The
Connection Machine
System

Connection Machine I/O System Release Notes

Version 6.0
November 1990

These release notes
replace all previous
CM I/O System
Release Notes

Thinking Machines Corporation
Cambridge, Massachusetts

First printing, November 1990

Thinking Machines Corporation
245 First Street
Cambridge, Massachusetts 02142–1264
(617) 234–1000/876–1111

Contents

Customer Support

Thinking Machines Customer Support encourages customers to report errors in Connection Machine operation and to suggest improvements in our products.

When reporting an error, please provide as much information as possible to help us identify and correct the problem. A code example that failed to execute, a session transcript, the record of a back-trace, or other such information can greatly reduce the time it takes Thinking Machines to respond to the report.

To contact Thinking Machines Customer Support:

U.S. Mail:	Thinking Machines Corporation Customer Support 245 First Street Cambridge, Massachusetts 02142–1264
Internet Electronic Mail:	customer–support@think.com
Usenet Electronic Mail:	ames!think!customer-support
Telephone:	(617) 234–1000/876–1111

For Symbolics users only:

The Symbolics Lisp machine, when connected to the Internet network, provides a special mail facility for automatic reporting of Connection Machine system errors. When such an error occurs, simply press Ctrl–M to create a report. In the mail window that appears, the `To:` field should be addressed as follows:

`To:` customer–support@think.com

Please supplement the automatic report with any further pertinent information.

Customer Support

Thinking Machines Customer Support encourages customers to report errors in Connection Machine operation and to suggest improvements in our products.

When reporting an error, please provide as much information as possible to help us identify and correct the problem. A code example that failed to execute, a session transcript, the record of a backtrace, or other such information can greatly reduce the time it takes Thinking Machines to respond to the report.

To contact Thinking Machines Customer Support:

U.S. Mail:
Thinking Machines Corporation
Customer Support
245 First Street
Cambridge, Massachusetts 02142-1264

Internet
Electronic Mail: customer-support@think.com

UUCP
Electronic Mail: ames!think!customer-support

Telephone: (617) 234-1000 (876-1111)

For Symbolics users only:

The Symbolics Lisp machine, when connected to the Internet network, provides a special mail facility for automatic reporting of Connection Machine system errors. When such an error occurs, simply press Ctrl-M to create a report. In the mail window that appears, the To: field should be addressed as follows:

To: customer-support@think.com

Please supplement the automatic report with any further pertinent information.

1 About Version 6.0

Version 6.0 is the fifth release of the Connection Machine (CM) I/O system software, which provides a CM file system and permanent disk storage for CM data. A CM I/O software release shares the version number of the CM System Software release that supports it.

Version 6.0 CM I/O software supports both the CM-IOP and the VMEIO host computer, which allow data storage and acquisition devices to be integrated into the CM system. Version 6.0 also supports the following new CM I/O software features:

- Buffered I/O
- DataVault striping
- File permission checking

In addition to describing how to use these features and the two new data storage devices, these release notes also document other enhancements to the software and suggests work-arounds to some restrictions.

1.1 Porting Code from Previous Releases

Programs using Version 5.0, 5.1, or 5.2 of the CM System Software can be executed under Version 6.0. To take advantage of the new features in Version 6.0, relink these programs with the Version 6.0 libraries.

Programs developed under Version 4.3 or earlier releases must be executed in back-compatibility mode under Version 6.0.

2 Support for the CM-IOP and VMEIO Host Computer

Version 6.0 CM I/O software supports Thinking Machines Corporation's CM-IOP, a new data storage device that connects devices containing SCSI drives, such as StorageTek tape drives, to the CM system. Another data storage device newly supported in Version 6.0, the VMEIO host computer, allows devices with a VMEbus interface—such as magnetic tape drives and video framegrabbers—to be integrated into the CM system.

A CM file system directory tree can reside on the CM-IOP or VMEIO host computer. A secondary function of the CM-IOP or VMEIO host computer, therefore, is to store CMFS files.

You can use the CM-IOP and VMEIO host computer as either a client or a server.

- Used as a client, the CM-IOP and VMEIO host computer provide fast access to a DataVault, for example: they can use a local tape drive to back up a DataVault, and it can also use that tape drive to quickly load a database onto the DataVault.

  To use the CM-IOP or VMEIO host computer as a client, use UNIX I/O routines to transfer data between the CM-IOP or VMEIO host computer memory and a storage or acquisition device connected to it. Use the CMFS serial I/O calls, as explained in Chapter 5 of the *Connection Machine I/O System Programming Guide*, to transfer data between CM-IOP or VMEIO host computer memory and other CMFS data storage devices.

- Used as a server, the CM-IOP and VMEIO host computer enable the CM to quickly access a file residing on the machine's CMFS directory tree or a data acquisition device (such as a framebuffer) local to it.

  NOTE: The CM-IOP and VMEIO host computer used as a server enable the CM to read from or write to a tape drive. Doing so, however, is inefficient because the CM operates at a much higher speed than the usual tape speed (generally less than 3 megabytes/second).

  To use a CM-IOP or a VMEIO host computer as a server, write and install an appropriate device driver, as explained in Chapter 10 of the *Connection Machine I/O Programming Guide*. Then call CMFS read and/or write routines, as explained in Chapter 5 of the *Connection Machine I/O System Programming Guide*.

2.1 New VMEIO Device Driver Library Routines

With Version 6.0, there are two new library routines to use when writing device drivers for a CM-IOP or VMEIO host computer:

- **`CMFS-ioctl`**

  Support CM character-special device drivers.

- **`CMFS-mknod`**

  Create a new file that can call a specific device driver.

See Appendix D of the *Connection Machine I/O System Programming Guide* for the man pages for these calls.

3 CM I/O Feature Enhancements

3.1 Buffered I/O

Version 6.0 of the CM I/O software supports buffered I/O, which can increase the performance of a program doing many small sequential (but not real-time) I/O transactions.

In buffered I/O, a program first sets up a buffer inside the CM into which data is read, and from which data is written. Transfer of data into the buffer requires a relatively large latency period. Once the data is in the buffer, however, the actual transfer is performed quickly, relative to both synchronous and streaming I/O transfers.

Typically, a large amount of data will be read into the buffer from a DataVault; the buffer will then be emptied gradually by a number of calls to read small portions of that data. When the buffer is empty, another large transfer into the buffer takes place.

On writing, the reverse occurs. The buffer fills gradually with small transfers of data from the CM. Then, when the buffer is full, the buffer-full of data is transferred to the DataVault.

Note that the DataVault and CMIO bus are in use only during the actual transfers over the bus; they are not tied up for the duration of the buffered I/O.

3.1.1 New Library Calls for Buffered I/O

For buffered reads and writes, use the following calls in this order:

(1) **`CMFS-setbuffer`**

`CMFS-setbuffer` assigns a CM field to be used as a buffer for a particular open file, which is identified by a file descriptor. The size of the buffer sets the amount of data that can be transferred to or from the disk at one time.

(2) **`CMFS-buffered-read-file-always`**, **`CMFS-buffered-write-file-always`**

`CMFS-buffered-read-file-always` transfers data from the buffer to a CM field; **`CMFS-buffered-write-file-always`** transfers data from the CM field to the buffer.

These two calls return the number of bits read or written to or from the CM field per virtual processor.

If there is not enough data in the buffer to satisfy a **`CMFS-buffered-read-file-always`** call, the file server automatically transfers another buffer-full of data from the file into the buffer, allowing the read operation to complete. Likewise, whenever a call to **`CMFS-buffered-write-file-always`** fills the buffer, the file server automatically completes the transfer by writing the buffer's data to the file; when the buffer is empty again, the write operation will complete itself if necessary.

If a file-to-buffer or buffer-to-file transfer occurs, you do not regain control until the transfer is complete.

(3) **`CMFS-flush`**

To ensure that all buffered writes are actually sent to the disk, call either **`CMFS-flush`** or **`CMFS-close`** before exiting your program.

Call **`CMFS-flush`** before changing between reading and writing the file.

For a more complete explanation of buffered I/O, see Chapter 5 of the Version 6.0 *Connection Machine I/O System Programming Guide*. The man pages for these new library calls are in Appendix D of the *Connection Machine I/O System Programming Guide*.

3.2 DataVault Striping

DataVault striping, so named because a file is distributed or "striped" across more than one DataVault, allows you to store files larger than 40 gigabytes and access them at several times the I/O bandwidth associated with an unstriped DataVault. Striped DataVaults—either 2, 4, or 8 to a set—are each connected via a separate CMIO bus to the CM or another data storage device. One of the DataVaults, the "master", transparently coordinates the other DataVaults in the set; users, therefore, access the striped set via CMFS calls and commands to the master. Ask your site's system administrator which DataVault is configured as the master of the striped set.

To ensure proper performance, all the DataVaults in the striped set must be accessible to the environment requesting DataVault I/O. This requires all the DataVaults to be up and running with a CMIO bus connection to the working environment. (Operations that do not involve reading or writing of data, such as listing directory entries, require only a connection to the master DataVault's running file server.)

Within a CM I/O system, it is possible for one or more sets of striped DataVaults to co-exist with one or more non-striped DataVaults. It is not possible for a file to be stored on just one member of a striped DataVault set; if you want a file stored on just one DataVault, that DataVault must not be a member of a striped set.

3.3 File Permission Checking

In CMSS versions previous to 6.0, file permissions were set by **`CMFS-creat`** and **`CMFS-open`**, but not checked. Version 6.0 software, however, does check permissions once the CM file system server is restarted (by your site's system administrator) with the new **`-p 1`** option to **`fsserver`**.

To enable permission checking for a program written under CM system software previous to Version 6.0, relink the program with the Version 6.0 libraries. (Note that software linked with libraries previous to Version 6.0 will not work with a file server that has permission checking enabled.) Then verify that the permissions of existing files are correctly set; if they are incorrectly set, working programs may fail when run under Version 6.0. Also verify that programs set permissions correctly for existing or future files.

3.3.1 New File Permission Commands and Library Calls

Three new commands—**cmchmod**, **cmchgrp**, **cmchown**—and four new library calls—**CMFS-[f]chmod**, **CMFS-[f]chown**—allow users to change permissions on CMFS files. Appendixes C and D of the Version 6.0 *Connection Machine I/O System Programming Guide* provides man pages for these commands and calls.

3.4 CM Fortran Interface

New to CMSS Version 6.0 is a collection of CM Fortran utility subroutines that invoke CMFS routines:

- `SUBROUTINE CMF_FILE_OPEN( UNIT, PATH )`
- `SUBROUTINE CMF_FILE_CLOSE( UNIT )`
- `SUBROUTINE CMF_FILE_REWIND( UNIT )`
- `SUBROUTINE CMF_FILE_LSEEK( UNIT, OFFSET )`
- `SUBROUTINE CMF_CM_ARRAY_FROM_FILE( UNIT, DEST )`
- `SUBROUTINE CMF_CM_ARRAY_TO_FILE( UNIT, SRC )`

These subroutines are supported in Version 1.0 of CM Fortran and can be used in programs compiled with either the **-paris** or **-slicewise** compiler switches. They are described in Appendix A of the Version 6.0 *Connection Machine I/O System Programming Guide.*

IMPORTANT

Please update your copy of the Version 6.0 *Connection Machine I/O System Programming Guide*:

In Appendix A, `SUBROUTINE CMF_CM_ARRAY_FROM_FILE( UNIT, DEST )` is erroneously listed as `SUBROUTINE CMF_FILE_READ_ARRAY( UNIT, DEST )`, and `SUBROUTINE CMF_CM_ARRAY_TO_FILE( UNIT, SRC )` is erroneously listed as `SUBROUTINE CMF_FILE_WRITE_ARRAY( UNIT, SRC )`.

3.5 More New CMFS Commands

In addition to the new feature-specific commands listed in Sections 2.1, 3.1.1, and 3.3.1 of these Release Notes, the following CMFS user commands are new in CMSS Version 6.0.

- **`cmstat`**

  Print status information about a CMFS file, including its type, mode, owner, size, and time of last access, inode modification, and change of contents. In addition, geometry information is listed for parallel files.

- **`show CMFS directory`** (Lisp/Paris only)

  List the contents of a directory. Information about these files, including their size and modification time, is also printed. This command is available only on a Symbolics Lisp Machine.

The man pages for these commands are in Appendixes C and D of the Version 6.0 *Connection Machine I/O System Programming Guide*.

3.6 Modifications to CMFS Calls and Commands

The man pages for these commands and calls are in Appendixes C and D of the Version 6.0 *Connection Machine I/O System Programming Guide*.

- **`CMFS-open`**

  For simplification, the **`CMFS-O-RECORDWISE`** flag is desupported in CMSS Version 6.0, although the flag will still function as expected in programs developed under versions previous to 6.0 and executed under Version 6.0.

- **`CMFS-errno`**

  Please note the following new error numbers to which the CM file system can set **`CMFS-errno`**.

  181 **CMFS_ESTRIPE_NOTAVAIL**
  One or more of the file servers serving the striped CM file system is not running.

  182 **CMFS_ECMIOBUS_UNREACH**
  The requested CMIO bus is unreachable from the file server.

183 **CMFS_EPROTOCOL_MISMATCH**
There is no library support for the feature (for example, disk striping) that the file server is attempting to use, or the file server doesn't support the feature that the application program is trying to use.

184 **CMFS_ESYNC_LOST**
The synchronization has been lost on the network connection between the CMFS library and the CMFS file server.

185 **CMFS_ENO_FILE_SERVER**
The file server to which the front-end computer is trying to connect is not running.

- **dvcp**

  dvcp copies *file1*, a CMFS file, to *file2*. The following changes have been made in Version 6.0:

  If the physical width of *file1* matches the physical size of the part of the CM you are currently attached to, *file2* is created with the same geometry (including on-chip and off-chip bits) of *file1*.

  If the physical width of *file1* does not match the physical size of the part of the CM you are currently attached to, *file2* is created with:

  - a physical width that matches the number of physical processors in the part of the CM to which you are currently attached
  - a rank, dimension length, and dimension ordering that matches that of *file1*
  - on-chip and off-chip bits that may be different from those of *file1*

  If *file1* is a serial file, *file2* is also a serial file.

- **copyfromdv** and **copytodv**

  copyfromdv and **copytodv** now automatically copy the attribute files associated with the file(s) named by the *sourcepath* argument.

- **cmdd**

  cmdd now supports the **-a** option, which appends to the output file.

4 Bugs

4.1 New Bugs

The following bugs are new to CMSS Version 6.0. Note the some bugs affect the site system administrator only.

4.1.1 cmls-with-permission-checking-enabled Bug

If an **fsserver** process is running with permission checking enabled, using **cmls** to list the root directory returns an error. There are two work-arounds:

- Turn off permissions checking (execute **fsserver** ... **-p 0**) before executing **cmls** [*hostname*:]**/**.
- Use the **DVWD** environment variable to set the current working directory to the root directory. Then execute **cmls** without the *root-directory* argument. For example, from a C shell, execute

  ```
  % setenv DVWD [hostname:]/
  % cmls
  ```

  From a Bourne shell, execute

  ```
  $ DVWD = [hostname:]/
  $ export DVWD
  $ cmls
  ```

4.1.2 I/O-not-on-processor-boundary Bug

If you attempt to perform I/O when not on a processor boundary, the library calls **CMFS-read-file(-always)** and **CMFS-write-file-(-always)** return –1 without first setting **CMFS-errno**. As a result, the message reported by **CMFS-perror** may not correspond to the latest I/O error.

4.1.3 short-read-from-VMEIO Bug

You cannot use **CMFS-read-file-always** to read an amount of data elements less than the physical size of the CM from a *serial* file on a VMEIO host computer into the CM.

4.1.4 cmtar-on-StorageTek Bug

cmtar fails to put a StorageTek tape drive into fixed block mode, resulting in an error message from the SCSI driver and from **cmtar**. A patch for this bug is on the 6.0 CM-IOP release tape.

4.1.5 Ciprico-device-driver Bug

There is a bug in the Ciprico device driver in the 6.0 VMEIO software. The bug manifests itself when multiple Ciprico SCSI boards are installed in the CM-IOP. (This bug does not affect VMEIO host computers.) To fix the driver:

1. Edit the following portion of **/sys/tmc/cip.c** as described below:

```
int
cipc_intr(unit)
      int unit;
{
   register struct cipc_softc *cipc = &cipc_softc[unit];
   register struct cipc_qio *qio;
   register struct scsi_unit *un;
   register STATBLK *sh;
   register byte error = 0;
   register long resid;
```

   Change the shaded line to:

```
register long resid = 0;
```

2. Since the CM-IOP installation is automated by the install-cmiop script, it is necessary to rebuild the kernel by hand, as follows:

```
# cd /sys/sun4/CMIOP
# make
# mv /vmunix /vmunix.old
# mv vmunix /
# fastboot
```

If you are not using the script to install the CM-IOP software, make the above modification to the driver during the installation and rebuld the kernel only once.

4.2 Fixed Bugs and Workarounds

The following bugs were reported in versions previous to CMSS Version 6.0. Most of these bugs have been fixed in CMSS Version 6.0; for those that have not been fixed, workarounds are published here.

4.2.1 cmls Reports the Correct Size of Files

In previous releases, **cmls** reported incorrectly the size of files larger than 2^{31} bytes. In CMSS Version 6.0, however, **cmls** reports the correct size of files larger than 2^{31} bytes.

4.2.2 bad-file-geometry-message Bug Fixed

In CMSS versions previous to Version 6.0, sometimes execution of a CMFS command that opens a file, such as **cmdu**, **cmfind**, and **cmtar**, erroneously sent to the standard output the following message for .bitmap files and for attribute files.

```
CMFS: ignoring bad file geometry info for filename.
```

In CMSS Version 6.0, this message only occurs for CMSS Version 5.1 files with bad geometry—files for which this message is valid. See Section 4.2.4.

This bug was reported in the March 1990 edition of *In Parallel*.

4.2.3 fcntl-streaming-flag-not-cleared Bug Fixed

In CMSS versions previous to Version 6.0, the streaming-attribute bit was not automatically cleared. In CMSS Version 6.0, the streaming-attribute bit is automatically cleared.

This bug was reported in the March 1990 edition of *In Parallel*.

4.2.4 Lisp Files Created Under CMSS 5.1.

Lisp files created under CMSS 5.1 were assigned incorrect geometries.

To create a correct geometry for a Version 5.1 Lisp file, create an empty file with the right attributes and copy those attributes over the file's bad attributes. An easy way to do this is:

```
% cmattach -g desired-geometry other-arguments
% fstest
        opencreat filename mode permissions
        quit
% cmdetach
% cmcp .filename.attr .file-with-bad-geometry.attr
```

4.2.5 Symbolics–DataVault TCP Connection

For a variety of reasons, the TCP connection between a Symbolics front end and DataVault can break; if it breaks, the front end will not try to reinstate the connection. When the TCP connection is broken, all subsequent CMFS calls are cast into the debugger with an appropriate error message unless the call is wrapped with the following macro:

```
(defmacro with-tcp-connection (&body body)
    `(condition-case ()
         ,@body
       (sys:network-error (progn
                            (setq cmfs-lisp::*initialized*
                              nil)
                            ,body))
))
```

A call with this wrapper would look something like:

```
(with-tcp-connection (cmfs:close "/"))
```

This macro can also be used with the **cmls** command.

