Chapter 8

Tape I/O

This chapter provides general information about tapes and tape-handling terminology and explains how to run the CMOST utilities for handling both UNIX-format tapes and IBM-format tapes:

- **`cmtar`** and its command-line interface utilities, **`cmdump`** and **`cmrestore`**, archive to tape and restore files. See Section 8.2.
- **`cmdd`** copies files to or from tape, performing any specified format conversion (such as EBCDIC ↔ ASCII). See Section 8.3.
- **`ibmtape`** imports and exports IBM data sets and handles tape labels. See Section 8.4.

IMPORTANT

In a CM-5 system, the tape drive itself may be connected to a VMEIO host computer, a CM-IOPG, a control processor, or an Integrated Tape System (ITS).

The utilities described in this chapter are not supported on the ITS. Their ITS counterparts are described in the *ITS User's Guide*.

Copyright © 1994 Thinking Machines Corporation

8.1 Using Tapes: An Overview

UNIX-type files (that is, regular UNIX files plus UNIX-compatible or UNIX-like files, such as SFS or CMFS files, respectively) have no format — they are simply raw sequences of bytes existing on a disk or tape, or other media. An application that uses a UNIX file is responsible for intepreting the file in an appropriate way. For instance, a text editor would interpret a file as sequences of bytes terminated by newlines, while a database program might interpret a file as a series of fixed-length records.

On an IBM system, however, files are treated differently. The IBM operating system (typically MVS) handles most of the user's file-processing needs. It allows the user to select from a variety of common file formats and allows a user's application to read and write these files without requiring a lot of code written for a specific file format.

Interestingly enough, when handling tapes more of the structure of the process is visible to the IBM user than to the UNIX user. Here are a few terms used by both the IBM and the UNIX communities:

- *Tape file*

  A *tape file* is a sequence of records (IBM) or bytes (UNIX) on a tape, terminated by a tape mark (defined below).

  Do not confuse a tape file with a data file. For instance, a whole UNIX directory structure can be archived into one tape file using a **`tar`** program, and individual UNIX data files can be extracted later. Or, a single UNIX file or IBM data set may span multiple tape files.

- *Tape mark*

  A *tape mark* is simply an indicator written on the medium by the drive (and unreadable by the user) at the end of the file, separating that file from the beginning of a next one. The drive can read the tape marks, allowing it to find a tape file's beginning. Two tape marks are typically written to indicate the end of the last file on the tape.

  There are two other special marks on the tape: the BOT marker and the EOT marker, physical indicators designating the tape's beginning and end. An application is notified of an EOT condition on reading or writing. More data may then be written, but the consequences could be dire — on older round tapes, the tape would frequently fall off the reel and require the operator to reload the tape.

Copyright © 1994 Thinking Machines Corporation

- *Tape block*

  A *tape block* is the smallest unit of data that may be addressed on a tape. There may or may not be a one-to-one mapping between records in a data set and blocks on a tape; for performance enhancement, tape blocks are usually much larger than records. If a block is indeed larger than a record, the data set records are packed into the tape block in some logical fashion. (Tape blocks are sometimes referred to as tape records, in which case the records of an IBM data set are called logical records and the tape blocks themselves are referred to as physical records.)

  A drive can position itself on a specific block on the tape by performing FSR (forward space record) or BSR (backward space record) operations. In addition, most tape drives support a high-speed access method called fixed-block mode, which allows collections of similar-sized records to be extracted at once. The normal mode of operation is variable-block mode.

All the preceding terms describe attributes that are common to all tapes, whether they originate from a UNIX system or an IBM system. They are also common to all tape styles, such as 9-track (reel-to-reel), QIC (cartridge), 3480 (IBM square tapes), or EXABYTE (8mm video).

8.2 Creating and Restoring Tape Archives: The cmdump and cmrestore Commands

The **cmdump** and **cmrestore** commands are executed from the command line and provide interactive user interfaces to the **cmtar**[9] program, which both creates tape archives of files and restores them to a CM-5 file system. These commands are easy to use and suitable for most archiving applications, supporting the **-c**, **-x**, **-b**, **-v**, **-f**, **-M**, and **-V** options to **cmtar**. (Execute **cmtar** directly to use the options not supported by **cmdump** and

9. **cmtar** is derived from the GNU tar program, which was written by John Gilmore and modified by many people, and whose GNU enhancements were written by Jay Fenlason. The GNU programs are made available by the Free Software Foundation.

Copyright © 1994 Thinking Machines Corporation

cmrestore — the ones that invoke concatenation and archive comparison, for example. See the **cmtar** man page in the Chapter 12.)

