Appendix B

Implementation of Messages

B.1 Data Network Packets

The discussion in Chapter 2 treated CMMD messages largely as though they were isolated units, with each node sending or receiving only one whole message at any time. In reality, however, a node may send and receive simultaneously, and many messages may be in transit at any one time. How is the message traffic really handled?

CMMD and CMAML messages travel through the Data Network in quanta called *Data Network packets*, or simply *packets*. Packets are 1 to 5 words in length on the CM-5, and 1 to 18 words in length on the CM-5E. They are labeled internally by their *type* (e.g., as a part of a block transfer, as an active message, etc.). The type is implemented very efficiently as a *hardware tag* (a special feature of the Data Network designed specifically to enable this kind of packet segregation). User applications never alter CMMD's internal hardware tags, and only the most sophisticated applications allocate their own hardware tags. (The procedure for doing so is explained in Appendix A.)

Each *data packet* is part of a block transfer. Data packets contain a few words (4 or 16) of data *per se*, plus some additional information indicating which message the data belongs to (i.e., which rport; see Chapter 2), and its offset from the start of that message. *Active message* packets, or simply *active messages*, contain a function pointer and a small number of arguments or other data words (see Chapter 12).

Figure 23 illustrates a typical incoming data stream as seen by a single node or the host. This example includes active messages, block transfer data from two separate transactions, plus some additional active messages generated

Copyright © 1995 Thinking Machines Corporation

internally by the CMMD system software (called *RTS*s and *ACK*s; these are described in the next section).

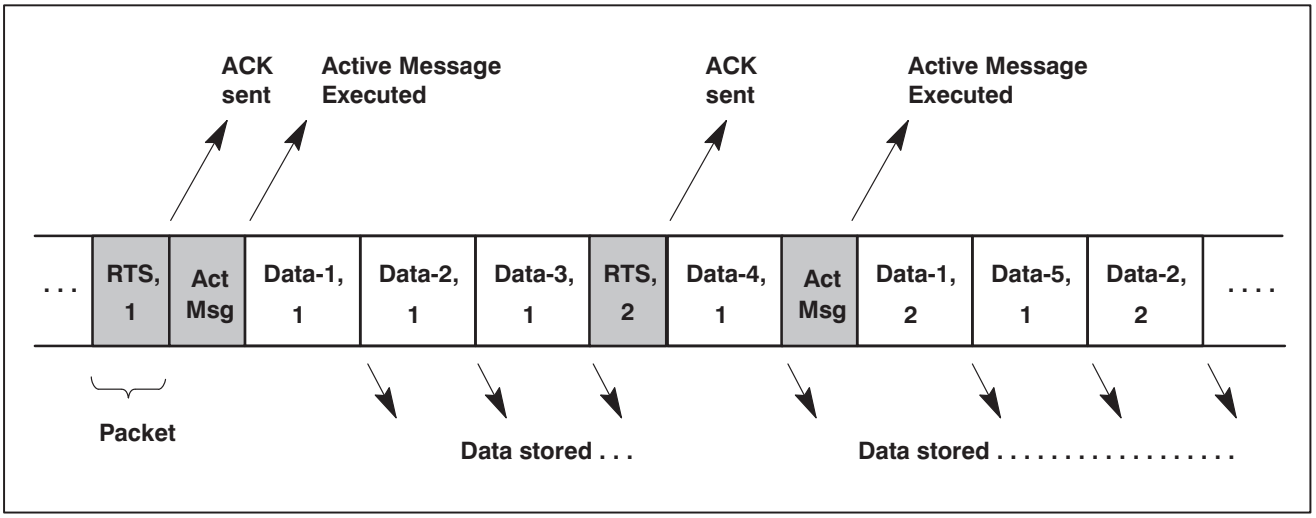

Figure 23. A sample incoming data stream for a single node.

B.2 Handshaking

As outlined in Chapter 2, nodes must exchange some preliminary information before they can exchange data. The exchange is called the "handshake." In CMMD, this is implemented *directly in terms of active messages*. When a node wants to send data to another node (via one of the cooperative functions, or the asynchronous functions, or virtual channels), it first sends a *request to send* (RTS). When the destination node receives an RTS, it eventually sends back an *acknowledgment* (ACK). Once an ACK has been received by the sender, the actual block data can be transferred.

