Chapter 6

File Operations

This chapter describes how to use CMOST I/O commands and CMFS library calls to complete tasks associated with using files on the Connection Machine system. To find out how to

- create and open a file, see Section 6.1.
- change a file's permissions, see Section 6.2.
- change a file's owner or group, see Section 6.3.
- move a file's pointer, see Section 6.4.
- truncate or extend a file, see Section 6.5.
- close a file, see Section 6.6.
- copy a file, see Section 6.7.

This chapter does not discuss reading from and writing to a file; these tasks are described in Chapters 7.

6.1 Creating and Opening Files

This section discusses the CMFS routines **`CMFS_open`** and **`CMFS_creat`**. **`CMFS_open`** is the more flexible call, supporting several flags that enhance its obvious primary function. Sections 6.1.1 and 6.1.2 contain sample code fragments and information about the calls' arguments.

A successful call to either **`CMFS_open`** or **`CMFS_creat`** opens the designated file and returns a *file descriptor,* a non-negative integer that identifies the file. The file pointer is set to the beginning of the file by default, and the owner and group of the file are set to the UNIX default (see Section 6.3). If the call

Copyright © 1994 Thinking Machines Corporation

is not successful, it returns −1 and sets **CMFS_errno** to a number representing the cause of the error.

A file can be opened by more than one process at a time. A single process can have open up to 224 CMFS[5] file descriptors and 64 SFS file descriptors. Up to 250 files can be open on any one CMFS file system, and up to 1888 files can be open on any one SFS file system.

6.1.1 CMFS_open

CMFS_open can be used either to open an existing CMFS file, or to open a file after first creating it (as is true of a UNIX file, a CMFS file must exist before it can be opened). When called from its C interface, **CMFS_open** takes two or three arguments, depending on whether or not the routine must create the file as well as open it:

- To create and open a new file, call **CMFS_open** with three arguments:
  - a pathname (no more than 255 characters)
  - the **CMFS_O_CREAT** flag ORed with any other relevant flags (see Table 7)
  - a *mode*, a description of the file's read/write/execute permissions, given by ORing together a combination of the values listed in Table 8
- To open an existing file, call **CMFS_open** with two arguments:
  - a pathname (no more than 255 characters)
  - one or more of the flags in Table 7, as appropriate to the application

When called from its Fortran interface, **CMFS_OPEN** must always be supplied with all three arguments, and the flags in Table 7 must be specified via their decimal equivalents (see the header file **<cm/cm_file.h>**).

If your CMFS file system is configured to check permissions, you cannot specify a flag from Table 7's left column that requires permissions greater than those allowed by the file's mode. For example, if the file was created

5. These CMFS numbers are valid on CMFS file servers running DataVault Software Version 2.0 Beta and later.

Copyright © 1994 Thinking Machines Corporation

with read-only permissions across the board, you cannot specify **CMFS_O_WRONLY** or **CMFS_O_RDWR** in the second argument. See Section 6.2 for more information about permissions.

For example, the following C code fragment creates and opens a file named **myfile** for writing only, gives read and write permissions to everyone, and returns the file descriptor **my_fd**:

```
int my_fd;
my_fd =  CMFS_open ("dv1:/myfile", CMFS_O_CREAT |
         CMFS_O_WRONLY, 0666);
```

The same code fragment written in Fortran is:

```
INTEGER my_fd
my_fd =  CMFS_OPEN ("dv1:/myfile", 01000 | 001, 0'666')
```

6.1.2 CMFS_creat

CMFS_creat takes two arguments:

- a pathname (no more than 255 characters)
- a *mode,* a description of the file's read/write/execute permissions (see Section 6.2), given by ORing together a combination of the values listed in Table 7

The following C example creates a file named **myfile** on the DataVault named **dv1**, and allows everyone to read and write it:

```
int my_fd;
my_fd = CMFS_creat ("dv1:/myfile", 0666);
```

Note that the code sample above is the same as

```
my_fd =  CMFS_open ("dv1:/myfile",
         CMFS_O_CREAT|CMFS_O_WRONLY|CMFS_O_TRUNC,
              0666);
```

From Fortran, call **CMFS_CREAT** as in the following example:

```
INTEGER my_fd
my_fd = CMFS_CREAT ("dv1:/myfile", 0'666')
```


Copyright © 1994 Thinking Machines Corporation

Table 7. `CMFS_open` flags.