The
Connection Machine
System

Connection Machine I/O System Programming Guide

Version 6.0
November 1990

Thinking Machines Corporation
Cambridge, Massachusetts

First printing, November 1990

Thinking Machines Corporation
245 First Street
Cambridge, Massachusetts 02142–1264
(617) 234–1000/876–1111

Contents

Appendixes

Figures and Tables

Figures

Tables

Figures and Tables

Figures

Tables

About This Manual

Objectives of This Manual

This manual introduces CM users to the CM I/O system and its file system. It describes how to program the CM I/O system using CM I/O system calls from within programs, and how to manipulate the CM file system using user commands. It also provides adaptable sample programs that use the CM I/O system.

Although the CM graphics display is an output device, it is not part of the CM I/O system. Graphics is a separate system within the CM system, and is described in the *Connection Machine Graphics Programming* manual set.

Intended Audience

We assume that readers of this manual know how to program a CM from a front end using one of the CM languages. We also assume that the reader is familiar with the basic concepts of the UNIX file system.

Readers wishing to acquire more information about the CM system, about using a CM from a front end, and about the CM languages may consult the following volumes of the Connection Machine documentation set:

CM User's Guide
Connection Machine Programming in C/Paris
*Connection Machine Programming in C**
*Connection Machine Programming in *Lisp*
Connection Machine Programming in Fortran

Readers wishing to acquire more information about the UNIX file system—both its everyday use in managing files and its use in I/O programming—may consult the VAX ULTRIX documentation set, available from Digital Equipment Corporation, and the SunOS documentation set, available from Sun Microsystems. Specific references are provided below.

In the VAX ULTRIX documentation set:

- For information about the UNIX file system:
  The Little Gray Book: An ULTRIX Primer
  Supplementary Documents, Volume I: General User. Part I: Overview

- For information about UNIX I/O programming:
  Supplementary Documents, Volume II: Programmer
  Part I: Programming Considerations
  Part 4: System Programming

In the SunOS documentation set:

- For information about the UNIX file system
  Getting Started with SunOS: Beginner's Guide
  Doing More with SunOS: Beginner's Guide
- For information on UNIX shell environment variables:
  Setting Up Your SunOS Environment: Beginner's Guide
- For getting started in low-level I/O programming:
  Programming Utilities and Libraries. Chapter 2: SunOS Programming

Revision Information

This manual replaces *CM I/O Programming Guide* Version 5.2, *CM I/O System Reference Manual* Version 5.1, and all previous CM I/O system release notes.

Organization of This Manual

Part I Programming the CM I/O System

Chapter 1 Introduction to the CM I/O System
This chapter describes the CM I/O system's hardware components and how they work as directed by the I/O library. It also introduces the concepts of CM I/O and serial I/O.

Chapter 2 The CM File System
Chapter 2 discusses the CM file system and introduces the I/O library's commands and environment variables, which prepare files to be used by programs.

Chapter 3 Introduction to CMFS Programming
Chapter 3 introduces the I/O library's calls and provides a foundation for using them to write programs that perform CM I/O and/or serial I/O.

Chapter 4 Creating, Opening, and Closing Files
This chapter describes how to create, open, and close files.

Chapter 5 Reading and Writing
This chapter describes reading and writing files, differentiating between CM I/O and serial I/O. It also explains how to prepare files for reading and writing.

Chapter 6 Transposing Data
This chapter describes transposing and byte-swapping data.

Chapter 7 Running and Debugging Programs
This chapter explains how to run programs that contain code that uses the CM I/O system. It also introduces debugging procedures.

Chapter 8 Sample Programs
This chapter contains three sample programs that perform I/O.

Part II Advanced Programming Topics

Chapter 9 Getting the Best Performance from CM I/O
This chapter introduces some techniques for obtaining good performance from the CM I/O system.

Chapter 10 CM Character-Special Files
This chapter introduces special files and explains how to write a device driver to manage them.

Appendixes

Appendix A Accessing the CM File System from CM Fortran
This appendix explains how to use the collection of CM Fortran utility subroutines that invoke I/O routines.

Appendix B Wildcard Files
This appendix introduces wildcard files and offers some caveats about using them.

Appendix C CMFS User Commands and Environment Variables
This appendix contains manual pages for the I/O library's commands and environment variables.

Appendix D CMFS Library Calls
This appendix contains manual pages for the I/O library's calls.

Notation Conventions

The table below displays the notation conventions observed in this manual.

Convention	Meaning
typewriter	UNIX and CM system software commands, command options, and file names. Also, Paris, C, and Lisp language elements, such as keywords, operators, and function names, when they appear embedded in text.
UPPERCASE	Fortran language elements, when they appear embedded in text.
italics	Parameter (argument) names and placeholders in command and system call formats.
`typewriter`	Code examples and code fragments.
% boldface `typewriter`	In interactive examples, user input is shown in **bold typewriter** and system output is shown in `regular typewriter` font.
|	"Or," to indicate a choice among two or more arguments.
[ ]	Indicates optional arguments or system call variations. For example, **CMFS-[f]stat** refers to both the system calls **CMFS-stat** and **CMFS-fstat.**

Customer Support

Thinking Machines Customer Support encourages customers to report errors in Connection Machine operation and to suggest improvements in our products.

When reporting an error, please provide as much information as possible to help us identify and correct the problem. A code example that failed to execute, a session transcript, the record of a back-trace, or other such information can greatly reduce the time it takes Thinking Machines to respond to the report.

To contact Thinking Machines Customer Support:

U.S. Mail:	Thinking Machines Corporation Customer Support 245 First Street Cambridge, Massachusetts 02142–1264
Internet Electronic Mail:	customer–support@think.com
Usenet Electronic Mail:	ames!think!customer-support
Telephone:	(617) 234–1000 (617) 876–1111

For Symbolics Users Only

The Symbolics Lisp machine, when connected to the Internet network, provides a special mail facility for automatic reporting of Connection Machine system errors. When such an error occurs, simply press Ctrl-M to create a report. In the mail window that appears, the To: field should be addressed as follows:

```
To:  customer-support@think.com
```

Please supplement the automatic report with any further pertinent information.

Part I
Programming the CM I/O System

Part I

Programming the CM/IO System

Chapter 1

The CM I/O System

The CM I/O system facilitates the storage and retrieval of massive amounts of data and significantly enhances CM application performance by providing high-speed data transfer among the components of the CM system. The CM I/O system is both flexible and easy to use; it can incorporate a wide variety of I/O devices and access any of these devices through software designed specifically for the CM I/O system.

This chapter describes the hardware components of a typical CM I/O system and demonstrates how these components interact when directed to perform an I/O transaction.

1.1 Components of the CM I/O System

Figure 1 shows the hardware components of a CM I/O system. It consists of one or more data storage devices, listed below, connected to the CM and to the front-end computer(s).

- The *DataVault* is the CM system's main storage device. It provides up to 40 gigabytes of disk storage. The *DataVault file server computer*, a microcomputer inside the DataVault, manages the DataVault's files. The DataVault is dual-ported to allow higher bandwidth data transfer and greater system flexibility.

  The DataVault can support *striping*, which means it works as a team with one, three, or seven other DataVaults to provide storage space for bigger files and faster access to them. The set of striped DataVaults functions as a single logical storage device; one of the DataVaults, designated *master*, receives instructions from the CM I/O system and controls the rest of the DataVaults, the *slaves*.

- The *VMEIO host computer* typically connects VME-based I/O devices, such as video framegrabbers and magnetic tape drives, to the CM system. Such devices provide convenient means of backing up and retrieving data. A file server process

can run on the VMEIO host computer, providing access to files stored on the host itself.

- The *CM-IOP* also supports the VME protocol. Typically, though, it connects storage devices with SCSI drives, such as StorageTek tape drives, to the CM system. Like the VMEIO host computer, the CM-IOP itself can provide secondary storage for files.

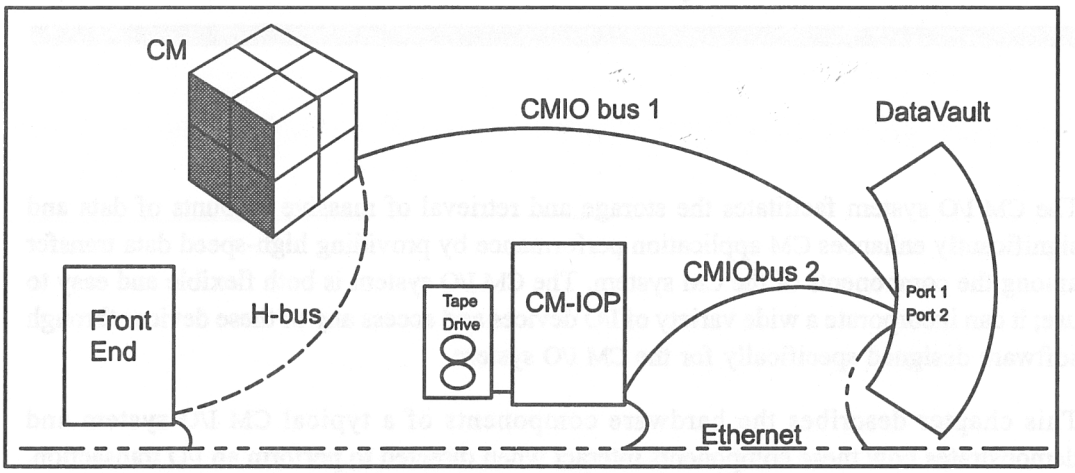

Figure 1. A typical CM I/O system.

The data storage devices receive and send data through Thinking Machines Corporation's high-speed multidrop *CMIO bus*, which can transfer data at 50 megabytes per second. A CMIO bus connects each data storage device to the CM and/or to one or more other data storage devices. In Figure 1, for example, you could do

- CM–DataVault transfer over CMIO bus 1
- CM-IOP–DataVault transfer over CMIO bus 2

A CMIO bus is connected to CM processors through a *CMIOC* (CM I/O controller) interface board. Each set of 8K processors can have a CMIOC. A CMIO bus can be connected to more than one CMIOC, enabling more than one set of 8K processors to access the bus.

As shown in Figure 1, every data storage device is connected to the front-end computer(s) and to the other data storage devices via an Ethernet, which conveys control and status messages. In the absence of an appropriate CMIO bus connection, the Ethernet can transfer data, albeit slowly compared to the CMIO bus speed. For example, since the CM I/O system in Figure 1 has no CMIO bus connection between the CM and the CM-IOP, CM–CM-IOP data transfer would proceed over the Ethernet. If a need were to arise in this system for frequent CM–CM-IOP data transfers, a CMIO bus connection should be installed between the CM and the CM-IOP.

1.2 I/O Processes on the Connection Machine System

The CM I/O system is modeled on the typical client-server relationship—a program running on a machine acting as client requests that a component running a file server process either send or receive data.

- The CM can only be a client
- The DataVault can only be a server
- The CM-IOP, VMEIO host computer, and front end can be either client or server

When the CM is the client, you can perform *CM I/O*—that is, use any server to read to or write from the CM—using the *CMFS (CM File System) library* commands, calls, and environment variables introduced in Chapters 2 and 3. During CM I/O, the following actions occur, seemingly simultaneously (see Figure 2):

- An Ethernet conveys the I/O instruction from the front end (where the CM application program is running) to the server (A), which either receives data or sends it, depending on whether the instruction is to write or read (D). The server then sends a completion message back to the front end via the Ethernet (E).
- The front end's H-bus conveys the I/O instruction from the front end to the CM (B), where the sequencer(s) to which the front end is attached tells the processors about the I/O request. Then the CMIOC(s) controlled by the sequencer(s) either sends data to or receives it from the server, depending on whether the instruction is to write or read (C). If a CMIO bus connects the CM and the server, the bus transfers the data; if not, an Ethernet transfers the data, albeit slowly.

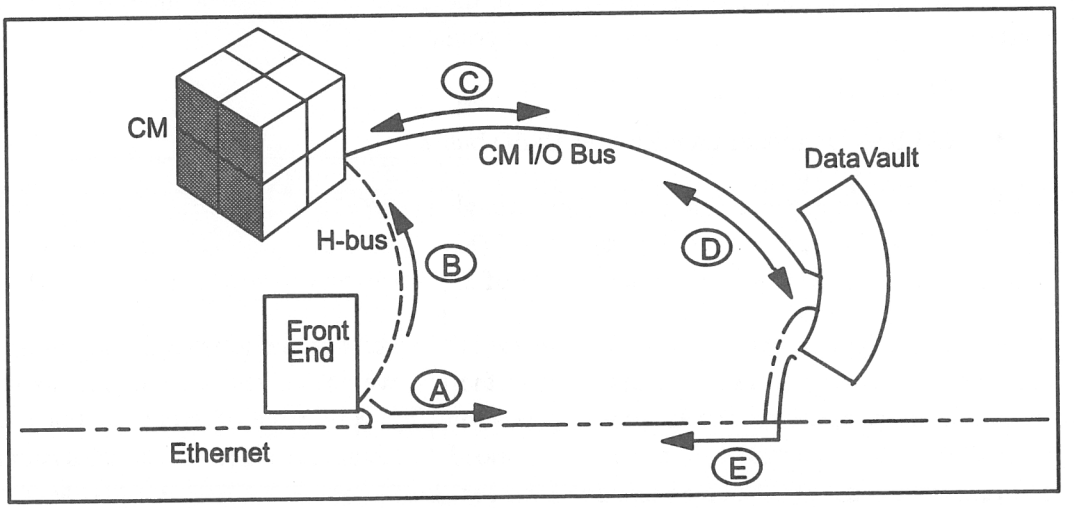

Figure 2. The CM I/O process discussed above. The CM is acting as client, and the DataVault is acting as server.

1.2.1 Serial I/O

When a CM-IOP, VMEIO host computer, or front end is the client, you can perform *serial I/O*—that is, use any server to read to or write from the client's memory. (*Serial* emphasizes that the CM is never involved in I/O transactions that occur when it is not the client.) Use serial I/O to perform operations auxillary to CM I/O: for example, loading a database into a DataVault or placing CM-generated data onto tape, either for backup purposes or for moving to another computer system.

Serial I/O operations also use the CMFS library commands, calls, and environment variables introduced in Chapters 2 and 3. During serial I/O, the following actions occur, seemingly simultaneously:

- An Ethernet conveys the I/O request from the client to the server. Depending on whether the instruction is to write or a read, the server either receives data from or sends it to the client. The server then sends a completion message to the client via the Ethernet.
- The client either sends or receives the data, depending on whether the instruction is to write or read. If a CMIO bus connects the client and the server, the bus transfers the data; if not, the transfer occurs over the Ethernet (less quickly).

Chapter 2

The CM File System

This chapter introduces the CM file system and the files that it stores. It also describes the CMFS library commands, which are used at shell level to prepare files for manipulation by programs containing CMFS library calls (introduced in Chapter 3). The CMFS library is part of the CM System Software.

2.1 Introduction to the CM File System

The CM file system (CMFS) exploits the great speed and massive storage capabilities of the CM I/O system. The CM file system is also easy to use, especially if you are familiar with the standard UNIX file system on a CM system's Sun and VAX front ends. The two file systems are similar in several ways: they organize files into directories, use path names to identify them, and treat I/O devices as files.

Unlike a UNIX file system, however, the CM file system supports multiple directory trees. Each data storage device can have a CM file system directory tree with its own root directory, **/**, as Figure 3 illustrates. The CM file system identifies each data storage device by its unique host name—the name of the machine followed by a colon (:). A file's full path name, therefore, includes a *hostname* component—for example, **`dv1:/big_project/data`**. (Because the hostname component contain a colon character, the name of the file itself cannot contain a colon.)

[NOTE: The *hostname* component of a set of striped DataVaults is a special case. Since the DataVaults function as a single logical device, the *hostname* of the set is the name of the master DataVault. The CM file system ignores requests to slave DataVaults.]

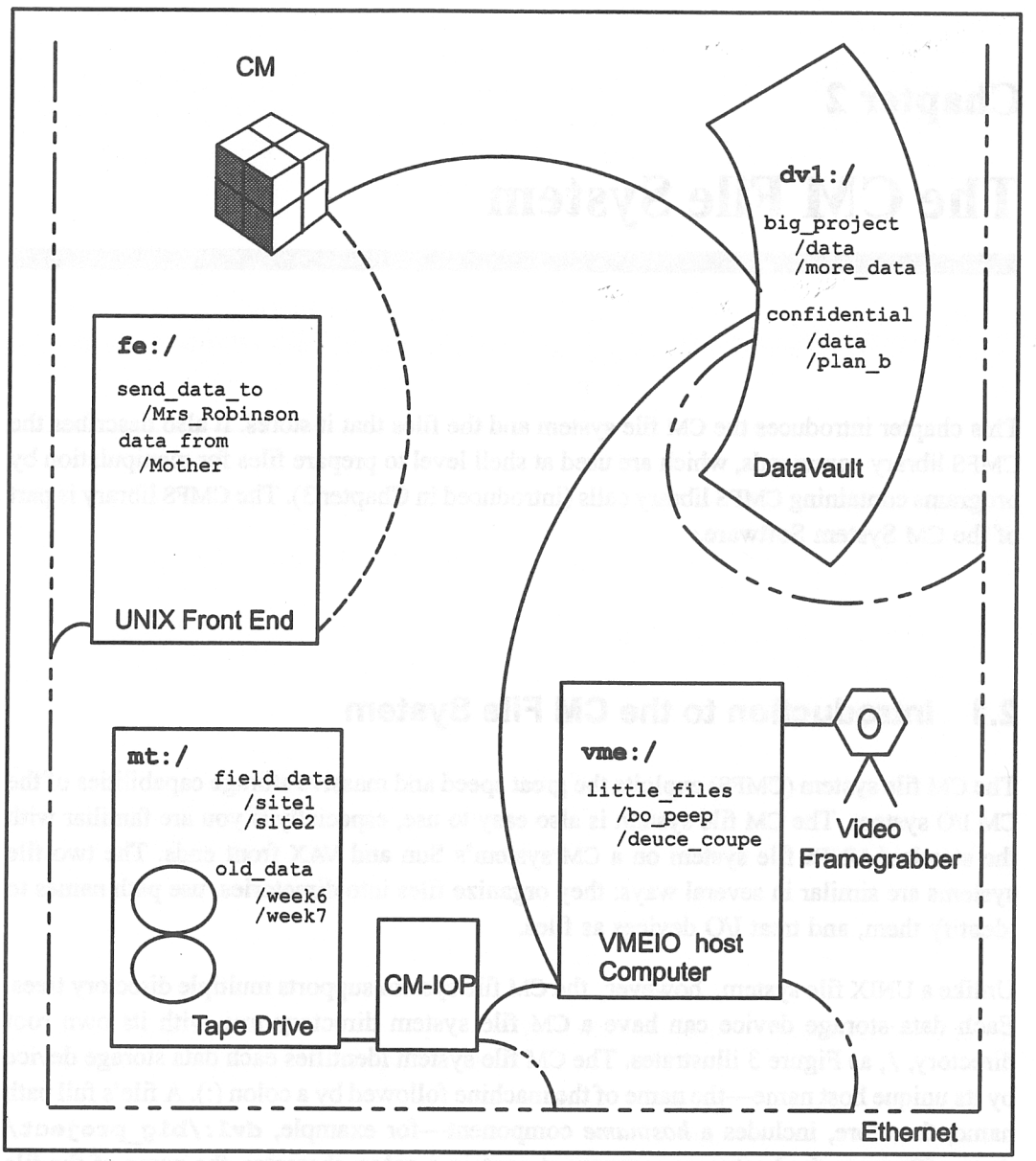

Figure 3. Files within a typical CM file system.

