NAME

CMAM – Introduction to the CM-5 Active Message communication layer.

DESCRIPTION

The CM-5 Active Message layer CMAM (<see-mam>) provides a set of communication primitives intended to expose the communication capabilities of the hardware to the programmer and/or compiler at a reasonably high level. A wide variety of programming models, ranging from send&receive message passing to variants of shared memory, can be implemented efficiently on top of CMAM without performance compromise. The basic communication primitive of CMAM is the *Active Message*: a message where the first word points to the message handler. On reception of an Active Message, the message handler is called with the remainder of the message as argument. The role of the message handler is to integrate the message data into the computation, by storing it, accumulating it, or whatever makes sense in the overlaying programming model. Its execution is atomic relative to other message handlers, thus long message handlers tend to back-up the network. Note that while Active Messages might be viewed as a restricted form of RPC (Remote Procedure Call), this is not the intent: message handlers should only integrate message data into the computation or, alternatively, reply to a simple request. They should not perform arbitrary computational tasks.

The CMAM library provides primitives to send Active Messages to remote nodes, to transfer bulk data to pre-arranged communication segments, and to synchronize processors using the CM-5 control network. Functions to initialize and control the usage of hardware message tags within the CMAM library allow it to co-exist with other message layers. Over all, the CMAM library should be viewed as a ''machine language for communication'' and not as an end-user communication library.

Sending Active Messages

The most important communication primitive is **CMAM**(3) which sends an Active Message to a remote node. It takes a pointer to a C function to call on the remote node and a buffer to pass as argument. **CMAM** is built on top of **CMAM_4**(3) which calls a remote function with 4 32-bit words as arguments. **CMAM_4** sends exactly one CM-5 network packet and passes the arguments from registers on the source node to registers on the destination node. In contrast, **CMAM** sends a header packet and one packet per 12 bytes of buffer and requires allocation of a temporary buffer on the destination node.

Bulk data transfer

The bulk data transfer function **CMAM_xfer**(3) augments **CMAM** by providing efficient bulk data transfer to pre-arranged communication segments. Before data transfer can occur, the recipient must allocate a communications segment using **CMAM_open_segment**(3), specifying a *base address*, a *byte count* and an *end-of-transfer function*. The sender (or multiple senders) can then transfer data to memory relative to the base address. When count bytes have been transferred, the end-of-transfer function is called on the recipient node. **CMAM_xfer** transfers a contiguous byte buffer sending 16 bytes in each CM-5 packet.

''Gather-transfer-scatter'' can be implemented using **CMAM_xfer_***N***i**(3) (*N*=1..4) which send a packet with *N* 32-bit words of data to a segment-base + offset on the remote node. (The next version of CMAM will provide more efficient gather-scatter through two functions, **CMAM_xfer_ivec** and **CMAM_xfer_dvec** transfering 32-bit element vectors (32-bit elements) and 64-bit element vectors with stride.)

Sending Indirect Active Messages

The function **CMAM_indirect_4**(3) is nearly identical to **CMAM_4** but performs an indirect function call on the remote node: a pointer to a pointer to a function (i.e. (**fun)()) is transmitted in the message. In some circumstances this allows one pointer to refer to both the function to be invoked and its environment and thus crams more data into a CM-5 packet. The buffer version **CMAM_indirect** is provided for completeness, but does not provide a clear advantage over **CMAM**.

Polling the network

The current implementation of CMAM does not use interrupts on message arrival, thus, to receive messages, the network interface status register must be polled. All CMAM functions sending messages poll the network so that a node sending messages will handle incoming ones. The most common situation

requiring explicit polling is when a node busy-waits for the reception of specific messages. For this purpose, **CMAM_wait**(3) waits (while polling) on a synchronization flag (a **volatile int**) to reach a given value, resets the flag to zero and returns. Other wait mechanisms may be implemented using **CMAM_poll**(3).

Sending and replying

All CMAM functions sending messages come in two variants to avoid deadlock in two-way communication. The *send* variant (**CMAM**, **CMAM_4**, **CMAM_xfer**, ...) sends messages on the ''left'' CM-5 data network and polls both the ''left'' and the ''right'' networks. The *reply* variant (**CMAM_reply**, **CMAM_reply_4**, **CMAM_reply_xfer**, ...) sends messages on the ''right'' network and only polls that network. Messages sent from the computation proper should use the send variant whereas message handlers wishing to send replies *must* use the reply variant. Reply-message handlers cannot send any messages themselves. Failure to follow these rules may result in a stack overflow: a handler using the send variant to reply and finding the outgoing network backed-up will accept messages from the network to drain it, nesting the appropriate handlers on the stack. Given that these handlers may attempt to send, the stack may grow arbitrarily. By using the reply variant, only messages on the reply network need to be accepted and, given that their handlers may not reply themselves, the stack will not grow arbitrarily.

