Chapter 15

Input/Output

Message-passing programs require flexible I/O. Any given application may want to have

- one node read or write a file
- all nodes read or write a file, with each node acting independently of the others
- all nodes read or write a file in some cooperative fashion

CMMD extends UNIX I/O to offer this flexibility. Extensions center on

- a set of *CMMD I/O modes*, which define the way nodes interact with a file
- a small number of *CMMD I/O functions*, to supplement standard UNIX I/O functions

This chapter first discusses general file I/O, and the CMMD extensions that support it. It then discusses sockets (Section 15.10), some special considerations concerning Fortran I/O (Section 15.11), plus three physical I/O routines provided for wizards only (Section 15.12).

Device Independence. CMMD I/O is device-independent. One set of operations accesses files from standard UNIX file systems, a Scalable Disk Array (SDA), or a DataVault.

Please note an important implication of this device independence: Users should not attempt to mix CMMD I/O with CMFS I/O routines. The reason is that CMMD associates internal I/O metadata with file descriptors (to describe the current position of file pointers, parallel blocking factors, etc.),

Copyright © 1995 Thinking Machines Corporation

and CMFS routines do not know about this metadata. Therefore, mixing CMMD and CMFS calls may cause CMMD's I/O metadata to be incorrect and may cause subsequent I/O operations to fail.

Languages. CMMD I/O can be used from all languages in which CMMD programs can be written. The following facts should be noted:

- Some CMMD I/O functions, which are callable from Fortran, CM Fortran, or C, accept a UNIX file descriptor argument when called from C. For such functions, users should always pass the corresponding Fortran unit number instead of a UNIX file descriptor.
- Some Fortran I/O operations do not fully meet UNIX expectations; Section 15.11 discusses how to handle these situations.
- CM Fortran Utility Library I/O routines cannot be called from CMMD programs. See Section 15.11 for information about making I/O calls from CM Fortran.
- Buffered I/O is not fully supported in CMMD. See Section 15.5 for details.

15.1 CMMD I/O Modes

In order to support UNIX functionality while simultaneously extending it to support parallel operations, CMMD introduces the concept of *I/O modes*. With every file descriptor in every node's program, CMMD always associates one of the following four CMMD I/O modes:

- *local independent* (keyword: `CMMD_local`)
- *global independent* (keyword: `CMMD_independent`)
- *global synchronous broadcast* (keyword: `CMMD_sync_bc`)
- *global synchronous sequential* (keyword: `CMMD_sync_seq`)

Throughout the text we refer to the I/O modes by their keyword, minus the `CMMD_` prefix, that is, `local`, `independent`, `sync_bc`, and `sync_seq`. Two groups of I/O modes are often referred to by the first or second word in their full name, that is, the *global modes*, meaning `independent`, `sync_bc`, and `sync_seq`, or the *synchronous modes*, meaning `sync_bc` and `sync_seq`.

Version 3.3, April 1995
Copyright © 1995 Thinking Machines Corporation

Each file descriptor or stream, when created, starts out in one of these modes. The mode affects the behavior of `read`, `write`, and other UNIX functions that take a file descriptor or stream as an argument.

The CMMD I/O modes support four basic "traffic patterns" of data movement between nodes and files. These are illustrated in Figure 20. (The figure shows *reading* only, although *writing* may be visualized by simply reversing the directions of the arrows.)

Note that the I/O mode does not just affect the pattern of data movement. It also determines whether, and on which operations, the nodes are synchronized. (Refer to Chapter 10 for details on synchronization.)

In the `sync_bc` or `sync_seq` modes, operations on file descriptors automatically synchronize all the nodes (just like scans or reduces; see Chapter 9). In the `local` and `independent` modes, in contrast, files may be read or written without synchronizing the nodes.

In `local` mode, files may not only be *accessed* without synchronizing, but also *opened* without synchronizing. Also, a node may or may not share a file with other nodes, and the node that opens a file may be the only node that ever knows about the file. In `independent` mode, however, files *must* be opened by all nodes, and opening is a synchronizing operation. Files in `independent` mode are known by *all* nodes, but may be accessed without synchronizing.

In `independent` mode all nodes share a single file descriptor internally, thus referring simultaneously to a single file, but they maintain independent, distinct file pointers within that file. File descriptors in `independent` mode may thus be said to exist on a "per-process basis," and file descriptors in `local` mode may be said to exist on a "per-node basis." In the `local` case, both the file descriptors *and* the file pointers exist on a per-node basis. In the synchronous modes (`sync_seq` and `sync_bc`), both the file descriptors and the file pointers always exist on a per-process basis.

Copyright © 1995 Thinking Machines Corporation

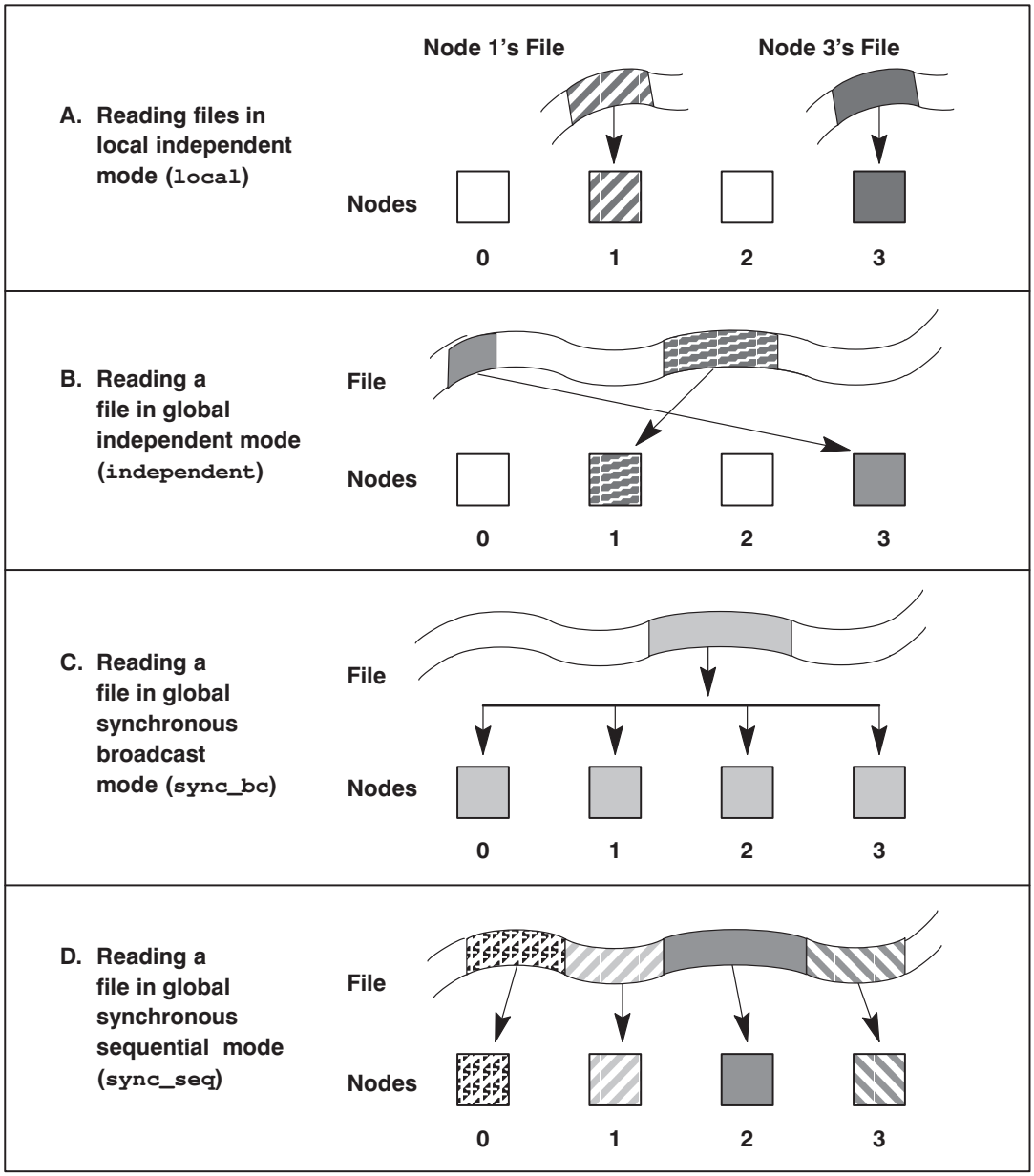

Figure 20. Four patterns for reading a file.

Version 3.3, April 1995
Copyright © 1995 Thinking Machines Corporation

In summary, as far as *reading and writing* are concerned, the following table constitutes a convenient conceptual grouping:

Reading and Writing

per-node	per-process
`CMMD_local_independent`	`CMMD_sync_seq`
`CMMD_global_independent`	`CMMD_sync_bc`

(the synchronous modes)

As far as *opening and closing* are concerned, the following table is more appropriate:

Opening and Closing

per-node	per-process
`CMMD_local_independent`	`CMMD_global_independent`
	`CMMD_sync_seq`
	`CMMD_sync_bc`

(the global modes)

Usage Note: File I/O operations performed in either of the global synchronous modes require that the file pointers on all the nodes be at the same position in the file. If they are not, then all file pointers are adjusted forward to match the file pointer that is the maximum distance into the file. This rule, called the *law of maximum*, can cause unexpected effects in code that does not take it into account.

15.1.1 Local Independent Mode

Completely independent I/O operations from individual nodes are supported by the `local` mode. In this mode, individual nodes can read and write different files independently, without having to coordinate their activities with other nodes. Nodes maintain their own local file descriptors and file pointers. (See Figure 20A.)

