Chapter 14

CMAML: Handlers and Auxiliary Functions

This chapter covers two main topics. Section 14.1 discusses active message handler functions and asynchronous messaging callback functions. Section 14.2 discusses the auxiliary functions that support the use of active messages. These include functions for polling, for enabling and disabling interrupts, and for saving and restoring registers.

14.1 Handler Functions

Handler functions are user-defined functions that are invoked immediately upon receipt of an active message (see Chapter 12). *Callback functions* are user-defined functions that can be invoked

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

In most cases a handler or a callback function is executed upon completion of some asynchronous event, and not upon reaching a given line of code in a program. Thus, handlers and callbacks are executed at unpredictable points in the main thread of control (also called the *foreground computation*). If the user has disabled interrupts, handler and callback functions can execute only when certain CMAML or CMMD functions are called in the foreground computation; otherwise, they might occur at any point in the foreground program. The sender of an active message, for

Copyright © 1995 Thinking Machines Corporation

example, cannot in general know the line of code at which the receiver will execute the handler.

There are many ways in which the user code might limit this uncertainty, such as by disabling interrupts, or by synchronization using blocking message passing or global reductions. However, many CMAML programs are written without the need to limit the asynchrony, and sometimes even to take advantage of it.

Like active messages, callback functions may run at unpredictable times with respect to the foreground computation. Callbacks for `CMAML_scopy` and `CMAML_pcopy` are executed after the invoking operation has run to completion, but before the control of the program returns to the foreground.

A partial tabulation of possibilities follows:

Operation	Where Operation Is Invoked	When Its Callback Is Run
`CMAML_scopy`	Foreground	KNOWN (right before `scopy` returns)
`CMAML_pcopy`	Foreground	KNOWN (right before `pcopy` returns)
`CMAML_scopy`	Reply handler	N/A (illegal to call `scopy` in reply handler)
`CMAML_scopy`	Request handler	N/A (illegal to call non-copy communication function in request handler)
`CMAML_scopy`	RPC handler	UNKNOWN (but prior to reentering foreground)

Copyright © 1995 Thinking Machines Corporation

CMAML_pcopy	CMAML_scopy's callback	UNKNOWN (but prior to reentering foreground)
CMMD_send_async	CMAML_scopy's callback	UNKNOWN
CMMD_receive_async	CMMD_send_async's callback	UNKNOWN
CMMD_send_async	RPC handler	UNKNOWN
CMMD_receive_async	RPC handler	UNKNOWN

Keep in mind that CMMD_send_async, CMMD_receive_async, CMAML_scopy, CMAML_pcopy, and CMAML_rpc are themselves similar to callbacks, insofar as they are potentially scheduled for later execution. In a way, they are their own callbacks. A partial tabulation of the possibilities is

Operation	Where Operation Is Invoked	When Sending Takes Place
CMAML_scopy	Foreground	KNOWN (blocks until success)
CMAML_pcopy	Foreground	KNOWN (blocks until success)
CMAML_rpc	Foreground	KNOWN (blocks until success)
CMAML_rpc	scopy callback of an scopy called from Foreground	KNOWN (scopy blocks, its callback then executes, RPC blocks)
CMAML_scopy	Reply handler	N/A (illegal)
CMAML_scopy	Request handler	N/A (illegal)

Copyright © 1995 Thinking Machines Corporation

CMAML_scopy	RPC handler	UNKNOWN (but prior to reentering foreground)
CMAML_pcopy	CMAML_scopy's callback	UNKNOWN (but prior to reentering foreground)
CMMD_send_async	Anywhere	UNKNOWN (determined partially by the receiver)
CMMD_receive_async	Anywhere	UNKNOWN (determined partially by the sender)

Although active messages execute immediately and to completion once they are invoked at their destination node, the time at which they are invoked is always uncertain. There are several factors causing this. From the sender's point of view, the time of injection into the network is uncertain insofar as the sending function may be nonblocking or itself called from within a handler or callback. The time of traversal through the network is always uncertain, so that the knowledge that sending has taken place gives little information about when receiving will take place. Still more uncertainty is introduced by the receiving node itself, since it may or may not have interrupts enabled, or it may or may not be polling. All these factors should be kept in mind when using active messages.