Communicating with the host (control) processor

Individual processing nodes may send messages to the host (''control processor'' in CM-5 terminology) using special variants of all communication functions (e.g. **CMAM_host**(3), **CMAM_host_4**(3), **CMAM_host_reply**(3), **CMAM_host_xfer**(3), **CMAM_host_reply_xfer**(3), ...). The host can send to any processing node using the normal functions.

The difficulty in communication with the host lies in the separate address spaces: addresses of message handlers are different on the host than on the processing nodes. See **CMAM_host**(3) on possibilities to deal with the situation.

Initialization

CMAM_enable(3) must be called on the control processor before using CMAM. It initializes data structures on all nodes and sets-up default message tag usage. **CMAM_disable**(3) may be called on the control processor at the end of the program to wait for all processing nodes to return to the idle loop.

Barrier synchronization

CMAM provides access to the ''synchronous global OR'' of the CM-5 control network in the form of a fuzzy barrier. **CMAM_start_barrier**(3) signals the arrival of a processing node at the barrier and **CMAM_end_barrier**(3) blocks until all processing nodes have called **CMAM_start_barrier**. **CMAM_barrier**(3) implements a traditional barrier. When starting the barrier, processing nodes may enter a 1-bit value into the global OR computation which is returned at the end of the barrier. This value is sometimes useful for termination detection. Note that **CMAM_enable_barrier**(3) must be called on the host before the barrier functions can be used on the nodes.

Mixing message layers

CMAM has been designed such that it can be mixed with other message layers. However, the only other message layer available, CMMD, does not (yet?) co-operate. The node-to-node communication primitives of CMAM and CMMD cannot be mixed (although a program could alternate phases of CMAM use with phases of CMMD use). The global operations of CMMD which use the control network may be used simultaneously with CMAM node-to-node operations (in particular, the integer scan operations in CMMD are compatible with CMAM). However, the initialization of CMAM barrier operations interferes with CMMD global operations. Note that reportedly CMMD 3.0 will include Active Messages.

CMAM can share the network interface with other message layers (or with user additions to CMAM) by using different hardware message tags and by co-operating in the message tag-handler dispatch. The function **CMAM_set_tags**(3) lets the user specify which hardware message tags CMAM should use. Other message layers may use free tags provided that a handler is registered with CMAM using **CMAM_set_handler**(3) and **CMAM_set_reply_handler**(3).

USAGE

The CMAM header files assume the use of **gcc**(**1**) and contain ANSI-C function prototypes and gcc **inline** definitions. Conversion to **cc**(**1**) is certainly possible, but not provided. All programs using CMAM should (assuming U.C. Berkeley installation):

include #**include <cmam/cmam.h>**
be compiled with **gcc-2.1 -I/usr/cm5/local/include . . .**
be linked with **cmld -gld2 -L/usr/cm5/local/lib . . . -pe -L/usr/cm5/local/lib**
(The **-gld2** option to **cmld** loads the library necessary for programs compiled with **gcc-2.x**.)

SEE ALSO

CMAM_enable(3), **CMAM**(3), **CMAM_4**(3), **CMAM_indirect**(3), **CMAM_indirect_4**(3), **CMAM_open_segment**(3), **CMAM_xfer**(3), **CMAM_xfer_N**(3), **CMAM_xfer_4i**(3), **CMAM_barrier**(3), **CMAM_set_tags**(3), **CMAM_host**(3), **CMAM_host_4**(3), **CMAM_host_indirect**(3), **CMAM_host_indirect_4**(3), **CMAM_host_xfer**(3), **CMAM_host_xfer_N**(3), **CMAM_host_xfer_4i**(3),

Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser, *Active Messages: a Mechanism for Integrated Communication and Computation*. Proc. of the 19th Int'l Symposium on Computer Architecture, Gold Coast, Australia. May 1992. (Also available as Technical Report UCB/CSD 92/675, CS Div., University of California at Berkeley.)

AUTHOR

Thorsten von Eicken, tve@CS.Berkeley.EDU

NAME

CMAM_enable, CMAM_disable – CMAM initialization

SYNOPSIS

#include <cmam/cmam.h>

void CMAM_enable (void);

void CMAM_disable (void);

DESCRIPTION

CMAM_enable must be called on the host (‘‘control processor’’ in CM-5 terminology) before any other CMAM functions are used. It runs an initialization function on all processing nodes to set the default packet tag assignments (see **CMAM_set_tags**(3)) and to initialize other CMAM data structures.

CMAM_enable does not change the state of the network interface and therefore does not conflict with other message layers. Note that the CMAM barrier functions (e.g. **CMAM_barrier**(3)) require separate initialization (conflicting with CMMD global operations) performed by **CMAM_enable_barrier**(3).

CMAM_disable runs an echo function on all processing nodes to ensure that they are all in the ‘‘idle loop’’.

SEE ALSO

cmam(3), **CMAM_set_tags**(3), **CMAM_enable_barrier**(3), **CMAM**(3).