- Use **cmdump** to back up files to tape. **cmdump** provides a robust interface that allows users to browse the current CM-5 file system and mark the files to be backed up. It is easy to mark an entire tree for backup, for example, or to search through a tree for filenames that meet certain criteria. Each file is archived with its relative pathname only; that is, the leading *hostname* **:/** portion of the path is stripped off before archiving.

  As when using **cmtar** or UNIX **tar**, you must have read permission on each file and directory that you archive. Archives created by **cmdump** cannot be restored using UNIX **tar**. They can, however, be restored using GNU tar.

- Use **cmrestore** to extract files from the tape archive and place them into the current CM-5 file system. When executed, **cmrestore** builds a tree representation of the archive's directory information and then provides an interface that lets you browse the tree and mark the files to be extracted. Files are extracted relative to the directory that is current when **cmrestore** is invoked.

  As when using **cmtar** or UNIX **tar**, you must be owner of the extracted files, and you must have write permission on the directory to which you restore them. Also, if a file with the same name as an extracted file already exists in the directory, you must have write permission on that file, as it will be overwritten.

Both **cmdump** and **cmrestore** by default automatically handle the any attribute files associated with the CMFS files on which they operate. Both commands also support shell-like globbing (wildcarding), Emacs-style command-line editing, command history, and selectable output pagination. Table 10 shows the Emacs-style editing and command history key sequences.

To run **cmdump** or **cmrestore**, log in to a VMEIO host computer, a CM-IOPG, or a CM-5 control processor that is networked both to the machine that hosts the relevant CM-5 file system and to the tape drive.

Set the environment variables **CMFS_PATHTYPE**, **DVHOSTNAME** and **DVWD** for the appropriate CM-5 file system before invoking the **cmdump** or **cmrestore** command: once you have invoked the command, the environment variables are unavailable. Although you can change the current working directory within **cmdump** and **cmrestore** (see "Command Descriptions," page 80), you cannot change the CM-5 file system to which the directory is relative; it is

Copyright © 1994 Thinking Machines Corporation

particularly important, therefore, to set the current CM-5 file system appropriately before invoking **cmdump** and **cmrestore**.

To invoke **cmdump** or **cmrestore**, simply type the program name, along with any flags, at the shell prompt. The flags are listed below.

-a Runs **cmdump** and **restore** under "explicit mode," turning off the default "automatic CMFS attribute file handling mode." Under automatic attribute handling mode, any attribute files associated with marked CMFS files are automatically dumped or restored. Explicit mode is not recommended for general use.

-b *number* Sets the archive's blocking factor. The default blocking factor of a Sun-4 is 3200; for a VAX, it is 20. **-b** is required for **cmrestore** if the archive is not blocked in the default record size.

-f *filename* Specifies the archive file. The default for a Sun-4 is **/dev/rstc0**. For a VAX, the default is **/dev/rmt0h**. Multiple archives are supported; see the **cmtar** manual page in Chapter 12.

-n Disables Emacs-style editing and history. If input is from a tty, Emacs-style editing and history is enabled by default. If input is *not* from a tty, editing and history are disabled by default.

-s Disables output pagination. If output is going to a tty (the default), output pagination is enabled by default. If output is not to a tty, output pagination is disabled by default.

-v Enables verbose mode. By default, verbose mode is disabled.

-V *vol−name* Specifies a volume name for the archive. By default, no volume name is given. The **-V** flag must be used on **cmrestore** if the archive was created with a volume name.

At the resulting prompt, type an appropriate sequence of commands. The commands supported by **cmdump** and **cmrestore** are listed in "Command Descriptions," page 80.

Copyright © 1994 Thinking Machines Corporation

Table 10. cmdump and cmrestore Emacs-mode command-line editing and command history key sequences (based on the GNU Readline Library).

