Chapter 13

CMAML: 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.

13.1 How Block Data Transfers Work

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

- First, the destination node and the source node(s) 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 be transferred.
- Next, one or more nodes send data to the designated receive 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 transmitted messages may invoke callback functions when the transmission completes.

Copyright © 1995 Thinking Machines Corporation

- The destination node then frees the receive port or resets it for another use.

Figure 19 shows the implementation of block data transfers on the sending and receiving nodes.

Node Preparation for the Transfer

The preliminary setup work outlined above, accomplished by the destination node and the sending node(s), is usually done using active messages.

The *receive port* (or *rport*) is an opaque object 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 callback function to be invoked when the transfer completes. (The user-accessible information contained in an rport is described in Section 13.1.2.)

Copyright © 1995 Thinking Machines Corporation

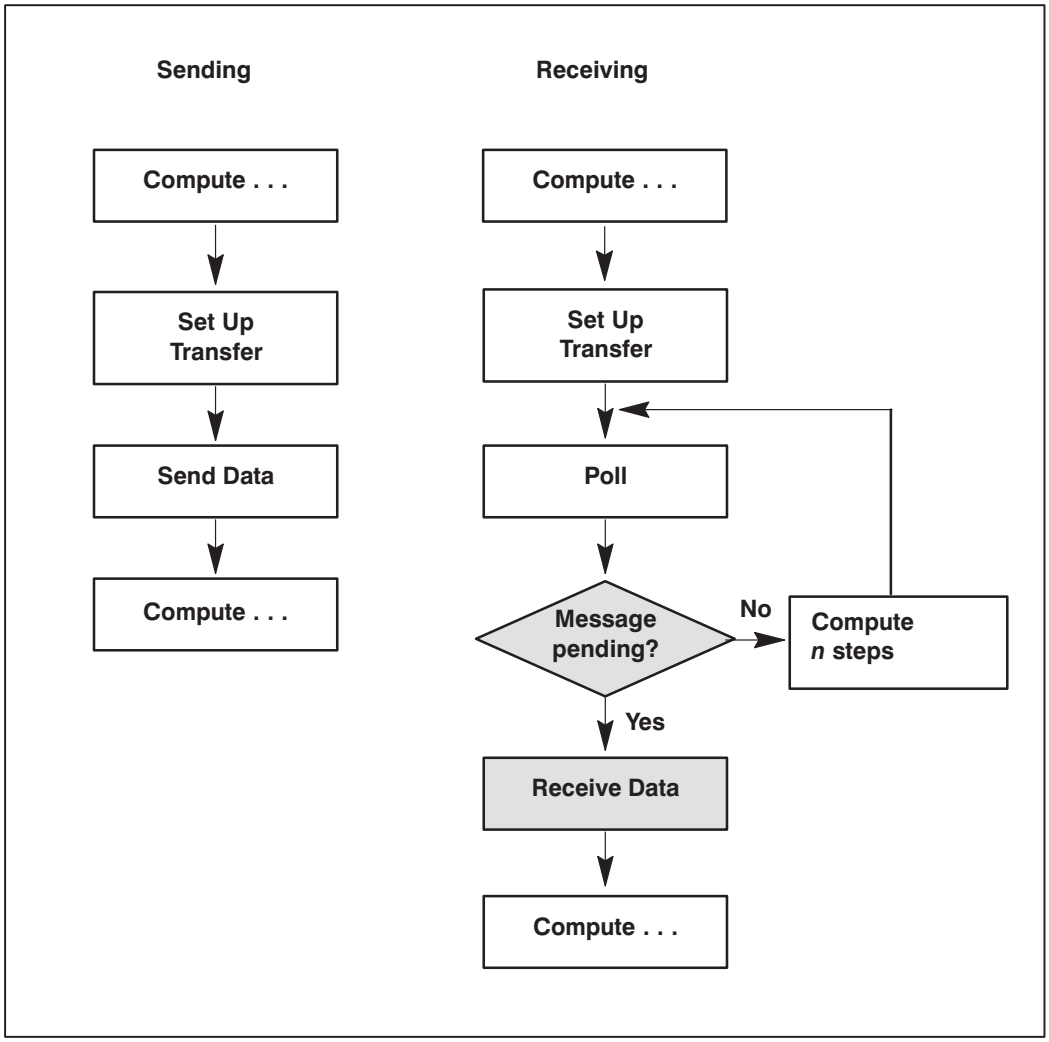

Figure 19. Block data transfers.