AUTHOR

Thorsten von Eicken, tve@CS.Berkeley.EDU

NAME

CMAM, CMAM_reply – Send Active Message

SYNOPSIS

#include <cmam/cmam.h>

typedef void CMAM_handler (void *buffer, int byte_count);

void CMAM (int node, CMAM_handler *fun, void *buffer, int byte_count);

void CMAM_reply (int node, CMAM_handler *fun, void *buffer, int byte_count);

void CMAM_set_size (int max_buffer_size);

DESCRIPTION

CMAM sends an Active Message to the remote **node**, calling the handler **fun** and passing the **buffer** as argument. The call

CMAM(node, foo, buf, buf_len);

will allocate a buffer of size **buf_len** bytes on the remote **node**, transfer the source buffer into the allocated buffer packing 12 bytes per CM-5 network packet, invoke

foo(*allocated_buffer***, buf_len)**

and finally deallocate the buffer. Note that since the buffer is deallocated after the handler returns all data must be copied to its final destination. Transferring large amounts of data is best performed using **CMAM_xfer**(3) which avoids the copy and packs more data into each CM-5 packet.

At initialization time a small number (typ. 16) of buffers of size **CMAM_SZ** are pre-allocated for **CMAM**. If needed, additional buffers are allocated using **malloc**(3). The current implementation limits the message size to the size of the pre-allocated buffers. **CMAM_set_size** may be used to change the size of the buffers used. **CMAM_set_size** must be called while no messages are partially received.

CMAM sends the Active Message on the left data network and polls both, the left and right networks. **CMAM_reply** is identical to **CMAM**, but uses the right data network and polls only the right side. **CMAM_reply** should be used within message handlers for replies to avoid the danger of stack overflow (see **cmam**(3)). Neither **CMAM** nor **CMAM_reply** may be used from within a reply-message handler.

CMAM may be used to send Active Messages from the host (''control processor'' in CM-5 terminology) to the processing nodes. However, note that the executable running on the host is different from the one running on the processing nodes, therefore the address of functions differ. The linker **cmld**(1) arranges for functions named **CMPE_**... or **cmpe_**... on the processing nodes to be resolved correctly on the host. To send Active Messages to the host from processing nodes, see **CMAM_host**(3).

PERFORMANCE

CMAM is built on top of **CMAM_4**(3) and sends three 32-bit words of data per message.

SEE ALSO

cmam(3), **CMAM_xfer**(3), **CMAM_host**(3), **CMAM_4**(3).

AUTHOR

Thorsten von Eicken, tve@CS.Berkeley.EDU

NAME

CMAM_open_segment, CMAM_open_this_segment, CMAM_shorten_segment, CMAM_query_segment, CMAM_kill_segment – Manipulate data transfer segments

SYNOPSIS

#include <cmam/cmam.h>

typedef int CMAM_end_xfer (void *info, void *base_address);

int CMAM_open_segment (void *base_address, int byte_count,
CMAM_end_xfer *end_xfer_fun, void *info);

int CMAM_open_this_segment (int segment_num, void *base_address, int byte_count,
CMAM_end_xfer *end_xfer_fun, void *info);

void CMAM_shorten_segment (int segment_id, unsigned int delta_count);

void CMAM_kill_segment (int segment_id);

int CMAM_query_segment (int segment_id);

DESCRIPTION

CMAM segments are used in conjunction with **CMAM_xfer**(3) and its many variants to transfer bulk data between nodes. A segment is opened by the recipient of the data using **CMAM_open_segment**, specifying a **base_address** relative to which received data is stored and the number of bytes to be received. **CMAM_open_segment** returns a segment identifier for the highest free segment or -1 if no segments are available (note that segment identifiers can be negative, thus check for ''!= -1''). After **byte_count** bytes have been received, the **end_xfer_fun** is called with **base_address** and **info** as arguments. The **end_xfer_fun** should return a new **byte_count** or 0 if the segment is to be closed.

CMAM_open_this_segment is similar to **CMAM_open_segment**, but specifies which segment should be opened. It returns -1 if the requested segment is already in use. **CMAM_open_this_segment** can be used in all-to-all communication to open the same segment on all nodes. **CMAM_open_segment** always allocates the highest free segment, so it is best to use low segment numbers with **CMAM_open_this_segment**.

Notes: The segment identifiers returned by **CMAM_open_segment** and **CMAM_open_this_segment** encode the segment number and the alignment of the **base_address**. More precisely: **segment_id** = (**segment_num** << **CMAM_SEG_SHIFT**) | (**base_address** & 7).

Currently, there are 256 segments available.

The **base_address** of closed segments is set to -1. As a result, transferring data to a closed segment typically raises an unaligned-store exception inside of a CMAM handler.

If **CMAM_open_segment** is called with a 0 **byte_count**, **end_xfer_fun** is called immediately.

