Chapter 11

The Active Message Layer

This chapter describes 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 define custom network protocols and to perform specialized low-level communication.

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.

11.1 CMAML

CMAML provides transport functions for two types of message transfer:

- *Active messages* are one packet long. 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 Cost, Australia: ACM Press, May 1992.

- *Block data transfers*, as their name implies, transfer a single 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 an attached handler function.

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

Users familiar with the CMAML machinery will find it easy to construct a 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.

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 protocol.

11.2 Network Interactions

Programming "at the CMAML level" involves direct interactions with the CM-5's Network Interface. Programming at this level, therefore, requires some understanding of basic Data Network behavior.

Before presenting any of the CMAML functions, this chapter provides some introduction to the mechanisms employed in message transmission, especially those regarding interrupts and polling.

11.2.1 Receiving an Interrupt

Consider the normal *flow of control* in an application. (Figure 11 sketches a very simple example.)

Recall that, in a CMMD program, each node runs an identical copy of the application. The flow of control on each node thus provides the *main thread of control* for the application on that node.

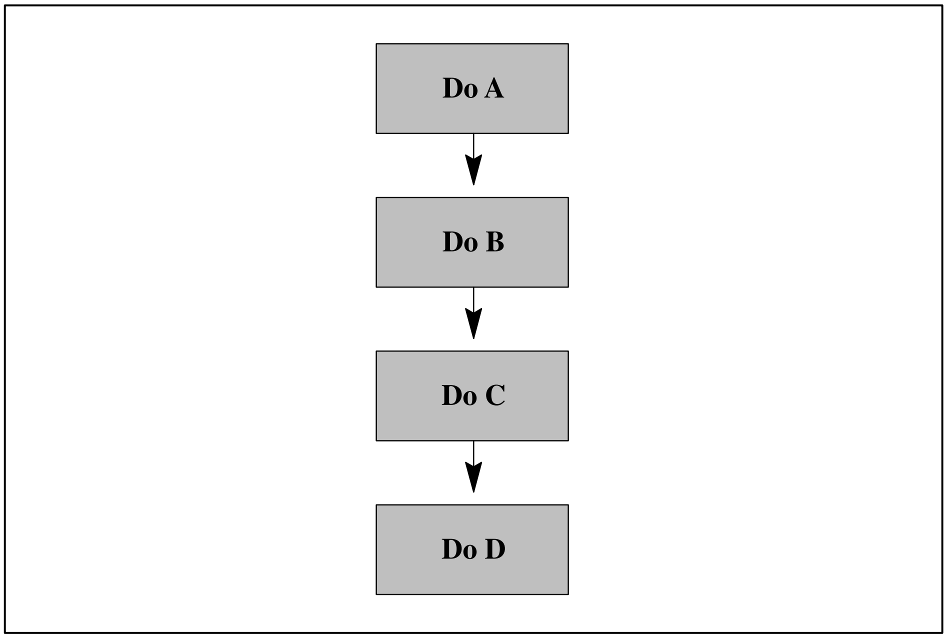

Figure 11. Main Thread of Control in a User Program

What happens to this thread of control when an asynchronous message is to be received? If interrupts are disabled, nothing happens until the program itself polls for messages. (See Figure 12.)

Note that a program makes a single function call to begin polling, and then automatically continues polling for as long as there are messages to be received.

If interrupts are enabled, however, the arrival of a message packet generates an interrupt (see Figure 13), via a call to the *CMMD interrupt handler.*