If interrupts are enabled, then active message handlers can be invoked at unpredictable points in the midst of the foreground computation. Under some circumstances this can fatally corrupt the foreground computation (by overwriting unprotected registers), unless the handler is carefully written to avoid such corruption. Functions and guidelines for dealing with this problem are described below.

The combination of active messages and enabled interrupts can be a very powerful programming model, but it can also be very subtle and difficult to debug. You should give careful consideration to the interaction between the foreground computation, communication, and handlers or callback functions. This interaction is always entirely the user's responsibility.

Copyright © 1995 Thinking Machines Corporation

14.1.1 Properties and Restrictions of Handlers

The effects of a handler are determined not only by its context, but by the context in which it is called. One can write a routine that is sometimes called as a subroutine by the main thread of control, and at other times called as a handler. However, because of its potential ability to interrupt the background computation, any function intended for use as a handler has the following properties and is constrained by the following restrictions:

- Handlers may interrupt the background computation, but they themselves 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 must 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 (except for `CMAML_reply`).
- Handlers may not call CMMD I/O functions.
- If called from within the dynamic scope of the CMMD interrupt handler, handlers must save any SPARC FP and VU state that they modify and restore it before returning. (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 SPARC FP registers, VU data and control registers, and VU instruction extension registers. (See Section 14.2.)

Copyright © 1995 Thinking Machines Corporation

- Any function called by a handler must obey the same restrictions that the handler itself must obey.

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

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

PLEASE NOTE

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 a protocol. (In fact, one of the purposes of CMAML is to facilitate writing such protocols.)

The request functions (`CMAML_request`, `CMAML_request_to_host`, `CMAML_request_va`, `CMAML_request_vapop`, `CMAML_request_arrpop0`, `CMAML_request_arrpop1`, `CMAML_request_arrpop2`) are the only CMAML functions that can be called with interrupts enabled. All other CMAML functions require that interrupts be disabled and polling used instead.

Version 3.3, April 1995
Copyright © 1995 Thinking Machines Corporation

14.2 Active Message Auxiliary Functions

There are three types of auxiliary functions for active messages:

- Functions used for polling the Data Network while network interrupts are disabled:
  ```
  CMAML_poll
  CMAML_request_poll
  CMAML_reply_poll
  CMAML_wait
  CMAML_poll_wait
  CMAML_poll_while_lt
  ```
- Functions used for enabling and disabling interrupts:
  ```
  CMAML_enable_interrupts
  CMAML_disable_interrupts
  CMAML_interrupt_status
  CMAML_inside_interrupt_handler
  ```
- Functions used for saving and restoring the contents of registers:
  ```
  CMAML_save_fpuregs
  CMAML_restore_fpuregs
  CMAML_save_dataregs
  CMAML_restore_dataregs
  CMAML_save_ctrlregs
  CMAML_restore_ctrlregs
  ```

This section provides reference pages for these functions.

Copyright © 1995 Thinking Machines Corporation

CMAML Polling Functions

Poll one or both network interfaces, either until no further messages are found or until a certain flag value is reached.

SYNOPSIS

C Syntax

```
void CMAML_poll(void)
void CMAML_request_poll(void)
void CMAML_reply_poll(void)

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)
```

Fortran Syntax

```
SUBROUTINE CMAML_poll()
SUBROUTINE CMAML_request_poll()
SUBROUTINE CMAML_reply_poll()

SUBROUTINE CMAML_wait(flag, value)
SUBROUTINE CMAML_poll_wait(flag, value)
SUBROUTINE CMAML_poll_while_lt(flag, value)
    INTEGER flag, value
```

ARGUMENTS

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

RETURNS

These functions return after they have accomplished the appropriate polling activities. They have no return values.