Rports are identified by nonnegative integers. A given rport 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 rport number to the sending node, together with a request for a given number of bytes of serial data to be sent to that rport.

Copyright © 1995 Thinking Machines Corporation

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

  (Typically, all nodes call `CMAML_allocate_this_rport`, then globally `AND` the return values to make sure all nodes returned `TRUE`. If the global `AND` is not `TRUE`, then some node(s) failed to allocate the rport. This process is repeated until success, although it is usually possible to "guess" a globally free rport the first time.)

At this time, the destination node initializes its receive rport's byte counter to the total number of bytes it expects to receive from all nodes sending to it, 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, and `CMAML_pcopy` to send a parallel block. Arguments to the function specify the node ID, the rport number, and the number of bytes being sent.

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

Implementation Note: `CMAML_scopy` and `CMAML_pcopy` require a "dest alignment" argument. This may seem strange, given that a "dest offset" argument also exists. However, the combination of the two allows `CMAML_scopy` and `CMAML_pcopy` to perform a simple preliminary calculation that ensures that packets they send are destined for doubleword-aligned locations on the *receiving* node. This is important for obtaining good performance.

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 an rport. This information can be exchanged easily using active messages.

Version 3.3, April 1995
Copyright © 1995 Thinking Machines Corporation

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.

Callback and Handler Functions

The `CMAML_scopy` and `CMAML_pcopy` calls may invoke callback functions upon completion. The receiving port may also specify a callback function to be called when the last expected byte of data has been received. Typically, this callback sets a flag to indicate the completion of the transfer.

The reason that the machinery is provided for associating a callback function with an `_scopy` or a `_pcopy` is that these functions do not necessarily block, but, depending on where they are called (inside an RPC handler, for instance), they might be scheduled for later execution. If this were not the case, then of course a callback would be unnecessary, since one could always simply place the desired function call lexically after the `_scopy` or `_pcopy`.

The Destination Node Handles the Block Data

CMAML software automatically uses the state contained in the receive port to correctly store the incoming destination packets in the destination block.

As each data byte arrives, the rport's byte counter is decremented. When this counter becomes exactly zero, the rport's callback function (if non-`NULL`) is scheduled for execution, which possibly takes place immediately. (If the counter always remains greater than zero, then its handler function is never scheduled for execution.) If the counter becomes negative — an unusual occcurrence, typically due to a user program error such as failing to disable interrupts or sending too many packets to an rport — then a fatal error occurs.

Copyright © 1995 Thinking Machines Corporation

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

An rport can be used only for a single block data transfer. After the transfer is complete, the rport must be either freed or reset for another use. The simplest way of resetting an rport is simply to reset its byte counter. Other items in the data structure can also be changed at this time. (See Section 13.1.2.)

Users *must* explicitly free any rports that they allocate. Generally speaking, this should be done as soon as possible, since the system will never free them by itself. Rports 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 rports, possibly fatal consequences ensue.

13.1.1 Blocking and Nonblocking Transfers

`CMAML_scopy` and `CMAML_pcopy` can be called either from a node's main thread of control or from a handler or callback function (see Chapter 14). When called from the main thread of control, they are blocking operations; they return only after the last data byte is sent and the callback (if any) has executed and returned.

When called from a handler or a callback function, `CMAML_scopy` and `CMAML_pcopy` are potentially nonblocking operations, that is, they are not necessarily invoked immediately but may be scheduled for later execution. Such execution can, but need not, occur almost immediately. For example, if an RPC is being sent from node 17, and if during the sending of this RPC an active message arrives, then the active message's handler will be invoked immediately and run to completion. But if that handler itself calls `CMAML_scopy`, that call will *not* take effect immediately (that is, it will not take precedence over the in-progress RPC); instead, it will be scheduled for later execution. 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.

Callback 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. (See Chapter 6 more informa-

Version 3.3, April 1995
Copyright © 1995 Thinking Machines Corporation

tion about callback functions, and Chapter 14 for more information about handlers.)

To illustrate some typical execution scenarios for the callback functions associated with block transfers, consider the following examples:

- The user calls `CMAML_scopy` from the top level of the program. It behaves as a blocking function, and its callback function executes immediately (that is, immediately after all the data is sent and immediately before the function returns).