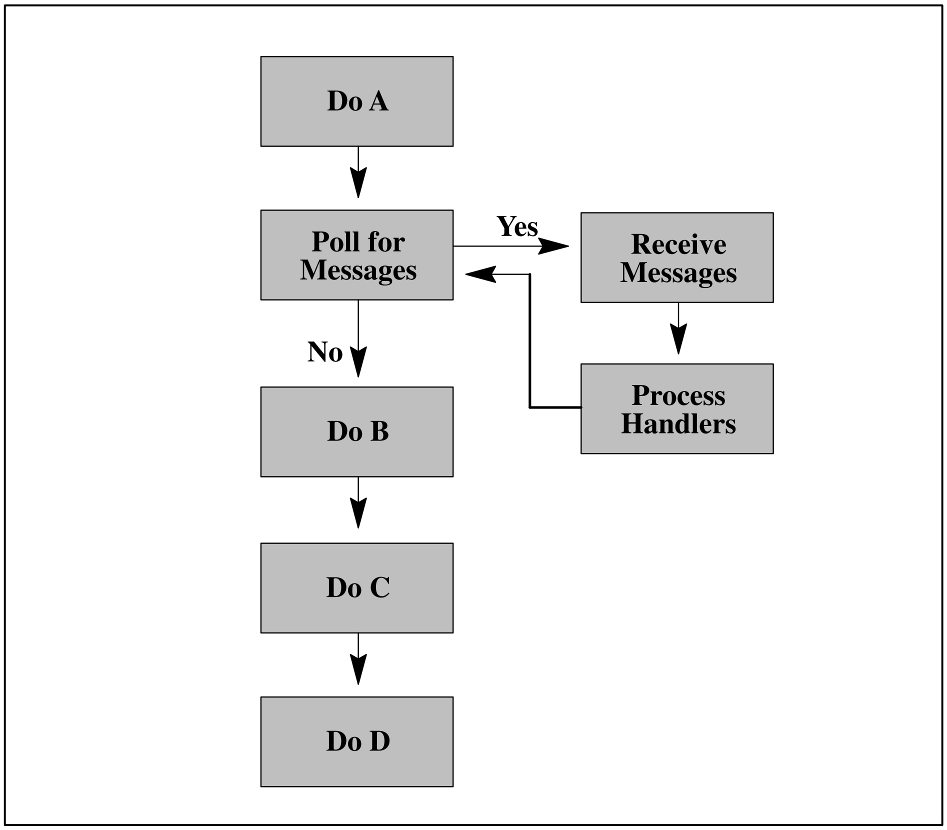

Figure 12. Flow of Control When You Poll for Messages

The interrupt handler starts its work by

- suspending the main thread of control
- disabling interrupts
- saving the state of the SPARC's IU (instruction unit) registers
- calling a polling function to handle the incoming messages

The polling function called by the CMMD interrupt handler is the same function called by user polling requests. The polling process, therefore, is identical for messages received via both polling and interrupts.

Note: The CMMD interrupt handler saves only the SPARC IU registers, not the SPARC FP or VU data/control registers. If your handler alters the state of these registers, you may be required to save and restore their state yourself. (See Sections 11.3.1 and 11.6.5.)

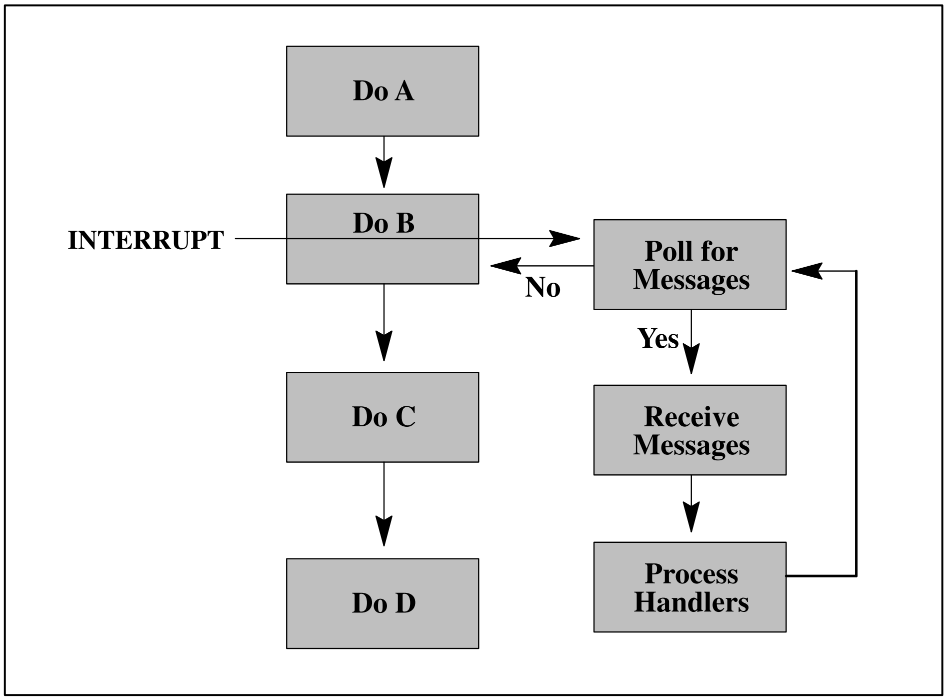

Figure 13. Flow of Control When Message Interrupts Are Enabled

11.2.2 Handling Incoming Messages

Once the internal polling routine is invoked, it proceeds to receive and process incoming packets until the node's network interface is empty.

Any data packets received are deposited into memory. If they terminate a transaction, data packets may also cause a handler function to be invoked. On the sending node, the handler is invoked when the last packet of the message has been injected into the Data Network. (The packet may not yet have been received at the destination node.) Handlers on the receiving node are not invoked until all packets have been received and processed.