Permissions Flags		Characteristics Flags	
`CMFS_O_RDONLY`	Open the file for reading only.	`CMFS_O_CREAT`	If the file does not exist, create it. You must also specify *mode*, the third argument (see Table 8).
`CMFS_O_WRONLY`	Open the file for writing only.	`CMFS_O_TRUNC`	If the file exists, truncate its length to 0 (thereby losing its contents).
`CMFS_O_RDWR`	Open the file for reading and writing.	`CMFS_O_APPEND`	Prior to each write, set the file pointer to the file's end.
		`CMFS_O_EXCL`	Return an error if `CMFS_O_CREAT` is also specified and the file already exists.
		`CMFS_O_FSYNC`	Disable asynchronous inode writing for write operations to the CMFS file system (but not for ftruncate operations); see the online `fsserver`(8) manual page.

NOTE: Construct the *flag* argument by ORing one of the *permissions flags* with zero or more of the *characteristics flags*. When used in a call from Fortran, these flags must be specified via their decimal equivalent — see **`<cm/cm_file.h>`.**

Table 8. `CMFS_creat` and `CMFS_open` *mode* values.

Owner Permission	Group Permissions	Others Permissions
read: `0400`	read: `0040`	read: `0004`
write: `0200`	write: `0020`	write: `0002`
execute: `0100`	execute: `0010`	execute: `0001`

NOTE: Construct the *mode* argument by ORing together one or more octal values from each column. For example, `0777` gives read, write, and execute permissions to everyone. See Section 6.2.

Copyright © 1994 Thinking Machines Corporation

6.2 Changing a File's Permissions

Upon its creation, a CMFS file's permissions are assigned via an octal value — for example, **0751** (see Table 8) — specified by the last argument to **CMFS_creat** or **CMFS_open**. From right to left, the first 3-bit value (e.g., **1**) indicates permissions given to persons who are not the owner and not in the owner's group, the second 3-bit value (e.g., **5**) indicates those given to the group to which the file belongs (usually, the group to which the owner belongs), and the third 3-bit value (e.g., **7**) indicates the permissions given to the file's owner (usually, the person who created the file).

The octal value **0751** therefore specifies read, write, and execute permissions (**4 + 2 + 1 = 7**) for the owner; read and execute permissions (**4 + 1 = 5**) permissions for the group; and execute permissions (**1**) for others. If a member of the group tries to write the file, an error is returned indicating insufficient permission; likewise, this error is returned if a person who is not the owner and not in the group tries to read or write the file. (NOTE: Permission checking is enabled only if the appropriate CMFS file system server is set to check permissions (the default). See your system administrator.)

To check a file's permissions, execute **cmstat** or **cmls -l**, or call **CMFS_[f]stat**. For example, the following **cmstat** and **cmls -l** output is for a file with permissions set to **0751**, described in the previous paragraph.

```
% cmstat myfile
File: "myfile"
    [...]
    Mode: (0751/rwxr-x--x) UID: (1155/jim)
    Gid: (10/staff)
    [...]

% cmls -l myfile
-rwxr-x--x  2  jim      886 Dec 31  14:39 myfile
```

To change the read/write/execute permissions of a file, call **CMFS_[f]chmod** or execute **cmchmod**. You must be either the current owner of the file or logged on as root. For example, to use **cmchmod** to give additional write permission to the group, you could type

```
% cmchmod 0771 myfile
```

Or, to use **cmchmod**'s alternate syntax, you could type

```
% cmchmod g+w myfile
```


Copyright © 1994 Thinking Machines Corporation

Please see the **cmchmod** man page in Chapter 12 of this manual for the complete syntax.

6.3 Changing a File's Owner or Group

A CMFS file is created with the following defaults:

- The file's owner ID is set to the effective owner ID of the process that creates it.
- The file's group ID is set to the effective group ID of the process that creates it if the set-gid bit of the file's parent directory is clear (this is the usual case). If the set-gid bit of the parent directory is not clear, the file's group ID is set to the group ID of the directory in which the file is created.