Even a UNIX front-end computer can have a CM file system directory tree, which is sometimes used to temporarily store smaller files used by both the CM and serial computers. In fact, all of the data storage devices can have a UNIX file system as well as a CM file system directory tree. *The two file systems, however, are completely separate.* (The UNIX file system holds the executable programs that run on the CM, as well as CM

system software and (possibly) users' private files. The CM file system holds only data files.) Consequently,

- Files in the UNIX file system are accessed only by the standard UNIX file system commands and calls; the CM file system are accessed only by CMFS commands and calls. (UNIX commands and calls applied to a CM file will fail; likewise, CMFS commands and calls applied to UNIX files will fail.)

- Each file system has its own working directory.

In UNIX, you set your current working directory using the command **cd**. In the CM file system, you set your current working directory using the CMFS environment variable **DVWD**[1]. To change and print your current working directory from a C shell, type

```
% setenv DVWD new-working-directory
% printenv DVWD
```

To change and print your current working directory from a Bourne shell, type

```
$ DVWD = new-working-directory
$ export DVWD
$ printenv DVWD
```

new-working-directory may or may not include a *hostname* component. You can, for example, use **DVWD** to specify a working directory relative to a default CMFS host set by another CMFS environment variable, **DVHOSTNAME**. To set **DVHOSTNAME** from a C shell, type

```
% setenv DVHOSTNAME default-host-name
```

To set **DVHOSTNAME** from a Bourne shell, type

```
$ DVHOSTNAME = default-host-name
$ export DVHOSTNAME
```

For example, if you set the environment variables as follows:

```
% setenv DVWD /big_project
% setenv DVHOSTNAME dv1
```

1. You can set **DVWD** and other CMFS environment variables under Lucid Lisp running on a Sun or VAX, but you cannot set them from within Lisp. The CMFS environment variables are not available on Symbolics front ends.

and in a later operation refer to the file `/data`, the system will interpret the full name of the file as `dv1:/big_project/data`. If you want instead to use the file `dv1:/confidential/data`, however, you would need to either explicitly specify the full name of the file or reset **DVWD** to `/confidential`.

If **DVWD**'s *new-working-directory* does not include a *hostname* component and you do not set **DVHOSTNAME**, the default CMFS host name is determined as follows:

- If you are attached to a CM, `/cm/configuration/configuration.lisp` (the CM system configuration file) provides the default CM file system host.
- If you are not attached to a CM, the front end's `/usr/local/etc/dv_hostname` may provide the default CM file system host. If that file does not provide a host name, the default host is the current system.

For convenience, you can have your `.login` file or your `.cshrc` file set **DVWD** and/or **DVHOSTNAME** each time you open a shell to work in the CM programming environment. Remember to supply a *hostname* component when accessing a file that resides on a different host than the host on which your current working directory resides.

2.1.1 About Parallel Files

Files stored within the CM file system are *CMFS files*. CMFS files are formatted for use either by the CM or by a serial computer. Files created and used by the CM are specially formatted to take advantage of the CM's massive parallelism; these are called *parallel files*. A parallel file consists of many streams of data, one stream per CM processor. In comparison, *serial files*—created on serial computers—consist of a single stream of data.

Just as every data set (array, shape, or pvar) used in a CM program has an underlying geometry that describes how it is laid out across the processors in the current VP set, so each parallel file has a geometry that tells how its streams of data are organized. When a parallel file is created, it inherits a geometry that reflects the following components of the current VP set:

- *Physical width.* The file's physical width is the same as the number of physical processors in the current VP set.
- *VP geometry.* The file's VP geometry reflects the VP geometry of the current VP set:

- The file's VP geometry *size* component tells the length of each of the file's axes, which is equal to the number of virtual processors along each axis of the current VP set.
- The file's VP geometry *shape* component tells how many axes the file has, which is equal to the number of dimensions in the current VP set.
- The file's VP geometry *virtual width* component is the total number of virtual processors in the current VP set..

The file in Figure 4, for example, has physical width = 8K, shape = 2D, size = 2 x 8K, and virtual width = 16K.

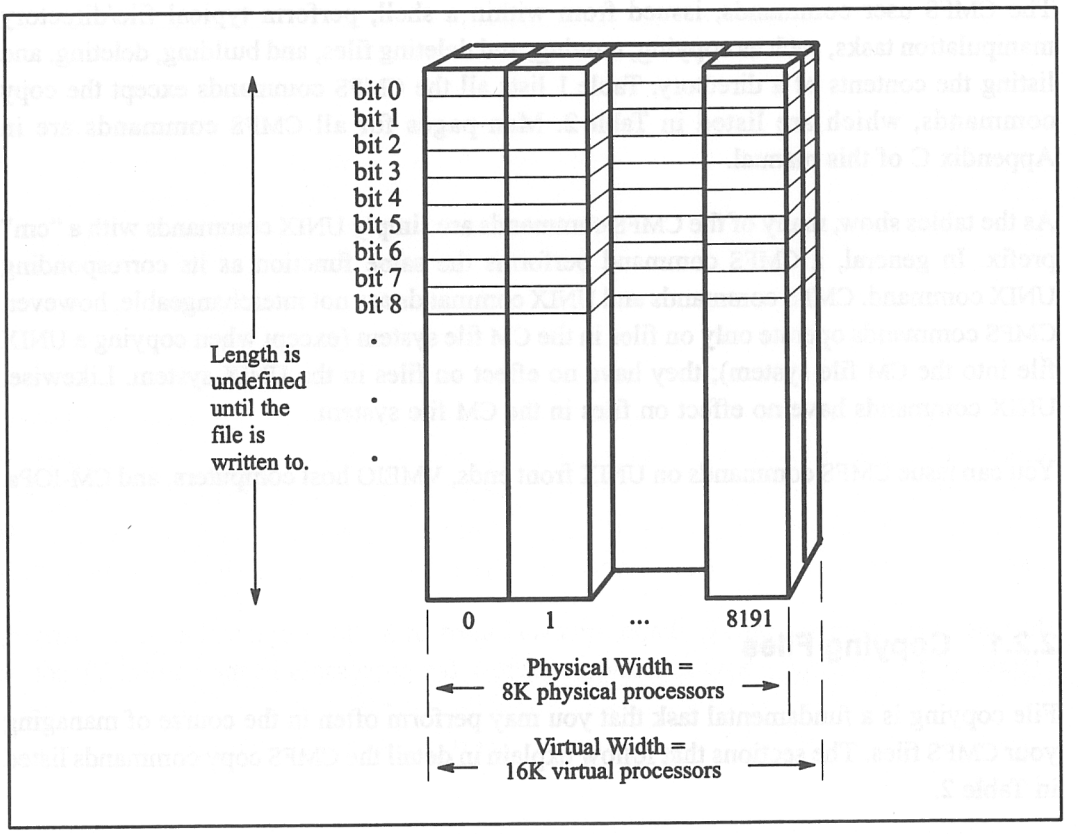

Figure 4. Schematic of a parallel file.
The file's VP geometry: size 2 x 8K, shape 2D.

An *attribute file* (named *.filename*`.attr`) associated with the parallel file records information about the file's geometry in binary format. The attribute file must remain with the parallel file with which it is associated. If you move the parallel file to a new location, be sure also to move its attribute file.

Serial files, since they consist of only one stream of data, do not have a geometry.

2.2 Manipulating Files: The CMFS User Commands

The CMFS user commands, issued from within a shell, perform typical file/directory manipulation tasks, such as copying, moving, and deleting files, and building, deleting, and listing the contents of a directory. Table 1 lists all the CMFS commands except the copy commands, which are listed in Table 2. Man pages for all CMFS commands are in Appendix C of this manual.

As the tables show, many of the CMFS commands are simply UNIX commands with a "cm" prefix. In general, a CMFS command performs the same function as its corresponding UNIX command. CMFS commands and UNIX commands are not interchangeable, however. CMFS commands operate only on files in the CM file system (except when copying a UNIX file into the CM file system); they have no effect on files in the UNIX system. Likewise, UNIX commands have no effect on files in the CM file system.

You can issue CMFS commands on UNIX front ends, VMEIO host computers, and CM-IOPs.

2.2.1 Copying Files

File copying is a fundamental task that you may perform often in the course of managing your CMFS files. The sections that follow explain in detail the CMFS copy commands listed in Table 2.

Like a UNIX copy command, a CMFS copy command makes a copy of the *source* file and places it into the *destination* file. If a CMIO bus connects the source file with the destination file, the copy commands use the bus to achieve good performance. If a CMIO bus is not available, however, the commands use the Ethernet to perform the copy.

Table 1. General CMFS user commands. Unless otherwise noted, you cannot execute these commands from within a Lisp environment.

Command	Use
cmchgrp	Change the group ownership of a file.
cmchmod	Change the permissions mode of a file.
cmchown	Change the owner of a file.
cmdf	Display free and used disk space.
cmdu	Summarize disk usage.
cmfind	Find files.
cmln	Make links to files or directories.
cmls	List a directory's contents (can be executed from Lisp).
cmmkdir	Make a directory.
cmmv	Move (rename) a file or directory.
cmrm	Remove (unlink) a file or directory.
cmstat	Print information about a file, including its geometry information, mode, size, and time of last access.
cmtruncate	Truncate or extend a file.
Show CMFS Directory	List the contents of a directory (can be executed from a Symbolics front end only).

Table 2. CMFS copy commands. You cannot execute these commands from within a Lisp environment.

Command	Location of the File to be Copied	Desired Location of the New File	Comments
cmcp	CM file system	CM file system	Can copy more than one file at a time.
cmdd	Raw data on tape	CM file system	See man page for other uses.
cmtar	Tape archive	CM file system	See man page for other uses.
copyfromdv	CM file system	UNIX file system	Can copy more than one file at a time.
copytodv	UNIX file system	CM file system	Can copy more than one file at a time.
dvcp	CM file system	CM file system	Must be attached to CM; copies one file at a time.

Copying Files within the CM File System: The cmcp and dvcp Commands

Both **cmcp** and **dvcp** copy files within the CM file system. **dvcp** uses the CM to perform the copy, so you must be attached to a CM to use it. You can issue **cmcp** regardless of whether or not you are attached.

dvcp can copy only one file at a time. For example,

```
% dvcp dv1:/mydata dv2:/mydata
```

copies **dv1:/mydata** to **dv2:/mydata**.

cmcp, however, can copy multiple files into a directory. For example,

```
% cmcp dv1:/mydata dv1:/mydata1 dv2:/
```

copies the files **dv1:/mydata** and **dv1:/mydata1** to the root directory of the CM file system on the device named *dv2*.

Copying Files between a UNIX System and a CM File System: The copytodv and copyfromdv Commands

To copy a file residing on a CM file system to a front end's UNIX file system, use **copyfromdv**. For example, to copy into the CMFS file **mydata_1** (residing in your current working directory on the device named *dv1)* into a UNIX file system, issue the following command:

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

mydata_2 now resides in your current UNIX working directory.

To copy the UNIX file **mydata_2** to the file **mydata2_revised** on the device named *dv1*, issue the **copytodv** command. For example,

```
% copytodv mydata_2 dv1:mydata2_revised
```

If the UNIX file you want to copy isn't on a CM front end, but is reachable by network:

- Use the UNIX command **rcp** to copy the file to a front end. Then use **copytodv** to copy it into the CM file system.

- Use the UNIX **rsh** command, followed by **copytodv**, to copy the file directly (if the CMFS software has been loaded on the networked computer and you have an account on the computer).
- Use NFS to mount the file system and treat it as though the file were on a front end.

Copying a Tape Archive File Into a CMFS File: The cmtar Command

To extract one or more files from a tape archive, log in to a machine loaded with CM System Software that is networked to both the tape drive and the machine on which you want to store the file (usually a DataVault). Generally, a VMEIO host computer or a CM-IOP with a local tape drive give the best performance.

To extract files from an archive located on a StorageTek tape drive connected to a VMEIO host computer, for example, log in to the VMEIO host computer and type

```
% cmtar -x -v -b 3200 -M -V volume_name [files]
```

This recommended form of **cmtar** extracts (**-x** option) *files* from the archive, which may span multiple tapes, indicated by the **-M** option. **-V** verifies that the tapes have the same volume name as *volume_name* and that multiple tapes of the same archive are read in the correct order. **-v** displays the name of each file extracted from the archive. **-b** specifies the blocking factor, which must be the same as that used when creating the archive. For a StorageTek tape drive connected to a VMEIO host computer on the same CMIO bus as the DataVault, a blocking factor of 3200 provides the best overall performance.

If you include the optional *files* argument, you must explicitly specify those files' corresponding attribute files. If you do not specify *files*, **cmtar** extracts all files in the archive, including the attribute files. **cmtar** places each extracted file in the current CMFS directory, giving it the name archived with it, unless you specify otherwise.

If you need the extracted files in a front end's UNIX directory also, use the UNIX **tar** command instead of **cmtar**, and then copy the file to the CM file system using **copytodv**.

Copying Raw Data from Tape into a CMFS File: The cmdd Command

Data-collection equipment, such as video cameras, can be used to gather data that is placed on tape in the form of a series of numbers or characters. To copy such raw data from tape into a CMFS file, log in to a machine loaded with CM System Software that is networked to both the tape drive and the machine on which you want to store the data. Generally, a VMEIO host computer or a CM-IOP with a local tape drive gives the best performance.

Then execute the CMFS command **cmdd** by typing something like

```
% cmdd -todv if=tape-drive-name of=hostname:filename bs=1600k
```

The **bs=** argument indicates how many blocks comprise a "chunk" of data written to the output file (the **of=** argument); *1600k* is a good value if the output file resides on a DataVault, because the DataVault block size is 16K bytes.

Chapter 3

Introduction to CMFS Programming

3.1 Introduction to the CMFS Library Calls

The CMFS library calls enable you to manipulate directories, files, and their data from within a program. The library provides an interface to both C/Paris and Lisp/Paris. Programs written in C* use the C/Paris interface, and programs written in *Lisp use the Lisp/Paris interface. Although the CMFS library does not provide an interface to Fortran/Paris, programs written in CM Fortran can access the CMFS library by either:

- using the C/Paris interface to call the CMFS routines as C functions (requires compiling with the *cmfs* and *paris* libraries)
- using a CM Fortran utility subroutine to invoke a CMFS routine (does not require compilation with the *cmfs* and *paris* libraries—Appendix A lists the available subroutines)

For simplicity, this manual refers to specific CMFS library routines using language-independent syntax. For example, when referring in general to the CMFS library call that closes a file, this manual uses **`CMFS-close`** rather than **`CMFS_close`** (the corresponding C/Paris syntax) or **`(CMFS:close)`** (the corresponding Lisp/Paris syntax). Manual pages containing language-specific syntaxes for all CMFS calls are in Appendix D of this manual.

As mentioned in Chapter 1, I/O on the Connection Machine system comes in two flavors: CM I/O and serial I/O. The emphasis of this chapter—indeed, of the rest of this manual—is on CM I/O. Section 3.3, however, describes the serial I/O process. Table 3 lists the CMFS library calls you can use to do either CM I/O[2] or serial I/O.

---

2. In general, **`CMFS-init`** must be called before any CM I/O routine is called—otherwise, CM I/O calls may not perform as expected. Depending on which language you are programming in, **`CM-init`** may be automatically called when you attach to the CM (C* and *Lisp), or you may have to explicitly call it (C/Paris and Lisp/Paris). For information about **`CM-init`**, see the *Paris Reference Manual*.

Table 3. CMFS calls that you can use while performing either CM I/O or serial I/O. Invoke the calls with the **CMFS-** prefix; for example, **CMFS-fchmod**.

CMFS-...	Comments
Calls that operate on files	
[f]chmod	Changes a file's mode.
[f]chown	Changes a file's owner and group.
close	Closes a file.
close-all-files	Closes all files; breaks TCP connections.
close-files-on-server	Closes server's files; breaks TCP connection.
creat	Creates or recreates a file.
fcntl	Controls various characteristics of a file
link	Creates a hard link to a file.
make-stat (Lisp/Paris only)	Creates a data structure for file information.
mknod	Creates a CM character-special file.
open	Opens or creates and opens a file.
perror	Writes an error message to an output file.
rename	Renames a file.
[f]stat	Obtains file status information.
unlink	Removes a file from its directory.
with-open-file (Lisp/Paris only)	Opens a file, processes it, and closes it.
Calls that operate on directories	
chdir	Changes the current working directory.
directory (Lisp/Paris only)	Lists matching path names.
opendir	Opens a directory and returns an identifying pointer.
readdir	Returns a pointer to the next directory entry.
telldir	Returns a pointer to the current directory location.
seekdir	Sets position of the next directory read operation.
closedir	Closes the directory and releases pointer.
mkdir	Makes a directory.
rmdir	Removes a directory.
scandir (C/Paris only)	Builds array of pointers to a directory's entries.

Calls that affect the CM system

errno	Stores the last error's error number.
file-system-reinitialize	Rereads the CM system configuration file to reinitialize internal CMFS data structures.
ioctl	Supports an I/O subsystem device driver.
make-statfs (Lisp/Paris only)	Creates a data structure for file system info.
statfs	Obtains file system statistics.
set_debug_mode	Enable/disable automatic debug-message printing.

Table 4. CMFS calls that you can use only while performing CM I/O. Invoke the calls with the **CMFS-** prefix; for example, **CMFS-read-file-always**.

CMFS-...	Comments
read-file(-always) **write-file(-always)**	Perform synchronous CM I/O (see Chapter 5).
streaming-iostat **partial-read-file-always** **partial-write-file-always**	Perform streaming CM I/O (see Chapter 5).
setbuffer **buffered-read-file-always** **buffered-write-file-always** **flush**	Perform buffered CM I/O (see Chapter 5).
bits-per-processor	Returns number of bits in each processor.
convert-ieee-to-vax-float	Converts floats from IEEE- to VAX-format.
convert-vax-to-ieee-float	Converts floats from VAX- to IEEE-format.
lseek	Moves an open file's pointer.
transpose-always	Transposes data between serial and parallel format.
transpose-record-always	Transposes structures of data between serial and parallel format.
[f]truncate-file	Truncates or extends a file.

3.2 CM I/O

When you perform CM I/O, you use a file residing on any CMFS server to read to or write from the CM. To perform CM I/O, you can use the calls listed in Table 3 and the calls listed in Table 4. You can use these calls to operate on parallel and serial files:

- In general, a program can use a parallel file to do CM I/O *only if the file's VP geometry information matches the VP geometry of the current VP set*, as described in Chapter 2. (The information about the file's geometry stored in the file's attribute file enables the file's data to be placed in the correct VP set processors during a read operation, and then written correctly into the file during a write operation.) Therefore, every time you use a parallel file to do CM I/O, make sure the current VP set has the same VP geometry as it did when the file was created.