Key Sequence	Use
Newline, Return	Accepts the line regardless of the cursor's location.
Delete	Deletes the character to the left of the cursor.
Ctrl-A	Moves to the beginning of the line.
Ctrl-B	Moves back one character.
Ctrl-D	Deletes the character beneath the cursor.
Ctrl-E	Moves to the end of the line.
Ctrl-F	Moves forward one character.
Ctrl-G	Quits the operation in progress.
Ctrl-K	Kills the text from the cursor to the end of the line.
Ctrl-L	Clears the screen, reprinting the current line at the top.
Ctrl-Q	Inserts next character typed verbatim.
Ctrl-T	Transposes the character before the cursor and the one underneath it.
Ctrl-U	Kills backwards to the beginning of the line.
Ctrl-W	Kills from the cursor to the previous whitespace.
Ctrl-Y	Yanks the most recently killed text to the cursor position.
Ctrl-_	Undoes the last change. Undoing multiple changes is possible.
Meta-Delete	Kills from the cursor to the start of the previous word.
Meta-B	Moves backward one word.
Meta-C	Capitalizes the current (or following) word.
Meta-D	Kills from the cursor to the end of the current word.
Meta-F	Moves forward one word.
Meta-L	Lowercases the current (or following) word.
Meta-R	Undoes all changes made to this line.
Meta-T	Transposes the word behind the cursor and the one in front of the cursor.
Meta-U	Uppercases the current (or following) word.
Meta-Y	Rotates the kill ring and yanks the new top. Works only if the prior command was Ctrl-Y or Meta-Y.
Meta-Ctrl-j	Toggles between Emacs and Vi editing modes.
Ctrl-P	Moves up through the history list.
Ctrl-N	Moves down through the history list.
Meta-<	Moves to the first line in the history list.
Meta->	Moves to end of input history (end of the line you're currently entering).
Ctrl-R	Searches backward, incrementally, starting at the current line, moving up through the history list as needed.
Ctrl-S	Searches forward, incrementally, starting at the current line and moving down through the history list as needed.

Copyright © 1994 Thinking Machines Corporation

A sample **cmdump** command sequence is given below, followed by a **cmrestore** command sequence.

```
% setenv DVHOSTNAME dv1
% setenv DVWD /big_project
% cmdump
Using /dev/rstc0 for archive
Current directory is dv1:/big_project
cmdump% ls
june4                    june3                june2
june1                    may31                may 30
jiml                sean                 test
cmdump% mark -r sean -n may30 may31
cmdump% display
dv1:/big_project/
dv1:/big_project/.may30.attr
dv1:/big_project/.may31.attr
dv1:/big_project/may30
dv1:/big_project/may31
dv1:/big_project/sean/
dv1:/big_project/sean/.abc.attr
dv1:/big_project/sean/.tree.attr
dv1:/big_project/sean/abc
dv1:/big_project/sean/tree
cmdump% unmark may31
cmdump% display
dv1:/big_project/
dv1:/big_project/.may30.attr
dv1:/big_project/may30
dv1:/big_project/sean/
dv1:/big_project/sean/.abc.attr
dv1:/big_project/sean/.tree.attr
dv1:/big_project/sean/abc
dv1:/big_project/sean/tree
cmdump% volname test
cmdump% execute
cmdump% (cmtar -c -M -V test -b 3200 -f /dev/rstc0 -T - -G)
cmtar: Removing datavault spec from names in the archive.
cmtar: Removing leading / from absolute pathnames in the archive.
cmtar: Prepare volume #2 (/dev/rstc0), then hit return


cmdump% exit
```


Copyright © 1994 Thinking Machines Corporation

```
% cmrestore
Retrieving directory info from archive(s) /dev/rstc0...done
Current directory is big_project
Files will be restored into CMFS directory dv1:/big_project
cmrestore% ls
big_project/sean/
big_project/may30
cmrestore% cd sean
cmrestore% ls
abc
tree
cmrestore% cd ..
cmrestore% mark -r sean
cmrestore% display
big_project/sean/
big_project/sean/.abc.attr
big_project/sean/.tree.attr
big_project/sean/abc
big_project/sean/tree
cmrestore% execute
(cmtar -x -M -b 3200 -f /dev/rstc0 -T -)
cmrestore% exit
%
```

The cmdump/cmrestore Command Descriptions

The commands supported by **cmdump** and **cmrestore** are listed below in alphabetical order. Note that although most of the commands are supported by both programs, a few are supported only by **cmdump**.

attr — toggles processing mode of attribute files

Syntax: **attr**

Description: By default, CMFS attribute files are marked and unmarked automatically along with their data file counterparts. **attr** toggles between this mode and "explicit" mode, where attribute files must be explicitly marked or unmarked.

If it becomes necessary to explicitly specify a CMFS attribute file on a command line, explicit mode must be used. Otherwise, explicit mode is not recommended.

