Chapter 7

Reading and Writing

This chapter gives an overview of the basic I/O process and describes the CMFS library routines and the CMOST I/O system calls that read and write. (See Chapter 2 for a complete list of I/O interfaces and where to find information about them.)

7.1 The Process

The basic I/O process is as follows:

1. Prepare for reading and/or writing.
   - If the file does not yet exist, create it with the pertinent read/write/execute permissions. See Section 6.1.
   - If the file already exists, verify that you have permissions to read, write, and/or execute the file. If you don't have the pertinent permissions, change them. See Sections 6.2 and 6.3.
   - If necessary, set the file pointer appropriately and/or modify the file's size. See Sections 6.4 and 6.5.
2. Read from and/or write to the file.
   - If the client is a CM-5 partition, use the routines that perform CM I/O; see Section 7.3.
   - If the client is a serial machine, such as a control processor, VMEIO host computer, or CM-IOPG, use the routines that perform serial I/O; see Section 7.4.
3. Close the file. See Section 6.6.

Copyright © 1994 Thinking Machines Corporation

7.2 File Padding

All write operations to the CMFS file system pad the end of the file to the next 512-byte boundary. Write operations to the SFS file system no longer pad in any way.

7.3 CM I/O

The CMFS library offers two sets of read/write routines:

- Synchronous I/O: **CMFS_read|write_file_always**, available from C* and CM Fortran. See Section 7.3.1.
- Physical I/O: **CMFS_read|write_file_physical**, available from C* only. See Section 7.3.2.

In addition to the CMFS routines, the CMOST operating system provides two system calls, **CM_read** and **CM_write**, which have a C interface.

7.3.1 CMFS Synchronous I/O Routines

The CMFS library provides two routines that perform synchronous CM I/O, i.e., reads to or writes from a partition's processing nodes:

- **CMFS_read_file_always**
- **CMFS_write_file_always**

You can call these routines from either C* or CM Fortran.

C* Syntax

```
#include <cm/cmfs.h>

int CMFS_read|write_file_always(int fd, void:void *pvar,
                                int count)
```


Copyright © 1994 Thinking Machines Corporation

CM Fortran Syntax

```
N = CMFS_READ|WRITE_FILE_ALWAYS(FD, ARRAY, COUNT)
INTEGER N, FD, COUNT
ANY_TYPE ARRAY(:, :, ..., :)          ! ANY TYPE, ANY
RANK
```

These routines take three arguments:

1. An integer specifying an open file descriptor.
2. The structure that holds or is to hold the data: in CM Fortran, an array; in C*, a pointer to a parallel variable.
3. An integer that specifies the number of bytes to read into or write from each element of the array or position of the parallel variable.

   CM Fortran only: This argument must be a multiple of 4 bytes and must begin on a cmword boundary. The cmword size is usually 512 bytes on a CMFS file system and 16 bytes on an SFS file system (execute **cmstat** or **CMFS_[f]stat** to verify).

These routines return one of the following values:

- The number of bytes read into or written from each array element or parallel variable position.
- **0** (**CMFS_read_file_always** only), which indicates that end-of-file has been reached.
- **-1**, which indicates failure. **CMFS_errno** is set to a number representing the cause of the error.

For more information about these calls, see their manual pages in Chapter 10.

These routines are optimized for performing infrequent, large transfers of data to and from the CM-5 partition. For more information about performance, see Chapter 3.

Copyright © 1994 Thinking Machines Corporation

7.3.2 CMFS Physical I/O Routines

The physical I/O routines are **CMFS_write_file_physical** and **CMFS_read_file_physical**. They have the same interfaces as **CMFS_write_file_always** and **CMFS_read_file_always**, namely,

```
int CMFS_write_file_physical(int fd,
                             void:void* source,
                             int bytes_per_position)

int CMFS_read_file_physical(int fd,
                            void:void* destination,
                            int bytes_per_position)
```

These routines are faster than the corresponding standard CMFS synchronous I/O routines and use less memory. However, note these points about their use:

- The data they read or write is not portable. If you use the **CMFS_write_file_physical** routine to write data to an external storage device, you can recover the data only by using the **CMFS_read_file_physical** routine to read the data onto the same set of physical processors, and into a parallel variable with the same memory layout as the parallel variable from which the original write occurred; you must also use the same file system. Essentially, you must use the same partition size, file system, and shape for both writing and reading; if you do a physical write, you must do a physical read.

