Appendix A

Extending CMAML

This 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 (for the CM-5) and the *NI Programmer's Handbook*, NI Version 2.2 (for the CM-5E). These manuals are 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

Copyright © 1995 Thinking Machines Corporation

A.1 Hardware Tag Allocation and Registration

CMAML provides functions for *allocating* (i.e., obtaining 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.

A.2 Hardware Tag Handler Functions

After allocating a hardware tag, programmers are responsible for *registering* a handler function for that tag with CMMD. A tag handler function is registered for each DR FIFO. These handler functions may be any SPARC ABI-compliant function or leaf routine, provided that they at least read the entire current message from the appropriate DR FIFO. Further interpretation of the message content is left to the handler.

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.

Version 3.3, April 1995
Copyright © 1995 Thinking Machines Corporation

A.3 CMAML Extension Functions

A.3.1 Allocating and Freeing Tags

The function

```
int CMAML_allocate_tag()
```

attempts to allocate a hardware tag for the caller. It returns a tag from 0 to 7, inclusive, if successful; it returns −1 if no tags were available.

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

The function

```
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.

The function

```
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 unpredictable results.

A.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 selec-

Copyright © 1995 Thinking Machines Corporation

tively by CMAML, as explained in Section 12.2.1 of the *CMMD Reference Manual* for Version 3.3.

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.

PLEASE NOTE

On the CM-5, the *handler* argument to `CMAML_set_[l|r]dr_handler` must point to a function containing *no more than 64 SPARC assembly instructions.* If you are writing your handlers directly in assembly code, then you can easily count the number of instructions to ensure that they do not exceed this limit. If your handlers are compiled, then you should disassemble their code and count the instructions. If the length of your handler exceeds 64 instructions, then there is a simple workaround: simply install a "stub" that does nothing but call your handler. (Such a stub will not contain more than a handful of instructions.)

Failure to observe this restriction will result in a fatal error.

Note that on the CM-5E there is no such limit. Hardware tag handlers can be arbitrarily long.

A.3.3 Processing Messages

Some aspects of CMNA programming are different between the CM-5 and the CM-5E. In particular, *CMNA code (that works in conjunction with CMMD) written for the CM-5 will not work on a CM-5E. It must be modified to use the special CM-5E functions described below*.

The relevant CMAML functions are

(for the CM-5):

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


Copyright © 1995 Thinking Machines Corporation

(for the CM-5E):

```
void CMAML_get_ldr_first(int ni_base_addr,
                         int ldr_fifo_status_all,
                         int first_word)
void CMAML_get_rdr_first(int ni_base_addr,
                         int rdr_fifo_status_all,
                         int first_word)
```

Once a tag has been allocated and its handler set, the `CMAML_get_[l|r]dr[_first]` functions may be called to process any message arriving via the specified Data Network interface.

The functions `CMAML_get_[l|r]dr` must be used on the CM-5 only. A fatal error will occur if they are used on a CM-5E. The functions assume that

- `ni_base_addr` contains the NI base address.
- `[l|r]dr_fifo_status` contains the current *NI-[L/R]DR_status* word, as obtained by calling `CMNA_[l|r]dr_status`. (You may bypass CMNA if you wish, and write directly in assembly code; however, this requires a detailed knowledge of the CMNA header files.)
- the `RECEIVE_OK` bit in the status word is set, i.e., there *is* a waiting message.

The functions `CMAML_get_[l|r]dr_first` must be used on the CM-5E only. A fatal error will occur if they are used on a CM-5. These functions assume that

- `ni_base_addr` contains the NI base address.
- `[l|r]dr_fifo_status_all` contains the current *NI-[L/R]DR-status-all* word, as obtained (for example) by calling `CMNA_mni_[l|r]dr_status_all_pop`. (You may bypass CMNA if you wish, and write directly in assembly code; however, this requires a detailed knowledge of the CMNA header files.) The *status-all* word is new to the CM-5E. It is a slightly different implementation of the CM-5's *status* word. The suffix "pop" means that if there is a message waiting, the function "pops" it from the NI. *You need not, and **must** not, ever pop the first word of a message by yourself. Rather, you should rely on the fact that on the CM-5E, the first word of every message is automatically popped as part and parcel of reading the* status-all *word.*

Copyright © 1995 Thinking Machines Corporation

- The RECEIVE_OK bit in the status-all word is set, i.e., there *is* a waiting message. On the CM-5E there is no ready-made macro analogous to the RECEIVE_OK that exists for the CM-5. On the CM-5E, the RECEIVE_OK bit happens to be the least significant bit, so you should include the following definition in your code:

  ```
  #define RECEIVE_OK_CM5E(status-all) ((status-all)&0x1)
  ```

  and use RECEIVE_OK_CM5E on the CM-5E wherever you would use RECEIVE_OK on the CM-5.