To find out which group the file is assigned to, and the owner of the file, look at the output of **cmstat** or **cmls -lg** (for an example, see Section 6.2). To find out whether you are a member of the file's group, look at the group database on a control processor. Look at the passwd database to find out the user ID associated with your login.

If permission checking is enabled on the file system server on which the file resides, the group and owner permissions (see Section 6.2) are checked at each attempt to access — open, stat, read, or write — a file. Access to the file is allowed under the following conditions:

- If the person attempting access is root.
- If the person attempting access is the owner and owner permissions are set to allow the access.
- If the person attempting access is not the owner but is a member of the file's group (as listed in the group database), and the file's group permissions are set to allow the access.
- If the person attempting access is neither the owner nor in the file's group, and the file's "other" permissions are set to allow the access.

If you are the owner of a file or logged on as root, you can change the group ownership of the file; you can change the owner of a file only if you are logged on as root. Make these changes by executing **cmchgrp** or **cmchown**,

Copyright © 1994 Thinking Machines Corporation

respectively, or by calling **CMFS_[f]chown**. (**CMFS_[f]chown** provides arguments for changing both group and file owner). For example, to change the group ID of **myfile** from **10/staff** to **102/proj**, type

```
% cmchgrp 102 myfile
```

or

```
% cmchgrp proj myfile
```

Likewise, to change the owner of **myfile** from **jim** to **sean**, type

```
# cmchown sean myfile
```

or, assuming that **2266** is the user ID listed for **sean** in the passwd database

```
# cmchown 2266 myfile
```

You can use **cmchown** to change both the owner of the file and its group:

```
# cmchown sean.proj myfile
```

Chapter 12 of this manual contains the manual pages for **cmchgrp** and **cmchown**. Chapter 10 contains the manual page for **CMFS_[f]chown**.

6.4 Moving a File Pointer

The position of the file pointer marks the point in the file at which a subsequent read or write starts. The file pointer can be at a negative value; if the file pointer value is negative, an error will occur only upon attempting to read or write from that location.

CMFS_creat and **CMFS_open** by default set the file's pointer to the beginning of the file. (You can set **CMFS_open**'s *flag* argument to **CMFS_O_APPEND**, which moves the pointer to the file's end before every write.) If you do not

Copyright © 1994 Thinking Machines Corporation

want to start reading or writing the file at its beginning, move the pointer by calling one of the four[6] supported CM-5 CMFS seek routines:

- **CMFS_serial_lseek:** A single call can seek up to (2 Gbytes −1 byte) into a file.
- **CMFS_physical_lseek**: A single call can seek up to 32 Gbytes into an SFS file and 1 Tbyte into a CMFS file.
- **CMFS_serial_dlseek**: A single call can seek up to (2^{51} −1) bytes into an SFS file and up to 1 Tbyte into a CMFS file.
- **CMFS_serial_llseek**: A single call can seek up to (2^{63} −1) bytes into an SFS file and up to 1 Tbyte into a CMFS file.

6.4.1 CMFS_serial_lseek, CMFS_serial_dlseek, CMFS_serial_llseek

CMFS_serial_lseek, **CMFS_serial_dlseek**, and **CMFS_serial_llseek** take three arguments:

1. A file descriptor of a file open for reading or writing.
2. The number of bytes to offset the pointer from the location specified by the third argument. The number can be positive or negative. The type of this second argument — and therefore the size of the largest number it can represent — depends on the routine:
   - Type **long** for **CMFS_serial_lseek** (2^{31} −1 maximum)
   - Type **double** for **CMFS_serial_dlseek** (2^{51} −1 maximum)
   - Type **long long** for **CMFS_serial_llseek** (2^{63} −1 maximum)
3. A flag (defined in **<cm/cm_file.h>**) that indicates where to start counting:

CMFS_L_SET	Count from the beginning of the file (byte 0).
CMFS_L_INCR	Count from the current position of the file pointer.
CMFS_L_XTND	Count from the end of the file.

6. Note that a fifth CMFS parallel seek routine, **CMFS_lseek**, does not work the same way on the CM-5 as it did on the CM-2 and CM-200; currently on the CM-5, its arguments and behavior are the same as **CMFS_serial_lseek**. Avoid calling **CMFS_lseek** on the CM-5, as its behavior may change in a future software release.