- These routines return the total number of bytes read or written, rather than the number of bytes read or written per-position.

- The I/O data read or written by these routines is padded. When the file pointer is not located at an alignment boundary appropriate to the I/O device where the file referenced through the **fd** argument resides, the file pointer is adjusted upward to such a boundary before the transfer of the data between the CM and the I/O device begins. Furthermore, garbage data is written to pad the length of the I/O transaction to the same alignment. For DataVault I/O, the alignment padding used is 512 bytes; for SDA I/O, it is 16 bytes times the number of SDA data disks.

Copyright © 1994 Thinking Machines Corporation

To anticipate the impact of padding before doing a read or write, use this routine:

```
int CMFS_physical_transfer_length(int fd,
                                  void:void* target,
                                  int bytes)
```

CMFS_physical_transfer_length takes the arguments of a **CMFS_read_file_physical** or **CMFS_write_file_physical** function call and returns the number of bytes that such a call would transfer. In other words, it returns the same value that a successfully completed call to the read or write function would return.

Because of the complications of padding, we recommend using a physical read only when the file pointer is in the same position it was in when the physical write was performed.

7.3.3 CMOST I/O System Calls

The CMOST I/O interface — **CM_read()** and **CM_write()** — allows C* users to read to or write from a parallel variable with more fine control than is possible with the CMFS library's read and write routines.

- **CM_read()** reads some number of bytes from an SDA file into a CM-5 buffer — an "NMO (New Mexico order) buffer," so named because of the shape of the data layout; see Appendix B for detailed information. Then, by calling the C* I/O function **CMC_get_from_io_buffer**, the data is copied into a parallel variable.

- **CM_write()**, conversely, writes some number of bytes from an NMO buffer to an SDA file. The data is first copied from the parallel variable into the NMO buffer by calling the C* I/O function **CMC_send_to_io_buffer**.

C* provides functions for allocating and freeing the NMO buffer — **CMC_alloc_io_buffer** and **CMC_free_io_buffer**, respectively. The syntax of the C* I/O functions is discussed in the *C* Release Notes*, Version 7.1.1.

Copyright © 1994 Thinking Machines Corporation

CM_read and **CM_write** have a C interface only:

C Syntax

```
#include <cmsys/cmio.h>

int CM_write (int fd, cmaddr_t pbuf, int nbytes)
int CM_read (int fd, cmaddr_t pbuf, int nbytes)
```

These routines take three arguments:

1. *fd*: an integer specifying the open file descriptor from which to read or to which to write. **CM_read** and **CM_write** use the seek pointer associated with the file descriptor for the usual purpose — to determine the location in the file at which to start the read or write operation.

2. *pbuf*: a pointer to a structure — defined below and in **<cmsys/cmio.h>** — that specifies the allocated buffer space into which the data is to be read, or from which the data is to be written.

```
typedef struct_cm_addr {
     union
          CM_memaddr_t cma_base;      /* parallel address */
                                      /* across all PNs */
          caddr_t cma_caddr;          /* Not needed for */
                                      /* SDA I/O       */
     }
     u_int cma_offset;                /* must be set to 0 */
                                      /* for SDA I/O    */
} cmaddr_t;
```

   For the union part of the structure, the **cma_base** field must be filled in; **cma_base** is the NMO buffer address, previously allocated. Although the other field, **cma_offset**, is not used for I/O to SDA or other UNIX files, you must set the **cma_offset** field to 0, or **CM_read|write()** will return an error. (The **cma_offset** field is used for CM5-HIPPI I/O, which is not discussed in this manual; see Thinking Machines' CM5-HIPPI documentation.)

3. *nbytes*: an integer that specifies the number of bytes to read into or write from the NMO buffer. The bytes are read or written using the NMO paradigm, as illustrated in Figure 7.

Copyright © 1994 Thinking Machines Corporation

For more detailed information about **`CM_read()`** and **`CM_write`**, see their manual pages in Chapter 11.