Copyright © 1995 Thinking Machines Corporation

`local` mode may be viewed, from the standpoint of *functionality*, as if the nodes were separate computers sharing a single file system. From the standpoints of *performance* and *resource consumption*, however, that is not an accurate model. In `local` mode, every node's I/O operations — including data transfers themselves — are processed internally by the host. This means that the host is a potential I/O bottleneck. Furthermore, the host internally maintains a separate UNIX file descriptor for every open file in `local` mode. This means that the nodes can easily swamp the host's file descriptor capacity. (The available number of this critical resource depends on the host's UNIX configuration, but is approximately 256.)

`local` mode is normally used for accessing standard UNIX file systems (rather than the SDA or the DataVault).

Here is an example of reading and writing a file in `local` mode:

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

/* Demonstration of local independent mode (CMMD_local). */

main()
{
  int fd, self = CMMD_self_address();
  int nnodes = CMMD_partition_size();
  char filename[32], x[32], y[32], z[32];

  CMMD_set_open_mode(CMMD_local); /* This determines the
                                     mode of all files
                                     subsequently opened;
                                     see Section 15.2 */

  /* Each node opens its own file with the the node number
     embedded in the filename.  Each node then creates a
     random string (called x) and writes it into its file. */

  sprintf(filename, "file_owned_by_node_%d", self);
  if(0>(fd=open(filename, O_WRONLY|O_CREAT, 0666)))
    CMMD_error("%d open1", self);

  srand(self);
  sprintf(x, "%032d", rand());
  if(0>write(fd, x, 32)) CMMD_error("%d write", self);
```


Version 3.3, April 1995
Copyright © 1995 Thinking Machines Corporation

```
  if(close(fd)) CMMD_error("close1 %d", self);

  /* Each node reads the string written by the next
     node into a string called y, by reading from the
     next node's file: */

  sprintf(filename, "file_owned_by_node_%d", (self+1)%nnodes);
  if(0>(fd=open(filename, O_RDONLY, 0666)))
    CMMD_error("open %d", self);

  if(0>read(fd, y, 32)) CMMD_error("read %d", self);

  if(0>close(fd)) CMMD_error("close %d", self);

  /* Each node now holds the next node's x in y.
     Check results by getting x again, this time using
     message passing, and putting x in z: */

  CMMD_send_and_receive((self+1)%nnodes, 0, z, 32,
              (self-1+nnodes)%nnodes, 0, x, 32);

  /* Check: */
  if(strncmp(y, z, 32))
    CMMD_error("miscompare");
}
```

15.1.2 Global Independent Mode

In `independent` mode, a single file can be accessed by *all* nodes independently. Thus, whereas in `local` mode the nodes may be regarded as separate computers sharing a single file system, in `independent` mode (and in fact in all three global modes), the nodes act like separate computers sharing a single *file*. (See Figure 20B.)

Every node, however, maintains its own pointer to the file, and can move that pointer about at will, reading or writing the file independently of the other nodes. In this way, I/O operations on files opened in `independent` mode are very similar to those of files opened in `local` mode, with a very important difference: the user may *change* the mode of files opened in `independent` mode to one of the synchronous modes (which are descibed

Copyright © 1995 Thinking Machines Corporation

in detail below). Files opened in `local` mode must remain in `local` mode for as long as they are open.

Another point of contrast between `independent` mode and `local` mode (and, in fact, between all global modes and `local` mode), is that files in `independent` mode consume internally only a single file descriptor apiece on the host. This conserves the number of file descriptors used on the host, which is often a critical resource. Thus, for example, if each node were to open the same file `foo` in `local` mode, and then open another file `bar` in `independent` mode, `bar` would use up *`partition_size`* file descriptors internally on the host, whereas `bar` would use up only one. (**Implementation Note:** The internal host file descriptors, which CMMD uses on behalf of node I/O operations, are invisible to the user; the user never sees them and never need worry about manipulating them.)

Here is an example of reading and writing a file in `independent` mode:

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

/* Demonstration of global independent mode
   (CMMD_independent). */

main()
{
  int fd, self = CMMD_self_address();
  int nnodes = CMMD_partition_size();
  char x[32], y[32], z[32];

  CMMD_set_open_mode(CMMD_independent);/* This determines
                                          the initial mode
                                          of all files
                                          subsequently opened;
                                          see Section 15.2 */

  /* Each node "globally" opens the same file. Each node
     then creates a random string (called x) and writes it
     into its own place in the file. */

  if(0>(fd=open("demo_file", O_RDWR|O_CREAT, 0666)))
    CMMD_error("%d open", self);
```


Copyright © 1995 Thinking Machines Corporation

```
  srand(self);
  sprintf(x, "%032d", rand());
  if(0>lseek(fd, self*32, SEEK_SET))
    CMMD_error("%d seek", self);
  if(0>write(fd, x, 32)) CMMD_error("%d write", self);

  /* Each node reads the string written by the next node
     into a string called y, by reading from the next node's
     position in the file: */

  if(0>lseek(fd, ((self+1)%nnodes)*32, SEEK_SET));
  if(0>read(fd, y, 32)) CMMD_error("read %d", self);

  if(0>close(fd)) CMMD_error("close %d", self);

  /* Each node now holds the next node's x in y.
     Check results by getting x again, this time using
     message passing, and putting x in z: */

  CMMD_send_and_receive((self+1)%nnodes, 0, z, 32,
              (self-1+nnodes)%nnodes, 0, x, 32);

  /* Compare: */
  if(strncmp(y, z, 32))
    CMMD_error("miscompare");
}
```

15.1.3 Global Synchronous Broadcast Mode

In `sync_bc` mode, for each file, all nodes read or write the same data as one another, on each read or write operation. (See Figure 20C.) Like the other two global modes (`independent` and `sync_seq`), files opened in `sync_bc` mode consume internally only one file descriptor apiece on the host.

On a write operation in `sync_bc` mode, each node must write from a buffer of data that is identical across all the nodes. However, only one copy of the data is written. On a read operation, all nodes must read the same amount of data, and each node's buffer is filled with a separate, identical copy of the data.

Copyright © 1995 Thinking Machines Corporation

sync_bc mode is particularly useful in reading — for example, when all nodes must obtain the same input from the user's terminal or the same information from a file (see Example 1 in Figure 21). It may also be convenient for writing messages to the terminal (for programs — or sections of programs — that are written in the data parallel style; refer to Chapter 1), since it is not necessary to explicitly select a single node to perform the write. For this reason CMMD internally opens stdout in sync_bc mode by default. Here is a trivial example of writing to stdout in sync_bc mode:

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

  main()
  {
    printf("Hello world, from all the nodes.\n");
  }
```

The output of this program looks like this:

```
prompt> hello_world
Hello world, from all the nodes.
prompt>
```

IMPORTANT

1. In sync_seq mode, a read or write call on a file or stream will synchronize all the nodes in the partition, just as if it were a call to CMMD_sync_with_nodes (or a scan or reduce routine). As with any globally synchronizing CMMD operation, each read or write in sync_seq mode must eventually be participated in by all nodes. The starting position for the operation is the maximum of the file pointers of the participating nodes. Following the operation, all file pointers are set to the last byte read or written.

2. In sync_bc mode, the I/O buffers on all nodes must be of the same length, and must contain the same data. If the buffers are not the same, file operations will not work correctly, and will produce undefined results.

Version 3.3, April 1995
Copyright © 1995 Thinking Machines Corporation

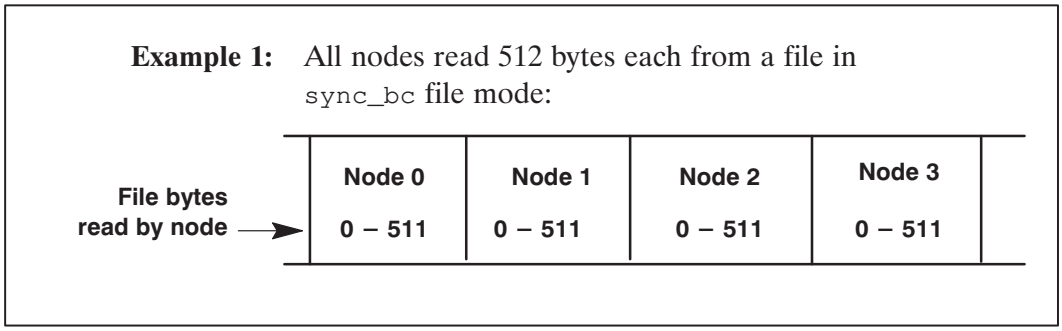

Figure 21. Effect of file mode on data received by nodes.

Here is an example of reading and writing a file in `sync_bc` mode:

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

/* Demonstration of global synchronous broadcast mode
   (CMMD_sync_bc). */

main()
{
  char x[32] = "Hello.", y[32];
  int fd, self = CMMD_self_address();

  /* Each node "globally" opens same file, in sync_bc
     mode.  Each node then writes an identical string x
     to the file. One node reads the file into y in CMMD_local
     mode, showing that it contains at least one copy of x,
     but fails to read beyond 32 bytes, showing that it
     contains only one copy. */

  CMMD_set_open_mode(CMMD_sync_bc);/* This determines the
                                      initial mode of all
                                      files subsequently
                                      opened; see
                                      Section 15.2 */
  if(0>(fd=open("file1", O_WRONLY|O_CREAT, 0666)))
    CMMD_error("pn%d open1", self);
```