Copyright © 1994 Thinking Machines Corporation

blocking — sets archive blocking factor

Syntax: **blocking** *blocking-factor*

Description: **blocking** sets the blocking factor that is passed to **cmtar** when the **execute** command is given.

cd — changes working directory

Syntax: **cd** *directory*

Description: In **cmdump**, **cd** changes the current directory for the **ls**, **find**, and **mark** commands. In **cmrestore**, **cd** changes the current directory within the archive's tree for the **ls** and **mark** commands.

display — shows a list of marked files

Syntax: **display**

Description: **display** prints a list of the currently marked files.

emacsedit — toggles Emacs-style editing and history

Syntax: **emacsedit**

Description: **emacsedit** enables/disables both the Emacs-style command-line editing and history mechanisms.

execute — runs cmtar

Syntax: **execute**

Description: **execute** runs **cmtar** with the appropriate arguments on the marked files.

exit, **quit** — exits **cmdump** or **cmrestore**

Syntax: **exit**, **quit**

Description: Execute this command to exit **cmdump** or **cmrestore** and return to the shell prompt.

find — finds files by name or by other characteristics

cmdump syntax: find *expression*

cmrestore syntax: not available in **cmrestore**

Description: **find** recursively descends the directory hierarchy from the current directory, seeking files that match a logical expression spe-

Copyright © 1994 Thinking Machines Corporation

cified via one of the available operators. It is almost identical to **cmfind** (see the man page in Chapter 12), with the following exceptions:

- Unlike **cmfind**, **find** does not take any *pathname* arguments.
- There are two new operators: **-mark** and **-unmark**. These provide a flexible method to (un)mark files based on any of the characteristics supported by **cmfind**.

NOTE: Do not specify **-print** and **-(un)mark** in the same invocation.

help, **?** — gives command help

Syntax: **help**, **?**

Description: **help** prints a brief synopsis of all available commands.

ls — lists the contents of a directory

cmdump syntax: **ls** [**-aAcdfFgilqrRstu1**] *files*

cmrestore syntax: **ls** *files*

Description: **ls** lists the contents of a directory. The **cmdump** version is identical to **cmls** (see the **cmls** man page in the appendix), with the exception of the **-C** (columnar output) flag, which here is the default, and which can be overridden by the **-1** (the numeral 1) flag.

The **cmrestore** version of **ls** is a simplistic one that lists names only, does not provide tabular output, and takes no arguments other than one or more file names.

mark — marks files

Syntax: **mark** [**-r**|**-n**] *files* [[**-r**|**-n**] *files*]

Description: **mark** flags files for backup (**cmdump**) or restoration (**cmrestore**). The **-r** flag specifies that all files following it (until the next **-n** flag) are directory names, and all files in the subtree rooted at those directories are to be marked. The **-n** flag (the default) specifies that all files following it (until the next **-r** flag) are not to be marked recursively.

In order for a file to be marked, it must reside in either the current directory or one of its subdirectories.

In verbose mode, all files being marked that aren't already marked are listed.

Copyright © 1994 Thinking Machines Corporation

Specifying a directory to be marked non-recursively has no effect. Specifying a regular file to be marked recursively is the same as specifying it non-recursively.

See also **unmark**.

more — toggles output pagination

Syntax: **more**

Description: **more** toggles output pagination via the UNIX **more** command. This command has no effect on the output of the forked **cmtar**.

prompt — changes the prompt

Syntax: **prompt** *string*

Description: **prompt** sets the command-line prompt of **cmdump** and **cmrestore** to *string*.

pwd — prints the working directory

Syntax: **pwd**

Description: **pwd** prints the name of the current working directory. In **cmdump**, this is a directory in a CM-5 file system. In **cmrestore**, this is a directory within the archive's tree.

status — displays the status of **cmdump** or **cmrestore**

Syntax: **status**

Description: This command displays status information about **cmdump** and **cmrestore**: archive filename, blocking factor, current directory, attribute file handling mode, verbose mode, prompt, Emacs-editing/history mode, and output pagination mode.

unmark — unmarks files

Syntax: **unmark** [**-r**|**-n**] *files* [[**-r**|**-n**] *files*]

Description: **unmark** specifies that previously marked *files* should be unmarked. The **-r** flag specifies that all files following it (until the next **-n** flag) are directory names, and all files in the subtree rooted at those directories are to be unmarked. The **-n** flag (the default) specifies that all files following it (until the next **-r** flag) are not to be unmarked recursively.

Copyright © 1994 Thinking Machines Corporation

In order for a file to be marked, it must reside in either the current directory or one of its subdirectories.

In verbose mode, all files being unmarked that aren't already unmarked are listed.

Specifying a directory to be unmarked non-recursively has no effect. Specifying a regular file to be unmarked recursively is the same as specifying it non-recursively.

`verbose` — toggles verbose mode

Syntax: **`verbose`**

Description: **`verbose`** toggles verbose mode. In verbose mode, **`cmdump`** and **`cmrestore`** print the names of the files as they are marked or unmarked.