- Since a serial file has no geometry, you can use it to do CM I/O regardless of the geometry of the current VP set. However, after it is read into the processors, its data elements must be transposed—rearranged—from serial to the parallel format compatible with the current VP set. Chapter 6 discusses transposition.

The calls listed in Table 4 need to check the VP geometry of the current VP set as well as the VP geometry of the file, so you must be attached to the CM to call them. (You do not have to be attached to the CM to use the calls listed in Table 3, but you can be.) If you are not attached, the call prints the message, *Attempt to do a parallel operation when not attached*, and the program aborts. In order for such a call to proceed *and* to produce results that make sense, the current VP set must have a VP geometry that matches the file's VP geometry information.

3.2.1 The CM I/O Programming Process

To perform CM I/O:

- Attach to the CM.
- Open the file and prepare the file for reading and/or writing.
- Read and/or write.
- Close the file.

The sections below give a general description of these tasks and provide a reference to a chapter that describes the tasks in detail.

Attach to the CM

See the *CM User's Guide* and Chapter 7 of this manual for information about attaching.

Open the File in the Correct VP Geometry

If the file does not yet exist, create it with the pertinent read/write/execute permissions. 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.
- Verify that the VP geometry of the current VP set matches the file's VP geometry information.

See Chapter 4 for detailed information.

Prepare the File for Reading and/or Writing

Set the file pointer appropriately and, if necessary, modify the file's size. See Chapter 5.

Perform I/O

Read the file into the CM and/or write data from the CM into the file. See Chapter 5.

- When reading, if the data came from a serial machine, rearrange the data elements before operating on it. You may also have to perform some format conversion:
  - If the data came from a Sun, byte-swap it before operating on the data.
  - If the data came from a VAX, convert any floating-point numbers from VAX format to IEEE format before operating on the data.

  See Chapter 6 for detailed information.
- When writing, if the data will be used on a serial machine, first rearrange the data elements. You may also have to perform some format conversion:
  - If the data will be used on a Sun, byte-swap it before writing.
  - If the data will be used on a VAX, convert any floating-point numbers from VAX format to IEEE format before writing.

  See Chapter 6 for detailed information.

Close the File

See Chapter 4.

3.3 Serial I/O

When you perform serial I/O, you use a file residing on any CMFS server to read to or write from a CM-IOP, VMEIO host computer, or front-end computer (not attached to the CM). *Serial I/O* emphasizes that the CM is never involved in I/O transactions that occur when it is not the client. A file transferred by serial I/O can be either parallel or serial in format. To perform serial I/O, you can use the calls listed in Table 3 and in Table 5.

Table 5. CMFS calls used only in performing serial I/O. Invoke the calls with the **CMFS-** prefix; for example, **CMFS-serial-lseek**.

CMFS-...	Comments
serial-lseek	Moves an open file's pointer
serial-read-file	Performs a serial read
serial-[f]truncate-file	Truncates or extends a file
serial-write-file	Performs a serial write

3.3.1 The Serial I/O Process

An application that performs serial I/O most often uses both UNIX and CMFS calls to read data from an I/O device into a CMFS file. For example, to read data from a tape drive connected to a CM-IOP into a CMFS file on a DataVault:

1. Open the tape device using the UNIX **open()** call.
2. Open the CMFS file residing on the DataVault using the CMFS open call (see Chapter 4):
   - If the file does not yet exist, create it with the pertinent read/write/execute permissions.

- 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.

3. Prepare the DataVault file for writing (see Chapter 5):
   - If necessary, call **`CMFS-serial-ftruncate-file`** to modify the file's size.
   - Set the file pointer appropriately by calling **`CMFS-serial-lseek`**.
4. Read the data from the tape drive into the DataVault file: Execute a loop consisting of a UNIX **`read()`** followed by a **`CMFS-serial-write-file`** (see Chapter 5).
5. Close the file (see Chapter 4).

The file can then be used to perform CM I/O. If the data came from a serial computer, transpose it (see Chapter 6) after reading it into the CM. If the data was on tape as a backup copy of a parallel file, be sure to take the file's attribute file off the tape, too.

Chapter 4
Creating, Opening, and Closing Files

4.1 Creating Files

To create a CMFS file, call **CMFS-creat** with two arguments: a path name (no more than 255 characters), and a *mode* (a description of the file's permissions). The call sets the owner and group of the file to the UNIX default; you specify the read/write/execute permissions in the second argument by ORing together a combination of the following:

0400 Read by owner	0040 Read by group	0004 Read by others
0200 Write by owner	0020 Write by group	0002 Write by others
0100 Execute by owner	0010 Execute by group	0001 Execute by others

If it is successful, **CMFS-creat** returns a *file descriptor* (a non-negative integer that identifies the file), opens the new file for writing (regardless of its assigned mode), and sets the file pointer to the beginning of the file. If it is not successful, the call returns –1 and sets **CMFS-errno** to a number representing the cause of the error.

The new CMFS file is parallel by default—its geometry matches the geometry of the CM's current VP set. To create a new CMFS file that has no geometry, see Appendix B.

The following C/Paris example creates a file named *myfile,* which resides on the DataVault named *dv1*, and gives read and write permissions to everyone:

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

From Lisp/Paris, call **CMFS:creat** as in the following example:

```
(SETQ my_fd
(CMFS:creat "dv1:/myfile", #O666))
```

4.2 Opening Files

Call **CMFS-open** to open a CMFS file that already exists. When performing CM I/O, however, first check the file's geometry to be sure the current VP set has a VP geometry that matches the file's VP geometry. (The geometries are irrelevant when performing serial I/O.) If they do not match, any subsequent CM I/O read, write, seek, or truncate call will fail, causing your program to abort.

(In a Lisp front-end environment, it is possible to detach from or reconfigure the CM after opening a file. However, the CM file system does not support this. The only CMFS calls that function properly on an open file after the CM is detached or reconfigured are the serial I/O calls (listed in Table 5, in Section 3.3) and **CMFS:close**, **CMFS:close-all-files**, and **CMFS:close-all-files-on-server**.)

4.2.1 Checking a File's Geometry

If you don't know the VP set geometry from which a file was created, you can find out by executing **cmstat**. A partial sample **cmstat** output is listed below.

```
% cmstat myfile
File: "myfile"
   [...]
Created on physical CM size: 8192
Width in virtual processors: 16384
   [...]
Geometry has rank = 2, vp-ratio = 2
Axis 0: len= 128 bits (off, on) chip=4,2 order=news vp-ratio= 2
Axis 1: len= 128 bits (off, on) chip=5,2 order=news vp-ratio= 1
```

From the **cmstat** output, you can tell that *myfile* was created in a VP set that comprised 8K physical processors and 16K virtual processors. Since the output lists two axes, you can tell that the VP set had a 2D geometry (*rank* also indicates the number of dimensions of the CM configuration). Finally, the *len* field listed with each axis indicates that the VP set had a 128 x 128 geometry.

Therefore, any program calling a CMFS library routine that operates on *myfile* must create a VP set with 16K virtual processors configured in a 128 x 128 geometry.

4.2.2 Calling CMFS-open

To open an existing file, call **CMFS-open** with two arguments: a path name and a *flag* argument. Construct *flag* by ORing one of the flags from the column on the left with zero or more flags from the column on the right (these flags are defined in **<cm/cm_file.h>**):

CMFS-O-RDONLY	Open the file for reading only.	CMFS-O-CREAT	If the file does not exist, create it. You must specify mode as a third argument (see Section 4.1).
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.

Note the following caveats:

- If your CM file system checks permissions, you cannot specify a flag from the column on the left that requires permissions greater than those allowed by the file's mode. For example, if the file was created with read-only permissions, you cannot use **CMFS-open** to open the file for writing.

  To check a file's permissions, execute **cmstat** (or call **CMFS-[f]stat**—see its man page in Appendix D of this manual). For example:

```
% cmstat myfile
File: "myfile"
      [...]
      Mode: (0666/-rw-rw-rw) UID: (1155/username)
      Gid: (10/staff)
      [...]
```

  To change a file's owner or group, call **CMFS-chown** or execute **cmchown** or **cmchgrp**; to change the permission of a file, call **CMFS-chmod** or run **cmchmod**. You must be either the current owner of the file or logged on as root. Appendix D of this manual contains the manual pages for these commands and calls.

- If you call **CMFS-open** without specifying the CMFS-O-CREAT flag, the file named in the call's first argument must already exist.

If successful, **CMFS-open** returns a file descriptor. If the call is not successful, it returns –1 and sets **CMFS-errno** to a number representing the cause of the error. At most, any program can have 29 file descriptors open simultaneously; at most, 29 files at a time can be open on each DataVault.

The following C/Paris example opens a *dv1* file named *myfile* for reading only:

```
int my_fd;
my_fd = CMFS_open ("dv1:/myfile", CMFS_O_RDONLY);
```

From Lisp/Paris, call **CMFS:open** as in the following example:

```
(SETQ my_fd
(CMFS:open "dv1:/myfile", CMFS:O-RDONLY))
```

4.3 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.

In C/Paris, it is not necessary to explicitly close a file if the program 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/Paris example closes the file represented by the file descriptor *my_fd*:

```
CMFS_close (my_fd);   /* my_fd had been previously declared as
                         an int */
```

A program running on a Lisp machine should always explicitly close each file:

```
(CMFS:close my_fd)    ;  my_fd had been bound to an open stream
```

If you think there might be files left open either by a previous user of the Lisp machine or due to an abort to top level, then you should call **CMFS:close-all-files** instead.

Chapter 5
Reading and Writing

This chapter explains how to perform CMFS read/write operations.

- Section 5.1 discusses how to prepare your open file for reading and/or writing.
- Section 5.2 discusses how to call the reading and writing routines used when the CM is the client.
- Section 5.3 discusses how to call the reading and writing routines used when a VMEIO host computer, CM-IOP, or front end is client.
- Section 5.4 discusses reading and writing attribute files.

Part III of this manual contains manual pages for all calls described in this chapter.

5.1 Preparing to Read and Write

After you have opened a file, prepare to read and/or write it by making sure the file pointer is set appropriately and modifying the file's size if necessary. The sections that follow discuss these tasks.

5.1.1 Moving a File Pointer

A file pointer marks the position of the file pointer in an open file. The position of the file pointer marks the point in the file at which a subsequent read or write starts.

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 want to start reading or writing the file at the beginning of the file, move the pointer by calling **CMFS-lseek** or **CMFS-serial-lseek**, depending on whether you are performing CM I/O or serial I/O, respectively.

The CMFS **-lseek** calls take three arguments: a file descriptor, an *offset* (a positive or negative integer), and a flag (defined in **<cm/cm_file.h>**) representing the point at which counting is to start. The flags are:

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

For **CMFS-lseek**, the *offset* argument indicates the number of bits *per processor* to offset the file pointer from the location specified by the third argument. To use **CMFS-lseek**, therefore, your program must be attached to the CM because it must determine the number of virtual processors in the current VP set.

For **CMFS-serial-lseek**, however, the *offset* argument indicates the number of *bytes* to offset the file pointer from the location specified by the third argument. Your program, therefore, does not have to be attached to the CM to use **CMFS-serial-lseek**. (Usually, although not necessarily, **CMFS-lseek** is used to move the file pointer of a parallel file, and **CMFS-serial-lseek** is used to move the file pointer of a serial file stored in the CM file system.)

If successful, **CMFS-lseek** and **CMFS-serial-lseek** return a file pointer value of type long. **CMFS-lseek** measures this value in bits per processor from the beginning of the file. **CMFS-serial-lseek** measures this value in bytes from the beginning of the file.

The following C/Paris example performs a "32-bit seek" on the file represented by the file descriptor *my_fd;* the file pointer moves forward through the file a total of (32 x the number of virtual processors) bits, starting from bit 0.

```
position = CMFS_lseek(my_fd, 32, CMFS_L_SET);
/* my_fd had been previously declared as an int */
```

From Lisp/Paris, perform the 32-bit seek by calling **CMFS-lseek** as in the following example:

```
(CMFS:lseek my_fd 32 CMFS_L_SET) ; my_fd is bound to an open
                                 ; stream
```

5.1.2 Changing a File's Size

A CMFS truncate call changes the size of a file. It takes two arguments: a path name (or a file descriptor) and a positive integer indicating whether to shorten or lengthen the file.

- If the file was previously larger than *length*, the call shortens the file to the length specified and the extra data is lost.
- If the file was previously smaller than *length*, the call extends[3] the file to the length specified. The contents of the blocks allocated in the gap between the end of the file and the location of the file pointer are undefined. Use this technique to pre-allocate physical disk blocks as contiguously as possible.

Depending on both whether you are performing CM I/O or serial I/O and whether your file is open or closed, call one of the four CMFS truncate routines listed below:

- When performing CM I/O, call **`CMFS-truncate-file`** or **`CMFS-ftruncate-file`**. The *length* argument is measured in bits per processor (the call, therefore, must determine the number of virtual processors in the current VP set). **`CMFS-ftruncate-file`** requires that the file be open for writing, so its first argument is a file descriptor rather than a path name.
- When performing serial I/O, call **`CMFS-serial-truncate-file`** or **`CMFS-serial-ftruncate-file`**. The *length* argument is measured in bytes. **`CMFS-serial-ftruncate-file`** requires that the file be open for writing. Its first argument, therefore, is a file descriptor rather than a path name.

`CMFS-lseek` and **`CMFS-serial-lseek`** (Section 5.1.1) can also extend a file by moving the file pointer past the end of the file. The contents of the blocks allocated in the gap between the end of the file and the location of the file pointer are undefined.

3. Attribute files associated with DataVault files cannot be extended; likewise, neither attribute files nor data files that reside on a VAX front end can be extended.

5.2 CM I/O: Reading and Writing

5.2.1 Introduction to CM I/O

When the CM is client, the CM I/O routines that read or write can operate on any CMFS file[4] *whose VP geometry information matches the VP geometry of the current VP set.* (If the file's VP geometry information does not match the VP geometry of the current VP set, the call returns an error and the program may abort.) The only exceptions to this rule are *wildcard* files (see Appendix B) and special files, such as tapes containing either raw or archived data.

Performance Note

Although a CM can read from or write to a tape within the CM file system, it is usually inefficient to do so: since the CM operates at a much higher speed than the usual tape speed (generally less than 3 megabytes/second), it logs much idle time. Therefore, whenever an application requires using data from tape, first copy the data from the tape to a CMFS file, and then read it into the CM; whenever an application requires placing CM data on tape, first write the data into a regular CMFS file, and then copy it to tape.

CM I/O may require transposing—rearranging—the organization of the file's data elements from serial to parallel format. Specifically, if you read a serial file into the CM, you *must* transpose the data before operating on it, and you *may* have to perform some floating-point format conversion on it. Likewise, if you plan to use a parallel file on a serial computer, transpose the organization of the data elements from parallel to serial before writing it to a file and perform any necessary floating-point format conversion. See Chapter 6 for a discussion of these tasks.

4. CM I/O calls cannot directly access a UNIX file system. To do CM I/O using a file that resides in a UNIX file system, either copy the file into a CM file system using a CMFS copy command (see Chapter 2) or use the serial I/O calls described in Section 4.2.

You can do CM I/O by employing any of three types of I/O listed below:

- Use *synchronous* I/O in data applications that require one or a few large read or write operations per session. See Section 5.2.2.
- Use *streaming* I/O (a kind of asynchronous I/O) in data applications that require real-time processing of data. See Section 5.2.3.
- Use *buffered* I/O in data applications that require frequent non-real-time transfers of small or large amounts of data. See Section 5.2.4.

The main difference among these three CM I/O methods is the amount of overhead that each actual data transfer incurs. Figure 5 shows a time-scale comparison of these CM I/O methods.

5.2.2 Synchronous I/O

Synchronous I/O is optimized for performing infrequent large transfers from a CMFS file to the CM, and vice versa. Table 6 lists the calls used to perform synchronous I/O.

Table 6. Synchronous I/O calls.
Invoke the calls with the `CMFS-` prefix; for example, `CMFS-read-file`.

CMFS-...	Comments
`read-file`	Reads data into specified virtual processors.
`read-file-always`	Reads data into all virtual processors.
`write-file`	Writes data from specified virtual processors.
`write-file-always`	Writes data from all virtual processors.

These calls take three arguments: an open file descriptor; a field ID that specifies the location in each processor into which to read data, or from which to write data; and an integer that specifies the number of bits per processor to read or write.

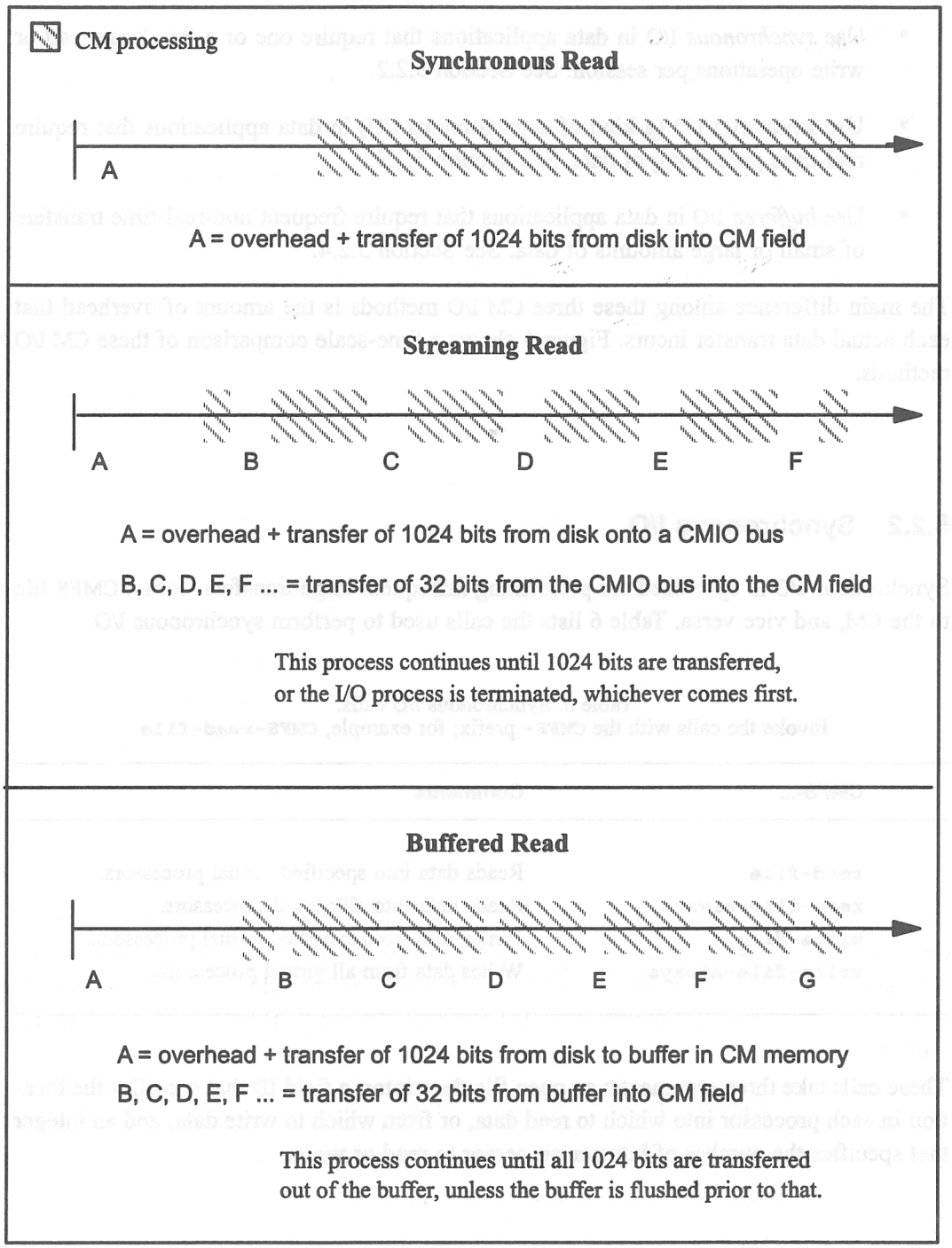

Figure 5. Time-scale comparison of methods of reading.
Time is represented horizontally.