Copyright © 1994 Thinking Machines Corporation

When successful, each routine returns the current file pointer value (of the same type as the routine's second argument), measured in bytes from the beginning of the file. If unsuccessful, these routines return **-1**, and **CMFS_errno** is set to indicate the cause of the error.

The following example uses **CMFS_serial_lseek** to move the file pointer 1 Mbyte into the file represented by the descriptor **my_fd**:

```
n = CMFS_serial_lseek(my_fd, 1000000, CMFS_L_SET);
```

6.4.2 CMFS_physical_lseek

CMFS_physical_lseek takes four arguments:

1. A file descriptor of a file open for reading or writing.

2. A positive or negative number (of type **long**) of cmwords to offset the file pointer from the position specified by the fourth argument. 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).

   When **CMFS_physical_lseek** returns successfully, this variable's value is the number of cmwords contained from the beginning of the file to the new file pointer position.

3. An additional number of bytes — additional to the number specified by the second argument — to offset the file pointer from the position specified by the fourth argument.

   When **CMFS_physical_lseek** returns successfully, this variable's value is the number of bytes in the space between the last cmword boundary and the new file pointer position. For instance, if the file pointer's position is on a cmword boundary, this variable's value is 0.

4. A flag (defined in **<cm/cm_file.h>**) that indicates where to start counting:

CMFS_L_SET	Count from the beginning of the file (byte 0).
CMFS_L_INCR	Count from the current position of the file pointer.
CMFS_L_XTND	Count from the end of the file.

Copyright © 1994 Thinking Machines Corporation

The combination of the second and third arguments allows one seek operation to move up to 32 Gbytes into an SFS file and up to 1 Tbyte into a CMFS file.

CMFS_physical_lseek returns **0** on success and **-1** on failure.

The following call to **CMFS_physical_lseek** moves the file pointer 5 Gbytes into an SFS file represented by the descriptor **my_fd**:

```
n = CMFS_physical_lseek(my_fd, 335544320, 0, CMFS_L_SET);
```

6.5 Changing a File's Size

A call to one of the CMFS truncate routines changes the size of a file. Each call takes arguments that indicate the file's pathname or file descriptor and the desired new length of the file:

- If the current length of the file is larger than the desired new length, the call shortens the file to the length specified and the extra data is lost.
- If the current length of the file is smaller than the desired new length, the call extends the file to the length specified. (The CM-5 CMFS lseek routines (Section 6.4) can also extend a file by moving the file pointer past the end of the file and then writing.)
  - When a CMFS file is extended in this way, the contents of the blocks allocated in the gap between the end of the file and the location of the file pointer are undefined. You can use this technique to pre-allocate DataVault physical disk blocks as contiguously as possible.
  - When an SFS file is extended in this way, a hole is left in the file.

There are five[7] CMFS truncate routines supported on the CM-5:

- **CMFS_serial_[f]truncate_file**: A single call can specify a file length of up to 2 Gbytes −1 byte. This routine is also available at the shell level by executing **cmtruncate**.

7. Note that **CMFS_[f]truncate_file** is currently not supported on the CM-5.

Copyright © 1994 Thinking Machines Corporation

- **CMFS_physical_ftruncate**: A single call can specify a file length of up to 32 Gbytes for an SFS file and 1 Tbyte for a CMFS file. This routine is also available at the shell level by executing **cmptruncate**.
- **CMFS_serial_lftruncate_file**: A single call can specify a file length of up to $(2^{51} - 1)$ bytes for an SFS file and up to 1 Tbyte for a CMFS file.
- **CMFS_serial_dftruncate_file**: A single call can specify a file length of up to $(2^{63} - 1)$ bytes for an SFS file and up to 1 Tbyte for a CMFS file.

6.5.1 CMFS_serial_[f]truncate_file, CMFS_serial_lftruncate_file, and CMS_serial_dftruncate_file

CMFS_serial_[f]truncate_file, **CMFS_serial_lftruncate**, and **CMFS_serial_dftruncate** take two arguments:

1. A file descriptor of a file open for writing. (Only **CMFS_serial_truncate_file** accepts the pathname of a file.)
2. The new length, in bytes, of the file described by the first argument. The type of this second argument — and therefore the size of the largest number it can represent — depends on the routine:
   - Type **long** for **CMFS_serial_[f]truncate_file** ($= 2^{31} - 1$ maximum)
   - Type **double** for **CMFS_serial_dftruncate_file** ($= 2^{51} - 1$ maximum)
   - Type **long long** for **CMFS_serial_lftruncate_file** ($= 2^{63} - 1$ maximum)

When successful, these routines return **0**. If unsuccessful, these routines return **-1** and **CMFS_errno** is set to indicate the cause of the error.