- The user sends an active message whose handler calls `CMAML_scopy`. The handler is received on another node while top-level code is executing there, i.e, either by interrupting user code or via a call to `CMAML_poll` from user code. In either case, the active message handler executes immediately (active message handlers *always* execute immediately), and the scopy and its callback function both run to completion even before the handler itself completes.

- The user sends an RPC active message whose handler calls `CMAML_scopy`. The handler is received on another node *not* during the execution of top-level user code, but rather at a moment when top-level user code itself is calling `CMAML_rpc`. As part of its transmission algorithm, `CMAML_rpc` attempts to receive any incoming packets that may be present at the same time (in order to keep the network as clear for sending as possible). Consequently, it receives the active message in question, and executes its handler immediately. However, when the handler calls `CMAML_scopy`, the call to `CMAML_scopy` is scheduled for later execution rather than taking over at that moment, in order to ensure that the currently executing RPC call finishes quickly, for it is a defining characteristic of active messages that they are usually sent quickly. (Obviously, `CMAML_scopy` "knows" whether it is being called from top-level user code or from within some CMMD function such as `_rpc`; this is made possible by an internal mechanism invisible to the user.)

- The user has called `CMAML_poll` from top level, and a data packet arrives that happens to be the last packet in an rport transfer. As soon as the packet's data is stored in the rport's buffer, the rport's callback function is executed. The callback function itself calls `CMAML_scopy`, and this copy is executed immediately (*not* scheduled for later execution), since it so happens that scopies have priority over callbacks.

Copyright © 1995 Thinking Machines Corporation

13.1.2 Contents of the Receive Port Data Structure

An rport is an opaque object containing information relating to a single array transfer. It contains the following user-accessible attributes or "fields." 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 rport is allocated, `FALSE` otherwise.

`byte_counter` — Initially this is the expected number of bytes to receive via this rport. As bytes arrive, the value counts down toward zero.

`handler` — The callback function; either `NULL` or a pointer to a user-supplied callback function that executes when the `byte_counter` reaches zero. (The regrettable inconsistency of its being termed a "handler" arises for historical reasons.)

`handler_arg2` — Either `NULL` or a pointer to a user-supplied argument that will be passed automatically to the callback function. (The callback function is called with two arguments: the rport number and this supplied argument.)

`total_nbytes` — The total number of bytes to be received.

13.2 Sending and Receiving Arrays

The following CMAML functions are used to send and receive arrays:

To send arrays:
```
CMAML_scopy
CMAML_pcopy
```

To manipulate receive ports:
```
CMAML_allocate_rport
CMAML_allocate_this_rport
CMAML_free_rport
```


Copyright © 1995 Thinking Machines Corporation

```
CMAML_number_of_rports
CMAML_number_of_free_rports
CMAML_get_rport_addr
CMAML_set_rport_addr
CMAML_get_rport_allocated

CMAML_get_rport_byte_counter
CMAML_set_rport_byte_counter

CMAML_get_rport_handler
CMAML_set_rport_handler
CMAML_get_rport_handler_arg2
CMAML_set_rport_handler_arg2

CMAML_get_rport_total_nbytes
CMAML_set_rport_total_nbytes
```

Their reference pages are provided in this section.

Copyright © 1995 Thinking Machines Corporation

CMAML_scopy
CMAML_pcopy

Block copy of serial (`scopy`) or parallel (`pcopy`) buffer.

SYNOPSIS

C Syntax

```
int CMAML_scopy(int dest_node, int rport, int dest_align,
          int dest_offset, int source_addr, int nbytes,
          void (*handler)(), void *handler_arg)

int CMAML_pcopy(int dest_node, int rport, int dest_align,
          int dest_offset, int source_addr, int nbytes,
          void (*handler)(), void *handler_arg)
```

Fortran Syntax

```
INTEGER FUNCTION CMAML_scopy
          (dest_node, rport, dest_align, dest_offset,
          source_addr, nbytes, handler, handler_arg)
    INTEGER dest_node, rport, dest_align, dest_offset
    INTEGER source_addr, nbytes
    EXTERNAL handler
    <ANY> handler_arg

INTEGER FUNCTION CMAML_pcopy
          (dest_node, rport, dest_align, dest_offset,
          source_addr, nbytes, handler, handler_arg)
    INTEGER dest_node, rport, dest_align, dest_offset
    INTEGER source_addr, nbytes
    EXTERNAL handler
    <ANY> handler_arg
```

ARGUMENTS

The arguments to these functions are as follows:

dest_node	The destination node number.
rport	The destination receive port number. Can be the host node.