Copyright © 1995 Thinking Machines Corporation

```
if(0>write(fd, x, 32)) CMMD_error("pn%d write1", self);
if(close(fd)) CMMD_error("close1 pn%d", self);

CMMD_set_open_mode(CMMD_local); /* Subsequent opens will
                                   now be in local mode*/
if( self == 0 ) {
  if(0>(fd=open("file1", O_RDONLY, 0666)))
    CMMD_error("pn%d open2", self);
  if(0>read(fd, y, 32)) CMMD_error("read1 pn%d", self);
  if(strncmp(x, y, 32)) CMMD_error("miscompare1");
  if(0>lseek(fd, 32, SEEK_SET))
    CMMD_error("pn%d seek", self);;
  if(0!=read(fd, y, 32))
    CMMD_error("read2 pn%d should give error", self);
  if(close(fd)) CMMD_error("close2 pn%d", self);

  /* Conversely, node 0 (only) writes a single copy of
     string x to its own file, in CMMD_local mode.  All
     nodes read the file, in CMMD_sync_bc mode, to show
     that all nodes receive a copy of the same string. */

  if(0>(fd=open("file2", O_WRONLY|O_CREAT, 0666)))
    CMMD_error("pn%d open3", self);
  if(0>write(fd, x, 32)) CMMD_error("pn%d write2", self);
  if(close(fd)) CMMD_error("close3 pn%d", self);
}

CMMD_set_open_mode(CMMD_sync_bc);
if(0>(fd=open("file2", O_RDONLY, 0666)))
  CMMD_error("pn%d open4", self);
bzero(y, 32);
if(0>read(fd, y, 32)) CMMD_error("pn%d read3", self);
if(strncmp(x, y, 32)) CMMD_error("miscompare2");
if(close(fd)) CMMD_error("close4 pn%d", self);
```


Copyright © 1995 Thinking Machines Corporation

15.1.4 Global Synchronous Sequential Mode

`sync_seq` mode allows each node to read (or write) sequential portions of a file. (See Figure 20D.) For write operations, each node contributes a buffer of data (of arbitrary size) to be written, and the data is written into the file sequentially, according to the node number, that is, node 0 first, node 1 next, etc. For read operations, each node requests a specific number of bytes of data, and the data is distributed across the nodes sequentially by node number. Each node's block of data begins where the preceding node's data block ends. (See Example 2 in Figure 22.)

Like the other two global modes (`independent` and `sync_bc`), files opened in `sync_seq` mode internally consume only one file descriptor apiece on the host. Note that `sync_seq` mode allows the amount of data read (or written) by each node to be different on a given synchronized read or write call; the buffer on each node can have a different size. Nodes not having any data to contribute simply supply a buffer length of zero. (See Example 3 in Figure 22.)

IMPORTANT

In `sync_seq` mode, a read or write call on a file or stream will synchronize all the nodes in the partition, just as if it were a call to `CMMD_sync_with_nodes` (or a scan or reduce routine). As with any globally synchronizing CMMD operation, each read or write in `sync_seq` mode must eventually be participated in by all nodes. The starting position for the operation is the maximum of the file pointers of the participating nodes. Following the operation, all file pointers are set to the last byte read or written.

Copyright © 1995 Thinking Machines Corporation

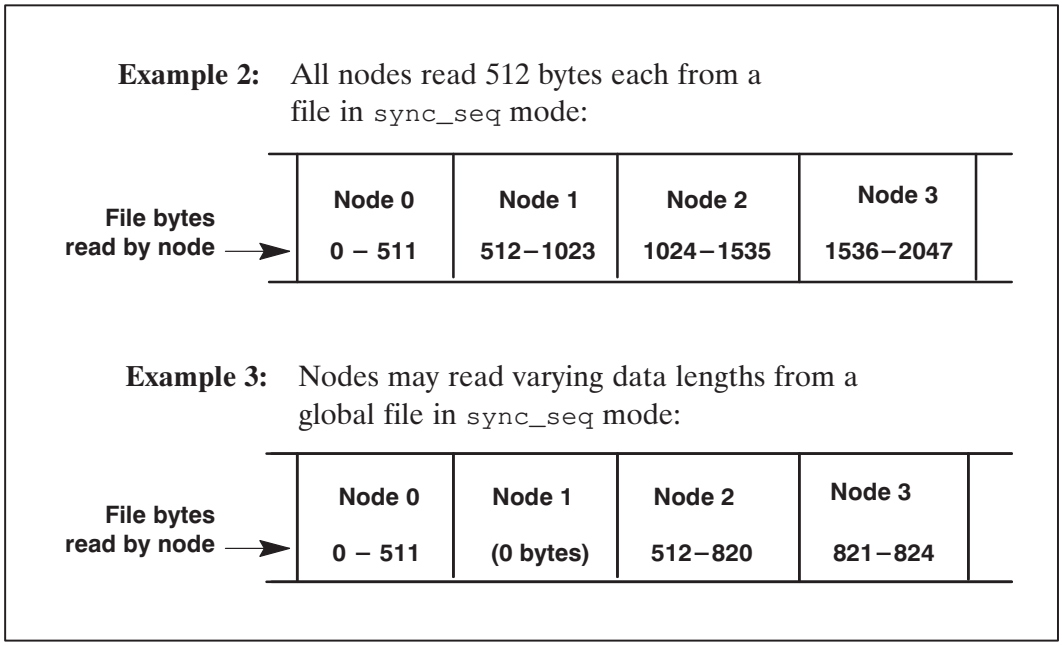

Figure 22. Nodes may read varying amounts of data in `sync_seq` mode.

Fortran. Fortran 77 and CM Fortran each have restrictions on the use of `sync_seq` mode. Please see Section 15.11 for details.

Here is an example of reading and writing a file in `sync_seq` mode:

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

/* Demonstration of global synchronous sequential mode
   (CMMD_sync_seq). */

main()
{
  char x[32], y[32];
  int fd, self = CMMD_self_address();

  /* Each node opens same file, in sync_seq mode. Each
     node then creates a string x, with the node number
     embedded in it, and writes x to the file. All nodes
```


Copyright © 1995 Thinking Machines Corporation

```
   then read successive 32-byte chunks from the file
   in independent mode, using lseek. */

CMMD_set_open_mode(CMMD_sync_seq);/* This determines the
                                     initial mode of all
                                     files subsequently
                                     opened; see
                                     Section 15.2 */
if(0>(fd=open("file1", O_WRONLY|O_CREAT, 0666)))
  CMMD_error("pn%d open1", self);
sprintf(x, "Hello from pn%d", self);
if(0>write(fd, x, 32)) CMMD_error("pn%d write1", self);
if(close(fd)) CMMD_error("close1 pn%d", self);

CMMD_set_open_mode(CMMD_independent);
if(0>(fd=open("file1", O_RDONLY, 0666)))
  CMMD_error("pn%d open2", self);
if(0>lseek(fd, self*32, SEEK_SET))
  CMMD_error("pn%d seek1", self);
if(0>read(fd, y, 32)) CMMD_error("read1 pn%d", self);
if(strncmp(x, y, 32)) CMMD_error("miscompare1");
if(close(fd)) CMMD_error("close2 pn%d", self);

/* Conversely, each node globally opens a file, in
   CMMD_independent mode.  Each node then writes x to
   successive 32-byte chunks in the file, using lseek.
   All nodes then read the file in sync_seq mode. */

if(0>(fd=open("file2", O_WRONLY|O_CREAT, 0666)))
  CMMD_error("pn%d open3", self);
if(0>lseek(fd, self*32, SEEK_SET))
  CMMD_error("pn%d seek2", self);
if(0>write(fd, x, 32)) CMMD_error("write2 pn%d", self);
if(close(fd)) CMMD_error("close3 pn%d", self);

CMMD_set_open_mode(CMMD_sync_seq);/* This determines the
                                     initial mode of all files
                                     subsequently opened;
                                     see Section 15.2 */`
```


Copyright © 1995 Thinking Machines Corporation

```
  if(0>(fd=open("file2", O_RDONLY, 0666)))
    CMMD_error("pn%d open4", self);
  bzero(y, 32);
  if(0>read(fd, y, 32)) CMMD_error("read2 pn%d", self);
  if(strncmp(x, y, 32)) CMMD_error("miscompare2");
  if(close(fd)) CMMD_error("close4 pn%d", self);
}
```

15.2 Setting the File Descriptor Mode

By default, the UNIX `open` call opens files in `local` mode.* Because applications often wish to open files for use in other modes, CMMD provides two mechanisms that change the default behavior of `open`: one that changes the default mode and one that accomplishes a single global open in a selected mode.

Similarly, an application often wants to change the mode of a file opened in one of the global modes *after* that file has been opened and accessed in a given mode. For example, all nodes may initially need to read a file's header information in `sync_bc` mode, then switch to `sync_seq` mode to distribute the file's data across the machine. CMMD provides a mechanism for this as well.

The CMMD functions that control the I/O mode of a file descriptor are as follows:

- `CMMD_set_open_mode` determines CMMD's current *open mode,* which is the default I/O mode of all subsequently opened files. If the open mode is set to one of the global modes, then each subsequent open must be performed as a global operation (i.e., like a scan or a reduce), with all nodes eventually participating in the operation.
- Alternatively, C applications can use `CMMD_global_open`, `CMMD_global_creat`, or `CMMD_global_fopen` to globally open a file in a particular I/O mode without regard for the current open mode. These global functions do not change the open mode for subsequent file opens.

* This does not apply to standard input, output, and error streams, which the system opens in `sync_bc` mode. See Section 15.4 for information on how to change modes for these streams.

Copyright © 1995 Thinking Machines Corporation

- To change the I/O mode of a file or stream opened in one of the global modes, once it is open, one or more nodes can call `CMMD_set_io_mode` or `CMMD_fset_io_mode`.
- To find out the I/O mode in which a file or stream has been opened, a node can call `CMMD_get_io_mode` or `CMMD_fget_io_mode`.

Reference pages for these functions are in Section 15.8.

15.3 I/O Error Handling

In general, UNIX error handling is not modified by CMMD I/O. However, CMMD does define three new error types, which are recognized by `perror`:

- `ECMMDBADMODE`

  The file descriptor passed into the I/O operation is currently set to a mode that is not permitted by the operation. For example, a file descriptor in `local` mode may not be passed to `CMMD_set_io_mode`.

- `ECMMDINCONSISTENT`

  The arguments to a global I/O operation are not identical across all nodes.

- `ECMMDNOMATCH`

  A global I/O function has been called on each node, but the function called is not the same on every node.

15.4 Using Append Mode for Global Output Files

Often an application program will make use of a "global output file," i.e., a file opened in one of the global modes, and written to by all the nodes. Typically such a file is opened in `independent` mode, especially when the nodes are expected to write their data at unpredictable times. For example, an application might need a *log file*, to which the nodes can write progress-update messages whenever necessary. Multiple messages may be written independently from different nodes, but all messages end up in the *same file*.

Copyright © 1995 Thinking Machines Corporation

Generally speaking, such files should be opened in (or later changed to) UNIX append mode, for the following reasons:

When a file (or any seekable device) is accessed in `independent` mode, each node has its own file pointer, which by default starts at the same point in the file. If nodes write data independently to the file without taking any special action to adjust their file pointers, each node's data starts at the same position in the file, overwriting data written by any preceding node. If all nodes write the same amount of data, only the data written last (by some arbitrary, unpredictable node) is retained.

When you place a seekable device that is to be accessed in `independent` mode into UNIX append mode, you cause any data written to the file to be appended at the end, thus avoiding data loss through overwrites.

In C programs, append mode is selected by an `fcntl` statement, or by opening a file with `O_APPEND` as the mode. For example, if you redirect `stdout` to a file, you can use the following `fcntl` statement to put that file in append mode:

```
fcntl(fileno(stdout), F_SETFL, O_APPEND);
```

Fortran programs use a CMMD function, `CMMD_set_append_mode`, to do the same thing. A reference page for this function is provided in Section 15.11.5.

Consider, for example, what would happen if we modified our "hello world" program (see page 316) to use `independent` mode. Each node would start with identical file pointers and would attempt to print its own unique message from the identical starting position in `stdout`. If `stdout` were instead a file or other seekable device, each node would overwrite the previous node's output; when the program finished, only one node's message would appear in the file.

NOTE: Data written to a terminal, or other nonseekable device, does not have this problem. Since output directed to a nonseekable device cannot overwrite previous output, it is always displayed as if in append mode. However, no harm is done by putting a nonseekable device into append mode, so, if your program directs output to arbitrary devices, which may be either seekable or nonseekable, it is best to explicitly request append mode.

Version 3.3, April 1995
Copyright © 1995 Thinking Machines Corporation

To get independent output from every node, yet write to a seekable device, you would use append mode. The modified "hello world" program would then look like this:

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

main()
{
  FILE *f;

  CMMD_set_open_mode(CMMD_independent);
  if(0>(f=fopen("logfile", "w")))
    CMMD_error("pn%d fopen", CMMD_self_address());

  fcntl(fileno(f), F_SETFL, O_APPEND);
  fprint(f, "Hello world from node %d\n",
         CMMD_self_address());
}
```

and the output would look something like this (note that in `independent` mode, the nodes do not respond in a predictable order):

```
prompt> hello_world
prompt> cat logfile
Hello world from node 27
Hello world from node 24
Hello world from node 2
Hello world from node 13
  ...
  ...
  ...
prompt>
```

Whereas if the `fcntl` statement were commented out, the log file would contain just one line, from a random node:

```
prompt> hello_world
prompt> cat logfile
Hello world from node 17
prompt>
```


Copyright © 1995 Thinking Machines Corporation

15.5 Buffered Streams: Supported, but Not Guaranteed

Buffered I/O calls accumulate data in a buffer, internally writing or reading the buffer with a UNIX-level I/O call only when full (or when explicitly directed by the user to flush the buffer). For instance, if `stdio` is used to write to a file, the actual UNIX `write` call occurs when the total number of bytes written to the file reaches 8K, rather than at each call to a `stdio` library routine. This means that it is difficult or impossible for the programmer to guarantee synchronous calls across the partition to the underlying I/O routines.

The CMMD I/O library provides only UNIX-level I/O calls. It is important for the programmer to realize, therefore, that functions such as `printf` and `scanf` in C, and formatted `read`/`write` operations in Fortran 77, use buffered I/O. Buffered I/O calls are permitted in all CMMD I/O modes. They are not, however, guaranteed to work as expected in `sync_seq` mode or in `sync_bc` mode, unless all nodes read or write the same number of bytes of data on a given synchronized call. The reasoning is that, since CMMD relies on the C run-time library to provide the buffered I/O code, CMMD has no control over the timing of when a buffer spills and actually performs a read or write.

Buffered I/O presents particular problems for Fortran applications, since much of Fortran I/O is buffered. Section 15.11 provides suggestions for avoiding these difficulties.

15.6 Functions That Support Extra-Large Files

Since UNIX uses only a 32-bit integer file pointer, seek operations on very large files are limited. CMMD therefore supplements the standard UNIX operations with functions that take an argument of type `double`, indicating a position within an extra-large file. These functions are: `dlength`, `dseek`, `dtell`, and `dftruncate`.

Please Note: `dftruncate` is the *only* truncate function supported in CMMD.

Each of these functions is sensitive to the mode of the file descriptor and must be called — with identical arguments — by all nodes when given file descriptors that are in either `sync_bc` or `sync_seq` mode.

Version 3.3, April 1995
Copyright © 1995 Thinking Machines Corporation

15.6.1 Managing I/O Rearrange Space

Prior to CMMD Version 3.3 there was a two-gigabyte limit on global I/O using the SDA or DV. The reason was that CMMD I/O makes use of a temporary, non-user-visible "staging area" called the *rearrange area* that is used by OS routines which themselves have an inviolable two-gigabyte limit. Even though the amount of memory on an individual processing node does not approach this limit, a large group of nodes can easily exceed it. For example, if 256 nodes each attempt to write sixteen megabytes in `sync_seq` mode, then the total size is four gigabytes. Thus, even though each node has called its I/O routine with parameters that are locally reasonable, the underlying two-gigabyte limit is nevertheless exceeded.

In CMMD Version 3.3 and higher, all global I/O routines implicitly loop over a rearrange area whose size is always less than or equal to two gigabytes. If the total I/O size is greater, then the operation is transparently broken down into a sequence of smaller operations. Hence, there is no longer a two-gigabyte limit.

Some of the most useful aspects of this rearrange mechanism have been exposed to the user. There are three new routines:

```
int CMMD_set_rearrange_size(int fd, int size);
int CMMD_get_rearrange_size(int fd);
int CMMD_max_rearrange_size(int fd);
```

`CMMD_set_rearrange_size` allows the user to specify the size of the rearrange area, on a per-node, per-file descriptor basis. *This number must be a nonnegative integer multiple of 64*. `CMMD_get_rearrange_size` allows the user to query a file descriptor for the same quantity. `CMMD_max_rearrange_size` returns an integer equal to the largest allowable second argument to `CMMD_set_rearrange_size`.

By default, CMMD will break down an I/O operation into the shortest possible sequence, using the largest possible staging area per iteration (up to approximately two gigabytes). However, if a program has very tight memory constraints, the user may wish to set the staging area size to a smaller value, thereby reducing the degree to which CMMD I/O encroaches upon the rest of the program's memory. (Of course, this reduction incurs a performance penalty due to the overhead of the additional looping.)

For example, if the user has a 512-node partition, and wishes to write out a file consisting of sixteen megabytes per node, then the total I/O size (eight gigabytes) will cause CMMD to transparently break down the operation into

Copyright © 1995 Thinking Machines Corporation

four two-gigabyte operations, each requiring four megabytes of staging area per node. However, if the user cannot allow CMMD to take up that much space, then he or she can call `CMMD_set_rearrange_size` on that file descriptor with a second argument of (for example) one-half megabyte. This means that CMMD will transparently break down the operation into 32 separate writes, each using one-half megabyte of staging area per node.

`CMMD_get_rearrange_size` and `CMMD_max_rearrange_size` may be called individually and asynchronously on any node. They will always return the same value on every node.

`CMMD_set_rearrange_size`, in contrast, is a synchronous function (like `CMMD_reduce_xxx`). It must be called by all nodes simultaneously, and they must all pass the same arguments. *It will generate a negative error return code if* ***any*** *of the following restrictions are violated:*

- *`size`* must be nonnegative
- *`size`* must be an integer multiple of 64
- *`size`* must be less than or equal to the current value returned by `CMMD_max_rearrange_size(`*`fd`*`)`
- All nodes must eventually call `CMMD_set_rearrange_size`
- All nodes must pass the same value of *`fd`*
- All nodes must pass the same value of *`size`*

It is *strongly* recommended that the user always check for a negative return value and take appropriate action.

For all files, the default return value of `CMMD_get_rearrange_size` is zero. This is a *special value* or "shorthand" for telling CMMD to use the largest possible staging area. Therefore the following two calls

```
CMMD_set_rearrange_size(fd, 0);
```

and

```
CMMD_set_rearrange_size(fd,
CMMD_max_rearrange_size(fd));
```

are always functionally equivalent.

Version 3.3, April 1995
Copyright © 1995 Thinking Machines Corporation

15.7 Other UNIX Support

CMMD provides the programmer with I/O routines for opening, reading, and writing files using both standard UNIX I/O semantics and extensions that provide for parallel I/O. Support for such UNIX functions is provided by linking an I/O server routine into the host program. This is done transparently for hostless programs. A "canned" host program including the I/O server routine is provided by the `cmmd-ld` link utility. Users who wish to write host/node programs must include explicit calls to the I/O server routine (see Chapter 19, Section 19.1).