The RTS contains information such as the desired message length, the message type (serial or parallel), the message tag, and the transaction type (ordinary message or virtual channel). The ACK contains, among other things, the length of message that it is willing or able to receive. The minimum of the two lengths is sent.

The RTS-ACK handshake is illustrated in Figure 24.

Version 3.3, April 1995
Copyright © 1995 Thinking Machines Corporation

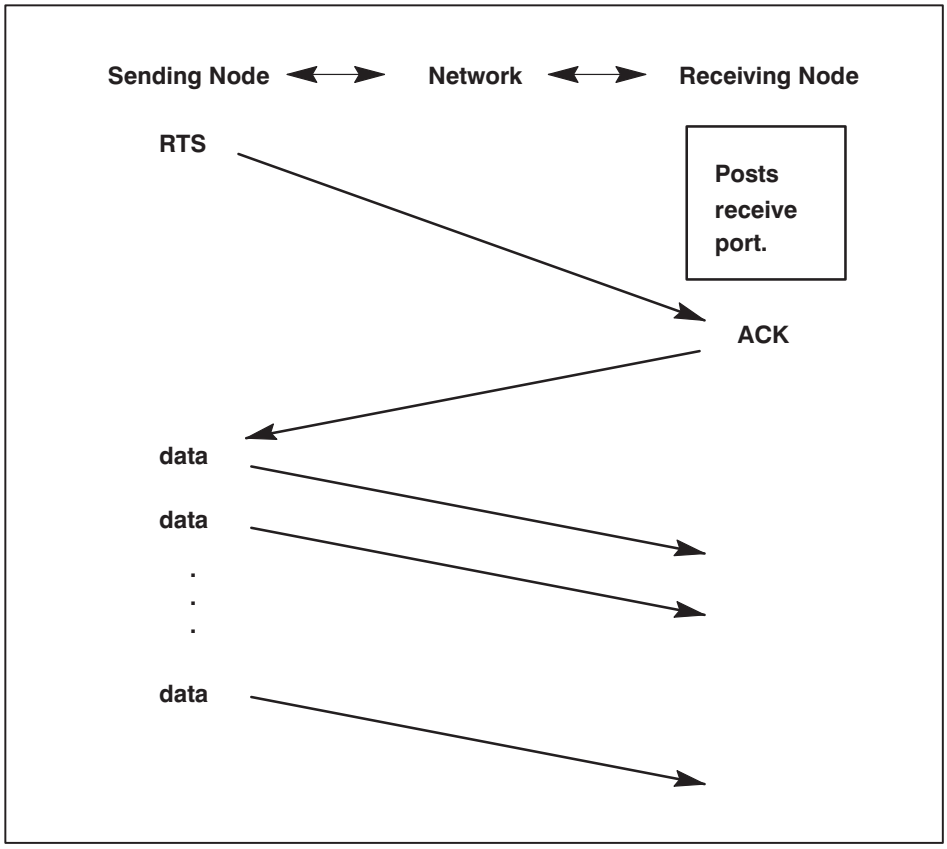

Figure 24. The basic "handshake" mechanism.

B.3 Processing Packets

All messages are sent *one* packet at a time. For active messages, the sending is complete after one packet, by definition. For block data transfers, in general, many packets must be sent.

Packets are atomic, that is, the order of the words within a Data Network never changes. However, the order in which packets arrive at a destination node is not necessarily the same as the order in which they were sent from a source node. In general, some "scrambling" occurs in transit. CMMD automatically *reorders* all block data packets upon arrival at their destination node. Active messages obviously have no need for reordering, since they are only one packet long.

Copyright © 1995 Thinking Machines Corporation

Due to the scrambling of packet order, sending is inherently simpler than receiving. Whereas a sending node always "knows" everything about every packet it sends (type, destination, etc.), a receiving node must read and decode this information from each incoming packet, and make appropriate decisions as to how to handle the packet. Thus a receiving node has considerably more work to do on average.

