Chapter 11

Interrupts and Polling

11.1 Interrupts and Polling

The asynchronous message passing functions *initiate* the sending or receiving of a message, but may return before the actual data transmission completes, or even starts. The transmission may occur at a later time.*

- If interrupts are enabled, transmission can occur "in the background," i.e. during interrupts, at arbitrary points in the program's execution. While this transmission is in progress, is is invisible to the user (except that the contents of a destination buffer change as a receive progresses).

- If interrupts are disabled, pending transmissions can complete only during CMMD functions that *poll*, i.e., that check for incoming message fragments or *packets* (which are described in Appendix B) and that transmit pending fragments when possible. The `_poll`, `_done`, and `_wait` functions exist for the purpose of *explicit* polling. Other CMMD functions (including the `_send` and `_receive` functions) poll *implicitl*y in the course of their duties.

You have the option, at most points in your program, of disabling network interrupts and polling the network by calling one of the polling functions or any CMMD function that polls as part of some other activity.

---

* The reader may wonder why it is necessary to complete an asynchronous send *later*. (The need to complete an asynchronous *receive* later is obvious — the sender may not even have begun to send it yet.) Two factors may prevent a message from actually being sent during an asynchronous send call, and thus require subsequent completion: (1) As noted in Chapter 2 (and elaborated in Appendix B), a message transmission requires a preliminary exchange of information in an internal "handshake," and the receiving node may always impose an arbitrary delay in its part of the handshake. (2) The Data Network, through which pieces of a message travel from node to node (see Appendix B), may become full or "jammed," and thus temporarily unable to accept more data.

Copyright © 1995 Thinking Machines Corporation

The CMMD polling function checks for pending messages on the Data Network. If it finds any, it processes them immediately, and checks for any more pending messages before returning. It does not return until there are (at that moment) no more messages to process. The CMMD interrupt handler, when it is invoked by an incoming message, simply calls the same CMMD polling function. (For more detailed information on this process, see Appendix B.) Enabling interrupts effectively gives incoming messages the power to immediately call a polling function on their own behalf. The ability to turn interrupts on or off thus allows you to choose whether (and when) to give network message reception priority over computation.

Interrupts are enabled by default. To disable interrupts, a node can call

```
CMMD_disable_interrupts
```

To re-enable interrupts, a node can call

```
CMMD_enable_interrupts
```

To explicitly poll the network for pending messages (and if any are pending, to cause them to be received), a node can call

```
CMMD_poll
```

IMPORTANT

1. The CMMD interrupt handler always disables interrupts while it is operative. Code that executes within the dynamic scope of the interrupt handler — such as handler functions — *must not* attempt to enable or disable interrupts. Doing so may in general cause a fatal error.

   Note that this also means that handlers cannot be interrupted by incoming messages, even if interrupts are otherwise enabled throughout the program.

2. When interrupts are disabled, CMMD relies on polling to keep the network from becoming blocked. If you disable network interrupts, you are responsible for calling the CMMD polling functions frequently enough, or in the right places, such that network message traffic keeps flowing.

Version 3.3, April 1995
Copyright © 1995 Thinking Machines Corporation

3. When a program is running in the host/node environment (described in Chapter 19), interrupts are disabled by default for the host program, and cannot be enabled in the host program.

11.1.1 Interrupts vs. Polling — Usage and Performance

An application can receive messages (active messages or otherwise) via either of two methods: explicit polling or automatic interrupts. You can switch between these two modes of message reception whenever you like on any node. (The one exception — which is absolutely crucial — is that *the interrupt state should never be modified within the dynamic scope of the interrupt handler itself*.) This section discusses how to choose the mode of message reception that is best for any given portion of your application.

When deciding whether your application should use polling, interrupts, or some alternation of both for the reception of messages, you should consider that invoking a polling function incurs less overhead than initiating an interrupt. However, this penalty is offset by the fact that when an interrupt takes place, the interrupting message is usually followed immediately by other messages that do *not* cause an interrupt, since they are processed directly by the polling function that the interrupt handler calls. Thus the overhead of the "leader" message — the message that actually triggers the interrupt — can often be amortized over many "follower" messages. (Note that strictly speaking, the word "messages" should be replaced by "packets"; see Appendix B for a detailed explanation.)