The CMMD I/O library also provides support for the following UNIX system and library calls. When called from the nodes, these functions operate as if they were called from an ordinary UNIX process. Individual nodes may call these functions at any time.

```
accept        fchown          ioctl         rmdir
access        fcntl           isatty        select
bind          fstat           link          sendto
chdir         fstatfs         listen        socket
chmod         getdtablesize   lseek         stat
chown         gethostbyname   lstat         statfs
close         gethostid       mkdir         symlink
connect       gethostname     open          tell
creat         getpagesize     perror        unlink
dup           getsockname     read          utimes
dup2          gettimeofday    readlink      write
fchdir        getwd           rename
```

Note that support for `ftruncate` is *not* provided. Applications should use the CMMD operator `dftruncate` instead. (See Section 15.8.)

IMPORTANT

1. CMMD emulations of UNIX system calls that take paths or file descriptors as arguments use blocking communications operations that synchronize the nodes. Accordingly, such system calls should *not* be made from within CMMD callback or handler functions, as deadlock may result.

Copyright © 1995 Thinking Machines Corporation

2. CMMD I/O modes apply *only* to system calls that take a file descriptor or stream as an argument. All other system calls operate in standard UNIX fashion, on one node independently, without regard for the CMMD I/O mode.

3. Any call to an OS function or a file system function must have word-aligned arguments.

The possibility of synchronization exists only for those UNIX I/O functions that take a file descriptor or stream as an argument, such as `read`, `write`, etc. Such operations coordinate the actions of nodes acting in concert to transfer data to and from the parallel I/O devices of the CM-5. UNIX calls that do not take a file descriptor, such as `mkdir`, `chdir`, `chmod`, etc. do not synchronize. They may be called by any node, and affect only that node.

Note, for example, that having all nodes call `mkdir` with the same arguments is not likely to do what was intended: one node will be successful, while the rest will fail because the directory already exists.

File Units. From the operating system point of view, a CMMD program is a single process. If one node changes the current directory by calling `chdir`, then it is changed for the process as a whole.

This single-process model influences how many files may be open at a given time. CMMD currently allows each process to have approximately 256 files open simultaneously, no matter what size partition the process runs on. Thus, a programmer on a large machine may find the limit on the number of open files is smaller than the number of nodes in the system. Each file opened in a global mode is treated as a single file, even though every node can access them. Thus, opening files in one of the global modes helps avoid problems associated with the limited numbers of file descriptors. It also provides performance advantages over local files that are shared by multiple nodes. The use of files opened in a global mode is, therefore, highly recommended.

Version 3.3, April 1995
Copyright © 1995 Thinking Machines Corporation

15.8 Reference Pages

This section contains reference pages for the CMMD functions that open files and control the I/O mode of files. The `global_open`, `global_creat`, and `global_fopen` functions are callable from C only, as are the `fset_io_mode` and `fget_io_mode` functions. All other functions in this section are callable from both C and Fortran.

- `CMMD_set_open_mode`
  `CMMD_get_open_mode`
- `CMMD_global_open`
- `CMMD_global_creat`
- `CMMD_global_fopen`
- `CMMD_set_io_mode`
  `CMMD_fset_io_mode`
- `CMMD_get_io_mode`
  `CMMD_fget_io_mode`

This section also contains reference pages for the three new functions that manage the rearrange area:

- `CMMD_set_rearrange_size`
  `CMMD_get_rearrange_size`
  `CMMD_max_rearrange_size`

Copyright © 1995 Thinking Machines Corporation

CMMD_set_open_mode
CMMD_get_open_mode

Set or retrieve the open mode (IO mode) for a file.

SYNOPSIS

C Syntax

```
int CMMD_set_open_mode(CMMD_file_mode_t io_mode)

CMMD_file_mode_t CMMD_get_open_mode(void)
```

Fortran Syntax

```
INTEGER FUNCTION CMMD_set_open_mode(io_mode)
    INTEGER io_mode

INTEGER FUNCTION CMMD_get_open_mode()
```

ARGUMENT

io_mode	The desired mode under which all subsequent calls to `open` are to be treated.

DESCRIPTION

`CMMD_set_open_mode` changes the behavior of UNIX `open` to match that given by the *io_mode* argument. Subsequent calls to `open` and `fopen` will use this *io_mode*. `CMMD_set_open_mode` returns 0 if successful, −1 otherwise.

NOTE: If the *io_mode* argument is not consistent across the nodes (that is, has the same value across all nodes) the `ECMMDINCONSISTENT` error state is set.

The companion function, `CMMD_get_open_mode` returns the current default setting for `open`.

The four possible *io_modes* are:

`CMMD_local`	Default. Treats each file operation as purely local to the calling node.

Version 3.3, April 1995
Copyright © 1995 Thinking Machines Corporation

`CMMD_independent`
Globally opens the file but allows each node to perform I/O operations independently on the file.

`CMMD_sync_seq` Globally opens the file and synchronizes all I/O operations. Read operations spread data across all nodes in processor order. Write operations output data from all nodes in processor order.

`CMMD_sync_bc` Globally opens the file and synchronizes all I/O operations. Read operations broadcast data to all nodes, write operations take data as if from a single node.

NOTE: Setting the *`io_mode`* to be one of the *global* modes, `CMMD_independent`, `CMMD_sync_seq`, or `CMMD_sync_bc`, causes all calls to `open` to be performed globally. Each node in the partition must therefore participate in each open call. An implicit synchronization is performed.

Copyright © 1995 Thinking Machines Corporation

CMMD_global_open

Opens a file in one of the three global IO modes.

SYNOPSIS

C Syntax

```
int CMMD_global_open(char *path, int flags, int mode)
```

ARGUMENTS

path	A character string containing the pathname for the file to be opened or created.
flags	Standard UNIX flags that indicate whether the file is to be opened for reading, writing, or both. (The flags provided by each CM I/O device differ — consult your I/O documentation for a specific list.)
mode	An integer representing the UNIX permission settings for the file (for example, global read/write access is 0666).

RETURNS

`CMMD_global_open` returns a CMMD descriptor in `CMMD_independent` mode for the opened file.

DESCRIPTION

This function opens the designated file in `CMMD_independent` mode across all nodes (optionally creating the file if one of the *flags* is `O_CREAT`).

In all cases `CMMD_global_open` synchronizes the nodes. All nodes must participate in the `CMMD_global_open`, and all must use the same arguments. Inconsistent arguments cause the `ECMMDINCONSISTENT` error condition to be set.

Version 3.3, April 1995
Copyright © 1995 Thinking Machines Corporation

NOTE

1. Global files are available to the nodes only; the host processor cannot access them.
2. This function is available for C programs only.
3. During CMMD global I/O, if all nodes call some function — but not the same one — with the same file descriptor, CMMD sets the `ECMMDNOMATCH` error state.

Copyright © 1995 Thinking Machines Corporation

CMMD_global_creat

Opens a file in global independent mode, creating the file if it does not already exist.

SYNOPSIS

C Syntax

```
int CMMD_global_creat(char *path, int mode)
```

ARGUMENTS

path A character string containing the pathname for the file to be opened or created.

mode An integer representing the UNIX permission settings for the file (for example, global read/write access is 0666).

DESCRIPTION

This function is similar to `CMMD_global_open`, except that, like the UNIX `creat` operator, it opens the file with the flags `O_WRONLY`, `O_CREAT`, and `O_TRUNC`.

NOTE

1. Global files are available to the nodes only; the host processor cannot access them.
2. This function is available for C programs only.
3. During CMMD global I/O, if all nodes call some function — but not the same one — with the same file descriptor, CMMD sets the `ECMMDNOMATCH` error state.

Version 3.3, April 1995
Copyright © 1995 Thinking Machines Corporation

CMMD_global_fopen

Opens a file in global independent mode and associates a stream with it.

SYNOPSIS

C Syntax

```
FILE *CMMD_global_fopen(const char *path, int mode)
```

ARGUMENTS

path A character string containing the pathname for the file to be opened or created.

mode A character string beginning with one of the following sequences:

- `r` Open a text file for reading.
- `w` Truncate text file to zero length or create it.
- `a` Append: Open/Create text file for writing at EOF.
- `r+` Open text file for update (reading and writing).
- `w+` Truncate or create text file for update.
- `a+` Open/create file for update, with writing allowed only at end of file.

DESCRIPTION

`CMMD_global_fopen` opens (in `CMMD_independent` mode) the file named by *path* and associates a stream with it. This allows buffered I/O to be done to and from the file. If the open succeeds, the function returns a pointer that is used to identify the stream in subsequent operations. If the open fails, the function returns a null pointer.

As with `CMMD_global_open`, `CMMD_global_fopen` is a synchronizing function. All nodes must call it with the same arguments.

Copyright © 1995 Thinking Machines Corporation

NOTE

1. Global files are available to the nodes only; the host processor cannot access them.
2. This function is available for C programs only.
3. During CMMD global I/O, if nodes call different functions with the same file descriptor, CMMD sets the `ECMMDNOMATCH` error state.

Version 3.3, April 1995
Copyright © 1995 Thinking Machines Corporation

CMMD_set_io_mode
CMMD_fset_io_mode

These functions change the I/O mode for a file or a stream.

SYNOPSIS

C Syntax

```
int CMMD_set_io_mode(int fd, CMMD_file_mode_t io_mode)
int CMMD_fset_io_mode(FILE *stream, CMMD_mode_t io_mode)
```

Fortran Syntax

```
INTEGER FUNCTION CMMD_set_io_mode(unit, io_mode)
    INTEGER unit
    INTEGER io_mode
```

ARGUMENTS

fd	File descriptor (or Fortran unit number) for the file whose mode is to be changed.
stream	Stream whose mode is to be changed.
io_mode	The desired mode in which the file is to be used. Allowable mode specifiers are the three "global" modes described in CMMD_set_open_mode:

```
CMMD_independent
CMMD_sync_seq
CMMD_sync_bc
```

RETURNS

The functions return 0 if successful, −1 otherwise.