A node receives one packet at a time. However, once it reaches out to the network for one packet, it "commits" to pulling in packets until its network interface buffer is empty. It suspends this activity only to execute any active messages that arrive as part of the data stream (which is the primary rationale for the rule of thumb that all active message handlers should be *short*). Active messages are always executed *immediately* and *to completion*.

Suppose, for example, that node 0 has disabled interrupts (see Chapter 11) to perform some critical, uninterruptable computation. At the end of the computation, it may call `CMMD_poll` to check the network for incoming messages. It will not return from this polling call until it has handled all pending packets: that is, until it has executed all active messages, retrieved all data, and — if any callback functions have been posted in the meantime — invoked and executed them (see Section 6.3). The order in which it does all this depends partly on the order in which the packets arrive and partly on the *priority* of the different tasks. (For more information on priorities, see Section 12.2.2.)

As another example, suppose that node 0 has made two `receive_async` calls (and sent their ACKs) followed by a `send_block` call. The `send_block` call puts node 0 into a blocking polling loop, waiting for the ACK. While polling, node 0 can receive the data for both its receive calls, as well as receiving and handling RTSs and active messages from other nodes. In fact, even after getting the ACK for its send call, it can continue to receive data; after every packet sent, a node automatically checks the network for incoming data.

To recapitulate, message transmission (whether active messages or data packets) takes place when one node sends Data Network packets to another node, and the second node pauses in its computation to receive and process those packets. Let us now take a closer look at the two ways in which such pauses can occur: polling and interrupts.

Version 3.3, April 1995
Copyright © 1995 Thinking Machines Corporation

B.3.1 Polling and Interrupts

To begin, consider the normal *flow of control* in an application. (Figure 25 shows a simple example.)

Recall that, in a CMMD program, each node runs an identical copy of the application. The flow of control on each node thus provides the *main thread of control* (also called the *foreground computation*) for the application on that node.

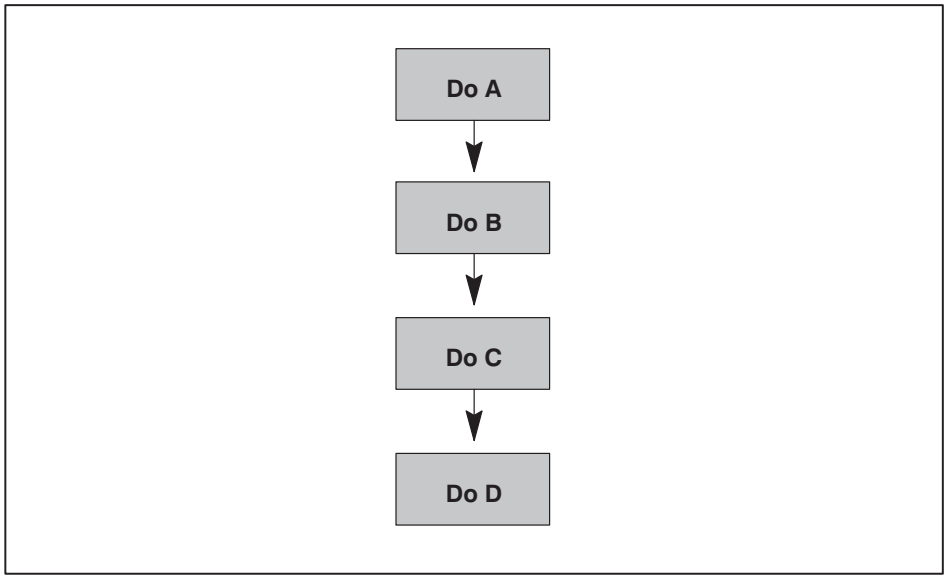

Figure 25. Foreground computation in a user program.

What happens to this thread of control when an asynchronous message is to be received? If interrupts are disabled, nothing happens until the program itself polls for messages. (See Figure 26.)