Obviously, such amortization will be less exploitable in applications that send many short messages with significant time gaps between messages. However, if in such an application the messages are also sent at random times, then it may be impractical (or simply tedious) to insert polling calls in exactly the right places throughout the program. Generally speaking, polling calls should be inserted only when the odds of finding a waiting messages are high:

- when you can predict arrival time
- when you are expecting multiple packets in rapid succession (as in a bulk data transfer)
- when packets arrive very frequently

Copyright © 1995 Thinking Machines Corporation

Usually it is difficult to determine analytically whether a given program will run faster with interrupts or with polling. Users often resort to experimentation to make this determination.

The choice between polling and interrupts can have not only performance implications, but semantic implications as well. For example, a section of a program in which interrupts are enabled may be at risk of executing non-atomically. (Whenever the atomicity of critical code sections is an issue, the programmer's strategy should include a careful consideration of interrupts.) Also, some programs may be semantically correct with interrupts enabled, yet may cause deadlock if interrupts are disabled. For example, the following program runs to completion when interrupts are enabled (i.e., when the program is invoked without an argument), yet it hangs when interrupts are disabled (i.e., when it is invoked with an argument):

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

main(argc, argv)
  int argc;
  char *argv[];
{
  int addend = 1, sum, self = CMMD_self_address();
  char buffer[256];

  if( argc > 1 )
     CMMD_disable_interrupts();

  if ( self == 0 ) {
    if( 0 > CMMD_send_block(1, CMMD_DEFAULT_TAG,
                            buffer, 256) )
      CMMD_error("PN %d, bad blocking send.\n", self);
  } else if( self == 1 ) {
    if( 0 > CMMD_receive_async(0, CMMD_DEFAULT_TAG,
                               buffer, 256, NULL, NULL) )
      CMMD_error("PN %d, bad asynch receive.\n", self);
  }
  sum = CMMD_reduce_int(addend, CMMD_combiner_add);
  CMMD_all_msgs_wait();
  printf("Sum = %d\n", sum);
}
```

The reason it hangs (when interrupts are disabled) is that node 1 is caught waiting for the global operation `CMMD_reduce_int` to complete on the Control Network. Because of this, the call to the polling function `CMMD_all_msgs_wait` is never reached. However, if interrupts are enabled, then the

Version 3.3, April 1995
Copyright © 1995 Thinking Machines Corporation

messages sent by node 0 *interrupts* node 1's global computation, thus allowing the transaction between node 0 and node 1 to complete.*

If you choose to poll, however, you should ensure that each node polls frequently enough to prevent pending messages from clogging the Data Network. Such polling calls must be explicitly coded into your application. Although network clogging is not inherently fatal, it can cause serious performance degradation.

Notice that if asynchronous functions are being used (see Chapter 6), then one of the two following conditions holds:

- Useful computation is being done pending message completion, so that the relevant question is: when during such computation should the data be sent or received? This question is essentially answered when you make a choice between polling or interrupts.
- The asynchronous functions are being used to avoid a deadlock that would otherwise occur because of the communication pattern, as when all nodes do a send and a receive using blocking functions. For example, a send to the next higher node and a receive from the next lower node would cause deadlock unless nonblocking (asynchronous) functions were used. The following example illustrates:

```
#include <cm/cmmd.h>
main
{
  int nn = CMMD_partition_size();
  int self = CMMD_self_address();
  int src_node = (nn+(self-1))%nn;
  int dst_node = (nn+(self+1))%nn;
  char src_buffer[256], dst_buffer[256];

  if(0 > CMMD_receive_async(src_node, CMMD_DEFAULT_TAG,
                            dst_buffer, 256, NULL, NULL))
    CMMD_error("PN %d, Receive error.\n", self);

  if(0 > CMMD_send_block(dst_node, CMMD_DEFAULT_TAG,
                         src_buffer, 256))
    CMMD_error("PN %d, Send error.\n", self);

  CMMD_all_msgs_wait();
}
```

* Notice the important point that this example illustrates: *Data Network interrupts operate independently and transparently with respect to the global (Control Network) functions*.

Copyright © 1995 Thinking Machines Corporation

This program would block if the `receive_aysnc` were replaced by a `receive_block`, *regardless of whether or not interrupts were enabled*.

PLEASE NOTE

If an application choose to poll for messages, it should poll often enough to avoid congestion in the Data Network. Commonly, what constitutes "often enough" can be determined only through experimentation. The point is that a node which has disabled interrupts can "maliciously" refuse to accept incoming messages, by failing to poll. This can cause other nodes that are trying to send to it to block, since the Data Network is full and they cannot send their messages, resulting effectively in deadlock.

11.1.2 Implicit Polling by CMMD Communication Function

As a general rule, all blocking CMMD message passing functions implicitly poll the Data Network. Specifically, the functions

