Chapter 12

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 in which nodes interact with a file, allowing local, global independent, and two versions of global synchronized operations
- special CMMD file descriptors, which contain CMMD I/O mode information, as well as standard *fd* information
- a small number of CMMD I/O functions, to supplement standard UNIX I/O functions

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

Please note one implication of this device independence: Users should not attempt to mix CMMD I/O with CMFS I/O functionality. CMMD I/O does not recognize CMFS flags, environment variables, etc.

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

- Fortran programs should use a unit number wherever a CMMD function calls for a file descriptor.
- Buffered I/O (such as formatted I/O in C) is not fully supported from CMMD. See Section 12.8 for details.
- Some Fortran I/O operations do not fully match UNIX expectations; Section 12.8 discusses how to handle these situations.
- CMF Utility Library I/O routines cannot be called from within CMMD programs. See Section 12.9 for information about making I/O calls from CM Fortran.

12.1 Node-Level UNIX Support

CMMD provides the node-level 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 UNIX functions is provided by linking an I/O server routine into the host program. A standard host program including the server is provided by the **`CMMD-ld`** link utility for users not wishing to write a host program.

The CMMD-IO library 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 12.10.)

IMPORTANT

1. CMMD emulations of UNIX system calls use blocking communications operators. Accordingly, system calls should *not* be made from within CMMD handler functions.

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.

Global extensions provided by CMMD apply *only* to those UNIX I/O functions that take a file descriptor or stream as an argument, such as **`read()`**, **`write()`**, etc. These extensions 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.

The CM-5 programmer may find the limit on the number of open files is to be smaller than the number of nodes in the system. (CMMD currently allows each process to have 253 files open simultaneously.) Aside from the performance disadvantage, having each node open its own distinct file is therefore discouraged. CMMD provides functions that allow nodes to share references to files, thus avoiding the restrictions of UNIX.

12.2 CMMD File Descriptors and I/O Modes

In order to support UNIX functionality while simultaneously extending it to support parallel operations, CMMD creates new file descriptors and introduces new I/O modes. The I/O modes defined by CMMD are:

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

Each file descriptor, when created, starts out in one of these modes.

All UNIX functions that take a file descriptor or stream as an argument are sensitive to the associated I/O mode. The major difference concerns synchronization across nodes. In the independent modes, operations on file descriptors proceed independently. In the synchronous modes, operations on file descriptors synchronize across all nodes in the process of performing the operation.

When a file descriptor is in one of the *independent* modes, all operations proceed independently on each node making an I/O routine call. No coordination with other nodes is required. Independent modes are used primarily to read ordinary files on UNIX file servers. Individual nodes read and write different files independently. Each node maintains its own table of file descriptors and *file position indicators* (often called *file pointers*).

The distinction between *local independent* and *global independent* concerns the way in which the file is "known" by the nodes. A local independent file may be opened by a single node. *Global independent* files are opened by all nodes. In the global case, all nodes have a file descriptor pointing to the file along with individual file pointers. The primary reason for opening a file in one of the global modes is that nodes may thereafter take advantage of parallel I/O operations.

These operations, when used on a file located on a CM-5 I/O device, invoke underlying parallel I/O routines that quickly broadcast or distribute the data to all nodes.

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 "law of maximum" rule can cause unexpected effects in code that does not take it into account.

12.2.1 Local Mode

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

12.2.2 Global Independent Mode

Global independent mode (**`CMMD_independent`**) allows all nodes to access a single file for independent reading and writing. Files opened in this mode create only a single entry in the process's table of file descriptors. This conserves the number of open file descriptors. (Since UNIX limits the number of open files any process can have open at any given time, this mode allows all processors to have a pointer to the file without using up a critical resource.) Every node, however, maintains its own pointer to the file, and can move that pointer about at will, reading or writing the file in an independent manner. In this way, I/O operations on global independent files are nearly identical to that of files in local mode, with one important exception: the programmer may change the mode of global independent files to one of the synchronous modes described below, thereby achieving high-performance I/O.

12.2.3 Global Synchronous Broadcast Mode

Global synchronous broadcast mode (**`CMMD_sync_bc`**) allows nodes to simultaneously read (or write) the same data from a file or stream. The exact same data is read (or written) by each node, but the operation takes place in parallel to improve performace. On read operations, one node reads from the file and then broadcasts the data to the other nodes. On write operations, each node should create an identical buffer. However, only one buffer is written to the file, as if a single node did the write. This mode is particularly useful for having all nodes read the same input from the user's terminal or read in the same header information from a file. (See Example 1 in Figure 14.)