Receipt of an Active Message always causes a handler to be invoked immediately. All handlers execute before control returns from the polling function to the application's main thread of control.

11.2.3 Returning Control

When all messages have been received, and all handlers executed, the polling function returns. If it was called by the CMMD interrupt handler, the interrupt handler then

- restores the state it saved for the interrupted computation
- re-enables interrupts
- exits

11.2.4 Dynamic Scope of Control

We have now sketched out a system in which an application, on a given node, is executing some computation. The computation pauses to allow receipt of one or more messages; the messages themselves may call one or more handlers.

Each of these handlers executes within a particular *dynamic scope of control*. If the application itself polled for the handler-containing message, then the handler executes within the application's dynamic scope of control. If the CMMD interrupt handler was involved, then the user-written handler executes within the CMMD interrupt handler's dynamic scope, while the CMMD interrupt handler itself executes within the application's scope.

11.3 Handler Functions

A *handler* is simply a user-defined function that can be invoked in any one of a number of situations:

- upon receipt of an an Active Message
- upon receipt of all expected data on a receive port
- upon completion of **`CMAML_scopy`** or **`CMAML_pcopy`** operation
- upon completion of a **`CMMD_send_async`** or **`CMMD_receive_async`** operation

Note that handlers are, by definition, executed upon completion of some event, not upon attaining a given line of code in a program. They thus execute asynchronously to the application's main thread of control (also called its *background* *computation*. In fact, they may cause intermissions in the background computation whenever the application (either by polling or by enabling interrupts) allows it. Handlers, therefore, must be written so as not to interfere with the state of the background computation. The user is reponsible for the correct interaction between the background computation, communication, and handlers.

11.3.1 Properties and Restrictions of Handlers

Note, again, that handlers are defined by the context in which they are called, not by their content. One can write a routine that is sometimes called as a subroutine by the main thread of control, sometimes called as a handler. However, because of their potential ability to interrupt other code, any function intended for use as a handler has the following properties and is constrained by the following restrictions:

- Handlers are always invoked in an environment in which interrupts are disabled. (Either the application has disabled interrupts, or the CMMD interrupt handler has disabled them.)
- Handlers themselves may never enable or disable interrupts.
- Handlers must be careful not to corrupt static or global data structures. (For example, mallocs can be called only if interrupts are always disabled around them, in the main thread of control as well as in the handlers.)
- Handlers cannot contain any calls to the CM Run-Time System. (The RTS is *not* re-entrant.) For this reason, handlers cannot transmit or manipulate parallel arrays, nor can they contain any CM Fortran or C* statements that do so.
- Handlers must obey SPARC ABI calling conventions.
- Handlers may call only those communication functions that do not block while waiting for their completion.
- No CMMD I/O functions may be called in handlers.
- If called from within the dynamic scope of the CMMD interrupt handler, handlers must be careful to save and restore any SPARC FP and VU state that they have modified. (Recall that the interrupt handler itself saves and restores only the state of the SPARC integer registers.)

CMAML provides functions for saving and restoring the states of the SPARC FP registers, VU data and control registers, and VU instruction extension registers.

- Any function called by a handler must obey the same restrictions that the handler itself must obey. (Know the side-effects of your functions — it is *much* better to be safe than to be sorry.)

Note also the following two restrictions on the use of handlers:

- Users should never assume that handlers will execute in any given order.
- Users should never assume that a communication function called from within a handler will have completed by the time the handler returns.

11.4 Active Messages

An active message is a single network packet comprising a series of words. The first word (word 0) is the address of a handler function on the destination node. The remaining words are arguments to be passed to the function. (Standard SPARC ABI calling conventions apply.)

The current CMAML implementation uses a five-word format:

Function Address	**arg 1**	**arg 2**	**arg 3**	**arg 4**
word 0	**word 1**	**word 2**	**word 3**	**word 4**

Receipt of an active message automatically invokes the handler function with the specified arguments. The interpretation of the arguments is strictly under the control of the handler function. When the handler function returns, the node either resumes its polling for messages or (if it was interrupted to receive the active message) continues the execution of its interrupted computation.

Active messages are never buffered on arrival. When an active message is received (whether by polling or by interrupt), its handler is invoked immediately.

11.4.1 Requests, Replies, and RPCs

Three types of active messages exist: requests, replies, and RPCs (remote procedure calls). 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.
- May send only Reply messages from within its handler.
- Always blocks until complete.
- Uses the request interface to the Data Network to transmit data.
- Until the transmission succeeds, it alternates attempts to send with attempts to receive data. When attempting to receive, it polls both interfaces alternately.