Copyright © 1995 Thinking Machines Corporation

DESCRIPTION

The file whose mode is to be changed must be a global file (that is, opened with either `CMMD_sync_seq`, `CMMD_sync_bc`, or `CMMD_independent` mode as the default). Trying to use `CMMD_set_io_mode` on a local file causes `errno` to be set to `ECMMDBADMODE`.

`CMMD_set_io_mode` and `CMMD_fset_io_mode` may be called at any time. They do not require synchronization. It is therefore possible for a single node to switch a file in synchronous mode (`CMMD_sync_seq` or `CMMD_sync_bc`) to `CMMD_independent` in order to write out a piece of data, without having to synchronize. It is the responsibility of the programmer to ensure that such independent changes do not cause trouble for other nodes, which — not having participated in the change — continue to think of the file as being in the synchronous mode.

Because the system, rather than the programmer, opens standard input, standard output, and standard error streams, the first thing most programs typically do is set these streams to the desired modes. By default these stream are opened by the system in `CMMD_sync_bc` mode.

A common practice therefore is to have at the beginning of a program something like the following:

```
CMMD_fset_io_mode(stdin, CMMD_sync_bc);
CMMD_fset_io_mode(stdout, CMMD_independent);
CMMD_fset_io_mode(stderr, CMMD_independent);
```


Version 3.3, April 1995
Copyright © 1995 Thinking Machines Corporation

CMMD_get_io_mode
CMMD_fget_io_mode

These functions tell which I/O mode is associated with an open file or stream.

SYNOPSIS

C Syntax

```
CMMD_mode_t CMMD_get_io_mode(int fd)
CMMD_mode_t CMMD_fget_io_mode(FILE *stream)
```

Fortran Syntax

```
INTEGER FUNCTION CMMD_get_io_mode(unit)
    INTEGER unit
```

ARGUMENTS

fd	File descriptor (or Fortran unit number) for the file whose *io-mode* is being queried.
stream	The stream whose *io-mode* is being queried.

DESCRIPTION

CMMD_get_io_mode and CMMD_fget_io_mode return the *io-mode* for the specified file or stream. The calls can be used for local or global files. The *io-mode* returned will be one of the following:

```
CMMD_local
CMMD_independent
CMMD_sync_seq
CMMD_sync_bc
```

The functions return −1 and set errno to EBADF if the file descriptor does not identify an open file.

Copyright © 1995 Thinking Machines Corporation

CMMD_set_rearrange_size

Allows the user to control the amount of "scratch space" used by a particular file description during parallel I/O.

SYNOPSIS

C Syntax

```
int CMMD_set_rearrange_size(int fd, int size)
```

Fortran Syntax

```
INTEGER FUNCTION CMMD_set_rearrange_size(fd, size)
    INTEGER fd, size
```

ARGUMENT

fd	A file descriptor for a global file, intended to be used for `sync_sequential` I/O to the SDA or the DV.
size	The size (on a per-node basis) of the internal staging area or "scratch space" that CMMD is allowed to use for parallel I/O.

DESCRIPTION

This synchronous function allows the user to control the amount of "scratch space" used by a particular file during parallel I/O.

I/O to the SDA or DV makes use of a chunk of non-user-visible scratch space called the *rearrange area*. The size of the rearrange area (the *rearrange size*) is interpreted on a per-node basis, meaning that the total amount of scratch space that CMMD actually uses on a given I/O is equal to the per-node rearrange size times the partition size.

CMMD keeps track of the rearrange size on a per-file descriptor basis. The rearrange size used by a particular file descriptor *fd* may be changed using `CMMD_set_rearrange_size`. The default value for every file descriptor is zero. This does *not* mean that no rearrange area is used. Rather, it is a special value, or "shorthand," that tells CMMD to use the maximum possible amount of rearrange

Version 3.3, April 1995
Copyright © 1995 Thinking Machines Corporation

area. By default, CMMD chooses a per-node value whose product with `CMMD_partition_size` is approximately equal to two gigabytes.

`fd` may be any valid global file descriptor. *`size`* must be a nonnegative integer multiple of 64, less than or equal to the current value returned by `CMMD_max_rearrange_size(`*`fd`*`)` (see the man page for `CMMD_max_rearrange_size`). All nodes must call `CMMD_set_rearrange_size` with the same arguments.

SEE ALSO

`CMMD_get_rearrange_size`
`CMMD_max_rearrange_size`

Copyright © 1995 Thinking Machines Corporation

CMMD_get_rearrange_size

Returns the size of the per-node I/O rearrange area.

SYNOPSIS

C Syntax

```
int CMMD_get_rearrange_size(int fd)
```

Fortran Syntax

```
INTEGER FUNCTION CMMD_get_rearrange_size(fd)
    INTEGER fd
```

ARGUMENT

fd	A file descriptor for a global file, intended to be used for `sync_sequential` I/O to the SDA or the DV.

RETURNS

A nonnegative integer multiple of 64, equal to the size of the per-node I/O rearrange area for the given file descriptor, in bytes.

DESCRIPTION

(See `CMMD_set_rearrange_size`.)

This function returns the last value used as a second argument to `CMMD_set_rearrange_size`, which must always be a nonnegative integer multiple of 64. If `CMMD_set_rearrange_size` has never been called, then it returns zero. `CMMD_get_rearrange_size` is asynchronous, meaning that it may be called independently on any node, at any time.

Version 3.3, April 1995
Copyright © 1995 Thinking Machines Corporation

SEE ALSO

```
CMMD_set_rearrange_size
CMMD_max_rearrange_size
```

Copyright © 1995 Thinking Machines Corporation

CMMD_max_rearrange_size

Returns the largest size allowable for the per-node rearrange area.

SYNOPSIS

C Syntax

```
int CMMD_max_rearrange_size(int fd)
```

Fortran Syntax

```
INTEGER FUNCTION CMMD_max_rearrange_size(fd)
    INTEGER fd
```

ARGUMENT

fd	A file descriptor for a global file, intended to be used for `sync_sequential` I/O to the SDA or the DV.

RETURNS

A positive integer multiple of 64, equal to the greatest number allowable as a second argument to `CMMD_set_rearrange_size`.

DESCRIPTION

(See `CMMD_set_rearrange_size`.)

The per-node rearrange area is always limited such that its product with the partition size is approximately equal to two gigabytes. `CMMD_max_rearrange_size` returns this limit. Since passing zero as a second argument to `CMMD_set_rearrange_size` is a shorthand way of telling CMMD to "use the maximum possible rearrange size," the two calls

```
CMMD_set_rearrange_size(fd, 0);
```


Version 3.3, April 1995
Copyright © 1995 Thinking Machines Corporation

and

```
CMMD_set_rearrange_size(fd, CMMD_max_rearrange_size(fd));
```

are functionally equivalent. The first form is merely a convenient shorthand for the second. *Notice that* `CMMD_max_rearrange_size` *will never return zero.*

SEE ALSO

`CMMD_set_rearrange_size`
`CMMD_max_rearrange_size`

Copyright © 1995 Thinking Machines Corporation

15.9 Reference Pages, File Pointers

This section contains reference pages for the four functions that control the double-precision file pointers used with CMMD I/O. These functions are callable from both C and Fortran.

- `dlength`
- `dseek`
- `dtell`
- `dftruncate`

Copyright © 1995 Thinking Machines Corporation

dlength

Returns the length of a file, in bytes.

SYNOPSIS

C Syntax

```
double dlength(int fd)
```

Fortran Syntax

```
REAL*8 FUNCTION dlength(unit)
    INTEGER unit
```

ARGUMENTS

fd	The file descriptor (or Fortran unit number) for the file whose length is being queried.

DESCRIPTION

The function returns the length of the specified file in bytes, as a double-precision floating-point number.

Copyright © 1995 Thinking Machines Corporation

dseek

Positions the double-precision seek pointer within a file.

SYNOPSIS

C Syntax

```
double dseek(int fd, double offset, int whence)
```

Fortran Syntax

```
REAL*8 FUNCTION dseek(unit, offset, whence)
    INTEGER unit
    REAL*8  offset
    INTEGER whence
```

ARGUMENTS

fd	The file descriptor (or Fortran unit number) for the file whose pointer position is to be changed.
offset	Number of bytes of offset to be added to the starting position set by *whence*.
whence	One of the following constants:

SEEK_SET	Pointer is set to *offset* bytes from the beginning of the file.
SEEK_CUR	Pointer is set to current position plus *offset* bytes.
SEEK_END	Pointer is set to size of file plus *offset* bytes.

DESCRIPTION

The dseek function sets the seek pointer associated with the file referred to by the descriptor *fd*, in accordance with *offset* and *whence*. Note that *offset*, in these

Version 3.3, April 1995
Copyright © 1995 Thinking Machines Corporation

routines, is a double-precision floating-point number; this allows seeks of greater than 2 gigabytes.

Remember that each node keeps separate pointers for *fd*s in independent modes, but that CMMD keeps one global pointer for *fd*s in synchronous modes. For this reason,

- If the *fd*'s mode is `local` or `independent`, any node can make this call independently.
- If the *fd*'s mode is `sync_seq` or `sync_bc`, all nodes must make the call with identical arguments.

On success, this function returns the seek pointer location as measured in bytes from the beginning of the file. On failure, it returns −1 and sets `errno` to indicate the error. The seek pointer is not moved if the function call fails.

Copyright © 1995 Thinking Machines Corporation

dtell

Returns position of double-precision seek pointer within a file.

SYNOPSIS

C Syntax

```
double dtell(int fd)
```

Fortran Syntax

```
REAL*8 FUNCTION dtell(unit)
    INTEGER unit
```

ARGUMENT

fd	The file descriptor (or Fortran unit number) for the file whose pointer position is being queried.