Version 3.3, April 1995
Copyright © 1995 Thinking Machines Corporation

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. If it finds a message pending, it reads it from the network and calls 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 polls only the Request interface).

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

`CMAML_wait` polls repeatedly (on both interfaces) as long as the value pointed to by *`flag`* is less than *`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` is the same as `CMAML_wait`, except that `flag` is tested *after* the network is checked.

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

Copyright © 1995 Thinking Machines Corporation

Packet Interrupt Enabling and Disabling

CMAML functions for enabling and disabling packet interrupts, and for retrieving interrupt status.

SYNOPSIS

C Syntax

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

Fortran Syntax

```
INTEGER FUNCTION CMAML_enable_interrupts()
INTEGER FUNCTION CMAML_disable_interrupts()
INTEGER FUNCTION CMAML_interrupt_status()
INTEGER FUNCTION CMAML_inside_interrupt_handler()
```

RETURNS

All these functions return values of `TRUE` and `FALSE`:

- `CMAML_enable_interrupts` and `CMAML_disable_interrupts` return `TRUE` if they changed the interrupt status and `FALSE` otherwise.
- `CMAML_interrupt_status` returns `TRUE` if interrupts are enabled, `FALSE` otherwise.
- `CMAML_inside_interrupt_handler` returns `TRUE` if the current code is executing inside the interrupt handler, `FALSE` otherwise.

INTERRUPTS

It is the user's responsibility to *not* enable interrupts within the dynamic scope of the CMMD interrupt handler. Failing to observe this restriction can in general result in a fatal error.

Version 3.3, April 1995
Copyright © 1995 Thinking Machines Corporation

DESCRIPTION

The two functions `CMAML_[enable | disable]_interrupts` unconditionally enable (or disable) interrupts. They return `TRUE` if interrupts were successfully enabled (or disabled) and `FALSE` otherwise. (**NOTE:** these functions return `TRUE` only 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. **Remember:** 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.

Copyright © 1995 Thinking Machines Corporation

Saving and Restoring VU and SPARC FP Register States

CMAML functions for saving and restoring the state of the SPARC FP registers and VU data/control registers.

SYNOPSIS

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)
```

Fortran Syntax

```
SUBROUTINE CMAML_save_fpuregs(iu_fp_spillarea)
    REAL*8 iu_fp_spillarea
SUBROUTINE CMAML_restore_fpuregs(iu_fp_spillarea)
    REAL*8 iu_fp_spillarea
SUBROUTINE CMAML_save_dataregs(dp_dr_spillarea)
    REAL*8 dp_dr_spillarea
SUBROUTINE CMAML_restore_dataregs(dp_dr_spillarea)
    REAL*8 dp_dr_spillarea
SUBROUTINE CMAML_save_ctrlregs(dp_cr_spillarea)
    REAL*8 dp_cr_spillarea
SUBROUTINE CMAML_restore_ctrlregs(dr_cr_spillarea)
    REAL*8 dp_cr_spillarea
```

DESCRIPTION

The CMAML interrupt mechanism is optimized to incur the minimum possible overhead in the most common case. In particular, 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 that are called within the dynamic scope of

Version 3.3, April 1995
Copyright © 1995 Thinking Machines Corporation

the CMMD interrupt handler) are responsible for saving the states of these registers and restoring them.

These functions save the state of the VU control and/or data registers and the state of the SPARC FP registers. The computation that was interrupted 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 doubleword-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

Implementation Notes

1. The *`dp_dr_spillarea`* must be a pointer to a parallel memory region in broadcast (or "ALL VU") data space.

2. The space requirements of some regions are padded to be doubleword multiples.

3. 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.

Copyright © 1995 Thinking Machines Corporation