`volname` — specifies the volume name of the archive

`cmdump` syntax: **`volname`** *volume-name*

`cmrestore` syntax: not available in **`cmrestore`**

Description: **`volname`** specifies a volume name that is passed to **`cmtar`** when the **`execute`** command is given.

8.3 Storing and Retrieving Raw Data: The cmdd Command

If your application requires handling IBM data set tape labels, see Section 8.4.

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 an SFS or CMFS file, execute the command **`cmdd`**. Execute **`cmdd`** from a computer networked to both the tape drive and the machine that hosts the file system in which you want to store the data; generally, a VMEIO host computer or a CM-IOPG with a local tape drive gives the best performance.

Copyright © 1994 Thinking Machines Corporation

cmdd's flags specify

- whether the input is coming from a CM-5 (SFS or CMFS) file system (**-fromdv**). If the input is coming from a CM-5 file system, you are required to specify the name of the file via the **if=***name* argument. (Specifying the input is not necessary if the input is coming from standard input.)

- whether the output is going to a CM-5 file system (**-todv**). If the output is going to a CM-5 file system, you are required to specify the name of the file via the **of=***name* argument. (Specifyng the output is not necessary if the output is going to standard output.)

- whether to append the input to the output file (**-a**) rather than rewrite it if it already exists.

The **cmdd** command is especially suited to I/O on raw physical devices because it supports arguments that allow reading and writing in arbitrary record sizes: the input block size (**ibs=***n*) and output block size (**obs=***n*) are given in bytes. (If the input and output block sizes are the same, use the **bs=***n* argument instead.) The block size indicates how many blocks constitute a "chunk" of data coming from the input file or written to the output file. The default is 65,536. If the output file resides on a DataVault, however, 1600K is a good value, because the DataVault block size is 16 Kbytes. If the input is coming from a StorageTek tape drive, or if the output is going to a StorageTek tape drive, fixed-block mode may be specified by **ifbs=***n* or **ofbs=***n*, respectively, where $0 < n <$ 64K.

Other arguments specify

- conversions to perform in the course of copying, including byte swapping (**swap**), padding (**sync**), and EBCDIC $\leftrightarrow$ ASCII conversion (**conv=ascii**, **conv=ebcdic**), and uppercase-lowercase conversion (**lcase**, **ucase**)

- other miscellaneous specifications, including seeking to specific parts of the input or output files

For complete information about **cmdd**'s arguments, see its man page in Chapter 12.

Copyright © 1994 Thinking Machines Corporation

A sample **cmdd** command is shown below. It reads an EBCDIC tape (from a drive connected to a CM-IOPG), blocked in ten 80-byte EBCDIC card images per record, into **cmiop:/x**:

```
# cmdd -todv ifbs=800 if=/dev/rstc0 of=cmiop:/x \
bs=1600k conv=ascii,lcase
```

The resulting ASCII-format file has all lowercase characters.

8.4 Handling IBM Data Sets and Tape Labels: The ibmtape Command

If you have access to a StorageTek 4980 tape drive connected to a CM-IOPG, you can use the CMFS utility **ibmtape** to import and export IBM data sets on IBM Standard Label 3480-type tapes. **ibmtape**:

- performs the standard tasks of IBM label reading, writing, and verification
- optimizes the throughput between the StorageTek and the CMFS or SFS file system by double-buffering the data

  Because the StorageTek is a streaming tape unit — capable of reading and writing data as fast as the tape passes beneath the tape heads — it must be kept constantly busy in order to achieve the 2 Mbytes/second maximum tape-VMEIO transfer rate. (If the StorageTek is "starved" for data as it is writing a tape, or if the peripheral it is providing data to cannot receive the data as fast, the StorageTek must frequently reposition the tape head, which degrades performance). To ensure that the maximum rate is achieved, **ibmtape** allocates two buffers rather than one: when one buffer fills up, the second begins to fill as the first is simultaneously emptied, and so on.

8.4.1 About IBM-Format Tapes

On IBM systems, a file is known as a data set. A data set name (dsn) — roughly equivalent to a UNIX filename — consists of a sequence of no more than 44 characters, interspersed with the period character (.). A data set name carries with it an implied hierarchy. For instance, the dsn

Copyright © 1994 Thinking Machines Corporation

PAYROLL.JULY.SOFTWARE is really a subset of a larger data set called **PAYROLL.JULY**, which is, in turn, a subset of a data set called **PAYROLL**.

A data set consists of one or more records, a sequence of bytes. A record is typically the smallest unit of addressable data in a data set. Records come in two types in the IBM world:

- *Fixed-length records*, the most common record type, are all of the same length. They are used widely in databases. Under UNIX, a particular record in a file like this would be extracted by multiplying the record length by the number of the desired record and doing a seek operation. On an IBM system, however, the user merely issues a read call with the particular record number.