These calls return one of the following values:

- the number of bits read or written per processor
- 0, which indicates that the end-of-file has been reached
- –1, which indicates failure

If the call fails, in addition to returning –1, it sets `CMFS-errno` to a number representing the cause of the error.

Chapter 8 contains a sample program that uses synchronous I/O.

5.2.3 Streaming I/O

Important

When doing streaming I/O, the current VP set geometry must match *both the file's VP geometry information and the file's physical width information.*

Streaming I/O (a kind of asynchronous I/O) provides a method of doing I/O for real-time applications. It eliminates interruptions caused by latency and overhead and enables CM processing to occur simultaneously with data transfer. Streaming I/O also provides a high sustained transfer rate for applications that use large amounts of data, such as database searches.

However, you should not use streaming I/O if you need to:

- Read or write *more than one* file during the I/O operation.
- Read *and* write a file during the I/O operation.
- Use the file server and/or its CMIO bus for another operation running concurrently. Streaming I/O is transparently disabled under timesharing, for example.

In essense, streaming I/O is one I/O transaction that consists of a large transfer followed by many small transfers that "stream" one after the other with no interrupting latency period. The operation performs all necessary handshaking and disk activity before it sends any data to its final destination: a CMIO bus and its associated buffers act as one "limitless" buffer, holding all data involved in the entire operation until the streaming transfers can occur uninterrupted.

Table 7 lists the streaming I/O calls in the order in which you must make them.

Table 7. Streaming I/O calls listed in required order.
Invoke the calls with the **CMFS-** prefix; for example, **CMFS-fcntl**.

	CMFS-...	Comments
1.	**CMFS-fcntl**	Activates streaming I/O for a particular file descriptor.
2.	**CMFS-read-file-always**	Reads entire amount of data from the file into buffer.
	CMFS-write-file-always	Writes entire amount of data from CM into buffer.
3.	**CMFS-streaming-iostat**	Returns the total number of bits per processor available to the forthcoming streaming calls.
4.	**CMFS-partial-read-file-always**	Transfers (relatively) small amount of data from buffer to CM.
	CMFS-partial-write-file-always	Transfers (relatively) small amount of data from buffer to file.
5.	**CMFS-streaming-iostat**	If no errors occur, returns the number of bits per virtual processor actually read or written during the subsequent transfers; if error occurs, returns –1

The following three calls set up the streaming I/O operation:

- **CMFS-fcntl** takes three arguments: an open file descriptor; **CMFS-F-SETFL** (a flag that specifies the operation); and **CMFS-FSTREAMING** (a flag that specifies a status flag). These flags are defined in **cm_file.h**. When successful, the call returns 0; when unsuccessful, the call returns –1 and sets **CMFS-errno** to a number representing the cause of the error.

- **CMFS-read|write-file-always** take three arguments: an open file descriptor; a field ID that specifies the location in each processor at which the data will eventually be placed (or from which to take immediately the data for placement into the buffer); and an integer that specifies the number of bits per processor that will be

read to each processor (or be taken immediately from each processor). These calls return –1 (and set **CMFS-errno**) to indicate failure; any other return value indicates success.

You cannot use **CMFS-read-file** and **CMFS-write-file** in streaming I/O operations.

- **CMFS-streaming-iostat** takes one argument: an open file descriptor. The **CMFS-streaming-iostat** entry in Table 7 lists the call's return values.

CMFS-partial-read-file-always and CMFS-partial-write-file-always do the actual buffer-to-processor or buffer-to-disk transfers. These calls take three arguments: an open file descriptor; a field ID that specifies the location in each processor at which to place the data (or from which the data came); and an integer that specifies the number of bits to be read into each processor (or that were written from each processor). When successful, these calls return the number of bits transferred per processor; when unsuccessful, these calls return –1 and set **CMFS-errno** to a number representing the cause of the error.

Invoke **CMFS-partial-read-file-always** or **CMFS-partial-write-file-always**[5] as many times as necessary to transfer all the data that is in the buffer. Call **CMFS-streaming-iostat** at the end of the entire streaming operation to verify the number of bits per processor actually transferred.

Chapter 8 contains a sample program that uses streaming I/O.

5.2.4 Buffered I/O

Although buffered I/O usually necessitates several short latency periods, its actual data transfers are very quick relative to both synchronous and streaming I/O. Therefore, use buffered I/O to increase the performance of an application doing many small, sequential non-real-time I/O operations. Buffered I/O, unlike streaming I/O, can read or write multiple files in a single I/O operation or perform both reads and writes using a single file.

Buffered I/O is a series of "I/O segments":

- A read segment consists of a relatively large disk-to-buffer read, followed by several small buffer-to-processors reads. If the buffer empties to the point that a

5. You cannot call both **CMFS-partial-read-file-always** and **CMFS-partial-write-file-always** within the same streaming operation.

buffer-to-processors read cannot complete, there is a latency period as disks work to refill the buffer so that another read segment can begin.

- A write segment consists of several relatively small processors-to-buffer writes followed by a large buffer-to-disk write. If the buffer fills to the point that a write cannot complete, there is a latency period as the buffer's data is written onto disks so that another write segment can begin.

The buffer must be flushed when switching between reading and writing segments.

Table 8 lists the buffered I/O calls in the order in which you must make them.

CMFS-setbuffer, which creates the buffer, takes three arguments: an open file descriptor, a CM field in the same VP set as the fields that will be used in subsequent buffered read and write calls, and an integer specifying the number of bits to be transferred into or out of buffer during the latency periods. For efficiency, the buffer should be an integral number of disk blocks. (You can determine the optimal block size by calling **CMFS-[f]stat** or executing **cmstat**.) **CMFS-setbuffer** returns 0 to indicate success or –1 to indicate failure (and sets **CMFS-errno**).

CMFS-buffered-read-file-always and **CMFS-buffered-write-file-always** take three arguments: an open file descriptor; a CM field to which or from which to transfer data; and an integer that specifies the number of bits to be transferred out of the buffer into the CM field. When successful, the calls return the number of bits read or written per processor; when unsuccessful, the calls return –1 and set **CMFS-errno**.

CMFS-flush takes one argument: an open file descriptor. The call returns 0 to indicate success or –1 (and sets **CMFS-errno**) to indicate failure.

Table 8. Buffered I/O calls listed in the required order.
Invoke the calls with the **CMFS-** prefix; for example, **CMFS-setbuffer**.

	Call CMFS-...	Comments
1.	**setbuffer**	Assigns a CM field to be the buffer.
2.	**buffered-read-file-always**	Performs initial disk-to-buffer read, subsequent buffer-to-processors reads, and all further necessary disk-to-buffer reads.
	buffered-write-file-always	Performs processors-to-buffer writes and subsequent buffer-to-disk writes.
3.	**flush**	Clears buffer when switching from reading to writing and vice-versa; before disk seeks, and when terminating buffered I/O.

5.3 Serial I/O: Reading and Writing

When a VMEIO host computer, CM-IOP, or front-end computer is the client, you can perform *serial I/O*—that is, read to or write from the client's memory—using the CMFS library's serial I/O calls listed in Table 9. Use serial I/O, for example, to:

- Run an application on a front end not attached to the CM if you want to put data from a front-end UNIX file into a CMFS file or vice versa.
- Run an application on a VMEIO host computer or on a CM-IOP if you want to place data from a tape into a CMFS file or vice versa (or place data from a framegrabber into a CMFS file).

Table 9. Calls that read and write data from a non-CM client.
Invoke the calls with the **`CMFS-`** prefix; for example, **`CMFS-serial-read-file`**.

CMFS-...	Comments
`serial-read-file`	Reads into the memory of a non-CM client.
`serial-write-file`	Writes from the memory of a non-CM client.

`CMFS-serial-read-file` and **`CMFS-serial-write-file`** take three arguments: an open file descriptor, a pointer to a buffer in the client's memory at which to place data or from which to write data, and an integer that specifies the number of bytes to read from the file or write into the file. If successful, these calls return the number of bytes read into or written from the client's memory. The calls return **`-1`** and set **`CMFS-errno`** if they fail.

5.4 Reading and Writing Attribute Files

Each CMFS file has an associated attribute file that contains information about it. The attribute file, named *.filename*`.attr`, is in the same directory as the file itself. The first 256 bytes of an attribute file are reserved for use by the CM file system.

You can use the rest of the attribute file to store any information you would like to record, back up, and keep in synchronization with the associated CMFS file; often this information is application-specific. Use the CMFS serial I/O calls, **`CMFS-serial-read`** and **`CMFS-serial-write`**, to read and write such information.

Chapter 6
Transposing Data

When sharing data between a CM and a serial computer, you must ensure that:

- the bits that make up each data element are in an order—either parallel or serial—appropriate to the computer using it. See Section 6.1.
- the floating-point numbers are in the correct byte ordering—either big-endian or little-endian—and in the floating-point format that the computer uses—either VAX format or IEEE format. See Section 6.2.

6.1 Transposing Data

In order for a serial computer to use a parallel file, you must transpose the data from parallel to serial format; the fastest way to do this is to transpose the data on the CM before writing it out to a file. Likewise, in order for a CM to use a serial file, you must transpose the data from serial to parallel format; the fastest way to do this is to transpose the data on the CM after you read it from the file.

To perform the transfer, use either **`CMFS-transpose-always`** or **`CMFS-transpose-record-always`**.

- **`CMFS-transpose-always`** transposes only a single data item in each processor at a time, so it is appropriate for a file comprised of single data items.
- **`CMFS-transpose-record-always`** is appropriate for a file comprised of structures.

These calls take the following arguments, in the order listed:

1. *field-id*: a field ID indicating the location in each processor of the data item to be transposed.

2. *length*: an integer indicating the length, in bits, of the data item to be transposed in each processor. For **CMFS-transpose-always**, *length* must be an integer power of 2 that is less than or equal to 256. For **CMFS-transpose-record-always**, *length* must be an integer multiple of 32.

3. *trans–type*: a flag indicating the kind of transposition: either from serial to parallel format—**CMFS-IO-READ-FROM-SERIAL-DATA**—or from parallel to serial format—**CMFS-IO-WRITE-TO-SERIAL-DATA**. These flags are defined in **<cm/cm_file.h>**.

4. *function* (for **CMFS-transpose-always** only): a flag indicating whether to use row-major or column-major arrangement:

   - If *trans-type* is **CMFS-IO-READ-FROM-SERIAL-DATA**, *function* is either **CMFS-read-from-row-major** or **CMFS-read-from-column-major**.
   - If *trans-type* is **CMFS-IO-WRITE-TO-SERIAL-DATA**, *function* is either **CMFS-write-to-row-major** or **CMFS-write-to-column-major**.

   These flags are defined in **<cm/cm_file.h>**.

The *function* argument is optional. If you do not supply a *function* argument, **CMFS-tranpose(-record)-always** places the transposed data in the CM processors in *send* order; that is, the transposed data elements are placed, one after the other, in sequentially ordered processors. Most CM applications use *NEWS* order, however. NEWS order does not necessarily place the data elements in sequentially ordered processors; rather, it places the data elements such that the processors' grid-like layout is exploited, providing much faster communication and, therefore, better application performance. For more information about send order and NEWS order, see the Chapter 2 of the *Paris Reference Manual.*

6.2 Converting Data Format

In order for a serial computer to use data generated on a CM, or for a CM to use data generated on a serial computer, all floating-point numbers must be either byte-swapped or format-converted, depending on whether the serial computer is a VAX or a Sun.

- VAX computers use VAX format floating-point numbers, while the CM uses IEEE format floating-point numbers. Depending on which type of computer will use the data next, call **`CMFS-convert-ieee-to-vax`** or **`CMFS-convert-vax-to-ieee`**:
  - Call **`CMFS-convert-ieee-to-vax`** before writing data from CM processors to a file.
  - Call **`CMFS-convert-vax-to-ieee`** after reading the data into the CM processors but before computing on it.
- Suns use "big-endian" (left-to-right) ordering of floating-point numbers. The CM uses "little-endian" (right-to-left) ordering. Depending on which type of computer will use the data next, therefore, byte-swap the data appropriately. It is easiest and fastest to do this by using Paris calls on the CM.

  For example, to byte-swap data from Sun ordering to CM ordering, create a routine something like this one:

```
if (SwapBytes == TRUE) {
   printf("Swapping bytes to CM byte order/n");
   CM_swap_2_1L(field_id,
                CM_add_offset_to_field_id(field_id, 24), 8)
   CM_SWAP_2_1L(CM_add_offset_to_field_id(field_id, 8),
                CM_add_offset_to_field_id(field_id, 16), 8);
}
```

  This routine assumes that the length of the data item in each processor is 32 bits.

Chapter 7

Running and Debugging Programs That Contain CM File System Code

7.1 Running Programs That Contain CM File System Code

This section contains information about compiling CMFS code and attaching to a CM in a way that will enable best system performance. Refer to the *CM User's Guide* for general information about performing these tasks.

7.1.1 Compiling CMFS Code

The CMFS software reserves the use of *CMFS_*, *CMIOC_* , *CMFS*, *CMFS-I*, *CMFS-LISP*, and *dv*. If used for any other purpose, these strings may produce compiler errors.

The following sections discuss language-specific compilation requirements.

C-Based Languages

Programs written in a C-based language that use the CM file system must be compiled with two libraries:

- *cmfs*, the CM file system library
- *paris*, the Paris library. Compiling with the paris library requires the math library, *m*, to be specified also. If your program performs only serial I/O, however, it is not necessary to link with the *paris* library.

The header file `<cm/cmfs.h>` contains function prototyping information (type declarations) for all CM file system operations. It is not required that this file be included

in C/Paris and C* programs that call CMFS routines. However, since it is used in the type checking performed by the C* compiler, C* programs that do not include it compile with warnings resulting from type checking.

Lisp-Based Languages

Programs written in Lisp that use the CM File System have no special compilation requirements. However, the *cmfs* library must be loaded. (The standard *Lisp band has the *cmfs* library, but the Paris band does not.)

7.1.2 Attaching to the CM

As discussed in the *CM User's Guide*, if you intend to run a program that uses CM I/O, you should attach to one or more CM sequencers that control at least one CMIOC. The CMFS routines called by your program are then able to access files residing on all data storage devices on the same CMIO bus as the CMIOC. (If the CMFS routines cannot access a data storage device via a CMIO bus, they use the Ethernet, which is slow compared to a CMIO bus.)

Therefore, before you attach to the CM, find out which sequencers control the CMIOCs that are on the same CMIO buses as the data storage devices to which your program should have CMIO bus access. Ask your site's system administrator or issue the **cmfinger** command. Among the information in the table **cmfinger** prints out is a line telling which sequencers contain CMIOCs, and which of these sequencers is free. For example,

```
% cmfinger
CM    Seqs Size   Front end   I/F   User     Idle     Command
-----------------------------------------------------------------

FOO   1    8K     wotan       0     karen    0h 06m   "cmattach"

                1024K memory size, 32-bit floating point
                framebuffers on sequencers 0 1 (seq 0 is free)
                CMIOCs on sequencers 0 1 (seq 0 is free)
          FOO has 1 free seq -- 0 -- totalling 8K procs
```

You can use the **-S** option to **cmattach** to specify that you want to attach to a particular sequencer or sequencers—valid combinations are 0–1, 2–3, and 0–3. Other methods of attaching to the CM are described in the *CM User's Guide*.

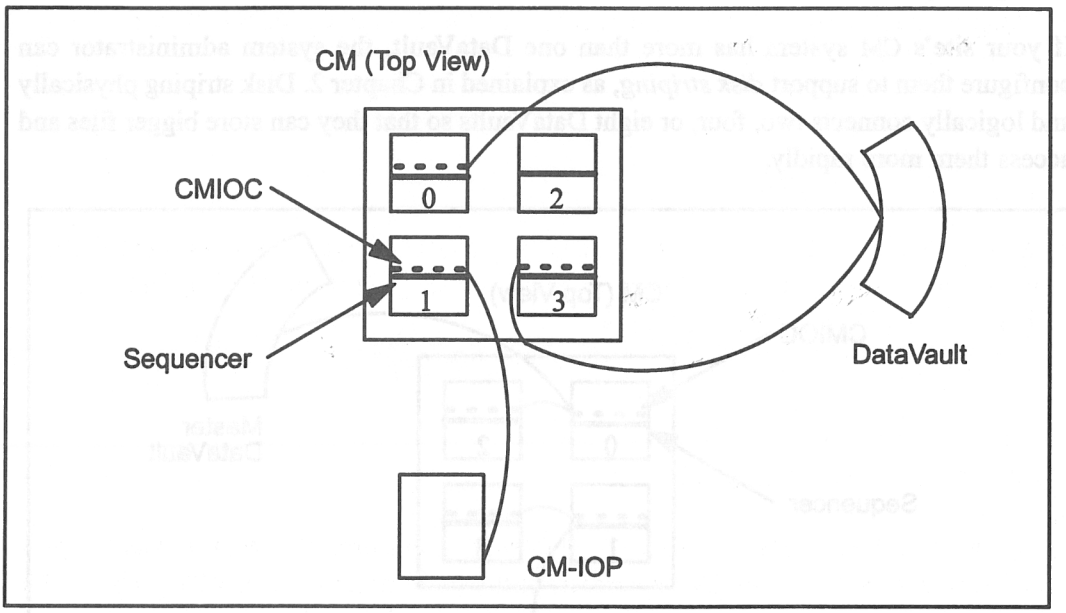

Figure 6. Top view of a 64K CM system with four sequencers. Sequencer 2 does not control a CMIOC, but the other sequencers do.

In a CM system like the one in Figure 4, for example, you can attach to one, two, or four sequencers, depending on the number of processors with which you want to work and on the data storage devices to which your program should have CMIO bus access.

- If you need CMIO bus access to the DataVault, attach to either sequencer 0, sequencer 3, sequencers 0 and 1, sequencers 2 and 3, or all four sequencers.
- If you need CMIO bus access to the CM-IOP, attach to either sequencer 1, sequencers 0 and 1, or all four sequencers.
- If you need CMIO bus access to both the DataVault and the CM-IOP, attach to either sequencers 0 and 1 or to all four sequencers.
- If you do not need access to either the DataVault or the CM-IOP, attach to sequencer 2 if possible so that you aren't needlessly tying up a sequencer that can access the CM File System.