Copyright © 1995 Thinking Machines Corporation

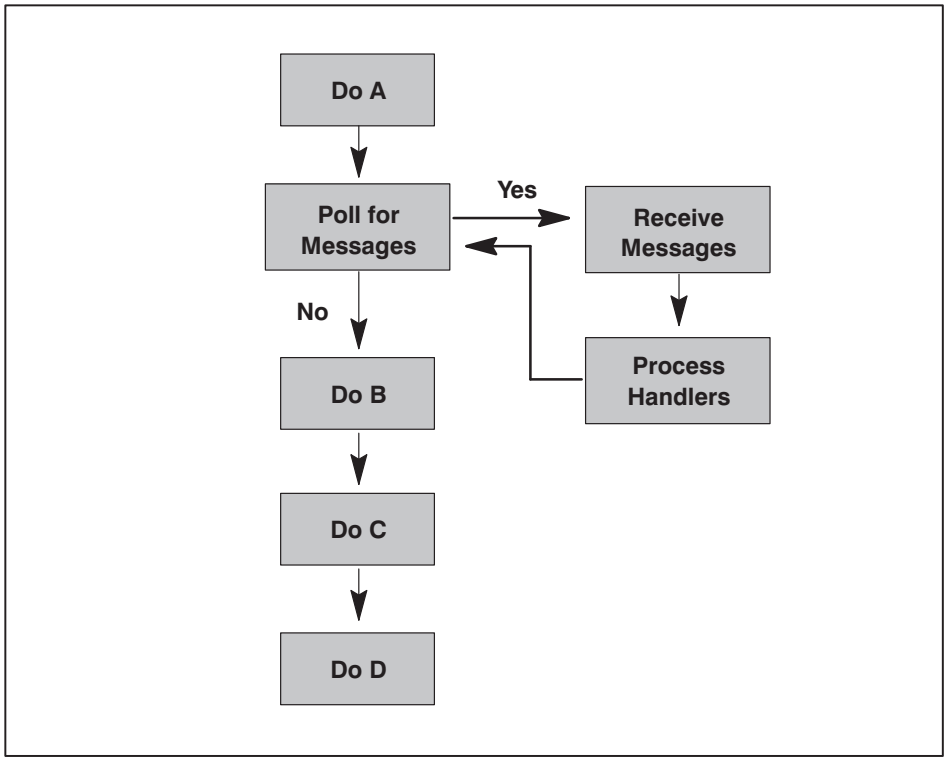

Figure 26. Flow of control when the user polls for messages.

The polling function called by the CMMD interrupt handler is the same function called by user polling requests (`CMMD_poll`). The polling process, therefore, is identical for messages received via both polling and interrupts.

Note that a program makes a single function call to begin polling, and then automatically continues polling for as long as there are messages to be received.

If interrupts are enabled, however, the arrival of a message packet generates an interrupt (see Figure 27), which immediately invokes the *CMMD interrupt handler*.

Version 3.3, April 1995
Copyright © 1995 Thinking Machines Corporation

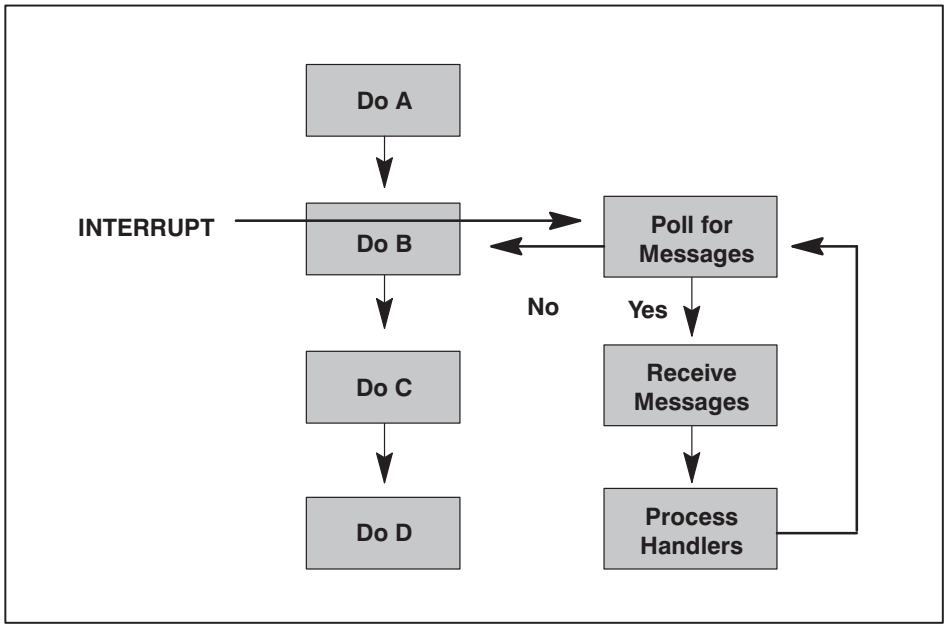

Figure 27. Flow of control when message interrupts are enabled.