The following example uses **CMFS_serial_truncate_file** to truncate the file represented by the descriptor **my_fd** to 1 Mbyte in length.

```
n = CMFS_serial_truncate_file(my_fd, 1048576);
```


Copyright © 1994 Thinking Machines Corporation

6.5.2 CMFS_physical_ftruncate

CMFS_physical_ftruncate takes three arguments:

1. A file descriptor of a file open for writing.
2. A positive or negative number (of type **int**) of cmwords you want the truncated file to contain. The cmword size is usually 512 bytes on a CMFS file system and 16 bytes on an SFS file system (execute **cmstat** to verify).
3. An additional number of bytes — additional to the number specified by the second argument — you want the truncated file to contain.

CMFS_physical_ftruncate returns **0** on success and **-1** on failure.

The following call to **CMFS_physical_ftruncate** sets the length of the SFS file represented by the descriptor **my_fd** to 5 Gbytes:

```
n = CMFS_physical_ftruncate(my_fd, 335544320, 0);
```

6.6 Closing Files

To close a parallel file, call **CMFS_close**, which takes one argument: an open file descriptor. **CMFS_close** deletes the file descriptor from the reference table; if the file descriptor is the last reference to the underlying file, the program can no longer use the file unless it is re-opened.

It is not necessary to explicitly close a file if the program that opened it exits successfully. However, since there is a limit on the number of active descriptors per program, a program that uses many descriptors should explicitly close each file when finished with it.

The following C example closes the file represented by the file descriptor **my_fd**:

```
n = CMFS_close (my_fd);
```

The same example in Fortran syntax is:

```
N = CMFS_CLOSE (my_fd)
```


Copyright © 1994 Thinking Machines Corporation

6.7 Copying Files

File copying is a fundamental task that you may perform often in the course of managing your CM-5 file system files. Table 9 lists the CMOST I/O copy commands. The sections that follow explain in detail all the CMOST I/O copy commands except those concerned with tape I/O; commands associated with tape I/O are discussed in Chapter 8.

Like a UNIX copy command, a CMOST I/O copy command makes a copy of the *source* file and places it into the *destination* file. If the source file is a CMFS file, it has an associated attribute file, named *.sourcefile* **.attr**; if you choose to keep track of CMFS attribute files (for more information, see Section A.2.1 in Appendix A), *.sourcefile* **.attr** should be copied into the same directory as the destination file and renamed *.destination file* **.attr**. Most CMOST I/O copy commands copy and rename the attribute file automatically by default, although some do not.

If the CM-5 network or one or more CMIO buses connect the host system with the file system in which the source file resides and with the file system in which the destination file resides, the CMOST I/O copy commands can use the CMIO bus(es) and/or network to achieve good performance. Otherwise, the commands use the Ethernet to perform the copy.

Copyright © 1994 Thinking Machines Corporation

Table 9. CMOST I/O copy commands.

Command	Synopsis
cmcp	Copies one or more CMFS, SFS, or UNIX files into another file or directory within that file system or a different one.
cmdd	Converts and copies files with various data formats. Especially useful for ASCII ↔ EBCDIC conversion. See Chapter 8 on tape I/O.
cmdump	Archives files. See Chapter 8 on tape I/O.
cmrestore	Extracts files from an archive made by **cmdump**. See Chapter 8 on tape I/O.
cmtar	Creates or extracts a tape archive. See Chapter 8 on tape I/O.
copyfromdv	Copies a CMFS, SFS, or UNIX file into an SFS file (or other UNIX-system file). **cmcp** has largely obsoleted **copyfromdv** and **copytodv**.
copytodv	Copies an SFS file (or other UNIX-system file) to an SFS, CMFS, or UNIX file. **cmcp** has largely obsoleted **copyfromdv** and **copytodv**.
dvcp	Copies one or more CMFS, SFS, or UNIX files into another file or directory within that file system or another file system, using the CM-5 networks and PNs for performance enhancement. Must be executed on a partition manager.
ibmtape	Imports/exports IBM data sets to/from a file system. See Chapter 8 on tape I/O.
pcp	Copies one or more SFS or other UNIX-system files into another file or directory within that file system or another UNIX-compatible one. Supports symbolic link copying. Must be executed on a partition manager.