DESCRIPTION

This function returns the current seek position in the specified file as a double-precision floating-point number.

Version 3.3, April 1995
Copyright © 1995 Thinking Machines Corporation

dftruncate

Truncates a file.

SYNOPSIS

C Syntax

```
int dftruncate(int fd, double length)
```

Fortran Syntax

```
INTEGER FUNCTION dftruncate(unit, length)
    INTEGER unit
    REAL*8  length
```

ARGUMENT

fd	The file descriptor (or Fortran unit number) for the file to be truncated.
length	The new length for the file (in bytes), expressed as a double-precision floating-point number.

RETURNS

The function returns 0 on success, −1 otherwise.

DESCRIPTION

This function changes the length of a file. If the supplied length is longer than the existing file, the file's length is extended; if shorter, the file is truncated. The file must be open for writing when this function is called.

Remember that each node keeps separate pointers for *fd*s in independent modes, but CMMD keeps one global pointer for *fd*s in synchronous modes. This means:

- If the *fd*'s mode is `local` or `independent`, any node can make this call independently.

Copyright © 1995 Thinking Machines Corporation

- If the *fd*'s mode is `sync_seq` or `sync_bc`, all nodes must make the call with identical arguments.

Version 3.3, April 1995
Copyright © 1995 Thinking Machines Corporation

15.10 Sockets

In CMMD, sockets are used exclusively for process-to-process communication. Sockets can carry data between

- two processes on a single partition
- two processes on different partitions of the same machine
- two processes on different machines

Sockets can make use of all four CMMD I/O modes. Thus, a wide variety of communication patterns is possible. For example,

- A single node in one process can talk to a single node in a second process.
- A single node, in `CMMD_independent` mode, can send data to an entire partition of nodes, who can receive it either in `CMMD_sync_seq` or `CMMD_sync_bc` mode.
- All nodes in one process, in `CMMD_sync_seq` mode, can send data to a single node in a second process.
- All nodes in one process can send data to all nodes in a second process.

CMMD provides a single socket function, `CMMD_global_socket`, which creates and opens a global socket. It also supports the UNIX systems calls `bind`, `connect`, `getsockname`, `listen`, `select`, and `sendto`.

15.10.1 Restrictions

- In Version 3.0 and higher of CMMD, sockets are available only on UNIX file systems. They are not supported for SDA (Scalable Disk Array) or DataVault files.
- Also in Version 3.0 and higher, all socket transmissions go through the host. Performance is therefore poor.
- Sockets cannot be used for inter-node communications within a single process.

Copyright © 1995 Thinking Machines Corporation

15.10.2 For Further Information

This manual does not explain socket programming. For discussions of socket programming in general, consult either of the following standard texts:

- W. Richard Stevens
  UNIX Network Programming
  Prentice Hall, 1990

- The *Network Programming Guide* in the Sun documentation set.

15.10.3 Reference Page, Sockets

This section contains the reference page for the `CMMD_global_socket` command.

Copyright © 1995 Thinking Machines Corporation

CMMD_global_socket

Creates a socket accessible to all nodes and opens it in global independent mode.

SYNOPSIS

C Syntax

```
int CMMD_global_socket(int domain, int type, int protocol)
```

ARGUMENTS

domain	The communications domain.
type	The socket type.
protocol	The socket protocol.

DESCRIPTION

This function creates a global socket (a socket accessible to all nodes, opened in global independent mode), with the given parameters (see the UNIX man page for `socket` for a description of these arguments).

I/O Modes. A socket is always opened in global independent mode, no matter whether the application has changed the open-mode default or not. If you want a socket to function in any other mode, you must change the mode with `CMMD_set_io_mode` after the socket has been opened. To do global synchronous sequential I/O to a socket, for example, you would first call `CMMD_global_socket` to create the socket. Second, you would call `CMMD_set_io_mode` to change the mode to `sync_seq`. You would then issue the `connect()` call, and then write your data globally in `sync_seq` mode.

Check Return Values. When you are reading data from a socket, you should always check the return value and continue reading until the read returns zero. You must do this because there is no guarantee that you will get all your data in a single transmission; context-switching (for example) might intervene.

Global and Single-Node Operations. For socket I/O, as for all other CMMD I/O, functions that take a file descriptor use CMMD semantics, while those that do not retain standard UNIX semantics. Thus, socket operations that take an *fd* ar-

Copyright © 1995 Thinking Machines Corporation

gument, such as opening, reading, writing, or closing the socket, can be done on all nodes; other operations, such as `bind`, `accept`, and `connect` can be called on one node only, because there is, in reality, only one socket involved.

Version 3.3, April 1995
Copyright © 1995 Thinking Machines Corporation

15.11 Fortran I/O from CMMD

CMMD I/O modes translate into Fortran terms in the following manner:

- `CMMD_local`: Each node does direct access I/O to its own file, accessing records according to the record argument.
- `CMMD_independent`: One file is shared, and each node can access a record in that file via the record argument.
- `CMMD_sync_bc`: All processors cooperatively access the same record in a file.
- `CMMD_sync_seq`: Processors access sequentially numbered records starting with the record accessed by processor 0.

  NOTE: CM Fortran programs generally can use `CMMD_sync_seq` mode only for direct access I/O.

This section discusses some items of particular concern to programmers writing Fortran applications using CMMD.

15.11.1 Conditional Write Statements

WRITE statements inside IF statements may produce different numbers of writes on each node. Writes in this context should, therefore always be done in `CMMD_independent` mode, not in `CMMD_sync_seq` mode.

15.11.2 Fortran 77 or CM Fortran?

An application can do I/O in only one version of Fortran. If your main program is in Fortran 77, you should use Fortran 77 I/O throughout. If your main program is in CM Fortran, you should use CM Fortran I/O throughout. Do not try to mix the two.

NOTE: This is a fundamental constraint. CM Fortran and Fortran 77 use different versions of `getfd`; at link time, the language used in the main program determines which version of `getfd` is used.

Copyright © 1995 Thinking Machines Corporation

15.11.3 Buffered I/O and Fortran 77

The CMMD I/O library provides only UNIX-level I/O calls. It is important for the programmer to realize that formatted `read`/`write` operations in Fortran 77 use buffered I/O. Buffered I/O calls are permitted in all modes. They are not, however, guaranteed to work as expected in `CMMD_sync_seq` mode. (`CMMD_sync_bc` mode should work, as long as all nodes read or write the same number of bytes of data.) Since CMMD relies on the C run-time library to provide the buffered I/O code, CMMD has no control over the timing of when a buffer spills and actually performs a read or write.

The conflict between UNIX and buffered I/O creates problems for Fortran I/O under the `sync_seq` and `sync_bc` modes, as the only provided Fortran I/O mechanisms are built on top of a layer of buffering.

Consider the concrete example of performing buffered writes to a line-buffered device. Every time a newline is encountered in the data, a write is performed. If this stream is in `CMMD_sync_seq` mode, these writes will attempt to synchronize. If only some nodes encounter a newline, requiring a write, they will idle awaiting synchronization. The other nodes, not having encountered a newline, could easily try some operation that attempts communication across the network. With several nodes awaiting a pending synchronization, there will be a collision in the CM-5's Control Network.

Several strategies for working around these problems exist, and are discussed in the following paragraphs.

Using the fileopt Option. Fortran 77 programs should always use the option `fileopt='buffer=`*`nnnn`*`'` when opening a file. This practice can prevent problems in global independent mode. Using `fileopt` can also make it possible to perform I/O to global `sync_seq` files, as long as all records in the file are the same size and all nodes read or write the same amount. The application must take the following two steps:

- The `open` statement must contain the option `fileopt='buffer=`*`nnnn`*`'`.
- The buffer size supplied in the open statement must be equal to the exact number of bytes transferred in a read or write.

  For sequential formatted I/O, *nnnn* is *linelength* + 1 (where 1 represents the newline character).

Version 3.3, April 1995
Copyright © 1995 Thinking Machines Corporation

For sequential unformatted I/O, *nnnn* is the data size + 8 (where 8 is the record marker).

For direct access I/O, *nnnn* is the data size.

Altering the Buffering Protocol. The function `CMMD_set_unit_buffer` alters the buffering protocol used by a UNIX stream that underlies a Fortran I/O unit. Normally, UNIX allocates a buffer when the stream is opened. When the buffer is full, it is flushed to the output device. In some cases, users would like to flush the buffer at the end of a line. In other instances, the buffer can be flushed after each character (this is equivalent to no buffering). The reference page for `CMMD_set_unit_buffer` can be found in Section 15.11.4.

Unbuffered I/O Functions. As another alternative, CMMD provides functions that allow Fortran programmers to write UNIX files by accessing the underlying, unbuffered, UNIX I/O calls. These functions are `CMMD_global_read` and `CMMD_global_write`. They are described in Section 15.11.4.

15.11.4 CM Fortran

Direct Access I/O. Direct access I/O is the best method of I/O for CM Fortran/CMMD programs. The programs should also use `CMMD_set_unit_buffer` to ensure the proper buffering policy.

In particular, note that in CM Fortran, sequential access I/O cannot be used with `CMMD_sync_seq` mode, because of an incompatibility in the implementations of CM Fortran and CMMD.

I/O from VU Memory. As of Version 3.1 and higher, global I/O synchronous-sequential or synchronous-broadcast can be done directly from VU memory.

Global Independent Mode. CM Fortran programs using formatted I/O in global independent mode should state explicitly the number of bytes being read or written by the `read` and `write` commands. They should avoid using the * specification, which has the potential to cause problems.