The interrupt handler starts its work by

- suspending the foreground computation
- disabling interrupts
- saving the state of the SPARC's instruction unit registers
- calling a polling function to handle the incoming messages

Handling Incoming Messages

Once the internal polling routine is invoked, it proceeds to receive and process incoming packets until the node's network interface is empty.

Any data packets received are deposited directly into memory. If a given packet is the last one in a block data transfer, it may also cause a *callback function* to be invoked (see Chapter 6 and Chapter 14). On the sending node, a callback function may be invoked after the last packet of the message has been injected into the Data Network (though not necessarily after the packet has been received at the destination node). Callback functions on

Copyright © 1995 Thinking Machines Corporation

the receiving node are not invoked until all packets have been received and processed.

It should be stressed that there is no set order for packets. A node may receive RTS or ACK messages, data packets, or active messages at any time during a polling cycle. Thus, a series of data packets for a single message may be interrupted by an active message, or by an RTS or an ACK. Each packet contains identification as to its type, so the node can deal with each type correctly.

Similarly, there is no certainty that all data packets for a particular message will arrive at once. If a pause occurs during transmission, the lack of packets to be processed will end the polling cycle. The node resumes its interrupted work (or its polling) until the next messages interrupt it, or until the application's polling discovers new messages waiting.

Receipt of an active message causes its handler to be invoked immediately. All handlers and callback functions execute before control returns from the polling function to the application's main thread of control.

Returning Control

When all messages have been received, and all handlers executed, the polling function returns. If it was called by the CMMD interrupt handler, the interrupt handler then

- restores the state it saved for the interrupted computation
- re-enables interrupts
- exits

B.3.2 More on Handshaking: The async and noblock Functions

The behavior of `CMMD_send_async`, `CMMD_send_noblock`, and `CMMD_receive_async` is determined largely by the application program. These functions do not block; rather, they allow the nodes to perform other work until both are ready to transmit and receive the message (this is called "overlapping communication with computation"; see Chapter 2).

Like `CMMD_send_block` and `CMMD_send_block_v`, the functions `CMMD_send_async` and `CMMD_send_noblock` first send out an RTS. In the

Version 3.3, April 1995
Copyright © 1995 Thinking Machines Corporation

case of CMMD_send_noblock, the source buffer is copied to a "safe" area first. Both functions return immediately after sending their RTS. The immediate return is what allows the calling program to continue doing useful work while waiting for an ACK.

The process of matching an RTS with a receive function (See Chapter 18) is always performed on the receiving node. Whenever an RTS arrives prior to a matching receive call, the RTS is stored internally by CMMD. There it "waits" for a matching receive call. Whenever CMMD_receive_async is called, it first checks to see whether an appropriate RTS is already waiting. If one is waiting, it sends an ACK immediately. If none is waiting, it internally posts its intent to receive, then returns, and when a matching RTS finally arrives, CMMD will notice that the asynchronous receive call has already been posted. In either case, the asynchronous receive returns immediately, so that the receiving node can continue to perform useful computations (rather than blocking, as would CMMD_receive_block).

By default, CMMD interrupts a node program whenever RTSs, ACKs, or data arrive. When interrupted, a node program will automatically remove the pending messages from the Data Network and process them.

Alternatively, a node program can disable interrupts (see CMMD_disable_interrupts and CMAML_disable_interrupts). This places the responsibility for checking for pending messages on the application. Typically, the application implements a customized compute-and-poll cycle:

Do some quantum of work;
Poll for incoming packets;
Repeat as necessary.

In either case, asynchronous functions continue their alternation of computation and message-handling until all data has been sent and received. They then must perform one or two further "clean-up" steps. The essential step is that the application must call the function CMMD_free_mcb in order to mark the call's message control block as being ready to be freed. In addition, the application may specify a callback function for a send_async or receive_async function call. If it does so, the callback function is invoked when the message-passing call completes. See Figure 28 and Figure 29.