A *Reply* message:

- Can only be sent 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.
- Must be sent with interrupts disabled.
- Blocks if called from the main thread of control; does not (and must not) block if called from a handler.

- If called as a blocking function, alternates send requests between the request and reply interfaces, alternates polling attempts on both interfaces, and alternates sending and polling attempts until the packet is sent.
- Can call other RPCs, Replies, or any other non-blocking 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 functions do not reply, or cause only an easily managable amount of communication activity by their reply.

For examples of these functions at work, see the sample programs at the end of this chapter.

11.4.2 Receiving Active Messages

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.

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 immediately invoked. If nothing were done to restrict this chain of automatic invocations of handler functions, sending a single active message could lead to a potentially unbounded series of uncompleted attempts to send a message.

To prevent this worst-case scenario from occuring, 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; 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.

RPC messages have no such automatic protection. They should be used in cases where the handler function either does not reply or causes at most a tolerable number of uncompleted transactions. 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.

11.5 Block Data Transfers

CMAML supports transfers of contiguous blocks of serial and parallel data. (In discussions of CMMD functions, these are referred to as serial and parallel arrays.) This block transfer mechanism is the mechanism that underlies CMMD's virtual channels. Virtual channels, in turn, underlie the blocking and asynchronous CMMD point-to-point message passing functions.

Sending a data block is a cooperative effort between a destination node and one or more source nodes.

- First, two or more nodes agree on the parameters for the transfer, e.g., which receive port will be used, whether the transfer will be serial or parallel, the number of bytes that will constitute the transfer.
- Next, one or more nodes send data to the designated port on the destination node. If multiple nodes are sending, they use offsets to position their data within the reserved memory area.
- The destination node handles the incoming data, keeping track of the number of bytes received.
- If desired, the port and/or the transmittal messages may invoke handlers when the transmittal completes.
- The destination node then frees the receive port or resets it for another use.

Node Preparation for the Transfer

First, two or more nodes agree on parameters for the transfer. This includes setting up a receive port on the destination node; making sure that the sending node(s) know the port number; and making sure that all nodes involved agree on the number of bytes to be sent, and on whether the transfer will involve serial or parallel data.

The *receive port* is a data structure that contains all the information (such as starting location and length) needed to process incoming data packets and store them in the destination node's memory. It may also specify a handler function to be invoked when the transfer completes. (The user-accessible information contained in a receive port is described in Section 11.5.2.)

Receive ports are identified by numbers. A given port can accept either serial or parallel data, but not both.

These necessary steps can be accomplished in a number of ways. Two such ways will serve as examples:

- **Example 1.** The destination node asks for any available receive port. It then uses active messages to send the port number to the sending node, together with a request for a given number of bytes of serial data to be sent to that port.

- **Example 2.** All nodes cooperatively allocate a particular receive port on every node, then send data to each other's ports in some predetermined pattern.

  (The nodes use **`CMAML_allocate_this_rport`**, then globally sum return values to make sure all nodes got **`TRUE`** returns. If the sum doesn't equal the partition size, somebody failed to allocate the port. Those who did get the port must then free it; all nodes collectively then try to allocate another specified port. This continues until all nodes successfully allocate the same port.)

At this time, the destination node should also initialize its port's byte counter to the number of bytes it expects to receive in a transfer, using the function **`CMAML_set_rport_byte_counter`**.

The Source Node(s) Send the Block Data

Source nodes may now send blocks of data to the destination node. They use the function **`CMAML_scopy`** to send a serial block, **`CMAML_pcopy`** to send a parallel block. Arguments to the function specify the node ID, the port number, and the number of bytes being sent.

If multiple source nodes are involved, they copy bytes into different offsets of the receive port. The total number of bytes sent by all nodes must equal the receive port's initial receive byte counter value.

PLEASE NOTE

(1) It is very important that the source node(s) and destination node agree on the number of bytes to be sent to a receive port. This information can easily be exchanged using active messages.

(2) It is an error to attempt to send a serial source to a parallel destination and vice versa.

(3) The number of bytes transferred by a **pcopy** represents the total for all vector units. Each vector unit receives one-fourth of the bytes transferred.

Handlers

The **CMAML_scopy** and **CMAML_pcopy** calls may invoke handlers upon completion. The receiving port may also specify a handler function to be called when the last expected byte of data has been received. Most often, this handler sets a flag to indicate the completion of the transfer.

The Destination Node Handles the Block Data

The destination node either enables interrupts and lets the CMMD interrupt handler take care of receiving and storing the array packets, or disables interrupts and explicitly polls for the data packets.

In either case, the CMAML software automatically uses the data in the receive port to correctly store the incoming destination packets in the destination block.

