Chapter 15

Changing Defaults

CMMD allows applications to change three defaults:

- An application can logically reset the size of the partition on which it runs, causing it to execute on a smaller number of nodes.
- Any node can change the method by which it decides whether an incoming message ”matches” the message that node had asked to receive.
- Any node can enable or disable safety checking for the application.

The functions that change these defaults are **CMMD_reset_partition_size** and **CMMD_set_message_matching_fcn**, and **CMMD_{enable|disable}_safety**. These functions are discussed in this chapter.*

15.1 Resetting Partition Size

FUNCTION **CMMD_reset_partition_size**

C SYNTAX

int **CMMD_reset_partition_size (**int *new_size***)**

FORTRAN SYNTAX

INTEGER FUNCTION **CMMD_reset_partition_size**
integer *new_size*

---

* CMMD Version 2.0 also allowed nodes to change the default number of MCBs. This facility is no longer available.

ARGUMENTS

new_size An integer between 1 and **`CMMD_partition_size()`**, specifying the number of nodes that are to form the new "logical partition."

RETURNS

`CMMD_reset_partition_size` returns the number of nodes that are now active as the "logical partition" for the application.

DESCRIPTION

The **`CMMD_reset_partition_size`** function allows an application to change the size of the "logical partition" in which it operates. The logical partition must be smaller than or equal to the physical partition; it consists of nodes 0 through (*new_size* – 1). All nodes with addresses *new_size* or greater abstain from activity while the new partition size is in effect.

Note: During the following discussion, a partition in which some nodes are abstaining because of a **`CMMD_reset_partition_size`** call is referred to as a "restricted logical partition."

This is a global function. Once called by a single node, it blocks until all active (non-abstaining) nodes have called it.

The **`CMMD_reset_partition_size`** function can be used at any reasonable point within the program. It can also be used multiple times during a program. However, it must be used only when no network activity is taking place; resizing the partition when there are messages in the Data Network can have evil effects for your program.

Proper use of this function demands that the application re-set the logical partition–size to the physical partition-size at the end of each block in which the size has been altered. An example of proper use is shown below.

Note that the original (physical) partition value is not automatically saved; as shown in this example, the program must explicitly save the original value, then re-use that value as an argument to **`CMMD_reset_partition_size`**. Note also the use of the **`if`** statement, which is needed to ensure that only the correct nodes run the restricted code:

```
int original_size = CMMD_partition_size();
int new_size1 = 32;
int new_size2 = 51;

change = CMMD_reset_partition_size( new_size1 );

if ((change .eq. new_size1) &&
   (CMMD_self_address() .lt. new_size1)) {
   <code to be run in 32-node restricted partition>
   change = CMMD_reset_partition_size( original_size );
}
change = CMMD_reset_partition_size( new_size2 )
if ((change .eq. new_size2) &&
   (CMMD_self_address() .lt. new_size2)) {
   <code to be run in 51-node restricted partition>
   change = CMMD_reset_partition_size( original_size );
}
```

Note that this code restores the partition to the full size at the end of each block. Once this occurs, the nodes that have been sleeping wake up and continue execution from the point of the original call to the function **CMMD_reset_partition_size**. The **if** statement prevents the newly awakened nodes from executing the code that was meant for the restricted set.

15.1.1 Effect of Restriction on CMMD Functions

- Calls to the **CMMD_partition_size** function return the size of the current logical partition; thus, they reflect the effects of the most recent call (if any) to the **CMMD_reset_partition_size** function.
- You can perform CMMD global functions within a restricted logical partition. Be careful to set restricted code within an **if** statement, as shown in the example above, with the node's address as the test; otherwise, nodes that slept when the partition was made smaller will execute code from the point at which they went to sleep, rather than being redirected to the point at which they wake up. If there are any global functions in the restricted code, there would be communications collisions or hanging programs
- You cannot perform global I/O to an SDA or a DataVault.

15.2 Changing the Message-Matching Function

When the CMMD library is initialized at the start of an application program, a default "matching algorithm" is in place on all nodes. This algorithm determines whether the sending node's ID and message tag match those requested by the receiving node.