(Only when using the asynchronous send and receive functions can a node query the MCB for status information; therefore, only when using these functions is the application on the node required to cooperate in freeing the MCB for reuse.)

Copyright © 1995 Thinking Machines Corporation

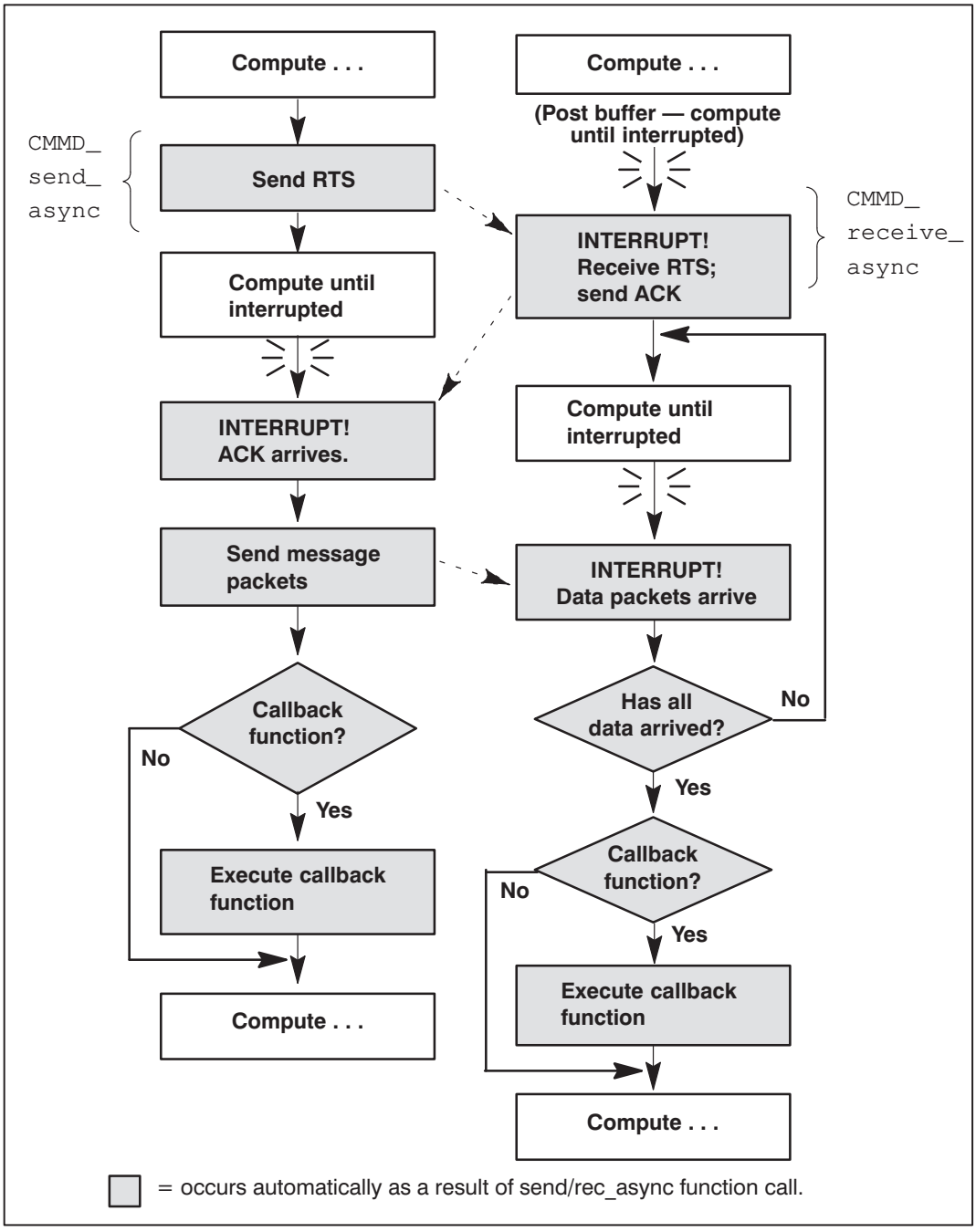

Figure 28. Async send, receive: Interrupts enabled.