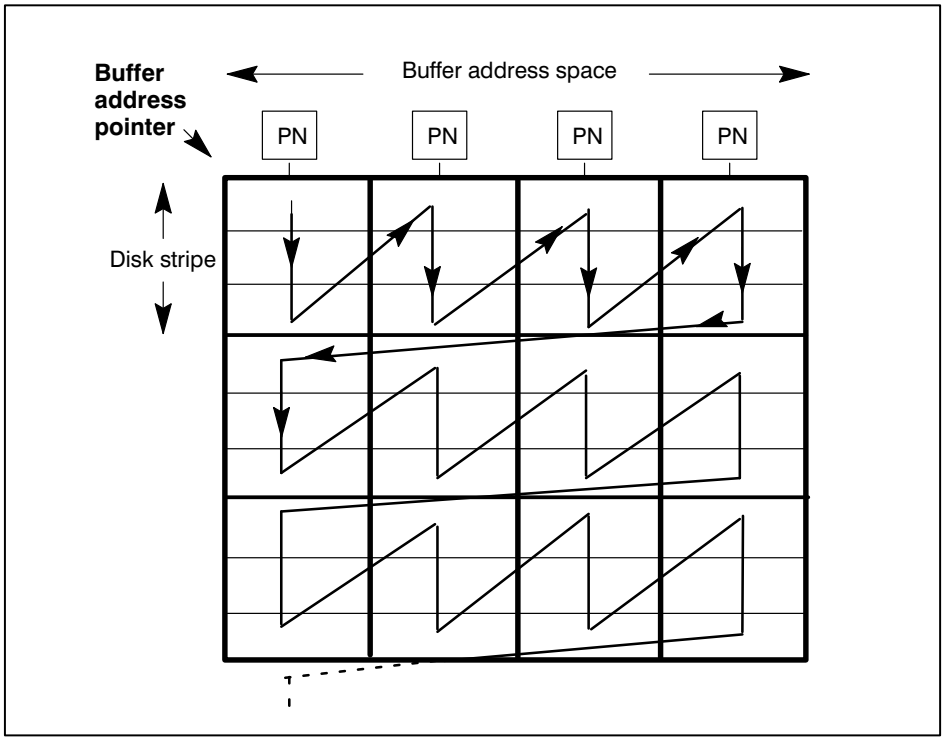

Figure 7. NMO: the order in which buffer packets are read and written. For more information about NMO, see Appendix B.

7.4 Serial I/O

The CMFS library provides two routines that perform synchronous serial I/O, i.e., read operations to or write operations from a serial computer's user memory:

- **`CMFS_serial_read_file`**
- **`CMFS_serial_write_file`**

You can call these routines from either C* or CM Fortran.

Copyright © 1994 Thinking Machines Corporation

C* Syntax

```
int = CMFS_serial_read|write_file(int fd,  char *buffer,
                                  int count)
```

CM Fortran Syntax

```
N = CMFS_READ|WRITE_FILE_ALWAYS(FD, BUFFER, COUNT)
INTEGER N, FD, BUFFER(1), COUNT
! The actual size of the buffer array is COUNT/4
```

These routines take three arguments:

1. An integer specifying an open file descriptor.
2. The location in the memory of the serial computer that holds or is to hold the data: a pointer to a buffer in C; an integer in Fortran.
3. An integer that specifies the number of bytes to read into or write from the buffer.

These routines return one of the following values:

- The number of bytes read into or written from the buffer.
- **0** (**CMFS_serial_read_file** only), which indicates that end-of-file has been reached.
- **-1**, which indicates failure. **CMFS_errno** is set to a number representing the cause of the error.

For more information about these calls, see their manual pages in Chapter 10.

7.4.1 CMFS_serial_read_file on a VMEIO Host Computer

When the address for the buffer argument is a VMEIO buffer address, there are two cases where **CMFS_serial_read_file** attempts to allocate a separate VMEIO buffer of (*count* + 8) bytes for the transfer:

- when the address passed in is not eight-byte aligned
- when the current file offset is not a multiple of eight bytes

Copyright © 1994 Thinking Machines Corporation

Therefore, if all the available memory is used by the user-specified buffer, the calling program will fail with the error **`CMFS_ENOBUFS`**.

Copyright © 1994 Thinking Machines Corporation