```
CMMD_send_block
CMMD_receive_block
CMMD_send_and_receive
CMMD_write_channel
```

poll the Data Network unconditionally and continously until their messages are complete and they return.

In the current implementation, neither the scalar scan/reduce functions (`CMMD_reduce_int`, `CMMD_scan_double`, etc.) nor the `sync`, `broadcast`, `concat`, or `global_or` functions implicitly poll the Data Network. All other communication functions, including the nonblocking message-passing functions

```
CMMD_send_async
CMMD_send_noblock
CMMD_receive_async
CMMD_open_send_channel
```


Version 3.3, April 1995
Copyright © 1995 Thinking Machines Corporation

```
CMMD_open_receive_channel
CMMD_close_send_channel
CMMD_close_receive_channel
CMMD_reset_channel
```

including `reduce_v`, `scan_v`, `scopy`, and `pcopy` (see Chapter 12), may or may not implicitly poll the Data Network. Users should make no assumptions either way.

11.1.3 Polling by CMMD Inquiry Functions

In addition to the polling by messaging operations, discussed in the preceding section, the CMMD inquiry functions automatically poll when interrupts are disabled:

```
CMMD_msg_done              CMMD_msg_wait
CMMD_all_sends_done        CMMD_all_receives_done
CMMD_all_msgs_done         CMMD_all_msgs_wait
CMMD_msg_pending           CMMD_mcb_pending
CMMD_is_channel_readable   CMMD_is_channel_writable
```

The two blocking functions, `CMMD_msg_wait` and `CMMD_all_msgs_wait`, poll continuously until their "wait condition" (i.e., completion of the desired messages) is met. They continue to poll even if there occur moments or periods, prior to the wait condition being met, in which the Data Network has no pending messages. The other functions, in contrast, "poll once," meaning that they return upon the *first* moment when they find no pending messages.

The polling behavior of these functions is present primarily as a convenience, since these functions are often called when interrupts are disabled, or would otherwise typically be preceded by a polling operation.

11.1.4 Implicit Polling by External Functions

In CMMD Version 3.0 and higher, a number of external functions, such as `printf`, have been implemented on top of CMMD blocking message-passing primitives. Calls to `libc` functions (or to any other external functions) may therefore poll; user code should not assume either that they will or that they will not do so.

Copyright © 1995 Thinking Machines Corporation

11.2 Function Dictionary

This section provides the reference pages for the functions associated with polling and interrupts.

For manipulating and querying the interrupt status:

```
CMMD_enable_interrupts
CMMD_disable_interrupts
CMMD_interrupt_status
CMMD_inside_interrupt_handler
```

For explicit polling:

```
CMMD_poll
```

Section 11.3 provides an example of disabling and enabling interrupts, and of explicit polling.

Version 3.3, April 1995
Copyright © 1995 Thinking Machines Corporation

CMMD_enable_interrupts
CMMD_disable_interrupts

Enable or disable network interrupts.

SYNOPSIS

C Syntax

```
int CMMD_enable_interrupts(void)
int CMMD_disable_interrupts(void)
```

Fortran Syntax

```
INTEGER FUNCTION CMMD_enable_interrupts()
INTEGER FUNCTION CMMD_disable_interrupts()
```

RETURNS

`CMMD_enable_interrupts` and `CMMD_disable_interrupts` return 1 if interrupts were successfully enabled/disabled (that is, if the interrupt state actually changed), and 0 otherwise.

DESCRIPTION

`CMMD_enable_interrupts` enables interrupts unconditionally, and returns 1 if interrupts were successfully enabled, and 0 otherwise. After a node calls this function, it may be interrupted at arbitrary times for the reception of a message from the network.

`CMMD_disable_interrupts` disables interrupts unconditionally, and returns the value that `CMMD_interrupt_status` returned *before* the interrupt state was changed. (That is, it returns 1 if the status changed from enabled to disabled, 0 if interrupts were already disabled.) This makes it convenient for you to save the prior interrupt state when you disable interrupts at the beginning of a block of code, and restore it when the code block completes. After a node calls `CMMD_disable_interrupts`, it will not be interrupted for the reception of network messages. Instead, the node must regularly call the CMMD polling function (described below) to detect and receive incoming messages.

Copyright © 1995 Thinking Machines Corporation

Warning: CMMD itself enables and disables interrupts at various times. Thus, user-defined functions that enable or disable interrupts are responsible for saving the prior interrupt state (for example, by saving the value returned by `CMMD_disable_interrupts`) and for restoring that state before returning.

PLEASE NOTE

The interrupt state should *never* be modified within the dynamic scope of the interrupt handler itself. (You can always determine whether you are inside the dynamic scope of the interrupt handler by calling `CMAML_inside_interrupt_handler`.

Version 3.3, April 1995
Copyright © 1995 Thinking Machines Corporation

CMMD_interrupt_status
CMMD_inside_interrupt_handler

These functions tell whether interrupts are enabled or disabled, and whether code is executing within an interrupt handler (where they must remain disabled).

SYNOPSIS

C Syntax