However, any node executing a CMMD program can substitute a custom matching function. These user-written functions are not constrained by the "exact match" rules used by the default matching function. Instead, they can define, for meassages sent to their particular node, their own methods for interpreting matches between requested and supplied length, tag, and node values.

The function that sets the matching function is **`CMMD_set_msg_matching_fcn`**; this function can be called as often as desired, either to define new user-written matching functions or to re-institute the default function, at any time during the program.

15.2.1 How CMMD Uses the Matching Function

Implementation Note: When a node calls a CMMD communications operator to send a message (whether cooperative, asynchronous, or via virtual channels), the node generates an RTS (request to send) that is transmitted to the destination node. Similarly, when a node calls a CMMD operator to receive a message, it expects to receive an RTS from the source node.

The matching function is used by the destination node to determine which RTS goes with which receive operation. When any RTS arrives at the destination node, the matching function compares it with the arguments passed to the receive operator. If the matching function returns **`TRUE`**, an acknowledgment is sent to the source node, and the actual data transmission begins.

PLEASE NOTE

(1) Only calls containing legal values (that is, values in the range 0 <= value < **CMMD_partition_size()**, **CMMD_host_ node()**) will be seen by the matching function. Out-of-boundary values will cause errors when they are given to their original CMMD functions.

(2) A call to **CMMD_msg_pending** or **CMMD_mcb_pending** will generate a call to the matching function that has the maximum possible integer specified for the receive-side length.

15.2.2 Setting the Matching Function

The function **CMMD_set_msg_matching_fcn** can be called by any node to set the matching function for that node. It takes a single argument, either a pointer to a user-defined matching function, or **NULL** to reinstate the default matching function. The matching function operates on all messages being *received* by its node.

The user-defined matching function should expect six arguments:

- the node, tag, and length values supplied to the receive function on the destination node.
- the node address of the source node, along with the message tag and length specified by the source node in its send request.

The function should compare the arguments, and return **TRUE** if it decides that they match, and **FALSE** otherwise. (For Fortran, return values are 1 or 0.)

The matching function must be prepared to handle the keywords **CMMD_ANY_NODE** or **CMMD_ANY_TAG**. However, it is free to assign any useful meaning to these keywords — it is *not* required to implement the default wildcard matches.

For example, suppose we want to define a special tag to be used by a subset of the nodes — when sending a particular type of message to a particular node. The receiving node could define a matching function to handle that particular tag differently from all other tags — accepting messages with the special tag from any node in the subset, rejecting messages with that tag from other nodes, treating all other tags in the default manner.

Let's assume that we choose 96 for our special tag value, and that nodes 0 to 63 constitute our chosen subset.

Our matching function needs to return **TRUE** in the case that the send tag is 96, and the source node address is any value from 0 to 63 inclusive. It needs to return **FALSE** if the send tag is the special value, and the source node address is greater than 63. In all other cases, it must perform a normal matching comparison between the addresses and tags.

Here is an example of a simple matching function that does this:

```
int my_matching_function(rnode,rtag,rlen
                        snode,stag,slen)
{
    int match;
    match = FALSE;
    /* compare stag with arbitrary, user-defined tag */
    if (stag == 96)
      if ((snode >= 0) && (snode <= 63)) match = TRUE;
    else
      if (((rnode == CMMD_ANY_NODE) || (rnode == snode))
       && ((rtag == CMMD_ANY_TAG) || (rtag == stag)))
         match = TRUE;
return(match);
}
```

To install this function on node 0, the following code will suffice:

```
if (CMMD_self_address() == 0)
    CMMD_set_msg_matching_fcn(&my_matching_function);
```

When this custom matching function is not longer required, it can be removed:

```
if (CMMD_self_address() == 0)
    CMMD_set_msg_matching_fcn(NULL);
```

Note: This function tests on the special case first, the default case second. You can also test for exact matches first, then for any "broader matches" you choose to define. In the sample function above, reversing the order of the tests would result in acceptance of all exact matches on all tags, plus the additional broader matching on tag 96.

