Request/Reply messaging does not degenerate into the unbounded case described above.

RPCs are a sort of general "workhorse" type of active message, which has no such automatic protection. RPCs should be used in cases where the handler function either does not reply or causes at most a tolerable (or predictable) number of uncompleted transactions, where *tolerable* is defined by the application. For example, the number of transactions might be under the control of the application or its user; thus, the amount of handler recursion would be a parameter that could be set to a workable value.

Note, however, that RPC recursion is internally represented as the queuing of RPC calling sequences on a compact, CMAML-managed queue, and *not* on the process stack. This not only saves space, but greatly facilitates debugging.


Copyright © 1995 Thinking Machines Corporation

## 12.3 Long Active Messages

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.

- `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


Version 3.3, April 1995
Copyright © 1995 Thinking Machines Corporation

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`

### 12.3.1 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.

### 12.3.2 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 functions 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


Copyright © 1995 Thinking Machines Corporation

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.*

### 12.3.3 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 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.



Copyright © 1995 Thinking Machines Corporation

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

## 12.4 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
```


Version 3.3, April 1995
Copyright © 1995 Thinking Machines Corporation

# Short Active Messages

These six functions send short (5-word) active messages on the CM-5 and CM-5E. Keep in mind that "5 words" means the total packet size, i.e., one word for the handler plus four words of data.

---

### SYNOPSIS

#### C Syntax

```
void CMAML_request
void CMAML_reply
void CMAML_rpc
    (int dest_node, void (*handler)(),
     int data1, int data2, int data3, int data4)

void CMAML_request_tohost
void CMAML_reply_tohost
void CMAML_rpc_tohost
    (void (*handler)(),
     int data1, int data2, int data3, int data4)
```

#### Fortran Syntax

```
SUBROUTINE CMAML_request
SUBROUTINE CMAML_reply
SUBROUTINE CMAML_rpc
          (dest_node, handler, data1, data2, data3, data4)
    INTEGER dest_node
    EXTERNAL handler
    INTEGER data1, data2, data3, data4

SUBROUTINE CMAML_request_tohost
SUBROUTINE CMAML_reply_tohost
SUBROUTINE CMAML_rpc_tohost
          (handler, data1, data2, data3, data4)
    EXTERNAL handler
    INTEGER data1, data2, data3, data4
```

---


*Copyright © 1995 Thinking Machines Corporation*

### ARGUMENTS

| | |
|---|---|
| *dest_node* | An integer $0 \leq n <$ *partition_size* specifying the destination node for the message. |
| *handler* | Address of a handler function on the destination node. |
| *data1, data2, data3, data4* | Handler function arguments. |

### DESCRIPTION

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. However, each function sends a different type of message:

`CMAML_request` sends a Request active message.
`CMAML_reply` sends a Reply active message.
`CMAML_rpc` sends an RPC active message.

`CMAML_request_tohost` sends a Request active message to the host.
`CMAML_reply_tohost` sends a Reply active message to the host.
`CMAML_rpc_tohost` sends an RPC active message to the host.

Note that the `_tohost` functions do not take a *dest_node* argument. The destination node (the host) is implicit in the name.

#### Requests, Replies, and RPCs

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

Request function:

- may be called only from top-level 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


*Version 3.3, April 1995*
*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. (Neither short nor long `_tohost` functions should ever be called in a hostless program.)


*Copyright © 1995 Thinking Machines Corporation*

**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*

# 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]
```

---

### 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 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


Version 3.3, April 1995
Copyright © 1995 Thinking Machines Corporation

`CMMD_send_block` and `CMMD_receive_block`), and *before* sending any `_tohost` active messages.

---


*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. |


Version 3.3, April 1995
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.


*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


Version 3.3, April 1995
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


*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.

---


*Version 3.3, April 1995*
*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)
```


*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
```


*Version 3.3, April 1995*
*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.<br><br>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


*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.


*Version 3.3, April 1995*
*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


*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.