Version 3.3, April 1995
Copyright © 1995 Thinking Machines Corporation

dest_align The address of the destination buffer, modulo 8.

A value of 0 means doubleword-aligned, 4 means word-aligned, and other values between 1 and 7 inclusive indicate byte alignments. (`CMAML_scopy` and `CMAML_pcopy` take *dest_align* into account in conjunction with the *dest_offset* parameter in order to determine the most efficient way to partition the send buffer into network packets.)

dest_offset The byte offset into the destination array (0 = no offset). Note that multiple nodes can send data into a single array by using different offsets.

source_addr The address of the source array.

nbytes For `scopy`: The number of bytes to transfer.

For `pcopy`: Total number of bytes to transfer across *all 4 VUs*. Must be a multiple of 16 (4 bytes per VU for 4 VUs).

handler Callback function to be invoked (or scheduled for execution) when the last byte is transferred. (Can be `NULL` if no handler function is required.)

handler_arg Callback function's single argument, a `void *` pointer. (Will be ignored if no callback function is specified, or if the callback function does not take an argument; is usually set to `NULL` in those cases.)

RETURN VALUES

`CMAML_scopy` and `CMAML_pcopy` return `TRUE` if the transfer was completed, `FALSE` if the operation was scheduled for execution later, and −1 upon encountering a nonfatal error. Nonfatal errors are the results of violating one or more of the restrictions listed in the **Restrictions** section below. It is strongly advised that all calls to `CMAML_scopy` or `CMAML_pcopy` include a check of the return value, with appropriate action taken by the user program if the value is −1.

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 written

Copyright © 1995 Thinking Machines Corporation

into the buffer associated with an allocated receive port on the receiving node. (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 buffer associated with the receive port *`rport`* on the node *`dest_node`*. The sender and receiver must agree on the number of bytes to be 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_counter` accessor.

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

`CMAML_scopy` and `CMAML_pcopy` may be called from user code, from a callback associated with `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 or callback, `CMAML_scopy` and `CMAML_pcopy` are potentially nonblocking operations, and may actually be scheduled for later execution.

Note that the callback functions for `CMAML_scopy` and `CMAML_pcopy` must have a calling sequence consisting of at most one argument, the *`handler_arg`* (which may, for example, be a pointer to an arbitrarily complex data structure). This is distinct from the callback associated with an rport, which takes up to two arguments (the rport number plus one auxiliary argument).

PLEASE NOTE

1. If any operation or handler is queued for execution, it is executed before control returns to user code. Other than this, there is no guarantee as to when execution takes place.

Version 3.3, April 1995
Copyright © 1995 Thinking Machines Corporation

2. It is the application's responsibility to make sure that the contents of the source buffer passed to CMAML_scopy or CMAML_pcopy do not change until the initial call executes. In practice, this means preserving them until the next possible return 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 vice versa. For parallel data, CMAML_pcopy copies *nbytes* total from source to destination. The same number of bytes (*nbytes* / 4) is transferred 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 of a multiple of 4 bytes per VU. Thus, the total number of bytes in *nbytes* must be a multiple of 16.

RESTRICTIONS

For CMAML_scopy:

- Arrays must be contiguous. No striding is allowed. (Not enforced, but is implicit.)
- The *nbytes* parameter must be between 1 and $(2^{25})-1$, inclusive. Violations cause a return value of -1.
- *dest_node* must be between 0 and CMMD_partition_size, inclusive. (The host node is always a valid destination for serial transfers.) Violations cause a return value of -1.
- Interrupts must be disabled. Violations cause a return value of -1.

For CMAML_pcopy:

- Arrays must be contiguous on each VU and span all VUs. (Not enforced, but is implicit.)

Copyright © 1995 Thinking Machines Corporation

- 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). Violation of the first condition is detected and causes a return value of -1.
- The *dest_node* must be between 0 and (CMMD_partition_size − 1), inclusive. (The host node is never a valid destination for a parallel transfer.) Violation causes a return value of -1.
- The sum of *dest_align* and *dest_offset* must be evenly divisible by 8. Violation causes a return value of -1.
- Interrupts must be disabled. Violation causes a return value of -1.

Whenever CMAML_scopy or CMAML_pcopy return -1, they take no other action whatsoever.

Version 3.3, April 1995
Copyright © 1995 Thinking Machines Corporation

Allocating and Freeing Receive Ports

These CMAML functions are used to allocate and free receive ports, and to gather information on the number of receive ports available.