- *Variable-length records* may be of different, arbitrary lengths. Under UNIX, a user would have to scan an entire file to locate a particular record. On an IBM, pointers to specific records are kept along with the file.

IBM tapes are labeled with an electronic version of the label found on the outside of the tape, written on the first few records of the tape itself. The label usually contains data descriptive of the contents of the tape, such as the data set name, the volume number, the owner of the tape, the number of the records, etc. The tape may also have a trailer label, which contains information such as checksums, block counts, and an indication as to whether or not there are more volumes to be read in a particular data set.

A program such as **dd** or **cmdd** can extract the tape label so it can be examined by the user (although the information contained in the label is meant to be read by a program, not a human — many of the options to **ibmtape** are used to verify tape labels). Tape labels serve several different purposes:

- They support the IBM concept of extending the file system out to a collection of tapes, allowing the computer to verify the contents of a particular set of tapes.

- They provide an element of protection. Jobs that run on an IBM must specify the data set name and volume serial numbers of the tapes they want to work with. If the wrong volume is inadvertently mounted on the drive, or if the volumes of the data set are mounted out of order, the operator is notified.

Copyright © 1994 Thinking Machines Corporation

- They describe the contents of the tape and give the information necessary to reconstruct the data set on disk as it has been written to tape (potentially across many separate volumes).

8.4.2 ibmtape Syntax

ibmtape {**-fromdv** | **-todv**} **if=***inputfile* **of=***outputfile* [**-vmeio**] **label=** {**isl** | **ibl** | **inl** | **osl** | **onl** | **obl** | **osl, wsl** | **osl, wnl**} **vsn=***list_of_vol_serial_nos* **dsn=***data_set_name* [**bs=***buffersize*] **rformat=***fmt*:*blen*:*rlen* [**stackers=***n*] [**-leds**] [**-compress**] [**-a**]

-fromdv | **-todv**
: These switches indicate the direction of motion of the data. If **-tosda** is specified, it is assumed that the input file is a StorageTek tape unit and the output file resides on the DataVault or SDA (depending on the setting of **CMFS_PATHTYPE**); if **-fromsda** is specified, the reverse is true. One and only one of these switches must be specified.

if=*inputfile*
: Specifies the name of the input file. If a data set is being written, this specifies the name of the file that is being read; if a data set is being read from tape, this specifies the name of the device to read it from (usually **/dev/rstc0** through **/dev/rstc3**). Whether the file is being moved to or from a CMFS or SFS must be specified by the **-todv** or **-fromdv** switch. When *inputfile* is a CMFS or SFS file, it can be specified using a full pathname (i.e., *hostname*:*filename*) or, if **DVWD** and/or **DVHOSTNAME** are set appropriately, a relative pathname.

of=*outputfile*
: Specifies the name of the output file. See **if=**.

-vmeio
: When executed on a CM-IOPG, this switch causes **ibmtape** to use the VMEIO hardware in the CM-IOPG to transfer the data to or from the CMFS or SFS, instead of sending it over the Ethernet.

Copyright © 1994 Thinking Machines Corporation

label={ **isl** | **ibl** | **inl** | **osl** | **onl** | **obl** | **osl, wsl** | **osl, wnl** }
Specifies the type of label processing to process against the given input or output data set. This option applies to the tape being read or written on the StorageTek tape unit. Since Standard-Label processing is most commonly used, we describe those options more fully:

isl Standard-Label processing on input tape.
Data is read from the IBM Standard Label that precedes the input dataset and compared against the values given in the command-line **dsn** and **vsn** options. If any of these fields do not match, the operator is alerted. If no **rfor-mat** option is specified, the data format, block size, and record size in the label are used for processing. When the end of volume on each tape is reached, the number of blocks read from the tape is compared with the block count given in the label. If these counts do not match, the operator is alerted. If the tape label indicates that the tape was written in compressed format, but the hardware the tape is loaded on is not capable of reading compressed media, the tape is rejected.

inl Non-labeled processing on input tape.

ibl Bypass label processing on input tape.

osl Standard-Label processing on output tape.
Data is read from the IBM Standard Label that the tape was initialized with and compared against the values given in the command-line **dsn** and **vsn** options. If these do not match, the operator is alerted; if the operator chooses to proceed, the tape is written, but the label is not altered.

onl Non-labeled processing on output tape.

obl Bypass label processing on output tape.

