Chapter 10

Interrupts and Polling

10.1 Interrupts and Polling

The CMMD library internally uses network interrupts to handle messages sent between nodes. This interrupt-driven system is not visible in programs that call the standard CMMD communication operators, but it is nevertheless present, and this internal use of interrupts can in some cases affect the performance of your code.

You have the option, at most points in your program, to disable network interrupts and explicitly "poll" the network at chosen points in your code by calling either **CMMD_poll** or a function that polls as part of some other activity.

The CMMD polling function checks for pending messages. If it finds any, it itself calls the CMMD interrupt handler to process the messages. When all pending messages have been processed, control returns to the application. (For more information on this process, see Section 10.2.) The choice of polling versus interrupts, therefore, has no effect on the syntax of most CMMD operators; it simply allows you to choose when to allow your program to pause for network message reception.

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 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. (It is an error to do so.) For more information, see Chapter 11.

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

(2) When interrupts are disabled, CMMD relies on regular polling to keep the network from becoming blocked. If you disable network interrupts, you are responsible for calling the CMMD polling function regularly to keep network message traffic flowing.

(3) When a program is running in the "host/node" environment (described in Chapter 16), interrupts are disabled by default for the host program, and cannot be enabled in the host program.

10.1.1 Interrupts vs. Polling — Usage and Performance

An application can receive messages (active or otherwise) via either of two methods: explicit polling or automatic interrupts. 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 active messages or other asynchronous messages, you should consider the following facts:

- Polling calls usually have less overhead than interrupts.

- Using polling rather than interrupts helps guarantee atomicity during critical sections of code.
- If you choose to poll, however, you must ensure that each node polls frequently enough to prevent pending messages from clogging the network. These polling calls must be explicitly coded into your application.
- Enabling interrupts prevents network clogging. Moreover, interrupts occur only when a message is waiting to be delivered. They thus avoid the potential costs of fruitless polling calls. One interrupt is more expensive than one polling call, but less expensive than repeated polling calls.
- It is an error to attempt polling while interrupts are enabled.
- You can switch between these two modes of message reception whenever you like, on any node. The exception is that interrupt state should never be modified inside the dynamic scope of the CMMD interrupt handler.

In general, receiving messages automatically via the CMMD interrupt handler is most efficient when your code cannot predict when the next message will arrive.

Polling is most efficient when the odds of any given polling call finding a waiting message are high: when you can predict arrival time, when you are expecting multiple packets (as in a bulk transfer of data), or when packets arrive frequently.

PLEASE NOTE

If an application chooses to poll for messages, it is very important that the application poll often enough to prevent congestion in the network.

When interrupts are enabled, Data Network message arrival causes the operating system to suspend computation on the node while the CMMD interrupt handler services the network. When interrupts are disabled, arriving messages pile up in the node's Network Interface, and then back up into the Data Network itself. A node that fails to drain its network interface may thus cause network congestion throughout the user's partition.

10.1.2 Polling and the CMMD Messaging Operations

When you use polling along with the standard CMMD message-passing functions, it's helpful to know how these functions send messages so that you have some idea of the amount of polling you must include in your program. (Note that these protocols are implementation-dependent. They may change in some future version.)

Cooperative and Asynchronous Messaging

For example, the cooperative and asynchronous message functions follow the same pattern:

- The source node sends an RTS (request to send) to the destination node.
- The destination node receives the RTS and sends back an ACKnowledge when it is ready to receive the specified message.
- The source node receives the ACK and begins transmitting the data to the destination.
- The destination node receives the data, concluding the message.

One difference between cooperative and asynchronous messages is that in cooperative message passing the source and destination nodes block while waiting for succesive parts of the transaction to happen. The source node, once it has sent its RTS, blocks until it receives the corresponding ACK from the destination node. The destination node blocks until it receives a corresponding RTS from the source node.

A second difference lies in the frequency of polling. Blocking functions poll continuously. Asynchronous functions poll once, at the time they send their handshake signal, and may poll some small number of times thereafter. Any further polling must be explicitly performed by the application, either via the polling functions themselves or via inquiry functions such as **`CMMD_msg_pending`**.

Channel Messaging

Channel functions have a more extended protocol, which allows them to avoid some of the overhead involved in the cooperative and asynchronous operations:

- In the initial handshake, the source sends an RTS to the dest, and the dest replies with an ACK. This establishes the two ends of the channel.
- In a write operation, the source node sends a complete block of data to the destination node.
- The destination node, when it has received the data, transmits a reset signal back to the source node, to let it know the channel can be reused.
- The write and reset steps are repeated as many times as the channel is used to send a message (except for the last time, when the channel is not reset).
- When the destination node has closed its end of the channel, it sends a close signal to the source. The receive side of the channel is then freed for reuse.
- When both nodes have closed the channel, the send channel is freed for reuse.