```
int CMMD_interrupt_status(void)
int CMMD_inside_interrupt_handler(void)
```

Fortran Syntax

```
INTEGER FUNCTION CMMD_interrupt_status()
INTEGER FUNCTION CMMD_inside_interrupt_handler()
```

RETURNS

`CMMD_interrupt_status` returns 1 if interrupts are enabled; it returns 0 otherwise.

`CMMD_inside_interrupt_handler` returns 1 if code execution is currently taking place within the dynamic scope of the interrupt handler; it returns 0 otherwise.

DESCRIPTION

Two interrupt status functions are `CMMD_interrupt_status` and `CMMD_inside_interrupt_handler`.

Because interrupts are automatically disabled inside interrupt handlers, the CMMD internal interrupt flags can be used to determine whether code is currently executing within an interrupt handler. The function `CMMD_inside_interrupt_handler` returns 1 if this is the case, and 0 otherwise. (Remember that it is an error for code executing inside an interrupt handler to try to alter interrupt status.

Copyright © 1995 Thinking Machines Corporation

Therefore, a value of 1 returned by `CMMD_inside_interrupt_handler` means that your code must not attempt to enable interrupts at this time.)

Version 3.3, April 1995
Copyright © 1995 Thinking Machines Corporation

CMMD_poll

If interrupts are disabled, polls for incoming messages for this node, and handles all that it finds. If interrupts are enabled, does nothing.

SYNOPSIS

C Syntax

```
void CMMD_poll()
```

Fortran Syntax

```
SUBROUTINE  CMMD_poll()
```

DESCRIPTION

`CMMD_poll` is the generic CMMD polling function. It checks for incoming messages, and if it finds one, automatically handles receiving it. `CMMD_poll` keeps checking the network until no messages are pending, and then returns.

Copyright © 1995 Thinking Machines Corporation

11.3 Polling Example (Using Channel Functions)

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

main()
{
  int scd, rcd, nwrites = 5, self = CMMD_self_address();
  char buffer[80];

  CMMD_disable_interrupts();
  if ( self == 0 ) {
    if( 0 > (scd = CMMD_open_send_channel(1, CMMD_DEFAULT_TAG,
                                          buffer, 80)) )
      CMMD_error("Bad open send channel.\n");
    while ( nwrites-- ) {
      /* CMMD_is_channel_writable polls, thus potentially
          receiving reset-channel notifications from the
          peer: */
      while ( ! CMMD_is_channel_writable( scd ) ) {
        /* Computation could be done here while you
            are waiting. */
      }
      /* CMMD_write_channel polls also: */
      if ( CMMD_write_channel( scd ) )
        CMMD_error("Bad write channel.\n");
    }
    if ( 0 > CMMD_close_send_channel( scd ) )
      CMMD_error("Bad state for closing send channel.\n");
    /* Optionally, you could poll here until the channel status
       changed to Free (to guarantee that your channel has
       been closed on *both* ends), but the next polling
       operation(s), whether implicit or explicit, will
       eventually poll and thus free it for you. */
```


Copyright © 1995 Thinking Machines Corporation

```
  } else if ( self == 1 ) {

    if( 0 > (rcd = CMMD_open_receive_channel(0, CMMD_DEFAULT_TAG,
                                             buffer, 80)) )
      CMMD_error("Bad open receive channel.\n");

    while ( nwrites-- ) {
      /* CMMD_is_channel_readable polls: */
      while ( ! CMMD_is_channel_readable( rcd ) ) {
        /* Computation could be done here while you
           are waiting. */
      }

      /* Process the received data here. */

      /* Do not reset the channel after the last write. */

      if ( nwrites )
        /* CMMD_reset_channel may or may not poll: */
        if( 0 > CMMD_reset_channel( rcd ) )
          CMMD_error("Bad reset.\n");
    }

    if ( 0 > CMMD_close_receive_channel( rcd ) )
      CMMD_error("Bad state for closing recv channel.\n");
  }
  CMMD_enable_interrupts();
}
```


Copyright © 1995 Thinking Machines Corporation