Important: A file or stream in **`CMMD_sync_bc`** mode must be read (or written) by all nodes in the partition. The starting position for the operation is the maximum of file pointers of the participating nodes. Following the operation, all file pointers are set to the last byte read.

The I/O buffers on all nodes must be of the same length. If the buffers are not the same, file operations will not work correctly, and will produce undefined results.

Example 1: All nodes read 512 bytes each from a global **`sync-bc`** file:

	Node 0	Node 1	Node 2	Node 3
Bytes	0 – 511	0 – 511	0 – 511	0 – 511

Example 2: All nodes read 512 bytes each from a global **`sync-seq`** file:

	Node 0	Node 1	Node 2	Node 3
Bytes	0 – 511	512–1023	1024–1535	1536–2047

Figure 14. Effect of file mode on data received by nodes.

12.2.4 Global Synchronous Sequential Mode

Global synchronous sequential mode (**`CMMD_sync_seq`**) allows nodes to read (or write) sequential portions of a file. The data read from (or written to) the file comes from each node in sequence, from node 0 up to the highest-numbered node. 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 by node number (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 ended. (See Example 2 in Figure 14.)

Note: The **`CMMD_sync_seq`** mode allows the amount of data read (or written) by each node to be different; the buffer on each node can have a different size. Nodes not having any data to contribute may set their buffer length to be zero. (See Example 3 in Figure 15.)

Example 3: Nodes read varying data lengths from a global **`sync-seq`** file:

	Node 0	Node 1	Node 2	Node 3
Bytes	0 – 511	(0 bytes)	512–820	821–824

Figure 15. Nodes can read varying amounts of data in `sync-seq` mode.

A file or stream in **`CMMD_sync_seq`** mode must be read (or written) by all nodes in the partition. The starting position for the operation is the maximum of file pointers of the participating nodes. Following the operation, all file pointers are set to the last byte read.

12.3 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 independent 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.

12.4 Setting the Default File Descriptor Mode

By default, the UNIX **`open()`** call opens files in local independent mode*. To change this behavior, CMMD provides two mechanisms that change the default behavior of **`open()`**. In one case, a function simply sets the open mode for subsequent file opens. In another (available only from C), a "global open" function both sets the mode for a single open and opens the file. A third function returns the open mode for any open file.

---

* This does not apply to standard input, output, and error streams, which the system opens in **`SYNC_BC`** mode. See Section 12.5 for information on how to change modes for these streams.

12.4.1 Setting the Open Mode

FUNCTIONS **CMMD_set_open_mode**
CMMD_get_open_mode

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 possible *io_modes* are:

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

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.

12.4.2 Opening a Global File

The routines described below are used to open global files, files accessible only on the nodes. These operators are only accessible from the C interface of CMMD.

FUNCTION **CMMD_global_open**

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).

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.

CMMD_global_open returns a CMMD descriptor in **CMMD_independent** mode for the opened file. Global files are available to the nodes only; the host processor cannot access them.

FUNCTION **CMMD_global_creat**

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**.

FUNCTION **`CMMD_global_fopen`**

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.

NOTE

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.

12.5 Changing the I/O Mode of a Global File

In the course of an application, it is often desirable to change the manner in which a global file is treated. For example, all nodes may initially need to read header information in **`CMMD_sync_bc`** mode, then switch to **`CMMD_sync_seq`** to distribute the rest of the data across the machine.

File descriptors in one of the global modes may therefore have their I/O behavior altered by calling **`CMMD_set_io_mode()`** or **`CMMD_fset_ io_mode()`**.

FUNCTIONS **`CMMD_set_io_mode`**
`CMMD_fset_io_mode`

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*

INTEGER FUNCTION **`CMMD_fset_io_mode(`***stream, io_mode***`)`**
integer *stream*
integer *io_mode*

ARGUMENTS

fd	CMMD 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.