- The *first_word* argument contains the first word of the message, if there is a waiting message. Note that if there is a waiting message, the function CMNA_mni_[l|r]dr_status_all_pop will already have read it from the NI, at the same time that it read the status-all word. (This is one of the ways in which the CM-5E is more efficient than the CM-5.)

The CMNA functions CMNA_mni_[l|r]dr_status_all_pop return a double-precision floating-point value. The number is meaningless as a floating-point quantity. The first 32 bits of it represent the status-all word, and the second 32 bits represent the first word of the waiting message. Therefore, the standard idiom for writing code to handle CMAML packets in CMNA looks like this:

```
{
  int statall;
  double statall_and_first;

  statall_and_first = CMNA_mni_ldr_status_all_pop();
  statall = ((int *)&statall_and_first)[0];
  if ( RECEIVE_OK_CM5E( statall ) )
    CMAML_get_ldr_first(NI_BASE, statall,
                        ((int *)&statall_and_first)[1]);
      ...
      ...
      ...
}
```


Copyright © 1995 Thinking Machines Corporation

which is slightly more complicated than the corresponding idiom for the CM-5, which looks like this:

```
{
  int stat;

  stat = CMNA_ldr_status();
  if ( RECEIVE_OK( stat ) )
    CMAML_get_ldr(NI_BASE, stat);
      ...
      ...
      ...
}
```

What follows is a typical CMNA code fragment, written in the two different styles.

First, for the CM-5:

```
#include <cm/cmna.h>

stat = CMNA_ldr_status();

if ( RECEIVE_OK( stat ) ) {

  if( USER_TAG( stat ) ) == my_tag ) {

    w1   = CMNA_ldr_receive();
    dw23 = CMNA_ldr_receive_double();
    dw45 = CMNA_ldr_receive_double();

    <process data accordingly>

    } else { /* not my tag */
      CMAML_get_ldr(NI_BASE, stat);
  }
}
```


Copyright © 1995 Thinking Machines Corporation

Next, for the CM-5E (notice that in order to access the hardware tag, you must introduce the macro `USER_TAG_CM5E`, which does not currently exist in CMNA):

```
#include <cm/cmna.h>

#define USER_TAG_CM5E (statall) (((statall) & \
                        CMAML_NI_HWTAG_MASK_ALL)>>3)
  ...
  ...
  ...
statall_and_first =
  CMNA_mni_ldr_status_all_pop();
statall = ((int *)&statall_and_first)[0];
w1      = ((int *)&statall_and_first)[1];

if ( RECEIVE_OK_CM5E( statall ) ) {

  if( USER_TAG_CM5E( statall ) == my_tag ) {

     dw23  = CMNA_ldr_receive_double();
     dw45  = CMNA_ldr_receive_double();

     <process data accordingly>

  } else {  /* not my tag */
    CMAML_get_ldr_first(NI_BASE, statall, w1);
  }
}
```

These examples show that you do not need to call the `CMAML_get_[l|r]dr[_first]` 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 hardware tags). Of course, you could just as well install your handlers using `CMAML_set_[l|r]dr_handler`, and process them using `CMAML_get_[l|r]dr[_first]`.

A.3.4 Using Hardware Tag Handlers

Note that hardware tag handlers on Cypress platforms must be no longer than 64 machine instructions. If this restriction is violated, the results are unpredictable and most likely will result in a segmentation fault or a bus

Copyright © 1995 Thinking Machines Corporation

error. If you wish to register a hardware tag handler that is longer than 64 instructions, you can register a short stub handler that simply calls the desired handler. This restriction does not apply on SPARC superscalar platforms such as the CM-5E.

Also note that the interface to hardware tag handlers is subject to change without notice from one release to another. Programmers coding at this level should be prepared to modify their code in future releases.

Copyright © 1995 Thinking Machines Corporation