6.7.1 General CMOST I/O Copy Commands

The **cmcp** command is the most general CMOST I/O copy command, akin to the UNIX **cp** command. The **copyfromdv** and **copytodv** commands — for which there are no UNIX equivalents — are more specialized versions of

Copyright © 1994 Thinking Machines Corporation

cmcp. You can execute these commands on any system on which CMOST has been installed (you need not be on a control processor or partition manager).

The cmcp Command

The **cmcp** command can copy the contents of a single file to another file, or copy multiple files into a directory. The source file and the destination file can reside in any CM file system — a CMFS file system, or an SFS or other UNIX file system; the source and destination file need not reside in the same file system or the same kind of file system.

cmcp takes at least two arguments: *sourcepath* indicates the file(s) to copy, and *destpath* indicates the name of the destination file or directory. Regardless of how many *sourcepath* arguments there are, the last argument always indicates the destination; if there is more than one *sourcepath* argument, the last argument must be a directory name rather than a file name. For example,

```
% cmcp dv1:/mydata dv1:/mydata1 sda1:/
```

copies the files **dv1:/mydata** and **dv1:/mydata1** to the root directory of the SFS file system on the SDA.

The copytodv and copyfromdv Commands

The **copytodv**[8] command copies an SFS file (or other UNIX-system file) into a file on any CM file system (SFS, UNIX, or CMFS). Similarly, **copyfromdv** copies a CMFS, SFS, or UNIX file into an SFS file (or other UNIX-system file). The calls each take two arguments, *sourcepath* and *destpath*, and support several flags, listed in the manual pages.

For example, to copy the CMFS file **mydata_1** (residing in your current working directory on the CMFS file system named **dv1**) into an SFS file system, issue the following command:

```
% copyfromdv dv1:mydata_1 sda1:mydata_2
```

mydata_2 now resides in your current working directory on the SDA.

8. The "dv" in the **copytodv**, **copyfromdv**, and **dvcp** command name is historical — in earlier software versions, these commands did strictly what their names imply. Now they have broader functionality.

Copyright © 1994 Thinking Machines Corporation

To copy the SFS file **mydata_2** to the file **mydata2_rev** in your current working directory on the SDA, issue the **copytodv** command:

```
% copytodv mydata_2 mydata2_rev
```

6.7.2 High-Performance Copy Commands: dvcp and pcp

dvcp and **pcp** are high-performance copy commands — they use the CM-5 networks and a partition of processing nodes (for temporary buffer space) to obtain best copy performance. Consequently, **dvcp** and **pcp** must be executed from a partition manager.

dvcp can handle CMFS, SFS, and UNIX files. **pcp** can handle SFS or UNIX files, but not CMFS files. **pcp** can copy links, which **dvcp** (and the CMFS file system in general) does not support.

As a side effect of copying an SFS file with **dvcp** or **pcp**, the file will become defragmented, which enhances I/O performance.

The **dvcp** and **pcp** commands copy one or more source files, specified by the *sourcepath* argument, to a destination file or directory, specified by the *destpath* argument. If *sourcepath* is not a single filename, *destpath* must be a directory name. For example,

```
% dvcp dv1:/my_new_data dv1:/my_old_data sda1:/mydata
```

copies the DataVault files **my_new_data** and **my_old_data** to the SDA directory **mydata**.

NOTE: **dvcp** does not automatically copy the attribute file associated with a CMFS source file, so if you are keeping track of attribute files, you must explicitly name the attribute file on the **dvcp** command line. If the name of the *destpath* source file is different from that of the *sourcepath* source file, be sure to rename the *destpath* attribute file to agree with the name of the *destpath* source file, i.e., .*destpath_source_name*.**attr**.

Copyright © 1994 Thinking Machines Corporation