Appendix C

Extending CMAML

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

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

Note: CMNA is a low-level macro set used for programming the NI chip. It is *not* intended for general use. CMNA is documented in *Programming the NI*, Version 7.1, available on request from Thinking Machines Corporation.

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

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

C.1 Hardware Tag Allocation and Registration

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

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

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

C.2 Hardware Tag Handler Functions

After allocating a hardware tag, programmers are responsible for "installing" or "registering" a hardware tag handler function for that tag with CMMD. A tag handler function is installed for each DR fifo. These handler functions may be any SPARC ABI–compliant function or leaf-routine, providing they at least read the message from the head of the appropriate DR fifo. Further interpretation of the message content is left to the handler.

Hardware tag handlers may be classifed into two groups: handlers that don't reply (those that process an arriving packet and return to the previous thread of computation), and handlers that do reply (those that process an arriving packet and in turn attempt to send or receive additional DR messages). It is the programmer's responsibility to handle all issues of critical sections, re-entrancy, nesting of handlers, queueing, etc.

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

C.3 CMAML Extension Functions

C.3.1 Allocating and Freeing Tags

int **`CMAML_allocate_tag()`**

Attempts to allocate a hardware tag for the caller. Returns a tag number of 0 to 7, inclusive, if successful; returns a –1 if no tags were available.

Currently, CMOST allows eight (8) hardware tags to be allocated by user-level software. In its current implementation, CMMD allocates the lowest 3 or 4 of these tags before invoking the user's application; it needs to be able to allocate one further tag the first time the application performs parallel I/O.

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

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

As explained above, CMMD allocates the lowest-numbered 3 or 4 tags (from the range 0–7, inclusive) for its own use, and needs to allocate one more if and when the application performs its first parallel I/O.

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

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

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

C.3.2 Registering Hardware Tag Handlers

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

These functions register the specified hardware tag handler functions for the LDR and RDR fifos, respectively, for the specified tag. (The LDR and the RDR are the two "halves" of the Data Network. They are used selectively by CMAML, as explained in Section 11.5.)

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

C.3.3 Processing Messages

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

Once a tag has been allocated and its handler set, the **CMAML_get_ldr** and **CMAML_get_rdr** functions may be called to process any message arriving via the specified Data Network interface.

The functions assume that

- **ni_base_addr** contains the NI base address, and **ldr_fifo_status** (or **rdr_fifo_status**) contains the current NI LDR (or RDR) fifo status.
- the **RECEIVE_OK** bit in the status word is a 1.

If these conditions are met, calling one of these functions will in turn invoke the pre-registered hardware tag handler for the LDR (or RDR) fifo.

C and assembly programs can call this function. For C, one typical code fragment would look like this:

```
#include <cm/cmna.h>
...
ldr_fifo_status = CMNA_ldr_status();
if(RECEIVE_OK(ldr_fifo_status))
    CMAML_get_ldr(NI_BASE, ldr_fifo_status);
```

Another typical code fragment might look like this:

```
#include <cm/cmna.h>
...
ldr_fifo_status = CMNA_ldr_status();
if(RECEIVE_OK(ldr_fifo_status)) {
    if(USER_TAG(ldr_fifo_status) == my_tag) {
    addr = CMNA_ldr_receive();
    dw1  = CMNA_ldr_receive_double();
    dw2  = CMNA_ldr_receive_double();
```

```
        <process data accordingly>
    } else /* not my tag */
            CMAML_get_ldr(NI_BASE, ldr_fifo_status);
    }
```

This second example demonstrates that you don't need to call the **CMAML_ldr_xdr** functions to process your own packets (i.e., packets that you recognize), but that you must do so for packets that you do not recognize (i.e., packets without your private tags).

C programs (but not assembly programs) can also use CMNA's **NI_BASE** symbol to get the NI base address, as follows:

```
#include <cm/cmna.h>
unsigned *ni_base_addr = ((unsigned *) NI_BASE);
```