CMAM_shorten_segment reduces the remaining byte count of the specified segment. If the count becomes ≤0, **end-of-xfer** is called. **CMAM_shorten_segment** is useful, for example, when the sender cannot provide as much data as the recipient asked for.

CMAM_query_segment returns the remaining byte count of the specified segment. It returns 0 for a closed segment.

CMAM_kill_segment closes the segment *without* calling the end of transfer function. Note that due to the reordering of packets in the network it is only safe to call CMAM_kill_segment on a segment for which it is known that no packets are underway.

PERFORMANCE

The overhead in opening and closing a segment (including the call to **end-of-xfer**) is about 10us.

SEE ALSO

cmam(3), **CMAM_xfer**(3), **CMAM_xfer_4**(3), **CMAM**(3).

AUTHOR

Thorsten von Eicken, tve@CS.Berkeley.EDU

NAME

CMAM_xfer, CMAM_reply_xfer – Transfer data block to segment

SYNOPSIS

#include <cmam/cmam.h>

void CMAM_xfer (int node, int seg_addr, void *buff, int byte_count);

void CMAM_reply_xfer (int node, int seg_addr, void *buff, int byte_count);

DESCRIPTION

CMAM_xfer transfers the block of data at address **buff** and of length **byte_count** to a segment on the remote **node** previously opened with **CMAM_open_segment**(3). The destination address for the data on **node** is specified as a segment plus an unsigned byte offset. The segment and the offset are encoded in **seg_addr** which is the sum ('+') of the remote segment id (as returned by **CMAM_open_segment**) and the unsigned byte offset. This encoding limits the offset to 24 bits.

On arrival at the destination node, the data is stored in memory and the byte count associated with the segment is decremented. If the count becomes zero, the end-of-xfer function is called as described in **CMAM_open_segment**(3).

CMAM_xfer sends packets on the left data network and polls both, the left and right networks. **CMAM_reply_xfer** is identical to **CMAM_xfer** but sends packets on the right data network and polls only the right side. **CMAM_reply_xfer** should be used within message handlers to send replies to avoid the danger of stack overflow (see **cmam**(3)).

CMAM_xfer can handle arbitrary block sizes and aligments of source and destination buffers. **CMAM_xfer** transfers most of the block in special XFER packets with 16 bytes of data each. These packets must be stored at a double-word boundary on the destination node. The pre-amble and post-amble necessary to reach this aligment is handled by **CMAM_xfer**.

PERFORMANCE

If the source and the destination buffers have the same aligment relative to double-words (i.e. (src_buff&7) == ((base_address+offset)&7)) most of the data is transferred with **CMAM_xfer_N** at 10Mb/s.

If the aligments differ by a word (i.e. (src_buff^(base_address+offset))&7 == 4) most of the block is transferred by **CMAM_xfer_No** at 7Mb/s.

If the source and destination buffers are badly mis-aligned (i.e. (src_buff^(base_address+offset))&3 != 0) the transfer rate is low...

SEE ALSO

cmam(3), **CMAM_open_segment**(3), **CMAM_xfer_N**(3), **CMAM_xfer_4**(3), **CMAM**(3).

AUTHOR

Thorsten von Eicken, tve@CS.Berkeley.EDU

NAME

CMAM_wait, CMAM_poll_wait, CMAM_poll, CMAM_request_poll, CMAM_reply_poll – Explicit check for pending messages

SYNOPSIS

#include <cmam/cmam.h>

void CMAM_wait(volatile int *flag, int value);

void CMAM_poll_wait(volatile int *flag, int value);

void CMAM_poll(void);

void CMAM_request_poll(void);

void CMAM_reply_poll(void);

DESCRIPTION

The current implementation of CMAM does not use interrupts on message arrival, thus, to receive messages, the network interface status register must be polled periodically. All CMAM functions sending messages poll the network so that a node sending messages automatically handles incoming ones. The most common situation requiring explicit polling is when a node busy-waits for the reception of specific messages (e.g. signalling the completion of a communication step). For this purpose, **CMAM_wait** polls both (left and right) data networks while waiting on a synchronization **flag** to reach **value** (i.e. ***flag** ≥ **value**), then subtracts **value** from **flag** before returning. (This scheme supposes that, in general, handlers increment a flag to signal when they have executed.) The difference between **CMAM_wait** and **CMAM_poll_wait** is that the former does not poll at all if ***flag** ≥ **value** at entry, while the latter polls at least once.

Other wait mechanisms may be implemented using **CMAM_poll** which polls both data networks, handling all pending messages. **CMAM_request_poll** and **CMAM_reply_poll** are similar, but poll only one network.

PERFORMANCE

SEE ALSO

cmam(3), **CMAM**(3), **CMAM_xfer**(3).