As each data byte arrives, the receive port's byte counter is decremented. When this counter becomes exactly zero, the receive port's handler function (if any) is scheduled for execution. (If the counter always remains greater than zero, then its handler function is never scheduled for execution.) If the counter becomes negative, a fatal error is signaled.

The Destination Node Cleans Up, or Prepares for the Next Transfer

A receive port can only be used for a single block data transfer. After use, the port must be either freed or reset for another use. The simplest way of resetting a port is simply to reset its byte counter. However, other items in the data structure can also be changed at this time. (See Section 11.5.2.)

Users *must* explicitly free any ports that they allocate. Ports are a shared, finite resource. They are used not only by these block transfers but also by CMMD message-passing functions, CMMD virtual channel functions, and CMMD I/O. If a program runs out of ports, fatal consequences ensue.

11.5.1 Blocking and Non-Blocking Transfers

`CMAML_scopy` and **`CMAML_pcopy`** can be called either from a node's main thread of control or from a handler function. When called from the main thread of control, they are blocking operations; they return only after the last data byte is sent and the handler (if any) has executed and returned.

When called from a handler, **`CMAML_scopy`** and **`CMAML_pcopy`** are non-blocking operations. Even if the **`CMAML_scopy`** or **`CMAML_pcopy`** calls do not execute by the time control returns from the handler, however, they always execute before control returns to the main thread.

Handler functions invoked by the block transfers themselves are invoked when the transaction has been completed. This may be after the return of the **`CMAML_scopy`** or **`CMAML_pcopy`** function.

11.5.2 Contents of the Receive Port Data Structure

A receive port is a data structure containing information relating to a single array transfer. It contains the following user-visible information. Except for the allocated flag, the user program must ensure that all these fields are correctly set before the transfer begins.

addr	The starting address of the memory block where the received data will be stored.
allocated	A read-only flag. **TRUE** if this port is allocated, **FALSE** otherwise.
byte_counter	Initially, this is the expected number of bytes to receive via this port. As bytes arrive, the value counts down toward zero.
handler	The handler function; either **NULL** or a pointer to a user-supplied handler function that executes when the **byte_counter** reaches zero.
handler_arg2	Either **NULL** or a pointer to a user-supplied argument to the handler function. (The handler function is called with two arguments: the receive port number and this supplied argument.)
total_nbytes	The total number of bytes to be received.

11.6 CMAML Function Dictionary

11.6.1 Active Messages: Requests, Replies, and RPCs

The CMAML module provides these functions for sending active messages:

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*)

ARGUMENTS

The arguments to these functions are as follows:

dest_node An integer specifying the destination node for the message.

handler Address of a handler function on the destination node.

data1, *data2*, *data3*, *data4*
Handler function arguments.

USE WITH INTERRUPTS

CMAML_request[**_tohost**]**()** may be called with interrupts enabled. All other active message functions must be called with interrupts disabled.

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.

(The **tohost** functions are for use in host/node programming.)

CMAML_request[**_tohost**]**()** and **CMAML_rpc**[**_tohost**]**()** check both the Request and Reply interfaces for incoming messages. **CMAML_reply**[**_tohost**]**()** checks only the Reply interface.

CMAML_request[**_tohost**]**()** must only be called from user code (not from within any handlers). **CMAML_rpc**[**_tohost**]**()** may be called from either user code or a handler. **CMAML_reply**[**_tohost**]**()** must only be called from a Request message handler.

When called from user code, active message functions are blocking operations, and do not return until the message has been sent. When called from within a handler, however, the active message functions are non-blocking operations, and may be queued for later execution. (In this case, they are guaranteed to complete before control returns to user code.)

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 is entirely different, and in fact can vary from run to run. You must therefore obtain the correct handler address from the host *each time* your program executes, and *before* sending any **_tohost** active messages.

11.6.2 Sending and Receiving Arrays

The following CMAML functions are used to send an array:

C SYNTAX

int **CMAML_scopy** **(**int *dest_node*, int *dest_rportno*, int *dest_align*,
int *dest_offset*, int *source_addr*, int *nbytes*,
void (**rport_handler*)(), void **rport_handler_arg***)**

int **CMAML_pcopy** **(**int *dest_node*, int *dest_rportno*, int *dest_align*,
int *dest_offset*, int *source_addr*, int *nbytes*,
void (**rport_handler*)(), void **rport_handler_arg***)**

ARGUMENTS

The arguments to these functions are as follows:

dest_node	The destination node number.
dest_rportno	The destination receive port number.
dest_align	The alignment of the destination array, in bytes.