Using Striped DataVaults

If your site's CM system has more than one DataVault, the system administrator can configure them to support *disk striping*, as explained in Chapter 2. Disk striping physically and logically connects two, four, or eight DataVaults so that they can store bigger files and access them more rapidly.

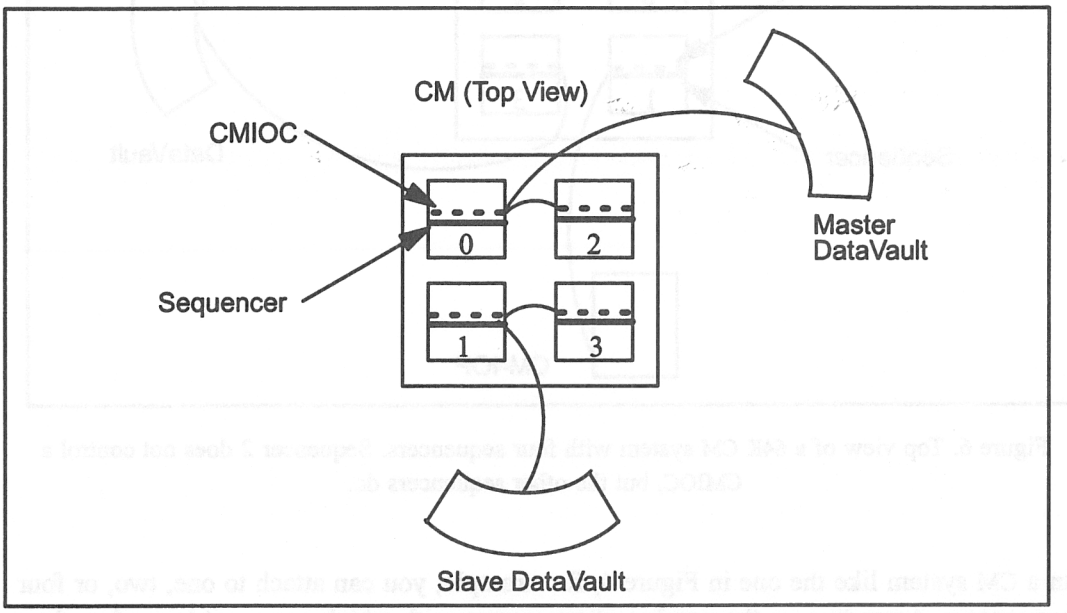

Figure 7. A set of two striped DataVaults.

If your program reads and/or writes a file that resides on striped DataVaults, your program must have CMIO bus access to all the DataVaults within the striped set to obtain good performance. Therefore, attach to a *set* of sequencers that provides CMIO bus access to every DataVault in the striped set. For example, to access a file on the set of striped DataVaults in Figure 7, attach to either sequencers 0 and 1, sequencers 2 and 3, or to all four sequencers. (An operation that only requires access to the file server—listing directory entries, for example—requires access only to the master DataVault.)

7.2 Debugging CMFS Code

When a CMFS library call does not complete successfully, it returns a special value (usually –1) to the application program and sets `CMFS-errno` to indicate the specific error. If the call was made using the C/Paris interface, simply use the library call `CMFS-perror` to

print the error message corresponding to the value of **CMFS-errno**. (See the **CMFS-errno** manual page in Appendix D for a list of error numbers and their corresponding descriptions.)

If the call was made using the Lisp/Paris interface, however, by default a Lisp error condition is signalled and the specific error condition is displayed by the debugger. The following Lisp function, for example, determines whether a file exists in the CM file system:

```
(defun probe-cmfs-file (name)
   "Return t if the named file exists; otherwise, return nil"
   (let ((stat-struct (CMFS:make-stat)))
      (= 0 (cmfs:stat name stat-struct))))
```

If *name* is a non existent file, the program enters the debugger, which provides the following error information:

```
> (probe-cmfs-file "12345")
>>Error: Error in CMFS function CMFS-STAT: "No such file or
directory"

CMI::CMFS-STAT:
   Required arg 0 (PATH): "12345"
   Required arg 1 (CM_STAT): #<Simple-Vector (UNSIGNED-BYTE 32)
      32 2E86626
:C  0: Proceed anyway
:A  1: Abort to Lisp Top Level
```

If you do not want to enter the debugger on an error, you can set the **CMFS:*enter-debugger-on-error*** variable to **nil** and print the error via **CMFS:perror** (as in the C/Paris interface). For example, the above Lisp function will not enter the debugger if you revise it as follows:

```
(defun probe-cmfs-file (name)
   "Return t if the named file exists; otherwise, return nil"
   (let ((stat-struct (CMFS:make-stat))
         (CMFS:*enter-debugger-on-error* nil))
      (declare (special CMFS:*enter-debugger-on-error*))
      (= 0 (CMFS:stat name stat-struct))))
```

7.2.1 Automatic CMFS Debugging Messages

You can activate and deactivate the printing of automatic CMFS debugging messages by setting an environment variable or by calling a CMFS routine. The environment variable is not available under Genera Lisp; the CMFS routine is available under all environments.

Environment Variable

The environment variable **CMFS_DEBUG** activates the printing of each CMFS library call's arguments and return value, which lets you know if a call is receiving an invalid argument or producing an unexpected return value.

If you are using a C shell, activate automatic CMFS debugging messages by typing

```
% setenv CMFS_DEBUG 1
```

If you are using a Bourne shell, activate automatic CMFS debugging messages by typing

```
$ CMFS_DEBUG=1
$ export CMFS_DEBUG
```

For convenience, you can have your **.login** file or your **.cshrc** file activate automatic CMFS debugging messages each time you open a shell to work in the CM programming environment. Deactivate automatic CMFS debugging messages by setting **CMFS_DEBUG** to 0.

To set **CMFS_DEBUG** under Lucid Lisp running on a front end that is either a Sun or a VAX, you must first kill Lisp. **CMFS_DEBUG** is not available under Genera Lisp.

CMFS Call

Although you cannot use **CMFS_DEBUG** under Genera, you can call the CMFS library routine **CMFS-set-debug-mode** from within your Genera application program. (Like most CMFS calls, **CMFS-set-debug-mode** is available under all supported CM programming languages.) **CMFS-set-debug-mode** allows you to explicitly control the portions of code for which CMFS debugging message are printed: if *val*, its only argument, is non-zero, CMFS debugging messages are printed until the routine is called again with a *val* argument of 0.

Chapter 8

Sample Programs

This chapter includes includes three programs:

- **createFile**, in Section 8.1.1, uses serial I/O to transfer data from a tape drive connected to a CM-IOP or VMEIO host computer to a CMFS file.
- **imageTranspose**, in Section 8.1.2, uses synchronous CM I/O to transfer data from a serial CMFS file into the CM.
- **showMovie**, in Section 8.2, uses streaming I/O to transfer data in real time from a parallel CMFS file to a framebuffer.

8.1 Synchronous I/O Programming Example

The program **imageTranspose**, listed in Section 8.1.2, is an image-processing[6] application. It uses data comprised of 1,024 32-bit images collected by a satellite (or other field equipment) and placed on magnetic tape in blocks of 2K bytes (not in a **tar** archive). Before running **imageTranspose**, this data is placed in a CMFS file, as described in Section 8.1.1.

imageTranspose reads the data from the file into a VP set consisting of 1024 x 1024 processors, transposes the data from serial to parallel format and, if necessary, byte-swaps

6. Image processing is a data-intensive application well suited to the CM system. In this sample program, the CM is configured as a two-dimensional grid of processors, and an image is represented by a 1024 x 1024 grid of processors. Each processor represents a pixel; each pixel's color is described by a 32-bit number. To enhance the image, calculations are performed simultaneously within the processors. For more information about image processing, see the *Connection Machine Graphics Programming* manual.

it.[7] Finally, **imageTranspose** creates a new CMFS file and writes the data to it in parallel format. (If **imageTranspose** were part of a real application, the program would perform an image enhancement before writing the new file; this step, however, is beyond the scope of this manual, so we've simply inserted a comment in the code where the image enhancement would take place.)

8.1.1 Placing the Raw Data Into a CMFS File

Executing cmdd

To take the raw data off of the tape, place the tape on a tape drive connected to a CM-IOP or VMEIO host computer and execute **cmdd** on the drive's host machine. For example, type:

```
% cmdd -todv if=/dev/tape of=dv1:/acct5/imagedata ibs=2048
obs=16384
```

The **ibs=** argument is the block size of the tape, and the **obs=** argument is the block size of the DataVault. (If you'd rather execute **cmdd** on a front-end computer, include the **-fromdv** argument and specify the **if=** argument as **vme1:/dev/tape**.)

If your site does not have a VMEIO host computer or a CM-IOP, you can place the tape on a drive connected to a front-end computer and execute **cmdd** there. The data transfer occurs slowly, however, because the Ethernet is used rather than a CMIO bus.

Writing a Program

Although executing **cmdd** is the most straightforward method of placing the raw data into a CMFS file, you can also write a program to do it. For example, **createFile** (listed on the next page) uses UNIX calls to open the tape device and to read the data into the host's memory. Then it uses CMFS serial I/O calls to move the data from the host's memory to a CMFS file, where **imageTranspose** can quickly access it.

For best performance, compile and run **createFile** on the host machine of the tape drive. **createFile** accepts two arguments—the raw data's path name, and the name to give to the CMFS file. A sample compilation and the program listing follows.

---

7. The CM and VAX computers uses little-endian ordering. If the data's byte ordering is big-endian, which Suns and IBM machines use, the data must be byte-swapped

```
% cc -o createFile createFile.c -lcmfs
% createFile vme1:/dev/tape dv1:/acct5/imagedata

/*                              createFile                                  */
/* This program reads raw data into the tape drive's host's memory          */
/*  and then writes it into a CMFS file using serial CM I/O.                */
/* You do NOT need to be attached to the CM to run this program--           */
/* it does not use the CM at all.                                           */
/* Compile: cc -o createFile createFile.c -lcmfs                            */

#include <stdio.h>
#include <cm/cm_file.h>

#define   ROWS 1024
#define   COLUMNS 1024
#define   INPUT_FILE av[1]
#define   OUTPUT_FILE av[2]
#define   BLOCK_SIZE 2048

main(ac, av)
char **av;
int ac;
{
    char   buffer[BLOCK_SIZE];
    int    tape_fd, file_fd;
    int    bytes_read, bytes_written;

    if (ac != 2) {
       printf("usage: %s input-file, output-file\n", av[0]);
       exit(1);
    }

/*                  Create a new CMFS file                                  */
    file_fd = CMFS_creat(OUTPUT_FILE, 0666);
    if (file_fd < 0) {
       CMFS_perror("OUTPUT_FILE error");
       exit(1);
    }

/*                          Open the tape device                            */
    tape_fd = open(INPUT_FILE, RDONLY);
    if (tape_fd < 0) {
```

```
        sprintf(ebuf, "Cannot open tape device %s", INPUT_FILE);
        CMFS_close(file_fd);
        exit(1);
    }

printf("Reading and writing the data into the CMFS file from %s\n",\
        OUTPUT_FILE);

/*                      Loop until end of tape                      */
    while ((bytes_read = read(tape_fd, buffer, BLOCK_SIZE)) > 0) {
/*                         Check for errors                         */
        if (bytes_read != BLOCK_SIZE) {
        if (0<bytes_read && bytes_read<BLOCK_SIZE)
            printf("short read from %s: expected %d bits, got %d\
                    bytes\n",INPUT_FILE, BLOCK_SIZE, bytes_read);
        else if (bytes_read = 0) {
            printf("End of file on %s\n", INPUT_FILE);
            break;
        }
/*              Write the data to the CM filesystem.                */
        if ((bytes_written = CMFS_serial_write_file(OUTPUT_FILE,
            buffer, bytes_read))) != sizeof (bytes_read)) {
            CMFS_perror("write error");
            break;
        }
    } /* end of while */

        if (bytes_read < 0) {
            sprintf(ebuf, "Error reading %s", INPUT_FILE);
            perror("read error");
        }
        close(tape_fd);
        CMFS_close(file_fd);
        exit(1)

}
```

8.1.2 Compiling and Executing imageTranspose

Compile **imageTranspose** with the CMFS, Paris, and math libraries, as shown below.

```
% cc -o imageTranspose imageTranspose.c -lcmfs -lparis -lm
```

When executed, **imageTranspose** accepts three arguments:

- The first argument provides the path name of the CMFS file—in this example, **dv1:/acct5/imagedata**.
- The second argument is the name to give to the new CMFS file in which to save the transposed, "image-enhanced" data.
- The third argument is optional: if the data needs to be byte-swapped after it is transposed, run the program with **-swap-bytes** as the third argument.

The procedure shown below explicitly attaches to sequencers 0 and 1—supposing that sequencer 0 or 1 (or both) contains a CMIOC that is the interface to the CMIO bus connected to the DataVault on which our CMFS file resides—and executes the program, listed on the following pages.

```
% cmattach -S0-1
Attaching to the Connection Machine system HAAGEN-DAZS... cold
booting... done.
Attached to 16384 processors on sequencers 0 and 1, microcode
version 6002
Paris safety is off.

Entering CMATTACH subshell. Type "exit" or control-D to detach
the CM...
%
% imageTranspose dv1:/acct5/imagedata dv1:/acct5/newdata
Calling CM_init()
Reading CM file test/testfile
Transposing data from serial to parallel format
All processors have expected data
Saving parallel data in test/output-file
Closing DataVault files
%
```

```
/*                        imageTranpose                                  */
/* This program reads raw data (in serial format) from a CMFS file into   */
/* the CM. Then it transposes the data into a two-dimensional geometry,   */
/* and save out the parallel-format data to a CMFS file.                  */
/*  usage: program [-swap-bytes] input-file output-file                   */
/* compilation: cc -o imageTranspose imageTranspose.c -lcmfs-lparis -lm    */

#include        <stdio.h>
#include        <sys/types.h>
#include        <cm/paris.h>
#include        <cm/cm_param.h>
#include        <cm/cm_file.h>

#define RANK     2  /* The number of dimensions: 2 in this example */
#define ROWS     1024
#define COLUMNS  1024
#define LENGTH   32 /* The length of the data item in each processor*/

int dim_array[RANK] = {ROWS,COLUMNS};

#define INPUT_FILE av[1]
#define OUTPUT_FILE av[2]
#define TRUE 1
#define FALSE 0

#ifdef max
#undef max
#endif
#define max(x, y) ((x > y) ? (x) : (y))

void
main(ac, av)
   int ac;
   char *av[];
{
   int              image_fd, enhanced_image_fd;
   int              bits_read, bits_written, number_right;
   CM_field_id_t    pixel_field_id;
   CM_geometry_id_t image_geometry_id;
   CM_vp_set_id_t   image_vp_set_id;
   char             ebuf[100];
   int              SwapBytes;
```

```
    if (ac < 3 || ac > 4 ) {
        printf("usage: %s [-swap-bytes] input-file output-file\n",\
                av[0]);
        exit(1);
    }

    if (strcmp(av[1], "-swap-bytes") == 0) {
        SwapBytes = TRUE;
        av++, ac--;                    /* Skip to the next argument */
    }

/*                    Set up for CM access                          */

    printf("Calling CM_init()\n");
    CM_init();

/*                  Create a CM geometry of 1024 x 1024 VPs         */

    image_geometry_id = CM_create_geometry(dim_array, RANK);
    image_vp_set_id = CM_allocate_vp_set(image_geometry_id);
    CM_set_vp_set(image_vp_set_id);

/*                  Allocate fields in CM memory                    */
    pixel_field_id = CM_allocate_stack_field(LENGTH);

/*                  Select all processors                           */
    CM_set_context();

/* Open and read the serial file containing image data into the CM */
    image_fd = CMFS_open(INPUT_FILE, CMFS_O_RDONLY);
    if (image_fd < 0) {
        sprintf(ebuf, "Cannot open CM file %s", INPUT_FILE);
        CMFS_perror(ebuf);
        exit(1);
    }

    printf("Reading CM file %s\n", INPUT_FILE);
    bits_read = CMFS_read_file_always(image_fd, pixel_field_id, LENGTH);

/*                        Check for errors                          */
    if (bits_read != LENGTH) {
```

```
        if (0<bits_read && bits_read<LENGTH)
            printf("short read from %s: expected %d bits, got %d\n",\
                    INPUT_FILE, LENGTH, bits_read);
        else if (bits_read = 0)
            printf("End of file on %s\n", INPUT_FILE);
        else if (bits_read < 0) {
            sprintf(ebuf, "Error reading %s", INPUT_FILE);
            CMFS_perror(ebuf);
        }
    }

/* Transpose the image from serial to parallel format, arranging it*/
/* in CM processors as a 1024 x 1024 grid.                         */
    printf("Transposing data from serial to parallel format\n");
    CMFS_transpose_always(pixel_field_id, LENGTH,\
        CMFS_IO_READ_FROM_SERIAL_DATA, serial_to_parallel_rearrangement);

/* Byte-swap the data if necessary. The SwapBytes flag specifies   */
/* whether the user wants the data byte-swapped. The following code*/
/* assume that LENGTH == 32 and that the data currently in the CM  */
/* is in "big-endian" format.                                      */
    if (SwapBytes == TRUE) {
        printf("Swapping bytes to CM byte order\n");
        CM_swap_2_1L(pixel_field_id,
                     CM_add_offset_to_field_id(pixel_field_id, 24), 8);
        CM_swap_2_1L(CM_add_offset_to_field_id(pixel_field_id, 8),
                     CM_add_offset_to_field_id(pixel_field_id, 16), 8);
    }

/*                    PERFORM AN IMAGE ENHANCEMENT                 */
/*           Code to manipulate the image would be here            */

/*                    Save the enhanced image                      */
    enhanced_image_fd = CMFS_open(OUTPUT_FILE,\
                        CMFS_O_CREAT|CMFS_O_WRONLY|CMFS_O_TRUNC,0666);
    if (enhanced_image_fd < 0) {
        sprintf(ebuf, "Cannot create output file %s", OUTPUT_FILE);
        CMFS_perror(ebuf);
        exit(1);
    }
    printf("Saving parallel data in %s\n", OUTPUT_FILE);
    bits_written = CMFS_write_file_always(enhanced_image_fd,\
                                    pixel_field_id, LENGTH);
```

```
/*                          Check for errors                          */
    if (bits_written != LENGTH) {
        if (0<bits_written && bits_written<LENGTH)
            printf("short write to %s: expected %d bits,\
                    wrote %d\n", OUTPUT_FILE, LENGTH, bits_written);
        else if (bits_written = 0)
            printf("End of file on %s\n", OUTPUT_FILE);
        else if (bits_written < 0) {
            sprintf(ebuf, "Error writing %s", OUTPUT_FILE);
            CMFS_perror(ebuf);
        }
    }

/*                       Close the CMFS files                         */
    printf("Closing DataVault files\n");
    CMFS_close(image_fd);
    CMFS_close(enhanced_image_fd);

}
```

8.2 Streaming (Asynchronous) I/O Programming Example

This sample program—a real-time application—uses streaming I/O to read a file containing movie frames into the CM and to then display them on a framebuffer.

```
/*                              showMovie                             */
/* This sample program illustrates streaming I/O. The CMFS file contains   */
/* 10,000 frames. Each frame is represented by a 2D grid of pixels where   */
/* each pixel is assigned to a CM processor. There are 8 bits per pixel,   */
/* which determine what the pixel displays. Using streaming I/O, the data  */
/* in the CMFS file is read into the CM and displayed on the graphics       */
/* display. (The functions that perform the display operations,             */
/* setup_framebuffer and framebuffer_write, are not shown in detail. Error */
/* checking has also been intentionally omitted for clarity.)               */

#define BITS_PER_IMAGE  8
#define IMAGES          10000

show_movie()
{
```

```
    int          fd;
    in           i;
    int          return_value;
    int          number_of_images_in_file;
    field_id_t   movie_field_id;

/* Open the CMFS file, "movie," in dv1:/images                    */
    fd = CMFS_open("dv1:/images/movie", CMFS_O_RDONLY);

/* Activate streaming I/O for the file.                           */
    CMFS_fcntl(fd, CMFS_F_SETFL, CMFS_FSTREAMING);

/* Initiate data transfer from the file to the CM. Since streaming */
/* I/O is activated for the file descriptor, this call only       */
/* performs I/O set-up procedures, not the actual data transfer.  */
    CMFS_read_file_always(fd, movie_field_id, IMAGES * BITS_PER_IMAGE);

/* Execution of the call below overlaps with the I/O set-up above */
    setup_framebuffer();

/* Check the initial return message to verify the total amount of */
/* data that will be transferred.                                 */
    return_value1 = CMFS_streaming_iostat(fd);

/* Compute the number of images in the file.                      */
    number_of_images_in_file = return_value/BITS_PER_IMAGE;

/* Transfer the images, one at a time, into CM memory and display */
/* display them on the graphics display.                          */
    for (i=0;  i<number_of_images_in_file;  i++)
    {
       /* Transfer one image into CM memory. */
       CMFS_partial_read_file_always(fd, movie_field_id, BITS_PER_IMAGE);
       /* Display the image. */
       framebuffer_write(movie_field_id, BITS_PER_IMAGE);
    }

/* Check the final return message to verify that all the data     */
/* was written to the CM (and, therefore, to the framebuffer).    */
    return_value2 = CMFS_streaming_iostat(fd);
    if (return_value2 != return_value1)
       printf("Some of the data did not get to the framebuffer\n");
    else printf("All of the data made it to the framebuffer");

/*                  Close the CMFS file.                          */
    CMFS_close(fd);
}
```

Part II
Advanced Programming Topics

Part II

Advanced Programming Topics

Chapter 9

Getting the Best Performance from CM I/O

This chapter offers guidelines for obtaining good performance from the CM I/O system. Issues addressed include CM–file geometry compatibility, CMIO bus access, reading and writing methods, and optimal file storage.