Version 3.3, April 1995
Copyright © 1995 Thinking Machines Corporation

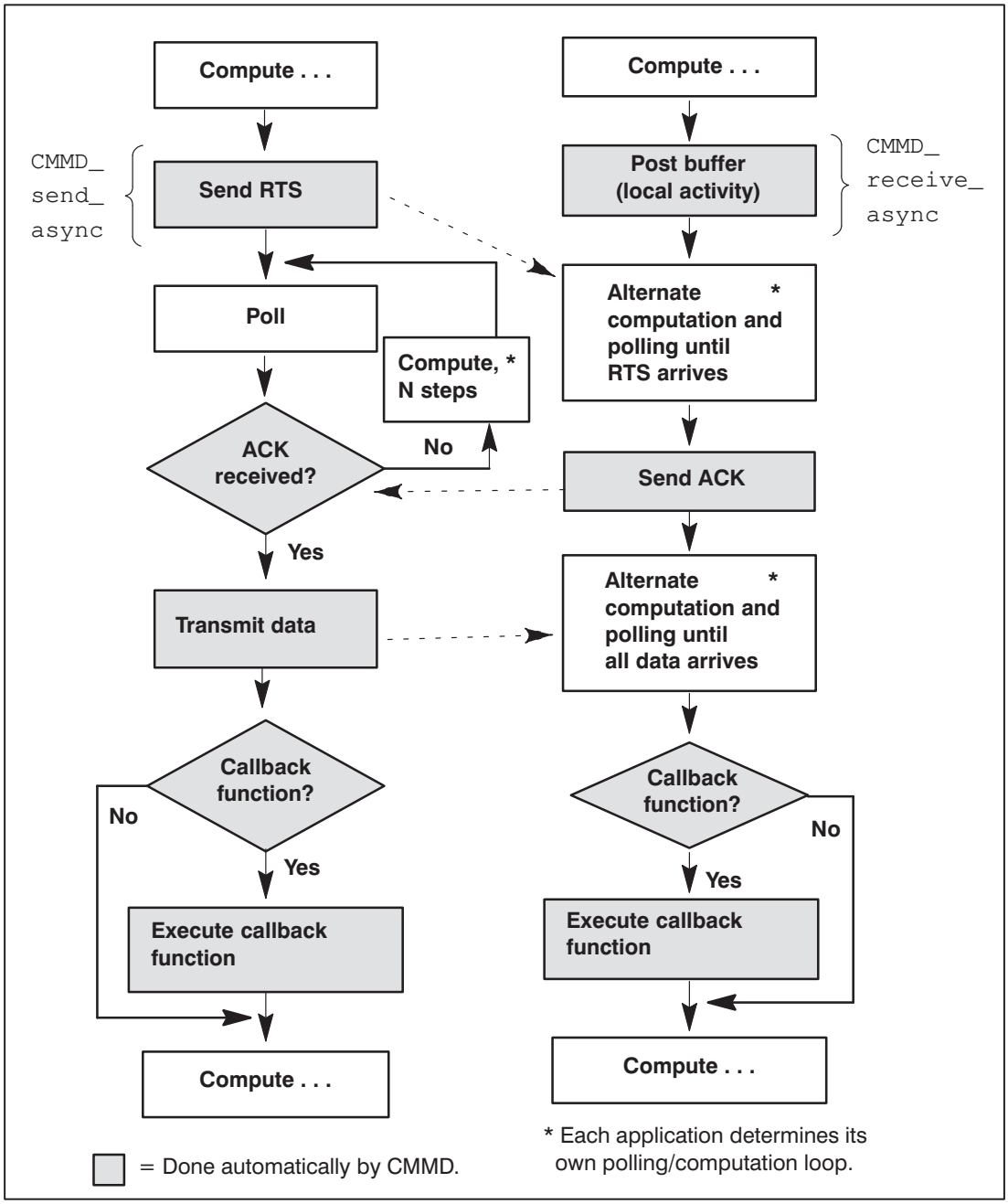

Figure 29. Async send, receive: Interrupts disabled.

Copyright © 1995 Thinking Machines Corporation