The channel-opening functions poll at handshake time. Other polling must be done by the application, typically by using the channel inquiry functions (described in Section 7.3.2).

Global Messaging

Global messaging involves the synchronous operation of all the nodes:

- When a node signals a global operation, it blocks until it detects that all the other nodes have signaled the same operation.
- When all nodes have signaled the operation, the global message is completed for all nodes.

Global operations use the Control Network for synchronization. They do not generally poll the Data Network.

If interrupts are enabled, global operations can be interrupted. If interrupts are disabled, an application can create a fatal deadlock, in the following manner: Node 1 calls **send_async**, then calls a global function; node 2 is to call **receive_block**, then call the same global function. If node 1 makes its global call before sending the message, and interrupts are disabled, there is no way for node 2 either to receive its message or make its own global call. The program will therefore hang.

10.1.3 Automatic Polling by CMMD Inquiry Functions

In addition to the polling by messaging operations, discussed in the preceding sections, the CMMD inquiry functions operators 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. The other functions poll once, at the time they are invoked.)

This is primarily a convenience, since these functions are often called when interrupts are disabled, and would otherwise typically be preceded by a polling operation.

10.1.4 Polling by External Functions

In CMMD Version 3.0, a number of external functions, such as **printf**, have been implemented on top of CMMD 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.

10.2 Enabling and Disabling Network Interrupts

The following functions are used to enable and disable interrupts, to check whether interrupts are currently enabled, and to use the current interrupt state to detect when a function has been called from within an interrupt handler.

FUNCTION **CMMD_enable_interrupts**
CMMD_disable_interrupts
CMMD_interrupt_status
CMMD_inside_interrupt_handler

C SYNTAX

int **CMMD_enable_interrupts(**void**)**
int **CMMD_disable_interrupts(**void**)**
int **CMMD_interrupt_status(**void**)**
int **CMMD_inside_interrupt_handler(**void**)**

FORTRAN SYNTAX

INTEGER FUNCTION **CMMD_enable_interrupts()**
INTEGER FUNCTION **CMMD_disable_interrupts()**
INTEGER FUNCTION **CMMD_interrupt_status()**
INTEGER FUNCTION **CMMD_inside_interrupt_handler()**

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.

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 from within an interrupt handler; it returns 0 otherwise.

DESCRIPTION

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

CMMD_disable_interrupts disables interrupts unconditonally, 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.

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 exiting.

Two futher status functions are **CMMD_interrupt_status** and **CMMD_inside_interrupt handler**.

CMMD_interrupt_status returns 1 if interrupts are enabled, 0 otherwise.

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.)

10.3 Polling

FUNCTION **CMMD_poll**

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.

10.4 Example (Using Channel Functions)

```
#include <cm/cmmd.h>
main()
{
  int scd, rcd;
  int nwrites = 5;
  char buffer[80];
  CMMD_disable_interrupts();

  if (CMMD_self_address() == 0) {
    scd = CMMD_open_send_channel(1, CMMD_DEFAULT_TAG, buffer, 80);
    if (scd < 0)
      CMMD_error(”Error return from CMMD_open_send_channel.”);

    while (nwrites--) {
      /* Note that CMMD_is_channel_writable polls.  */
      while (!CMMD_is_channel_writable(scd)) {
      /* Computation could be done here while you are waiting.  */
      }
      if (CMMD_write_channel(scd))
    CMMD_error (”Failure from CMMD_write_channel.”);
    }
    CMMD_close_send_channel(scd);

    /* You could poll here (until the status changed to Free) to
       guarantee channel closing, but the next network operation you
       do should take care of that, so there’s no real need.  */

  } else if (CMMD_self_address() == 1) {
    rcd = CMMD_open_receive_channel(0, CMMD_DEFAULT_TAG, buffer, 80);
    if (rcd < 0)
      CMMD_error(”Error return from CMMD_open_receive_channel.”);

    while (nwrites--) {
      /* Note that CMMD_is_channel_readable polls.  */
```

```
      while (!CMMD_is_channel_readable(rcd)) {
      /* Computation could be done here while you are waiting.  */
      }

      /* Process data in the buffer here.  */

      /* Don't reset the channel after the last write.  */
      if (nwrites) CMMD_reset_channel(rcd);
    }

    CMMD_close_receive_channel(rcd);

    /* You could poll here (until the status changed to Free) to
       guarantee channel closing, but the next network operation you
       do should take care of that, so there's no real need.  Turning on
       interrupts will also take care of it.  */
  }
  CMMD_enable_interrupts();
}
```