AUTHOR

Thorsten von Eicken, tve@CS.Berkeley.EDU

NAME

CMAM_enable_barrier, CMAM_disable_barrier, CMAM_start_barrrier, CMAM_end_barrier, CMAM_barrier – Global barrier synchronization

SYNOPSIS

#include <cmam/cmam.h>

void CMAM_enable_barrier (void);

void CMAM_disable_barrier (void);

void CMAM_start_barrier (int bit);

int CMAM_end_barrier (void);

int CMAM_query_barrier (void);

int CMAM_barrier (int bit);

DESCRIPTION

These functions provide access to the ‘‘synchronous global OR’’ of the CM-5 control network in the form of a fuzzy barrier. Before using the CMAM barrier functions, the network interface must be initialized from the host (‘‘control processor’’ in CM-5 terminology) by calling **CMAM_enable_barrier**. Note that this conflicts with the initialization required by CMMD’s global operations (CMAM turns host participation off whereas CMMD turns it on by default). **CMAM_disable_barrier** reverts the initialization performed by **CMAM_enable_barrier**.

Calling **CMAM_start_barrier** signals to all other nodes that the current node ‘‘entered the barrier’’. A call to **CMAM_end_barrier** blocks until all nodes have called **CMAM_start_barrier**. While busy-waiting on the barrier, **CMAM_end_barrier** polls and services the network.

CMAM_query_barrier returns true if all nodes have called **CMAM_start_barrier** (i.e. if a call to **CMAM_end_barrier** would not block). **CMAM_query_barrier** does not poll the network.

CMAM_barrier provides a traditional barrier: it blocks until all other nodes call **CMAM_barrier** as well.

Both, fuzzy and traditional barriers provide a one-bit global OR in addition to the synchronization. When starting the barrier, each processor injects a 1-bit value (the LSB of **bit**). When ending the barrier, all processors receive the global OR of all 1-bit values.

Note that the host is not involved in any of these barriers.

PERFORMANCE

A call to **CMAM_barrier** when all processors are already in synch takes 5us.

SEE ALSO

cmam(3).

AUTHOR

Thorsten von Eicken, tve@CS.Berkeley.EDU

NAME

CMAM_4, CMAM_reply_4 – Send single-packet Active Messages

SYNOPSIS

#include <cmam/cmam.h>

void CMAM_4 (int node, void (*fun)(), ...);

void CMAM_reply_4 (int node, void (*fun)(), ...);

DESCRIPTION

CMAM_4 sends a single-packet Active Message to the remote **node**, calling the handler **fun** and passing up to 4 32-bit word arguments. The call

CMAM(node, foo, i1, i2, i3, i4);

sends a single CM-5 network packet to **node** and causes

foo(i1, i2, i3, i4);

to be invoked. **CMAM_4** is declared as a varargs function, but really takes up to four words of arguments in addition to **node** and **fun**, and passes them into the remote function. Due to SPARC calling conventions, these arguments can be any combination of integer and floating-point values (i.e., passed in registers i2 through i5). Note that **CMAM_4** always transmits a full CM-5 packet, thus there is no performance advantage in passing less than four words of arguments to the remote function.

CMAM_4 sends the Active Message on the left data network and polls both, the left and right networks. **CMAM_reply_4** is identical to **CMAM_4**, but uses the right data network and polls only the right side. **CMAM_reply_4** should be used within message handlers for replies to avoid the danger of stack overflow (see **cmam**(3)). Neither **CMAM_4** nor **CMAM_reply_4** may be used from within a reply-message handler.

CMAM may be used to send Active Messages from the host (''control processor'' in CM-5 terminology) to the processing nodes. However, note that the executable running on the host is different from the one running on the processing nodes, therefore the address of functions differ. The linker **cmld**(1) arranges for functions named **CMPE_**... or **cmpe_**... on the processing nodes to be resolved correctly on the host. To send Active Messages to the host from processing nodes, see **CMAM_host**(3).

PERFORMANCE

CMAM_4 takes 2us, **CMAM_reply_4** takes 1.6us. Handling either one at arrival takes 2us.

SEE ALSO

CMAM(3) extends **CMAM_4** in that it sends an Active Message of arbitrary length, passing a buffer as argument. **CMAM_indirect_4**(3) is similar to **CMAM_4** but may cram an additional word of information into the CM-5 packet in certain circumstances.

cmam(3), **CMAM**(3), **CMAM_xfer**(3), **CMAM_host**(3), **CMAM_indirect_4**(3).

AUTHOR

Thorsten von Eicken, tve@CS.Berkeley.EDU

NAME

CMAM_xfer_N, CMAM_reply_xfer_N, CMAM_xfer_No, CMAM_reply_xfer_No, CMAM_xfer_Nb, CMAM_reply_xfer_Nb – Transfer aligned data block to segment

SYNOPSIS

#include <cmam/cmam.h>

void CMAM_xfer_N (int node, int seg_addr, void *buff, int byte_count);

void CMAM_xfer_No (int node, int seg_addr, void *buff, int byte_count);

void CMAM_xfer_Nb (int node, int seg_addr, void *buff, int byte_count);

void CMAM_reply_xfer_N (int node, int seg_addr, void *buff, int byte_count);

void CMAM_reply_xfer_No (int node, int seg_addr, void *buff, int byte_count);

void CMAM_reply_xfer_Nb (int node, int seg_addr, void *buff, int byte_count);

DESCRIPTION

CMAM_xfer_N transfers the block of *quadwords* at address **buff** and of length **byte_count** to a segment on the remote **node** previously opened with **CMAM_open_segment**(3). The destination address for the data on **node** is specified as a segment plus an unsigned byte offset. The segment and the offset are encoded in **seg_addr** which is the sum ('+') of the remote segment id (as returned by **CMAM_open_segment**) and the unsigned byte offset. This encoding limits the offset to 24 bits.