*Version 3.3, April 1995*
*Copyright © 1995 Thinking Machines Corporation*

### 12.4.1 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`<br>`CMAML_popd` | Remove the data waiting in the network and store it at a specified address |
| `CMAML_poptofirst`<br>`CMAML_poptofirstd`<br>`CMAML_poptofirst_ra`<br>`CMAML_poptofirst_ra_d`<br>`CMAML_poptofirst_rla`<br>`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`<br>`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**


Copyright © 1995 Thinking Machines Corporation

**a bus error, for lack of sufficient data. Such delayed bus errors are difficult to debug.**

Reference pages for these functions follow.


Version 3.3, April 1995
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*

### 12.4.2 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

## 12.5 Examples

### 12.5.1 Example: Sending an RPC Message

RPC messages are useful for performing simple exchanges of data between nodes. For example, here is one way to use RPCs to perform store operations: sending an RPC message that triggers a store handler function on the destination node. (In this case, a single node uses RPCs to broadcast values to all other nodes in the partition.)

```
#include <cm/cmmd.h>
void store_handler(address, value)
  int *address;
  int value;
{
  *address = value;
}

void simple_flag_example(interrupts)
  int interrupts;
{
  int flag = -1;
  int self = CMMD_self_address();
  int dest_node = ((self + 1) % CMMD_partition_size());

  CMAML_disable_interrupts();
  CMAML_rpc(dest_node, store_handler, &flag, self);

  if(interrupts == 0) {
    CMAML_poll_while_lt(&flag, 0);
  } else {
    CMAML_enable_interrupts();
    while(* (int *)&flag < 0 )
    {}
  }

  CMMD_fset_io_mode(stdout, CMMD_independent);
  printf("PN=%3d: flag was -1 and is now %3d\n",
         self, flag);
  CMMD_fset_io_mode(stdout, CMMD_sync_bc);
}
```




Copyright © 1995 Thinking Machines Corporation

```
void main()
{
  printf("With interrupts on...\n");
  simple_flag_example(1);
  printf("With interrupts off...\n");
  simple_flag_example(0);
}

void example(intr)
  int intr;
{
  int value = -1;
  int self = CMMD_self_address();
  int dest_node = (self + 1) % CMMD_partition_size();

  CMAML_disable_interrupts();
  CMAML_rpc(dest_node, store_handler, &value, self);

  if(intr == 0) {
    CMAML_poll_while_lt(&value, 0);
  } else {
    CMAML_enable_interrupts();
    while(* (volatile int *) &value < 0);
  }

  printf("PN=%d: value was -1 and is now %d\n",
         self, value);
}
```


Copyright © 1995 Thinking Machines Corporation

### 12.5.2 Example: Sending Request/Reply Messages

Request/Reply messages are useful for exchanges of data between nodes that require a quick response from the destination node. For example, here is one way to use Request/Reply messages to perform a fetch operation: sending a Request for data that triggers a Reply containing the requested data:

```
#include <cm/cmmd.h>
int done_flag ={ 0 }; /* Same address on all nodes */
void reply_handler(address, value)
  int *address, value;
{
  done_flag++;
  *address = value;
}

void request_handler(srcaddr, dstaddr, pn)
  int *srcaddr, *dstaddr, pn;
{
  done_flag++;
  CMAML_reply(pn, reply_handler, dstaddr, *srcaddr);
}


main()
{
  int self = CMMD_self_address();
  CMMD_fset_io_mode(stdout, CMMD_independent);

  if(self == 0) printf("With interrupts on...\n");
  CMMD_sync_with_nodes();
  example(1);

  if(self == 0) printf("With interrupts off...\n");
  CMMD_sync_with_nodes();
  example(0);
}


example(intr)
  int intr;
{
```


Version 3.3, April 1995
Copyright © 1995 Thinking Machines Corporation

```
  int value = -1;
  int self = CMMD_self_address();
  int dest_node = (self + 1) % CMMD_partition_size();

  done_flag = 0;

  CMAML_disable_interrupts();

  CMAML_request(dest_node, request_handler, &self,
                &value, self);

  if(intr == 0) {
    /* Must wait until _all_ communication is done,

       that is, until done_flag == 2 */
    CMAML_poll_while_lt(&done_flag, 2);

  } else {
    CMAML_enable_interrupts();
    while(* (int *) &value == -1 || done_flag < 2);
  }

  printf("PN=%d: value was -1 and is now %d\n",
         self, value);
}
```


Copyright © 1995 Thinking Machines Corporation