SYNOPSIS

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)
```

Fortran Syntax

```
INTEGER FUNCTION CMAML_allocate_rport()
INTEGER FUNCTION CMAML_allocate_this_rport(rport)
    INTEGER rport
INTEGER FUNCTION CMAML_free_rport(rport)
    INTEGER rport
INTEGER FUNCTION CMAML_number_of_rports()
INTEGER FUNCTION CMAML_number_of_free_rports()
```

INTERRUPTS

These functions *cannot* be used with interrupts enabled.

DESCRIPTION

These functions provide an interface to the opaque 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 an rport (if one is available) and returns an rport ID between 0 and one less than `CMAML_number_of_rports`, inclusive. It returns −1 if no rports are available. `CMAML_allocate_this_rport` attempts to allocate the rport specified by its argument *rport*. It returns `TRUE` if the specified

Copyright © 1995 Thinking Machines Corporation

rport is allocated, and `FALSE` otherwise. (You can use the `this_rport` function to request a specific rport on every node, so that all nodes use the same rport.)

`CMAML_free_rport` marks the specified rport as free and returns it to the internal pool of available rports, so that it can be reallocated. Returns `TRUE` if the operation is successful, `FALSE` otherwise. A value of FALSE could mean that the rport was not allocated, that the rport number was invalid (e.g., negative, or too large), or that the rport has been reserved by internal software, and therefore not freeable by the user.

It is strongly advised that user programs always examine the return value of `CMAML_free_rport`, and take appropriate action as needed.

`CMAML_number_of_rports` returns the total number of rports with which the current version of CMMD is configured (typically close to 127). `CMAML_number_of_free_rports` returns the actual number of rports that are available for allocation, at the time of the call. That is, its value is equal to `CMAML_number_of_rports` minus any rports that the user program or CMMD itself has already allocated.

Note that the fact that two nodes return the same value for `CMAML_number_of_free_rports` does not mean that the *same* rports are free on those nodes.

Version 3.3, April 1995
Copyright © 1995 Thinking Machines Corporation

Manipulating Port Objects

CMAML functions that set and retrieve information from rport objects.

SYNOPSIS

C Syntax

```
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)
```

Fortran Syntax

```
SUBROUTINE CMAML_get_rport_addr(rport)
    INTEGER rport
SUBROUTINE CMAML_set_rport_addr(rport, value)
    INTEGER rport
    <ANY> value

INTEGER FUNCTION CMAML_get_rport_allocated(rport)
    INTEGER rport

INTEGER FUNCTION CMAML_get_rport_byte_counter(rport)
    INTEGER rport
SUBROUTINE CMAML_set_rport_byte_counter(rport, value)
    INTEGER rport, value

SUBROUTINE CMAML_get_rport_handler(rport)
    INTEGER rport
SUBROUTINE CMAML_set_rport_handler(rport, value)
```


Copyright © 1995 Thinking Machines Corporation

```
    INTEGER rport
    <ANY> value

SUBROUTINE CMAML_get_rport_handler_arg2(rport)
    INTEGER rport
SUBROUTINE CMAML_set_rport_handler_arg2(rport, value)
    INTEGER rport
    <ANY> value

INTEGER FUNCTION CMAML_get_rport_total_nbytes(rport)
    INTEGER rport
SUBROUTINE CMAML_set_rport_total_nbytes(rport, value)
    INTEGER rport
    <ANY> value
```

INTERRUPTS

These functions *cannot* be used with interrupts enabled.

DESCRIPTION

These CMAML functions set and retrieve information from rport objects.

An rport object is an opaque object containing information relating to a single block transfer. It contains the following user-accessible 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 rport 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 rport. As bytes arrive, the value counts down toward zero.

Version 3.3, April 1995
Copyright © 1995 Thinking Machines Corporation

`handler` The callback function; either `NULL` or a pointer to a user-supplied callback function that executes when the `byte_counter` reaches zero. (The regrettable inconsistency of its being termed a "handler" arises for historical reasons.)

`handler_arg2` Either `NULL` or a pointer to a user-supplied argument to the callback function. (The callback function is invoked with one or two arguments: the receive port number and this optional 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. Striding is not allowed.
- The `byte_counter` value of an rport cannot exceed $(2^{25})-1$.

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).

Copyright © 1995 Thinking Machines Corporation