Copyright © 1994 Thinking Machines Corporation

osl, wsl
Replace any existing Standard Label with the **dsn** and **vsn** specified on the command line.

osl, wnl
Replace any existing Standard Label with the given non-labelled data, rather than use it simply for verification.

vsn=*list of volume serial numbers*
Specifies the six-digit volume serial numbers (VSNs) of the different volumes, separated by commas. On input tapes, only the volumes specified are processed. For output tapes, enough VSNs must be specified to accommodate all the data, and extra VSNs are ignored.

dsn=*data set name*
Specifies the name of the current data set. The name is matched against the label of the input or output tape.

bs=*buffersize* Specifies the size of the buffer to allocate in VMEIO host computer or control processor main memory. *buffersize* must be a constant multiple of the tape's block length (as specified in the tape label or on the command line via the **rformat** argument).

Performance note: Each VMEIO has 32 Mbytes of on-board buffer memory available to the user. A small amount of this memory (approximately 48 Kbytes) is reserved for the system, leaving in the range of 31 Mbytes of memory available for the application or for use by **ibmtape**. In order to achieve maximum performance from the CM-IOPG system, it is usually wise to set *buffersize* as large as possible.

Note, however, that because **ibmtape** employs a double-buffering system, the system automatically allocates *two* buffers of size *buffersize* in VMEIO buffer memory. Keeping this in mind, the **bs** option can be no larger than about 15 Mbytes. Maximum performance is usually reached with a buffer size in the range of 5−10 Mbytes.

Copyright © 1994 Thinking Machines Corporation

rformat=*fmt*:*blen*:*rlen*

Specifies the format of the data on the tape. If this option is omitted for input Standard Label tapes, the formatting information is extracted from the label (an extra step that slows performance) and used. **rformat** must be provided for output data. *fmt* specifies the format of the records on the tape, and must be one of **f**, **fb**, **v**, **vb**, **vbs**, or **u**. *blen* specifies the length of each physical tape block. *rlen* specifies the length of each logical record on the tape (0 for variable-format tape). The record length must divide evenly into the block length, and the block length must divide evenly into the buffer size (that is, *bs* ≥ *blen* ≥ rlen, and *bs* % *blen* = 0, and *blen* % *rlen* = 0). The *fmt* fields are:

f	Fixed-length records
fb	Fixed-length blocked records
v	Variable-length records
vb	Variable-length blocked records
vbs	Variable-length, blocked, spanned records
u	Undefined

For more information, see the **ibmtape** manual page in Chapter 12.

stackers=*n* Causes the use of up to four input stackers. This allows the operator to overlap the time needed to rewind and unload one tape that has finished processing with the loading and reading or writing of a subsequent tape in the data set. It also allows more than 10 tapes to be loaded at one time (each stacker on a StorageTek typically holds 10 tapes).

-leds Displays the volume serial number of the tape currently being processed on the status display of the StorageTek drive.

-compress Enables use of the ICRC (Improved Cartridge Recording Capability) hardware on the StorageTek drives. (Note that not all drives have compression hardware.) This switch is needed only for output tapes; compressed tapes are uncompressed automatically on input and require no special processing. If compressed

Copyright © 1994 Thinking Machines Corporation

tapes are loaded on a drive that does not have the compression hardware, the operator is alerted and the tapes are rejected. New compressed tapes are labeled as such and cannot be read by tape units that do not have the ICRC hardware.

-a Specifies that the data set is to be appended to the destination file.

-multilabel
Informs **ibmtape** that the tape has multiple data sets on it. This option is required when reading multiple data sets from a single tape.

-notfirst Indicates that because the data set to be read is not the first one on the volume, do not rewind the tape, i.e., do not do label verification. This option is required when reading multiple data sets from a single tape.

8.4.3 Reading Multiple Data Sets from a Single Tape

Note that **ibmtape** does not yet support writing multiple data sets onto a single tape.

Typically, a data set spans one tape (see tape A in Figure 8) or a set of tapes. Occasionally, however, if they're small, several data sets are put on a single tape (see tape B in Figure 8). To use **ibmtape** to read a data set from a tape that contains multiple data sets, start at the beginning of the tape and read each data set, ignoring the data until coming to the data set of interest. This technique circumvents scanning through the tape to find a particular data set.