Value of 0 = doubleword, 4 = word, and other values between 1 and 7 inclusive indicate byte alignments. The alignment takes into account the *dest_offset* parameter.

dest_offset — The byte offset into the destination array (0 = no offset).

Note: CMMD computes the *dest_align* argument after factoring in *dest_offset*.

source_addr — The address of the source array.

nbytes (**`scopy`** only)
The number of bytes to transfer.

nbytes (**`pcopy`** only)
Total number of bytes to transfer across *all 4 VUs*.

This argument must be a multiple of 16 (4 bytes per VU for 4 VUs).

rport_handler — Handler function called after last byte is transferred. *rport_handler_arg* — Handler function's argument, a **`void *`** pointer.

Both these arguments can be **`NULL`** if no handler function is required.

RETURN VALUES

`CMAML_scopy` and **`CMAML_pcopy`** return **`TRUE`** if the transfer was completed, **`FALSE`** if the operation was queued for execution later, and –1 upon encountering a non-fatal error. Non-fatal errors are the results of violating one or more of the restrictions listed in the “Restrictions” section below.

DESCRIPTION

Both these functions transmit an array to the specified destination node. **`CMAML_scopy()`** is used for sending serial arrays (block or vector) while **`CMAML_pcopy()`** is used for sending parallel arrays. The transmitted array is copied into a previously allocated receive port. (See **`CMAML_allocate_rport`**, **`CMAML_free_rport`**, and related functions.)

Both **`CMAML_scopy`** and **`CMAML_pcopy`** copy *nbytes* from the source array at *source_addr* to the receive port *dest_rportno* on the node *dest_node*. The sender and receiver must agree a priori on the number of bytes transfered. The sender specifies this byte count as *nbytes*. The receiver sets the byte count field of its receive port with the **CMAML_set_rport_byte** accessor.

As each data byte arrives, the receive port's byte counter is decremented. When the counter becomes exactly zero, the receive port's handler function (if specified) is scheduled for execution. (If the counter is ever negative, then a fatal error is signaled, and if the counter is always greater than zero, then the handler function is never scheduled for execution.) Multiple senders may copy bytes into different offsets of the same receive port. Collectively, the number of bytes sent should equal to the receive port's initial receive byte counter value.

CMAML_scopy and **CMAML_pcopy** may be called from either user code or from a handler called by **CMMD_send/receive_async**, **CMAML_scopy**, **CMAML_pcopy**, or an active message handler. When called from user code, **CMAML_scopy** and **CMAML_pcopy** are blocking operations, and do not return until the last data byte is sent. When called from a handler, **CMAML_scopy** and **CMAML_pcopy** are non-blocking operations, and may actually be queued for later execution. If any operation or handler is queued for execution, it is executed before control returns to user code.

It is the user's responsibility to ensure that **CMAML_scopy** transfers data to a receive port that is set up for receiving serial data, and that **CMAML_pcopy** transfers data to a receive port that is set up for receiving parallel data. It is an error to attempt to send a serial source to a parallel destination and visa versa. For parallel data, **CMAML_pcopy** copies *nbytes* total from source to destination. The same number of bytes (*nbytes* / 4) is copied between each source and destination VU. Thus, the number of bytes must be evenly divisible by the number of VUs on a PN (currently 4). Also, **CMAML_pcopy** only supports transfers in multiples of 6 bytes. Thus, the smallest possible transfer is 4 bytes from each of the 4 VUs. The next smallest transfer is 8 bytes from each of the 4 VUs, etc.

RESTRICTIONS

For **CMAML_scopy**:

- Arrays must be contiguous.
- The *nbytes* parameter must be between 1 and $(2^{25})–1$, inclusive.
- *dest_node* must be between 0 and **CMMD_partition_size()**, inclusive.

For **CMAML_pcopy**:

- Arrays must be contiguous on each VU.
- Arrays must span all VUs.
- The *nbytes* parameter must be between 1 and (2^{25})–1, inclusive, and must be an exact multiple of 16 (4 bytes per VU for 4 VUs).
- The *dest_node* must be between 0 and **(CMMD_partition_size()**–1), inclusive.
- Source and destination arrays must be doubleword-aligned (that is, the sum of *dest_align* and *dest_offset* must be divisible by 8).