On arrival at the destination node, the data is stored in memory and the byte count associated with the segment is decremented. If the count becomes zero, the end-of-xfer function is called as described in **CMAM_open_segment**(3).

On arrival at the destination node, the data is stored in memory and the byte count associated with the segment is decremented. If the count becomes zero, the end-of-xfer function is called as described in **CMAM_open_segment**(3).

CMAM_xfer_N sends packets on the left data network and polls both, the left and right networks. **CMAM_reply_xfer_N** is identical to **CMAM_xfer_N** but sends packets on the right data network and polls only the right side. **CMAM_reply_xfer_N** should be used within message handlers to send replies to avoid the danger of stack overflow (see **cmam**(3)).

CMAM_xfer_N requires the source and the destination block the be doubleword aligned (i.e. (addr&7) == 0) and transfers floor(**byte_count**/16) quadword packets.

CMAM_xfer_No and **CMAM_reply_xfer_No** are similar to **CMAM_xfer_N** respectively **CMAM_reply_xfer_N** but require only word-aligment of the source data block (i.e. (**buff**&3) == 0).

CMAM_xfer_Nb and **CMAM_reply_xfer_Nb** transfer up to 8 bytes of any aligment to a remote segment.

All six function described in this man page are intended to be building blocks for more convenient block transfer functions such as **CMAM_xfer**(3).

PERFORMANCE

CMAM_xfer_N transfers data at up to 10Mb/s. A node sending and receiving xfers at the same time typically sends at 5Mb/s and receives at 5Mb/s. A node receiving two xfers to different segments accepts data at about 7.5Mb/s due to overhead incurred when switching between the two segments.

CMAM_xfer_No transfers data at most at 7Mb/s.

SEE ALSO

cmam(3), **CMAM_open_segment**(3), **CMAM_xfer**(3), **CMAM_xfer_4**(3), **CMAM**(3)

AUTHOR

Thorsten von Eicken, tve@CS.Berkeley.EDU

NAME

CMAM_xfer_[1234]i, CMAM_reply_xfer_[1234]i, CMAM_xfer_[12]d, CMAM_reply_xfer_[12]d – Transfer single data packet to segment

SYNOPSIS

#include <cmam/cmam.h>

void CMAM_xfer_1i (int node, int segment_id, unsigned int offset, int d1);

void CMAM_xfer_2i (int node, int segment_id, unsigned int offset, int d1, int d2);

void CMAM_xfer_3i (int node, int segment_id, unsigned int offset, int d1, int d2, int d3);

void CMAM_xfer_4i (int node, int segment_id, unsigned int offset, int d1, int d2, int d3, int d4);

void CMAM_xfer_1d (int node, int segment_id, unsigned int offset, double d1);

void CMAM_xfer_2d (int node, int segment_id, unsigned int offset, double d1, double d2);

void CMAM_reply_xfer_1i (int node, int segment_id, unsigned int offset, int d1);

void CMAM_reply_xfer_2i (int node, int segment_id, unsigned int offset, int d1, int d2);

void CMAM_reply_xfer_3i (int node, int segment_id, unsigned int offset, int d1, int d2, int d3);

void CMAM_reply_xfer_4i (int node, int segment_id, unsigned int offset, int d1, int d2, int d3, int d4);

void CMAM_reply_xfer_1d (int node, int segment_id, unsigned int offset, double d1);

void CMAM_reply_xfer_2d (int node, int segment_id, unsigned int offset, double d1, double d2);

DESCRIPTION

These functions transfer one CM-5 packet with 1 to 4 32-bit words of data to a segment on the remote **node** previously opened with **CMAM_open_segment**(3). The destination address for the data on **node** is specified as a segment plus an unsigned byte offset: **segment_id** (as returned by **CMAM_open_segment**) specifies the segment and **offset** specifies an unsigned 24-bit byte offset within the segment.

On arrival at the destination node, the data is stored in memory and the byte count associated with the segment is decremented. If the count becomes zero, the end-of-xfer function is called as described in **CMAM_open_segment**(3).

The **CMAM_xfer_**∗ functions send packets on the left data network and poll both, the left and right networks. The **CMAM_reply_xfer_**∗ functions are identical, but send packets on the right data network and poll only the right network. **CMAM_reply_xfer_**∗ should be used within message handlers to send replies to avoid the danger of stack overflow (see **cmam**(3)).