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_syc_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);
```

12.5.1 Using Append Mode for Global Output Files

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.

For this reason, you may wish to place a seekable device in UNIX append mode when it is to be accessed in independent mode. This will 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 a **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. Its syntax is

INTEGER **CMMD_set_append_mode(***unit***)**
integer *unit*

Consider, for example, what would happen if our "hello world" program used **CMMD_independent**. Each node would start with identical file pointers and would attempt to print its message from the identical starting position in **stdout**. If **stdout** was 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 non-seekable device, does not have this problem. Since output directed to a non-seekable device cannot overwrite previous output, it is always displayed as if in append mode. However, no harm is done by putting a non-seekable device into append mode, so, if your program directs output to arbitrary devices, which may be either seekable or non-seekable, it's best to explicitly request append mode.

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

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

main (argc, argv)
    int argc
    char *argv[];
{
  CMMD_fset_io_mode(stdout, CMMD_independent);
  fcntl(fileno(stdout), F_SETFL, O_APPEND);
  printf("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 any particular order):

```
>hello_world
Hello world from node 3
Hello world from node 10
Hello world from node 1
:
:
```

12.6 Retrieving the IO Mode of a File Descriptor

FUNCTIONS **CMMD_get_io_mode**
CMMD_fget_io_mode

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*

INTEGER FUNCTION **CMMD_fget_io_mode(***unit***)**
integer *unit*

ARGUMENTS

fd	CMMD 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.

12.7 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**.

12.7.1 Restrictions

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

12.7.2 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.

12.7.3 CMMD Socket Function

FUNCTION **`CMMD_global_socket`**

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).

12.8 Buffered Streams: Supported, but Not Guaranteed

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 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.

Buffered I/O, by its nature, removes control from the user over when the underlying read or write occurs. 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.

This restriction results in a difficulty for Fortran I/O under these modes, as the only provided Fortran I/O mechanisms are built on top of a layer of buffering. To allow Fortran programmers to access the underlying, unbuffered, UNIX I/O calls, we have provided CMMD routines that resolve to these calls. See the *CMMD User's Guide* for details.

Consider this concrete example: 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 combine network.

12.9 Nodal CM Fortran I/O

Nodal CM Fortran programs cannot use CMF Utility Library I/O calls. Instead, if a nodal program needs to write a parallel array to an SDA or DataVault, it should:

- Use **CMF_FE_ARRAY_FROM_CM** to convert the parallel array into a serial array stored in the node's serial (SPARC) memory.
- Use CMMD parallel I/O to write this serial array to disk

To read the array back from disk into memory, the program must:

- Use CMMD parallel I/O to read the array data into serial memory
- Use **CMF_FE_ARRAY_TO_CM** to convert the serial array into a parallel array stored in the node's parallel memory.

Please Note Also: 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.

12.10 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.

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 **`CMMD_sync_bc`** or **`CMMD_sync_seq`** modes.

FUNCTION **`dlength`**

C SYNTAX

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

FORTRAN SYNTAX

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

ARGUMENTS

fd	The CMMD 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.

FUNCTION **`dseek`**

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 CMMD 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 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 **CMMD_local** or **CMMD_independent**, any node can make this call independently.
- If the *fd*'s mode is **CMMD_sync_seq** or **CMMD_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.

FUNCTION **`dtell`**

C SYNTAX

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

FORTRAN SYNTAX

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

ARGUMENT

fd	The CMMD 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.

FUNCTION **`dftruncate`**

C SYNTAX

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

FORTRAN SYNTAX

INTEGER FUNCTION **`dftruncate(`***unit, length***`)`**
integer *unit*
real*8 *length*

ARGUMENT

fd	The CMMD 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.

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 **`CMMD_local`** or **`CMMD_independent`**, any node can make this call independently.
- If the *fd*'s mode is **`CMMD_sync_seq`** or **`CMMD_sync_bc`**, all nodes must make the call with identical arguments.

The function returns 0 on success, –1 on failure.

12.11 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 physical write call can be read only by a 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.

12.11.1 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.

12.11.2 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 this function:

FUNCTION **CMMD_physical_blocking**

C SYNTAX

int **CMMD_physical_blocking(**int *fd***)**

FORTRAN SYNTAX

INTEGER FUNCTION **CMMD_physical_blocking(***unit***)**
integer *unit*

ARGUMENT

fd CMMD file descriptor.

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.

12.11.3 Physical Read/Write Operators

FUNCTION **`CMMD_physical_write`**

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*

ARGUMENT

fd	CMMD file descriptor 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 read; 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.

FUNCTION **`CMMD_physical_read`**

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 CMMD file descriptor for the file to be read.

buffer Memory location into which data will be read. Must be doubleword-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 set 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.