9.1 CM–File Geometry Compatibility

CM I/O requires only that the VP geometry of the current VP set match the file's VP geometry information. Although CM I/O can proceed if the file's physical width differs from the number of physical processors in the current VP set, the difference in physical geometries may necessitate extensive routing and communication, resulting in poor application performance.

If you intend to use a file in a VP set with a different number of physical processors than the one in which the file was created, you can improve application performance by "changing" the file's physical geometry information:

1. Attach to the CM and create the VP set in which you want to be able to use the file.
2. Use **dvcp** to copy the file. The copy of the file has a physical width that matches the number of physical processors in the current VP set and a VP geometry that matches that of the original file.

You can then read the new file into the current VP set and use it in CM I/O operations.

9.2 CMIO Bus Access

As discussed in previous chapters, a CM I/O operation should have access to the files it uses via a CMIO bus. However, there may be occasions when you must run an application program on processors that do not have a CMIOC available to them. If your site's CM system has a front-end computer that is also a VMEIO host computer, however, the processors can use the VMEIO board as a pseudo-CMIOC.

The VMEIO board serves as the interface between the front end's VMEbus and a CMIO bus, thereby enabling CMIO bus access to a data storage device on that bus. If a CM I/O operation that does not have an appropriate CMIOC available needs to access a file residing on a device connected to that bus, the operation automatically uses the bus's VMEIO interface: in a CM I/O write operation, for example, data travels through the H-bus to the front end's VMEIO board, where it accesses a CMIO bus connected to the device on which the file resides. See Figure 8.

Although accessing a CMIO bus by this method is slower than accessing a CMIO bus directly from a CMIOC, it is faster than doing CM I/O via the Ethernet.

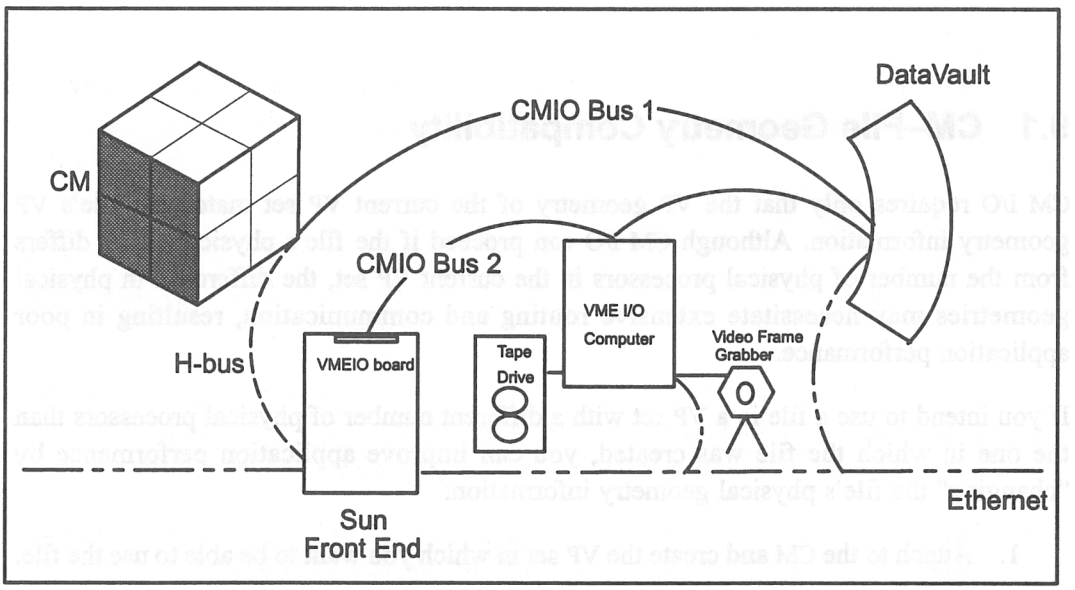

Figure 8. A CM system that has a Sun front end with a VMEIO board installed. If there are no CMIOCs available to provide access to CMIO bus 1, CM I/O proceeds through the VMEIO board to CMIO bus 2.

9.3 Using CM I/O Commands and Calls Efficiently

The CM I/O system design attempts to optimize the performance of read, write, and other I/O operations, since they are often performed several times per session. The following list offers hints for taking advantage of the optimization efforts.

9.3.1 Reading and Writing Files Most Efficiently

- Call **`CMFS-open`** or **`CMFS-creat`** only once per file per session. They are the slowest CM file system operations.

- Prior to writing a file, call **`CMFS-[f]truncate`** to extend it to a length longer than its current length. This allocates physical disk blocks so that they are as contiguous as possible.

- When doing synchronous CM I/O, whenever possible use **`CMFS-read-file-always`** and **`CMFS-write-file-always`** rather than **`CMFS-read-file`** and **`CMFS-write-file`**.

  Although it might seem that the conditional versions of the read and write operations would be faster or use less disk space if not all the processors are selected for an operation, the unconditional versions are faster in the current implementation.

  The performance of both the conditional and unconditional versions of these calls is dominated by the seek time on the 32 DataVault data disks for relatively small transfers and by the disk transfer rate for relatively large transfers.

- Use buffered I/O to increase the performance of a program doing many small, sequential (but non-real-time) I/O transactions to a file. Actual calls to the file server are only done whenever the buffer is full (write) or empty (read).

- Use streaming I/O to accomplish a kind of asynchronous I/O. Streaming I/O allows CM processing time to overlap with I/O communication and startup overhead, and it allows a match between the high transfer rate of CM-to-CMIOC transfers and the lower transfer rate of various I/O devices.

- If you need to read or write several fields in CM memory, allocate the fields as one large field, perform a single read or write on the large field, and then split the large field into small subfields again using the Paris call **`CM-add-offset-to-field-id`**. This is faster than performing a read or write for each individual field.

9.3.2 Copying Files Most Efficiently

- The CMFS utilities **cmtar**, **cmdd**, **copytodv**, and **copyfromdv** run much faster on a VMEIO host computer or CM-IOP than they do on a front-end computer that does not host a VMEIO board.

- When using **cmdd** or **cmtar** to copy data to or from the DataVault, specify a block size argument compatible with the DataVault's block size of 16K bytes. For example, **cmdd**'s **bs=** argument set to **1600k** helps to ensure good performance when reading raw data from a tape into a DataVault.

9.4 Choosing a Data Storage Device

In general:

- For fastest CM-to-file access, store large files on a DataVault. Striped DataVaults, each accessed by a separate CMIO bus, provide even faster access.

- Store smaller files accessed often by non-CM front-end code on the front end CMFS directory tree.

- If a file will be accessed by both the CM and a serial computer on another system, store the file in a CMFS directory tree residing on a VMEIO host computer or a CM-IOP. Files stored on these devices are transferred to the CM faster than files stored on a front end's CMFS directory tree and can be easily **cmtar**'ed to tape for transfer to another computer system.

Chapter 10

CM Character-Special Files

The CM file system views I/O devices such as tape drives, as CM character-special files. In a CM system, special files reside on a VMEIO host computer or a CM-IOP. The file server running on these devices uses CM character-special device drivers to manage the special files—that is, to run the I/O devices.

So that you can expand your CM system's I/O capabilities by adding devices to a VMEIO host computer or a CM-IOP, this chapter provides information on how to write and implement a CM character-special device driver.

Section 10.5 contains a sample device driver called `tape.c`.

10.1 The CMFS Library–Device Driver Interface

CM application programs request the use of a device via the same CMFS calls that act on regular files. To request access to an I/O device, for example, a user program calls **`CMFS-open`**. The file server process, **`fsserver`**, that runs on the VMEIO host computer[8] or CM-IOP interprets the call. **`fsserver`** uses the name of the device—indicated by **`CMFS-open`**'s *path* argument—to derive the device's inode (see Section 10.3.3). The inode specifies the device's major device number and minor device number. **`fsserver`** stores this information in memory so that it can quickly reference it when subsequent CMFS calls use the special file.

8. Since I/O devices are connected to the CM I/O system via a VMEIO host computer or a CM-IOP, the **`fsserver`** process that runs on the VMEIO host computer or CM-IOP is responsible for interpreting all calls to the I/O device. If the CMFS directory tree residing on the VMEIO host computer or CM-IOP is not the default, the *hostname:* component of **`CMFS-open`**'s *path* argument should be the name of the VMEIO host computer or CM-IOP (as given in the front ends' **`/etc/hosts`** table). Defaulting rules are listed in the **`fsserver`** man page.

The major device number is used to index the device switch cm_cdevsw, which contains one structure for each device configured into the **fsserver** process running on the VMEIO host computer or CM-IOP. The major device number, *n*, indexes the *n*th structure in the switch. That structure lists the pertinent device's function entry points. The entry points enable **fsserver** to call the driver routine corresponding to the CMFS library call.

Driver routines are the interfaces between **fsserver** and the device. The **fsserver** process running on the VMEIO host computer or CM-IOP translates the parameters of CMFS calls into parameters that are passed to a corresponding I/O routine of the device's driver. The device driver routine carries out the I/O request, and passes control back to the **fsserver** process.

10.2 Writing a Device Driver

The heart of a device driver consists of the routines that perform the operations requested via the CMFS library. This section provides templates for the routines, along with information about the arguments passed from **fsserver** to the driver.

Include the following files: **<sys/types.h>**, **<sys/ioctl.h>**, **"cm_ioctl.h"**, **"cm_param.h"**, **"cm_errno.h"**, **"cm_conf.h"**, **"cm_file.h"**, and any additional header files specific to your device.

You will not need the CM to test the device driver if you read and write to it using **CMFS_serial_read** and **CMFS_serial_write** calls.

10.2.1 The fsserver–Device Driver Interface: Routines

A typical **fsserver** call to a driver routine has the following format:

```
*cm_cdevsw(major_device_number).d_xxfile-operation(parameters);
```

parameters will have been translated from the parameters of the CMFS call into parameters reflecting the actual run-time environment of the driver. The types of all parameters used by drivers are defined in **cm_conf.h**. (See Section 10.4.1 for the contents of **cm_conf.h**.)

The first parameter of every required routine, *dev*, encodes the major and minor device numbers of the device. *dev* is of type dev_t, an unsigned short integer. Its high byte

contains the major device number, and its low byte contains the minor device number. The standard UNIX macros **major(dev)**, **minor(dev)**, and **makedev(major, min)**, defined on a Sun in **<sys/sysmacros.h>**,[9] can be used to encode and decode dev_t's.

Parameters subsequent to *dev* are routine-dependent.

Each driver must contain routines that handle open, close, read, write, and ioctl operations. The declarations of these routines are listed below. When implementing these routines, replace the *xx* placeholder with the name of your device. For example, a driver for a device named *mtape* would contain an **xxopen()** routine implemented as **mtapeopen()**.

xxopen(), xxclose()

```
xxopen (dev, flags)                xxclose (dev, flags)
dev_t dev;                         dev_t dev;
int flags;                         int flags;
```

fsserver calls the **xxopen()** routine to open a device for reading or writing. The **xxopen()** routine sets up the device and initializes data structures that the driver can use. (**fsserver** does not call the **xxopen()** routine on stat operations.) **fsserver** calls the **xxclose()** routine to close a CM character-special file after all file descriptors associated with the device are closed.

The *flags* argument is the same as that specified in the **CMFS_open()** library call.

These routines return 0 to indicate success; a returned error number indicates failure. (Error numbers are defined in **<cm/cm_errno.h>**.)

xxsize()

```
xxsize (dev, count)
dev_t dev;
int count;
```

The **xxsize()** routine allows the **fsserver** to advise the CM of an impending short read. (The CM I/O system architecture requires that all data must arrive once the CM has been programmed to receive it.)

The *count* argument indicates the number of bytes requested.

---

9. **<sys/sysmacros.h>** is included by **<sys/types.h>**.

The **xxsize()** routine must return the number of bytes that the driver is prepared to transfer to the CM in the next **xxread()** call.

xxread(), xxwrite()

```
xxread (dev, uio)                    xxwrite (dev, uio)
dev_t dev;                           dev_t dev;
struct cm_uio *uio;                  struct cm_uio *uio;
```

fsserver will call the **xxread()** routine in response to a **CMFS-read-file(-always)** or a **CMFS-serial-read-file()** call from the client program. **fsserver** will call the **xxwrite()** routine in response to a **CMFS-write-file(-always)** or a **CMFS-serial-write-file-always** call.

The *uio* argument is a pointer to a struct cm_uio. The cm_uio structure is defined in **<cm/cm_conf.h>**.

```
struct cm_uio {
    int uio_resid;          /* amount of data requested          */
    int uio_min;            /* minimum transfer size             */
    int uio_max;            /* maximum transfer size             */
    char *uio_buf;          /* buffer allotted for driver use    */
    int uio_bufsz;          /* size of the above in bytes        */
    int (*uio_xfr) ();      /* routine to do the transfer        */
};
```

The struct cm_uio that *uio* points to contains information on the transfer to be performed:

- The uio_resid field contains the number of bytes to be transferred to or from the device.
- The uio_min and uio_max fields contain the minimum and maximum number of bytes that can be transferred in a single call to the uio_xfr routine.
- The uio_buf field is a pointer to a buffer that the driver can use in performing the data transfer. For many applications, use of this buffer will improve the overall data transfer rate between the **fsserver** and the CM. In systems with a Thinking Machines Corporation proprietary VMEIO interface board, the buffer will be within the VMEIO interface's on-board memory rather than within the system's main memory.
- The uio_bufsz field contains the size of the uio_buf field in bytes.

- The uio_xfr field is a pointer to a function that transfers the data between the CM and **fsserver**. Depending on whether the data will be moved over a CMIO bus or over the Ethernet, the function calls either the VMEIO host computer's (or CM-IOP's) device driver or the Ethernet's device driver.

  xxread() and ***xxwrite()*** must call the uio_xfr routine as many times as necessary to satisfy the data transfer request. The arguments to the uio_xfr routine are (*uio*, *addr*, *length*), where *uio* is the pointer to the struct cm_uio passed to the driver, *addr* is a pointer to the data to be transferred, and *length* is the number of bytes to be transferred.

The ***xxread()*** routine *must* send the amount of data requested by the uio_resid field; this number will not be larger than the value returned by the ***xxsize()*** routine described above.The ***xxwrite()*** routine must accept the amount of data requested by the uio_resid field, even if it must simply throw it away.

xxread() and ***xxwrite()*** should return 0 for successful termination; otherwise, they return one of the error numbers defined in **<cm/cm_errno.h>**.

xxioctl()

```
xxioctl (dev, cmd, arg, need_to_swap)
dev_t dev;
int cmd, need_to_swap;
caddr_t arg;
```

fsserver calls the **xxioctl()** routine to perform a device-related task in response to the application program that calls **CMFS-ioctl** on a file descriptor associated with the device.

The *cmd* argument passed to the driver is the value passed by the user as the *cmd* argument to **CMFS-ioctl**. If the *arg* argument is to be used to pass data to the **xxioctl()** routine or to retrieve a result from the **xxioctl()** routine, the *cmd* argument must encode the size and direction of the transfer referred to by *arg*. This encoding is necessary for the **fsserver** and CMFS library to successfully transfer the data to which *arg* refers.

The encoding of *cmd* is done as follows: The top 4 bits encode the transfer direction information; the next 12 bits indicate the size of the data referred to by *arg*. The low 16 bits are ignored by the **fsserver** and CMFS library—they can be used to select a specific function within the **xxioctl()** routine. (See Figure 9.)

The encoding of this size and direction information is best done by using the macros in **<cm/cm_ioctl.h>** (Section 10.4.3 contains the code for **cm_ioctl.h**):

- `_CMFS_IO (top 8 bits of low 16 bits, low 8 bits)`
- `_CMFS_IOR (top 8 bits of low 16 bits, low 8 bits, type)`
- `_CMFS_IOW (top 8 bits of low 16 bits, low 8 bits, type)`
- `_CMFS_IOWR (top 8 bits of low 16 bits, low 8 bits, type)`

need_to_swap is an indication to the driver that the byte ordering of the **fsserver** and CMFS library differ from each other.

The **xxioctl()** routine returns 0 for successful termination; otherwise, it returns one of the error numbers defined in **<cm/cm_errno.h>**.