Copyright © 1995 Thinking Machines Corporation

SDA Access. Access to the SDA from CMF-on-a-node is done via normal CMF I/O calls. From CMF's point of view, all I/O occurs as if it were serial. For example, the code fragment

```
          integer, array( 10,10 ) :: a
CMF       layout a(:news, :news)

          open(unit=10, filename='foo', status='unknown')
          write( 10 ) a
```

causes each node to write 100 integers to unit 10. If the I/O mode is `sync_seq` and the file is on an SDA, then the I/O will occur at SDA speeds and the node contributions will appear in the file in processor address order.

No Utility Library I/O. Nodal CM Fortran programs cannot use CM Fortran Utility Library I/O calls.

15.11.5 Reference Pages, Fortran I/O Functions

This section contains the reference pages for the CMMD functions designed to assist in doing global CMMD I/O from Fortran programs:

- `CMMD_set_append_mode`
- `CMMD_set_unit_buffer`
- `CMMD_global_read`
- `CMMD_global_write`

Version 3.3, April 1995
Copyright © 1995 Thinking Machines Corporation

CMMD_set_append_mode

Sets a file into append mode. Available only in Fortran.

SYNOPSIS

Fortran Syntax

```
INTEGER CMMD_set_append_mode(unit)
    INTEGER unit
```

ARGUMENTS

unit	Fortran file unit number.

DESCRIPTION

This function (available only in Fortran) allows Fortran programs to place a seekable device into append mode. In this mode, any data written to the file is appended to the end of the file. In this way, any number of nodes can write independent I/O to a file without overwriting each other's data.

This functionality is particularly useful for output files, such as the UNIX `stdout` file.

The function returns a nonnegative value on success, −1 otherwise.

Copyright © 1995 Thinking Machines Corporation

CMMD_set_unit_buffer

Changes the buffering protocol of a UNIX stream. Available only in Fortran.

SYNOPSIS

Fortran Syntax

```
CMMD_set_unit_buffer(unit, size, policy)
    INTEGER   unit, size, policy
```

ARGUMENTS

unit	The Fortran unit number.
size	Size of the buffer to be allocated (if any).
policy	The buffer policy to be used:

0	_IOFBF	Fully buffers the input and output.
0200	_IOLBF	Line buffers the output; buffer is flushed when a newline is written, buffer is full, or input is requested.
04	_IONBF	No buffering on input or output.

DESCRIPTION

CMMD_set_unit_buffer alters the buffering protocol used by a UNIX stream that underlies a Fortran I/O unit. Normally, UNIX allocates a buffer when a stream is opened and flushes the buffer to the output device whenever it becomes full. In some cases, users would like to flush the buffer at the end of a line. In other cases, they would prefer the buffer to be flushed after each character (that is, they would prefer no buffering). CMMD_set_unit_buffer allows users to choose among these three options.

For further details on this function and on the significance of the constants used to set policy, see the UNIX setbuffer man page.

Version 3.3, April 1995
Copyright © 1995 Thinking Machines Corporation

CMMD_global_write
CMMD_global_read

These functions perform a global write or read of an unbuffered UNIX file from a Fortran program. Available only in Fortran.

SYNOPSIS

Fortran Syntax

```
INTEGER FUNCTION CMMD_global_read(unit, buffer, length)
INTEGER FUNCTION CMMD_global_write(unit, buffer, length)
    INTEGER unit
    CHARACTER*(*) buffer
    INTEGER length
```

ARGUMENTS

unit	Fortran file unit number.
buffer	Location into which data is to be read, or from which it is to be written.
length	Number of bytes to be read or written.

RETURNS

On success, these functions return the number of bytes written or read. On error, they return −1.

DESCRIPTION

The CMMD global read and write functions resolve to the underlying, unbuffered, UNIX I/O calls. Thus, they allow Fortran programmers to access these I/O mechanisms directly from either SPARC or VU memory. An application can use standard Fortran I/O functions to write data into character buffers, and then call the global CMMD I/O functions with the character data buffers as arguments.

Programmers should note the following issues regarding these functions:

Copyright © 1995 Thinking Machines Corporation

- The functions take unit numbers as arguments. Only flags (for example, `status=new`) that are checked at the time the units are opened are checked by these functions. Any other flags that affect writing to the unit (such as record format, etc.) are ignored.
- The file units must be in synchronous-sequential or synchronous-broadcast mode. The global read and write functions return errors if called on file units that are in `local` or in global independent mode.
- You can access any given file unit either by Fortran `read` and `write` statements, or by these CMMD functions, but not by both. (Using both may work, but such use is not supported.)
- Files created by `CMMD_global_write` are normal UNIX files, not standard Fortran files. In particular, they do not have record separators. This may or may not be a problem for a given application.

Version 3.3, April 1995
Copyright © 1995 Thinking Machines Corporation

15.12 For Wizards Only — Writing Device-Dependent Files

Typically, when data is written from the CM to an I/O device, the data is stored in a hardware-independent manner — in other words, in a format that doesn't depend on either the number of nodes in the CM or the number of disks (or bandwidth) in the I/O device. This allows the data to be read by a program running on a CM with a different hardware configuration.

Thus, before CM data is written to an I/O device, the data is rearranged into a hardware-independent I/O format. When data is read from a device, a reverse translation is performed. This translation is implicit in the standard CM I/O routines, and is transparent to programs that call these routines. However, the translation does incur a penalty in terms of speed and efficiency.

Where your application does not require a guarantee of hardware independence in a data file (for example, if you only need to write out a temporary file that is read back in during the same run of the program), you can avoid this translation step by calling a special set of I/O operators, described below.

WARNING

Use of these operators makes your files highly hardware-dependent: The files will be unreadable on any CM hardware configuration other than the one from which they were written (this includes partitions of a different size, etc.). Moreover, a file written by a `CMMD_physical_write` call can be read only by a `CMMD_physical_read` call. No other type of read will produce a useful result.

The format used by device-dependent file operations may change without notice in future software revisions.

Copyright © 1995 Thinking Machines Corporation

15.12.1 Physical Read/Write Restrictions

The operations described here have the following restrictions:

- They can be called only on a global file open in a global mode.
- The *offset* of the file pointer within the file must be a multiple of the physical blocking factor.
- The *buffer* argument must be aligned on a doubleword boundary. Its starting address must be identical on all nodes.
- The *length* argument must be a multiple of the physical blocking factor.

An error is signaled if these conditions are not met.

15.12.2 Physical Blocking Factor

The physical blocking factor of a file (or other device) is the minimum size of the data blocks read from and written to the device. You can get the physical blocking factor from a file descriptor by calling the `CMMD_physical_blocking` function.

15.12.3 Reference Pages

This section contains reference pages for

```
CMMD_physical_blocking
CMMD_physical_write
CMMD_physical_read
```


Copyright © 1995 Thinking Machines Corporation

CMMD_physical_blocking

Returns the physical blocking factor of the device on which the specified file is stored.

SYNOPSIS

C Syntax

```
int CMMD_physical_blocking(int fd)
```

FOrtran Syntax

```
INTEGER FUNCTION CMMD_physical_blocking(unit)
    INTEGER unit
```

ARGUMENT

fd	File descriptor (or Fortran unit number).

DESCRIPTION

`CMMD_physical_blocking` returns an integer representing the physical blocking factor of the device referenced by the *fd* argument. Currently, this is

- 16 bytes for a standard UNIX file
- 16 bytes times the number of disks for a Scalable Disk Array
- 512 bytes for a DataVault

This function is therefore useful when you want to know what type of file system holds a file that you want to read.

Copyright © 1995 Thinking Machines Corporation

CMMD_physical_write

Writes data, in physical-write order, from all nodes into the specified file.

SYNOPSIS

C Syntax

```
int CMMD_physical_write(int fd, void *buffer, int length)
```

Fortran Syntax

```
INTEGER FUNCTION CMMD_physical_write(unit, buffer, length)
    INTEGER unit
    <ARRAY> buffer
    INTEGER length
```

ARGUMENTS

fd	File descriptor (or Fortran unit number) for the file to be written.
buffer	Memory location from which data will be written. Must be doubleword-aligned, and must be the same across all nodes.
length	Total number of bytes to be written across all nodes. Must be a multiple of the blocking factor.

DESCRIPTION

`CMMD_physical_write` is a global function; it writes data from the specified buffer on all nodes simultaneously into the specified CM file. If successful, the routine returns the number of bytes written; otherwise, it returns −1.

Implementation Note: The data transfer performed by `CMMD_physical_write` is actually the second step in a synchronous-sequential write (or, for that matter, in a parallel write performed by data parallel software). The first step in such parallel writes is to reorder the data from its original position on the nodes into the physical-write order.

Version 3.3, April 1995
Copyright © 1995 Thinking Machines Corporation

CMMD_physical_read

Reads data, in physical-read order, from file onto all nodes.

SYNOPSIS

C Syntax

```
int CMMD_physical_read(int fd, void *buffer, int length)
```

Fortran Syntax

```
INTEGER FUNCTION CMMD_physical_read(unit, buffer, length)
    INTEGER unit
    <ARRAY> buffer
    INTEGER length
```

ARGUMENT

fd	File descriptor (or Fortran unit number) for the file to be read.
buffer	Memory location into which data will be read. Must be double-word-aligned, and must be the same across all nodes.
length	Total number of bytes to be read across all nodes. Must be a multiple of the file's blocking factor.

DESCRIPTION

`CMMD_physical_read` is a global read; it reads data into the specified buffer on all nodes simultaneously. It spreads data across all nodes in a predetermined order. If successful, the routine returns the number of bytes read; otherwise it returns −1.

Implementation Note: The data transfer performed by `CMMD_physical_read` is actually the first step in a synchronous-sequential read (or, for that matter, in a parallel read performed by data parallel software). The second step in such parallel reads is to reorder the data so that it looks as though it were read sequentially into each node.

Copyright © 1995 Thinking Machines Corporation