11.6.3 Port Allocation and Accessors

The following CMAML functions are provided for manipulating receive ports:

C SYNTAX

```
int  CMAML_allocate_rport(void)
int  CMAML_allocate_this_rport(int rport)
int  CMAML_free_rport(int rport)
int  CMAML_number_of_rports(void)
int  CMAML_number_of_free_rports(void)

void  *CMAML_get_rport_addr(int rport)
void  CMAML_set_rport_addr(int rport, void *value)

int   CMAML_get_rport_allocated(int rport)

int   CMAML_get_rport_byte_counter(int rport)
void  CMAML_set_rport_byte_counter(int rport, int value)

void  *CMAML_get_rport_handler(int rport)
void  CMAML_set_rport_handler(int rport, void *value)

void  *CMAML_get_rport_handler_arg2(int rport)
void  CMAML_set_rport_handler_arg2(int rport, void *value)

int   CMAML_get_rport_total_nbytes(int rport)
void  CMAML_set_rport_total_nbytes(int rport, int value)
```

USE WITH INTERRUPTS

These functions cannot be used with interrupts enabled.

DESCRIPTION

These functions provide an interface to receive ports. Receive ports are used in conjunction with **CMAML_scopy()** and **CMAML_pcopy()** for moving blocks of memory between nodes.

CMAML_allocate_rport allocates a port (if one is available) and returns a port ID between 0 and one less than **CMAML_number_of_rports()**. It returns –1 if no ports are available. **CMAML_allocate_this_rport** attempts to allocate the port specified by **rport**. It returns **TRUE** if the specified port is allocated, and **FALSE** otherwise. (You can use the **this_rport** function to request a specific port on every node, so that all nodes use the same port number.)

CMAML_free_rport marks the specified port as free and returns it to the internal pool of available ports. Returns **TRUE** if the specified port is allocated, **FALSE** otherwise.

CMAML_number_of_rports() returns the total number of rports that can be allocated. **CMAML_number_of_free_rports()** returns the actual number of rports that are available for allocation.

The remaining functions set and retrieve information from port objects.

A port object is a data structure containing information relating to a single array transfer. It contains the following user-visible information. (Except for the **allocated** flag, the user program must ensure that all these fields are correctly set before the transfer begins.)

addr The starting address of the memory block where the received data will be stored. All serial memory addresses are normal SPARC addresses. All parallel addresses are assumed to be in broadcast (or “ALL VU”) parallel space. Either data or instruction space addresses are acceptable.

allocated	A read-only flag. **TRUE** if this port is allocated, **FALSE** otherwise. **CMAML_allocate_rport** and **CMAML_allocate_this_rport** set this flag. **CMAML_free_rport** clears this flag.
byte_counter	Initially, this should be set to the expected number of bytes to receive via this port. As bytes arrive, the value counts down toward zero.
handler	The handler function; either **NULL** or a pointer to a user-supplied handler function that executes when the **byte_counter** reaches zero.
handler_arg2	Either **NULL** or a pointer to a user-supplied argument to the handler function. (The handler function is called with two arguments: the receive port number and this supplied argument.)
total_nbytes	The total number of bytes to be received. (**Note:** This value remains constant. It should equal the initial value of **byte_counter**.)

RESTRICTIONS

In general:

- Receive arrays must be contiguous.
- The **byte_counter** value of a port cannot exceed $(2^{25})–1$.
- If a handler is specified, it must be a pointer to a valid function address.

For parallel array transfers:

- Arrays must span all 4 VUs.
- The initial value of the **byte_counter** value must be an exact multiple of 16 (4 bytes per VU * 4 VUs).

11.6.4 Packet Interrupt Enabling and Disabling

The CMAML layer provides the following functions for enabling and disabling packet interrupts:

C SYNTAX:

int **CMAML_enable_interrupts(**void**)**
int **CMAML_disable_interrupts(**void**)**
int **CMAML_interrupt_status(**void**)**
int **CMAML_inside_interrupt_handler(**void**)**

Note: User-defined functions that perform floating-point computations within the dynamic scope of an interrupt handler are responsible for saving and restoring the register state of the floating-point units. (See Section 11.6.5 below.)

USE WITH INTERRUPTS

It is the user's responsibility to *not* enable interrupts within the dynamic scope of the CMMD interrupt handler.

DESCRIPTION

The two functions **CMAML_{enable**,**disable}_interrupts** unconditionally enable and disable interrupts. They return **TRUE** if interrupts were successfully enabled (or disabled) and **FALSE** otherwise. (**Note:** these functions only return **TRUE** if the interrupt state was actually changed.)

CMAML_interrupt_status returns the current state of interrupts; it returns **TRUE** if interrupts are enabled, and **FALSE** otherwise.

CMAML_inside_interrupt_handler returns **TRUE** if program execution is currently within the dynamic scope of the CMMD interrupt handler, and **FALSE** otherwise. A handler function can call **CMAML_inside_interrupt_handler** to determine whether it must save the state of the SPARC and VU registers.