Note also that this function has no test for length. You may want to add that test to any matching function that you write.

15.2.3 Matching Function Definition Function

FUNCTION **`CMMD_set_msg_matching_fcn`**

C SYNTAX

CMMD_matching_function_t
`CMMD_set_msg_matching_fcn`
`(`CMMD_matching_function_t **`(*matcher)`**
`(`int *req_node*, int *req_tag*, int *req_length*,
int *send_node*, int *send_tag*, int *send_length***`))`**

FORTRAN SYNTAX

INTEGER FUNCTION **`CMMD_set_msg_matching_fcn(`***func*)
INTEGER FUNCTION **`func(`***req_node, req_tag, req_length,*
*send_node, send_tag, send_length***`)`**
integer *req_node, req_tag, req_length,*
integer *send_node, send_tag, send_length*

ARGUMENTS

matcher Pointer to user-supplied matching function that replaces the CMMD matching function on this node. (If *matcher* is **`NULL`**, the default matching function is restored. The supplied matching function should take six arguments, as follows:

req_node	The node address requested by the destination node.
req_tag	The user tag requested by the destination node.
req_length	The length requested by the destination node.
send_node	The actual source node of a message.
send_tag	The actual user tag for the message.
send_length	The length in bytes of the message to be sent.

RETURNS

This function returns a pointer to the message-matching function that has been supplanted by the function invoked by this call. Thus, an application program can set a new matching function within some procedure, then revert to the previous function at the procedure's end.

DESCRIPTION

Matching determines whether the node and tag requested by a receive function match those of an incoming message. CMMD's default matching function returns **`TRUE`** if and only if:

> the requested node is either **`CMMD_ANY_NODE`** or the same as *send_node*

and

> the requested user tag is either **`CMMD_ANY_TAG`** or the same as *send_tag*.

The default function does not match on message length. Users who want to match on length must supply their own functions.

Supplying a valid function name for the *matcher* argument to **`CMMD_set_msg_matching_fcn`** establishes the specified function on that node. From then on, CMMD receive functions use the specified function (rather than the default function) to determine whether an incoming message corresponds to the message requested by the receive function. Like the CMMD default function, the user's matching function must use **TRUE** and **FALSE** as its return values. (Since these are integer functions, Fortran versions return 1 or 0.)

Supplying a **NULL (*matcher)** argument to **CMMD_set_matching_fcn** re-establishes the CMMD default function on calling node.

15.3 Safety Checking

FUNCTIONS **CMMD_enable_safety()**
CMMD_disable_safety()

C SYNTAX

void **CMMD_enable_safety(**void**)**
void **CMMD_disable_safety(**void**)**

FORTRAN SYNTAX

SUBROUTINE **CMMD_enable_safety()**
SUBROUTINE **CMMD_disable_safety()**

DESCRIPTION

These two functions enable and disable CMMD safety mode. When invoked, they set a global variable and return immediately, with no return value.

When safety mode is enabled, CMMD send and receive functions, channel functions, and associated utility routines signal errors if the user either passes recognizably illegal arguments or calls the functions under illegal circumstances. In particular, the following errors are signaled where appropriate:

```
Negative node number.
Node number larger than the current partition.
Negative user tag.
Null buffer with nonzero length.
Negative element size.
```

```
Negative element stride.
Negative element count.
Element size too large.
Element stride too large.
Element count too large.
Invalid MCB.
Invalid send MCB.
Invalid receive MCB.
Invalid channel ID.
Invalid send channel ID.
Invalid receive channel ID.
Parallel array address not doubleword-aligned.
Parallel array length not multiple of 16 bytes.
Strided parallel array message attempted.
Function illegally called from an interrupt handler.
```

CMMD code runs more efficiently with safety mode disabled. However, with safety mode disabled, any of the errors listed above may cause code to fail or crash at an obscure time or place, making the cause difficult to diagnose. By default, safety mode is disabled in the optimized CMMD library and enabled in the CMMD debug library. (The latter is used mostly for internal work.)