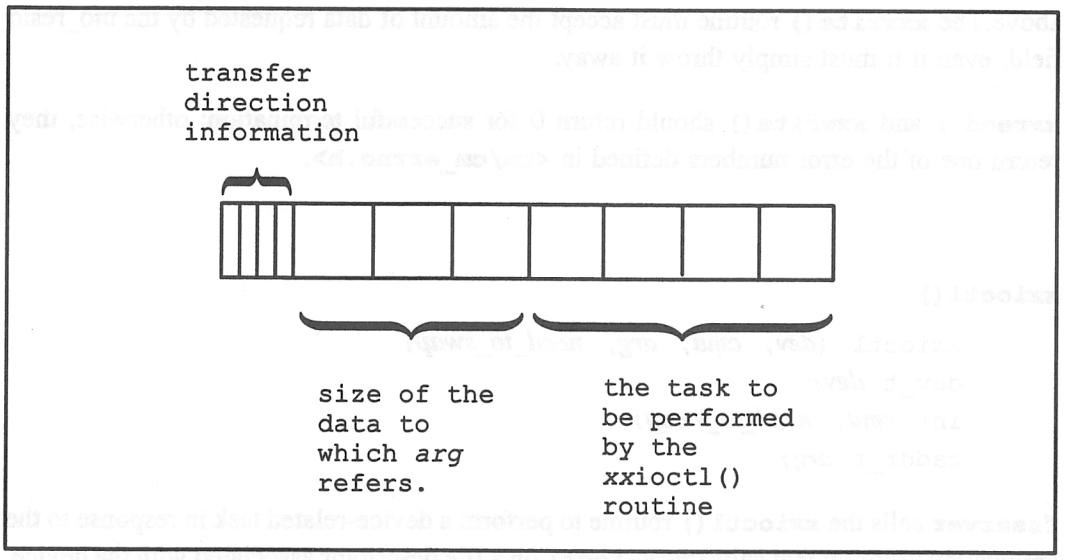

Figure 9. The ioctl int.

Available Support Routines

Several stub routines have been defined that may be useful to you.

```
nulldev (dev, flags)
dev_t dev; int flags;
```

nulldev() is a useful stub for open, close, or reset routines. It always returns 0, and does nothing.

```
nodev (dev, flags)
dev_t dev; int flags;
```

nodev() always returns **CMFS_ENODEV**.

```
zerosize (dev, count)
dev_t dev; int count;
```

zerosize() is a useful routine for write-only devices. Use it as a **xxsize()** routine that always returns 0.

```
countsize(dev, count)
dev_t dev; int count;
```

countsize() always returns *count*.

```
nullread (dev, uio)
dev_t dev; struct cm_uio *uio;
```

Use **nullread()** with **zerosize()** to always return *no data available*.

```
nullwrite (dev, uio)
dev_t dev; struct cm_uio *uio;
```

nullwrite(), useful for read-only devices, throws away all data written from the CM.

```
nullioctl (dev, cmd, arg, flag)
dev_t dev; int cmd, flag;
caddr_t arg;
```

nullioctl() is useful when there are no ioctl functions to perform. It always returns 0.

```
noioctl (dev, cmd, arg, flag)
dev_t dev; int cmd, flag;
caddr_t arg;
```

noioctl() always returns **CMFS_ENOTTY** for any value of *cmd*.

Although the cm_conf structure defines a **select()** and **reset()** routine slot, these are not currently used by the CMFS file server.

- The **reset()** routine tells the driver to terminate all current opens and return the software to an idle state.
- The **select()** routine determines whether a driver is ready to participate in an I/O operation.

10.3 Testing and Using a Device Driver

To use the device driver you have written, add it to the **fsserver** running on the VMEIO host computer or CM-IOP: add the driver's header files to **fsserver**'s **/h** subdirectory, add the files containing the driver's source code to **fsserver**'s **/fs** subdirectory, and edit the **fsserver**'s **/Makefile**. Then complete the steps listed in the following sections.

10.3.1 Add a Structure to the Device Switch

Add a structure representing the driver to the end of the device switch cm_cdevsw, in **fs/cm_conf.c** (see Section 10.4.2 for a sample **fs/cm_conf.c**). Below is a structure for a driver for a device called *tape*:

```
{
    tapeopen, tapeclose, taperead, tapesize,
    tapewrite, tapeioctl, nulldev, seltrue
    /* major device number                    */
};
```

Your driver's major device number will be equal to the number of structures in the device switch *before* you added your structure—the structures are numbered starting with zero.

Remember to declare the device driver's entry points in **/cm_conf.c**. For example,

```
extern int  tapeopen(), tapeclose(), taperead(), tapesize(),
            tapewrite(), tapeioctl();
```

10.3.2 Recompile fsserver and Start It

Recompile **fsserver** by typing **make fsserver**.

To restart the modified file server, type

```
./fsserver /directory-name
```

where *directory-name* is the root of the CMFS directory tree that resides on the VMEIO host computer or CM-IOP.

You may want to run the modified **fsserver** in the debugger *dbx*, at least at first.

10.3.3 Execute cmmknod

Execute the **cmmknod** command to make a special device entry in the CM file system directory tree served by your "new" **fsserver**. Type the following:

```
% cmmknod filename c major-device-number minor-device-number
```

where *filename* is the name by which the CM file system knows your device. Include the *hostname*: component of the path name—the name of the computer the device is connected to, as given in that computer's **/etc/hosts** table (for example, **vme1:/dev/new-device**).

Obtain the *major-device-number* from the device switch cm_cdevsw (see Section 10.3.1): if the structure you added is the *n*th one in the device switch, *major-device-number* is *n*–1 (structures in the switch are numbered from 0).

minor-device-number can specify the device's unit number, drive number, line number, and the like. If more than one minor device number is associated with the device, execute **cmmknod** as many times as necessary, specifying one minor device number at each execution. (The minor device number is not used by the CMFS library or by **fsserver**; it is passed to the driver via the *dev* argument to **CMFS-ioctl**, and can be used as you see fit.)

10.4 Files Used by a CM Character-Special Device Driver

10.4.1 cm_conf.h

```
/* CM Character device switch.*/

struct cm_cdevsw {
     int     (*d_open)();       /* dev_t dev, int flags          */
     int     (*d_close)();      /* dev_t dev, int flags          */
     int     (*d_read)();       /* dev_t dev, struct cm_uio *uio */
     int     (*d_readsize)();   /* u_long count                  */
     int     (*d_write)();      /* dev_t dev, struct cm_uio *uio */
     int     (*d_ioctl)();      /* dev_t dev, int cmd, caddr_t data,
                                   int need_to_swap              */
     int     (*d_reset)();      /* dev_t dev, int flags          */
     int     (*d_select)();     /* dev_t dev, int rw             */
};

struct cm_uio
     int     uio_resid;         /* amount of data requested      */
     int     uio_min;           /* minimum transfer size         */
     int     uio_max;           /* maximum transfer size         */
     char    *uio_buf;          /* buffer available for driver
                                                      use        */
     int     uio_bufsz;         /* size of the above in bytes    */
     int     (*uio_xfr)();      /* routine to do the transfer    */
};

/*
 * the arguments to the uio_xfr routine are
 * (*uio_xfr) (addr, length);
 */

#ifdef FILE_SERVER
extern struct cm_cdevsw cm_cdevsw[];
extern int nchrdev;
#endif
```

10.4.2 cm_conf.c

```
#include "cm_conf.h"

/*
How the device interface works:

When the file server receives an OPEN message for a file that is flagged as a CM
character-special device, the open routine listed in the cm_cdevsw structure for
that device number is called, and the result returned to the calling program. OPEN
returns zero for success, the appropriate CMFS_ERRNO for failures.

The same applies for the CLOSE routine.

The READ routine is expected to transfer data from the device and the CM. It may
call uio->uio_xfr as many times as necessary to deliver the data. The READ routine
MAY return a short count; it adjusts the uio_resid field to reflect the amount of
data actually transferred to the CM. READ returns zero for success, the appropri-
ate CMFS_ERRNO for failure.

The WRITE routine is expected to accept data from the CM and transfer it to the
device. It may call uio->uio_xfr as many times as necessary to read in the data.
The WRITE routine MUST actually read the requested amount of data from the CM.
WRITE returns zero for success, the appropriate CMFS_ERRNO for failure.

The IOCTL routine is for accepting a transparent message from the user.

The RESET routine is used to inform the driver that any current OPENS should be
terminated and the software returned to an idle state.

The SELECT routine is used to determine if a driver is ready to perform I/O.

*/

extern int nulldev();       /* always returns true                  */
extern int nullread();      /* returns zero bytes read              */
extern int nullwrite();     /* tosses data                          */
extern int fillread();      /* fills                                */
extern int noioctl();       /* IOCTL routine that always returns
                               "error"                              */
extern int nullioctl();     /* IOCTL routine that always returns
                               "success"                            */
extern int nodev();         /* Always returns error (CMFS_ENXIO)*/
extern int seltrue();       /* Select routine that always returns
                               "ready"                              */
extern int zerosize();      /* returns zero bytes ready to be
                               read                                 */
extern int countsize();     /* returns argument                     */
```

```
extern int tapeopen(), tapeclose(), taperead(), tapesize(),
           tapewrite(), tapeioctl();

struct cm_cdevsw cm_cdevsw[] =

/*open,             close,            read,             readsize,
  write,            ioctl,            reset,            select
*/
{
    {
         nodev,    nodev,    nullread,zerosize,          /* 0 */
         nullwrite,noioctl,nodev,    seltrue
    },
    {
         nulldev, nulldev, nullread,zerosize,          /* 1 */
         nullwrite,noioctl,nulldev, seltrue
    },
    {
         nulldev, nulldev, fillread,countsize,         /* 2 */
         nullwrite,nullioctl,nulldev,seltrue
    },
    {
         tapeopen,tapeclose,taperead,tapesize,         /* 3 */
         tapewrite,tapeioctl,nulldev,seltrue
    },
};

int nchrdev = sizeof (cm_cdevsw) / sizeof (cm_cdevsw[0]);
```

10.4.3 cm_ioctl.h

```
#ifndef _CMFS_IO

/*
Ioctl's have the command encoded in the lower word, and the size of any in or out
parameters in the upper word.  The high 2 bits of the upper word are used to
encode the in/out status of the parameter; for now we restrict parameters to at
most 4095 bytes.
The IOC_VOID field of 0x20000000 is defined so that new ioctls can be distin-
guished from old ioctls.
The CM file system library requires that ioctl's use this convention for encoding
length, since the parameters are really sent over the network to a remote
```

```
fileserver for interpretation.
*/

#define CMFS_IOCPARM_MASK 0xfff /* parameters must be < 256
                                   bytes                    */
#define CMFS_IOC_VOID 0x20000000  /* no parameters          */
#define CMFS_IOC_OUT  0x40000000  /* copy out parameters    */
#define CMFS_IOC_IN  0x80000000   /* copy in parameters     */
#define CMFS_IOC_INOUT (CMFS_IOC_IN|CMFS_IOC_OUT)

#define  _CMFS_IO(x,y)      (CMFS_IOC_VOID|('x'<<8)|y)
#define  _CMFS_IOR(x,y,t)   (CMFS_IOC_OUT|((sizeof(t)&
                            CMFS_IOCPARM_MASK)<<16)|('x'<<8)|y)
#define  _CMFS_IORN(x,y,t)  (CMFS_IOC_OUT|(((t)&
                            CMFS_IOCPARM_MASK)<<16)|('x'<<8)|y)
#define  _CMFS_IOW(x,y,t)   (CMFS_IOC_IN|((sizeof(t)&
                            CMFS_IOCPARM_MASK)<<16)|('x'<<8)|y)
#define  _CMFS_IOWN(x,y,t)  (CMFS_IOC_IN|(((t)&
                            CMFS_IOCPARM_MASK)<<16)|('x'<<8)|y)

/* this should be _IORW, but stdio got there first */
#define _CMFS_IOWR(x,y,t)   (CMFS_IOC_INOUT|((sizeof(t)&
                            CMFS_IOCPARM_MASK)<<16)|('x'<<8)|y)
#endif /* _CMFS_IO */
```

10.5 tape.c (A Sample Driver)

```
#include <sys/types.h>
#include <sys/mtio.h>
#include <sys/ioctl.h>

#include "cm_ioctl.h"
#include "cm_mtio.h"
#include "cm_param.h"
#include "cm_errno.h"
#include "cm_conf.h"
#include "cm_file.h"

/* Tape drive device driver for the CM file system */

#define T_NOREWIND      0x04
#define T_HIGH_DENSITY  0x08
```

```
#define NTAPE 4

#define swapshort(x)    (((x<<8)|((x>>8)&0xff))&0xffff)
#define swaplong(x)     (swapshort(x<<16)|swapshort(x>>16))

struct cmfs_tape_info {
    int tape_fd;
    int tape_flags;
    int tape_bufsz;             /* buffer size to use for tape I/O */
} cmfs_tape_info[NTAPE];

/* Flags */
#define OPEN 1
#define AT_EOF 2

#define TAPE_BUFSZ (10*1024)

extern int errno;

int tapeopen(dev, flags)
dev_t dev;
int flags;
{
    int unit;
    char unix_dev[80];
    int ret;
    struct cmfs_tape_info *cti;

    unit = minor(dev) & 0x3;
    cti = &cmfs_tape_info[unit];

#ifdef sun
    sprintf(unix_dev, "/dev/%srst%d",
            minor(dev) & T_NOREWIND ? "n" : "",
            minor(dev) & T_HIGH_DENSITY ? unit + 8 : unit);
#endif
#ifdef vax
    sprintf(unix_dev, "/dev/%srmt%d%s",
            minor(dev) & T_NOREWIND ? "n" : "",

 minor(dev) & T_HIGH_DENSITY ? unit + 8 : unit);
#endif
#ifdef vax
    sprintf(unix_dev, "/dev/%srmt%d%s",
            minor(dev) & T_NOREWIND ? "n" : "",
            unit,
            minor(dev) & T_HIGH_DENSITY ? "h" : "l");
#endif
```

```
    flags = flags & (CMFS_O_RDONLY|CMFS_O_WRONLY|CMFS_O_RDWR);
    cti->tape_fd = open(unix_dev, flags);

    if (cti->tape_fd < 0)
        return(errno);

    cti->tape_flags = OPEN;
    cti->tape_bufsz = TAPE_BUFSZ;

    return (0);
}

int tapeclose(dev, flags)
dev_t dev;
int flags;
{
    int unit;
    struct cmfs_tape_info *cti;

    unit = minor(dev) & 0x3;
    cti = &cmfs_tape_info[unit];

    cti->tape_flags = 0;
    return (close(cti->tape_fd));
}

int taperead(dev, uio)
dev_t dev;
struct cm_uio *uio;
{
    int unit;
    struct cmfs_tape_info *cti;

  int bytes_read, size;
    int ret, err;

    unit = minor(dev) & 0x3;
    cti = &cmfs_tape_info[unit];

    if (!(cti->tape_flags&OPEN))
        return (CMFS_EINVAL);

    /*
     * Read from the tape and write to the I/O system.
     */
    ret = 0;
```

```
    while (uio->uio_resid > 0) {
      size = min(uio->uio_bufsz, uio->uio_resid);
        if (cti->tape_flags & AT_EOF) {
            /* Can't read anymore, send back zeros */
            bzero(uio->uio_buf, size);
            bytes_read = size;
        } else {
            bytes_read = read(cti->tape_fd, uio->uio_buf, size);
            if (bytes_read == 0) {
                cti->tape_flags |= AT_EOF;
                bzero(uio->uio_buf, size);
            } else if (bytes_read < 0) {
                cti->tape_flags |= AT_EOF;
                ret = errno;
               bzero(uio->uio_buf, size);
            }
        }

        if (err = (*uio->uio_xfr)(uio, uio->uio_buf, bytes_read)) {
            return(err);
        }
        uio->uio_resid -= bytes_read;
    }
    return ret;
}

int tapewrite(dev, uio)
dev_t dev;
struct cm_uio *uio;
{
    int unit;
    struct cmfs_tape_info *cti;
    int size, bytes_written, ret;
    int blocks, bytes_left, err;
    char *p;

    unit = minor(dev) & 0x3;
    cti = &cmfs_tape_info[unit];

    if (!(cti->tape_flags&OPEN))
        return (CMFS_EINVAL);

    ret = 0;
    while (uio->uio_resid > 0) {
        /*
         * Write the tape with a blocksize of cti->tape_bufsz
         */
```

```
        size = min(uio->uio_bufsz, uio->uio_resid);
        blocks = size / cti->tape_bufsz;
        bytes_left = size % cti->tape_bufsz;
        if ((blocks > 0) && bytes_left && (size > bytes_left))
            size -= bytes_left;
        p = uio->uio_buf;
        if (err = (*uio->uio_xfr)(uio, p, size)) {
            return(err);
        }
        while (blocks-- > 0) {
            if (!(cti->tape_flags & AT_EOF)) {
                bytes_written = write(cti->tape_fd, p, cti->tape_bufsz);
                if (bytes_written < 0) {
                    cti->tape_flags |= AT_EOF;
                    ret = errno;
                }
            }
            p += cti->tape_bufsz;
            uio->uio_resid -= cti->tape_bufsz;

    }
        if (bytes_left > 0) {
            if (!(cti->tape_flags & AT_EOF)) {
                bytes_written = write(cti->tape_fd, p, bytes_left);
                if (bytes_written < 0) {
                    cti->tape_flags |= AT_EOF;
                    ret = errno;
                }
            }
            uio->uio_resid -= bytes_left;
        }
    }
    return ret;
}

int
tapesize(dev, count)
dev_t dev;
int count;
{
    int unit;
    struct cmfs_tape_info *cti;

    unit = minor(dev) & 0x3;
    cti = &cmfs_tape_info[unit];
```

```
    if (!(cti->tape_flags&OPEN))
        return (0);

    if (cti->tape_flags & AT_EOF)
        return(0);

    return (count);
}

int tapeioctl(dev, cmd, data, need_to_swap)
dev_t dev;
int cmd, need_to_swap;
caddr_t data;
{

 int unit;
    struct cmfs_tape_info *cti;
    int ret = 0;

    unit = minor(dev) & 0x3;
    cti = &cmfs_tape_info[unit];

    if (!(cti->tape_flags&OPEN))
        return (CMFS_EINVAL);

    switch(cmd) {
    case CMMTIOCTOP:
        {
            struct cm_mtop *mop = (struct cm_mtop *)data;

            if (need_to_swap)
                swap_mtop(mop);
            if (ioctl(cti->tape_fd, MTIOCTOP, mop) < 0)
                ret = errno;
            if (need_to_swap)
                swap_mtop(mop);
        nn}
        break;

   case CMMTIOCGET:
        {
            struct cm_mtget *mg = (struct cm_mtget *)data;

            if (ioctl(cti->tape_fd, MTIOCGET, mg) < 0)
                ret = errno;
            if (need_to_swap)
                swap_mtget(mg);
```

```
        }
        break;

    case CMMTIOCSETREC:
        {
            int reclen;

            reclen = *(int *) data;

  if (need_to_swap)
                reclen = swaplong(reclen);

            cti->tape_bufsz = reclen;
        }
        break;

    default:
        ret = CMFS_ENOTTY;
        break;
    }

    return ret;
}

static
swap_mtop(mop)
struct cm_mtop *mop;
{
    mop->mt_op = swapshort(mop->mt_op);
    mop->mt_op = swaplong(mop->mt_count);
}

static
swap_mtget(mg)
struct cm_mtget *mg;
{
    mg->mt_type = swapshort(mg->mt_type);
    mg->mt_dsreg = swapshort(mg->mt_dsreg);
    mg->mt_erreg = swapshort(mg->mt_erreg);
    mg->mt_resid = swapshort(mg->mt_resid);
}
```