11.6.5 Saving and Restoring VU and SPARC FP Register States

The CMAML interrupt mechanism incurs as little overhead as possible; thus, it does not automatically save the states of the SPARC FPU registers, vector unit data or control registers, or the vector unit instruction extension registers. Any handler functions that include SPARC FPU or VU calculations (and are called within the dynamic scope of the CMMD interrupt handler) are responsible for saving the states of these registers and restoring them.

The following CMAML functions are provided for use in saving and restoring the state of the SPARC FP registers and VU data/control registers:

C SYNTAX

void **`CMAML_save_fpuregs(`**double **iu_fp_spillarea*`)`
void **`CMAML_restore_fpuregs(`**double **iu_fp_spillarea*`)`
void **`CMAML_save_dataregs(`**double **dp_dr_spillarea*`)`
void **`CMAML_restore_dataregs(`**double **dp_dr_spillarea*`)`
void **`CMAML_save_ctrlregs(`**double **dp_cr_spillarea*`)`
void **`CMAML_restore_ctrlregs(`**double **dr_cr_spillarea*`)`

DESCRIPTION

These functions save the state of the VU control and/or data registers and the state of the SPARC FP registers. Interrupt handlers may depend on the state of these registers and may not work properly if they are unexpectedly modified by a handler function.

The save/restore functions must be called by any handler function that can be called within the dynamic scope of an interrupt handler. (A handler function can call **`CMAML_inside_interrupt_handler()`** to determine whether it has been called from an interrupt handler, and then save the registers if necessary.)

The *spillarea* arguments are pointers to user-defined memory regions used to hold the values of these registers. The address of each region must be double-word-aligned, and each region must be large enough to contain the block of registers specified by the function. Currently, the required sizes are:

For **`CMAML_save_fpuregs`**:	136 bytes in serial (SPARC) memory
For **`CMAML_save_dataregs`**:	512 bytes in parallel (VU) memory
For **`CMAML_save_ctrlregs`**:	104 bytes in serial (SPARC) memory

For the Curious: The space requirements of some regions are padded to be doubleword multiples.

Note: The *dp_dr_spillarea* must be a pointer to a parallel memory region in broadcast (or “ALL VU”) data space.

Implementation Note: All CMMD or CMAML functions that internally use the VU vector data registers (such as parallel memory array transfers) guarantee that the state of the VU registers is preserved.

11.6.6 Network Polling Functions

These functions are used for polling the Data Network while network interrupts are disabled:

C SYNTAX

void **`CMAML_poll()`**
void **`CMAML_request_poll()`**
void **`CMAML_reply_poll()`**
void **`CMAML_wait(`**volatile int **flag*`,` int *value*`)`
void **`CMAML_poll_wait(`**volatile int **flag*`,` int *value*`)`
void **`CMAML_poll_while_lt(`**volatile int **flag*`,` int *value*`)`

ARGUMENTS

flag	A volatile pointer to an integer variable.
value	An integer value for comparison with the *flag* argument.

USE WITH INTERRUPTS

These functions cannot be used with interrupts enabled.

DESCRIPTION

`CMAML_poll()` is the generic CMAML polling function. It checks both network interfaces for messages, and if it finds one pending, handles reading it from the network and calling the appropriate handler function. It keeps checking the network until no messages are pending, and then returns.

`CMAML_request_poll()` polls only for incoming Request messages (that is, it only polls the Request interface).

`CMAML_reply_poll()` polls only for incoming Reply messages (that is, it only polls the Reply interface).

`CMAML_wait(`**flag*, *value*`)` polls repeatedly (on both interfaces) until the value pointed to by **`flag`** is equal to **`value`**. The flag is tested *before* the network is checked, so this function may not poll at all. If the network is polled at least once, then before returning, **`value`** is subtracted from **`flag`**.

`CMAML_poll_wait(`**flag*, *value*`)` is the same as **`CMAML_wait`**, except that **`flag`** is tested *after* the network is checked.

`CMAML_poll_while_lt(`**flag*, *value*`)` is the same as **`CMAML_wait`**, except that **`value`** is not subtracted from **`flag`** before the function returns.

11.7 Examples

11.7.1 Example: Sending an RPC Message

RPC messages are useful for performing simple exchanges of data between nodes. For example, here's 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;
  volatile int value;
{    *address = value;  }
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;
{
```

```
    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(* (int *) &value < 0);
    }
    printf("PN=%d: value was -1 and is now %d\n",
              self, value);
}
```

11.7.2 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's 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;
{
    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)
         CMAML_poll_while_lt(&done_flag, 2);
         /* Must wait until _all_ communication is done,
            that is, until 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);
}
```