CMAM_xfer_4i and **CMAM_xfer_2d** require the destination to be doubleword aligned (i.e. (addr&7) == 0). The other functions transfering less than 4 32-bit words require word alignment of the destination.

PERFORMANCE

SEE ALSO

cmam(3), **CMAM_open_segment**(3), **CMAM_xfer**(3), **CMAM_xfer_N**(3), **CMAM**(3).

AUTHOR

Thorsten von Eicken, tve@CS.Berkeley.EDU

NAME

CMAM_host, CMAM_host_reply, CMAM_host_4, CMAM_host_reply_4 – Send Active Messages to host

SYNOPSIS

#include <cmam/cmam.h>

void CMAM_host (void (*fun)(), void *buffer, int byte_count);

void CMAM_host_reply (void (*fun)(), void *buffer, int byte_count);

void CMAM_host_4 (void (*fun)(), ...);

void CMAM_host_reply_4 (void (*fun)(), ...);

DESCRIPTION

CMAM_host, **CMAM_host_reply**, **CMAM_host_4**, and **CMAM_host_reply_4** are similar to **CMAM**(3), **CMAM_reply**(3), **CMAM_4**(3), respectively **CMAM_reply_4**(3), but send an Active Message to the host processor (''control processor'' in CM-5 terminology) instead of a processing node.

Note that the executable running on the host is different from the one running on the processing nodes and therefore the address of functions differ. While the linker **cmld**(1) arranges for functions named **CMPE_**... or **cmpe_**... on the processing nodes to be resolved correctly on the host, no similar symbol resolution is provided the other way around. One solution is to broadcast the address of all host handlers to the processing nodes at the beginning of a program. Alternatively, a table of handlers may be set-up on the host, its base address broadcast to the processing nodes and **CMAM_host_indirect**(3) used to send Active Messages to the host. Finally, the **cmld** shell script could be modifed to include resolution of host symbols within the processing node executable.

PERFORMANCE

These functions are very slow because an O.S. trap is required for every packet.

SEE ALSO

cmam(3), **CMAM**(3), **CMAM_4**(3), **CMAM_indirect**(3).

AUTHOR

Thorsten von Eicken, tve@CS.Berkeley.EDU

NAME

CMAM_host_xfer, CMAM_host_reply_xfer – Transfer data block to segment on host

SYNOPSIS

#include <cmam/cmam.h>

void CMAM_host_xfer (int seg_addr, void *buff, int byte_count);

void CMAM_host_reply_xfer (int seg_addr, void *buff, int byte_count);

DESCRIPTION

CMAM_host_xfer and **CMAM_host_reply_xfer** are similar to **CMAM_xfer**(3) respectively **CMAM_reply_xfer**(3) but transfer data to the host processor (''control processor'' in CM-5 terminology) instead of a processing node.

PERFORMANCE

These functions are very slow because an O.S. trap is required for every packet.

SEE ALSO

cmam(3), **CMAM_xfer**(3), **CMAM_host**(3).

AUTHOR

Thorsten von Eicken, tve@CS.Berkeley.EDU

NAME

CMAM_host_xfer_N, CMAM_reply_host_xfer_N, CMAM_host_xfer_No, CMAM_reply_host_xfer_No, CMAM_host_xfer_Nb, CMAM_reply_host_xfer_Nb – Transfer aligned data block to segment on host

SYNOPSIS

#include <cmam/cmam.h>

void CMAM_host_xfer_N (int seg_addr, void *buff, int byte_count);

void CMAM_host_xfer_No (int seg_addr, void *buff, int byte_count);

void CMAM_host_xfer_Nb (int seg_addr, void *buff, int byte_count);

void CMAM_host_reply_xfer_N (int seg_addr, void *buff, int byte_count);

void CMAM_host_reply_xfer_No (int seg_addr, void *buff, int byte_count);

void CMAM_host_reply_xfer_Nb (int seg_addr, void *buff, int byte_count);

DESCRIPTION

CMAM_host_xfer_N, **CMAM_host_xfer_No**, **CMAM_host_xfer_Nb**, **CMAM_host_reply_xfer_N**, **CMAM_host_reply_xfer_No** and **CMAM_host_reply_xfer_Nb** are similar to **CMAM_xfer_N**(3), **CMAM_xfer_No**(3), **CMAM_xfer_Nb**(3), **CMAM_reply_xfer_N**(3), **CMAM_reply_xfer_No**(3) respectively **CMAM_reply_xfer_Nb**(3) but transfer data to the host processor (''control processor'' in CM-5 terminology) instead of a processing node.

PERFORMANCE

These functions are very slow because an O.S. trap is required for every packet.

SEE ALSO

cmam(3), **CMAM_xfer_N**(3), **CMAM_host**(3).

AUTHOR

Thorsten von Eicken, tve@CS.Berkeley.EDU

NAME

CMAM_host_xfer_[1234]i, CMAM_reply_host_xfer_[1234]i, CMAM_host_xfer_[12]d, CMAM_reply_host_xfer_[12]d – Transfer single data packet to segment on host

SYNOPSIS

#include <cmam/cmam.h>

void CMAM_host_xfer_1i (int segment_id, unsigned int offset, int d1);

void CMAM_host_xfer_2i (int segment_id, unsigned int offset, int d1, int d2);

void CMAM_host_xfer_3i (int segment_id, unsigned int offset, int d1, int d2, int d3);

void CMAM_host_xfer_4i (int segment_id, unsigned int offset, int d1, int d2, int d3, int d4);

void CMAM_host_xfer_1d (int segment_id, unsigned int offset, double d1);

void CMAM_host_xfer_2d (int segment_id, unsigned int offset, double d1, double d2);

void CMAM_host_reply_xfer_1i (int segment_id, unsigned int offset, int d1);

void CMAM_host_reply_xfer_2i (int segment_id, unsigned int offset, int d1, int d2);

void CMAM_host_reply_xfer_3i (int segment_id, unsigned int offset, int d1, int d2, int d3);

void CMAM_host_reply_xfer_4i (int segment_id, unsigned int offset, int d1, int d2, int d3, int d4);

void CMAM_host_reply_xfer_1d (int segment_id, unsigned int offset, double d1);

void CMAM_host_reply_xfer_2d (int segment_id, unsigned int offset, double d1, double d2);

DESCRIPTION

CMAM_host_xfer_*[1..4]*__i__, **CMAM_host_xfer**_*[1..2]*__d__, **CMAM_host_reply_xfer**_*[1..4]*__i__ and **CMAM_host_reply_xfer**_*[1..2]*__d__ are similar to **CMAM_xfer**_*[1..4]*__i__(3), **CMAM_xfer**_*[1..2]*__d__(3), **CMAM_reply_xfer**_*[1..4]*__i__(3) respectively **CMAM_reply_xfer**_*[1..2]*__d__(3) but transfer data to the host processor (‘‘control processor’’ in CM-5 terminology) instead of a processing node.

PERFORMANCE

These functions are very slow because an O.S. trap is required for every packet.

SEE ALSO

cmam(3), **CMAM_xfer**(3), **CMAM_host**(3).

AUTHOR

Thorsten von Eicken, tve@CS.Berkeley.EDU

NAME

CMAM_set_tags, CMAM_set_handler, CMAM_set_reply_handler – Control hardware message tag usage

SYNOPSIS

#include <cmam/cmam.h>

void CMAM_set_tags (int CMAM_tag, int CMAM_indirect_tag, int CMAM_xfer_tag);

typedef void CMAM_tag_handler (void *NI_base);

void CMAM_set_handler (int tag, CMAM_tag_handler *handler);

void CMAM_set_reply_handler (int tag, CMAM_tag_handler *handler);

DESCRIPTION

CMAM has been designed such that it can be mixed with other message layers or extended by the user: CMAM can be directed to use any of the available hardware packet tags and handlers for non-CMAM packets can be registered with CMAM.

The CMAM initialization function **CMAM_enable**(3) sets CMAM up to use hardware tags 1, 2, and 3 for CMAM, CMAM_indirect, and CMAM_xfer, respectively, packets. After initialization, these tag assignments can be changed by calling **CMAM_set_tags** on the host (''control processor'' in CM-5 terminology) while the processing nodes are idle. As of version 7.1.4 of CMOST, only tags 0 through 3 are available to user message handlers. The remaining tags 4 through 15 are reserved (but not all used) by CMOST. Please complain to Thinking Machines Corp. for not leaving more tags to the user!

For multiple message layers to operate concurrently, a certain amount of coordination in handling incoming packets is necessary. Whenever CMAM polls the network and discovers the arrival of a packet, it dispatches to the appropriate tag-handler based on the hardware packet tag. The dispatch is performed by branching into a table indexed by the tag, each table ''entry'' holding 64 SPARC instructions. **CMAM_set_handler** takes a pointer to a tag-handler and copies 64 instructions into the appropriate entry in the left data network table. Similarly, **CMAM_set_reply_handler** copies into the right data network table. While copying the instructions, both functions adjust the PC-relative offsets of CALL instructions. The offsets of branch instructions are not modified (the offsets are too small to be adjustable in most cases). When called, the tag-handler receives the base address of the Network Interface chip (e.g. 0x20000000 on the PNs) as argument.

CMAM does not export any of its handlers for other message layers to call when they receive a CMAM packet. Several existing functions could be used, please contact the author if you need this type of functionality.

SEE ALSO

cmam(3), **CMAM_enable**(3), **CMAM**(3), **CMAM_indirect**(3), **CMAM_xfer**(3), **CMAM_host**(3).

AUTHOR

Thorsten von Eicken, tve@CS.Berkeley.EDU