To get the data from the third data set of Figure 8's tape B, for example, execute the following commands:

```
% ibmtape -todv if=/dev/nrstc0 of=dataset1 \
-multilabel rformat=fmt:blen:rlen label=label vsn=vol#  \
dsn=name bs=buffersize
% ibmtape -todv if=/dev/nrstc0 of=dataset2 -multilabel \
-notfirst rformat=fmt:blen:rlen label=label vsn=vol#  \
dsn=name bs=buffersize
% ibmtape -todv if=/dev/nrstc0 of=dataset3 -multilabel \
```


Copyright © 1994 Thinking Machines Corporation

```
-notfirst rformat=fmt:blen:rlen label=label vsn=vol#  \
dsn=name bs=buffersize
```

Note three things about this command sequence:

- The name of the input file is **/dev/nrstc0**. The **n** in **nrstc0** tells the tape drive to not rewind the tape after the operation completes (UNIX and CMOST default behavior is to rewind after each operation).
- The **-multilabel** option informs **ibmtape** that the tape contains more than one data set.
- The **-notfirst** option informs **ibmtape** that the data set about to be read is not the first one on the tape, and therefore to not verify the tape label, which would require rewinding the tape.

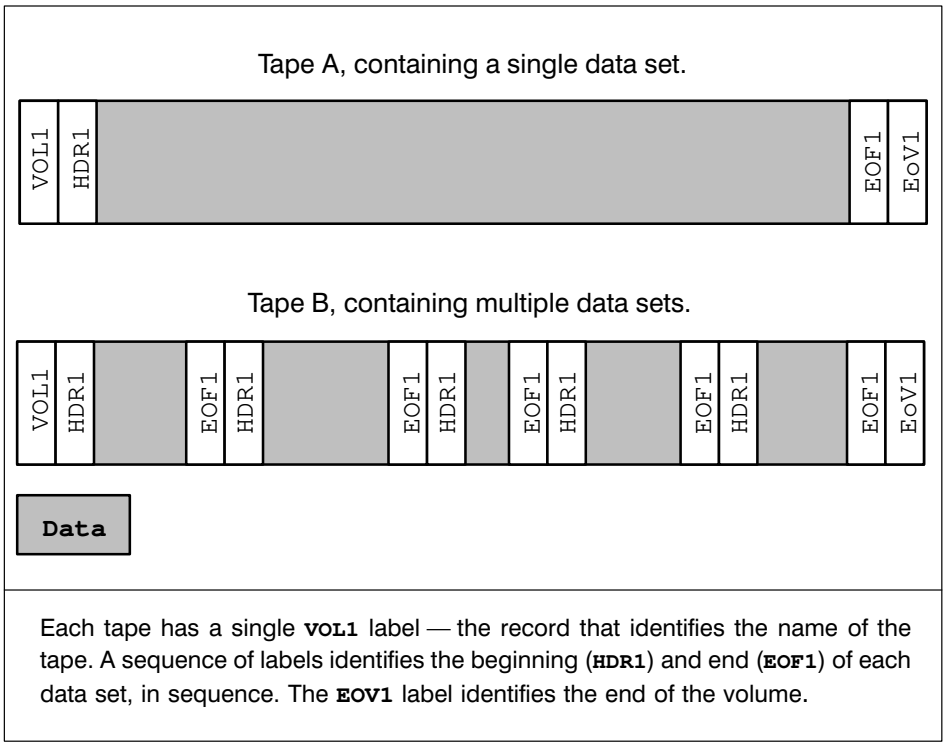

Figure 8. Single data set and multiple data set tape layout.

Copyright © 1994 Thinking Machines Corporation

8.4.4 ibmtape Performance Issues

In general, there are two potential causes for most **ibmtape** performance degradation:

- Using variable-record format when fixed-record length format would do.

  In general, the StorageTek system can process the various fixed-record formats (**f**, **fb**) much faster than the variable-record formats (**v**, **vb**, **vbs**, and **u**): because CM-5 file systems do not support structured files, variable-record files require additional processing as they are read from tape. For instance, the data streams of variable-record files contain start-record and end-record indicators, which most application programs want to remove or skip. To facilitate this, **ibmtape** creates an index into the data set as it is read. This index is called a control file and is named *dataset_name*.**ctl**. Creating the control file can be quite time-consuming since the entire input buffer must be scanned before it is written to disk. Although **ibmtape**'s double buffering provides extra time for the program to scan the buffer, extremely short variable-length records can still cause a considerable performance degradation. Therefore, if a variable-record format is being used to store data of primarily fixed record length, it would be advantageous to switch to a fixed-length blocked or fixed-length unblocked format if at all possible.

- Reading short blocks frequently (unfortunately, this cannot always be helped, as it is a direct result of the way the tape was written — something that one does not always have control over).

  Short *blocks* — shorter, that is, than the block size specified by **ibmtape**'s **rformat** option — also degrade performance. If the StorageTek encounters a short block, it pauses as it reports the short block size to **ibmtape**. **ibmtape** must then take the drive out of fixed-block mode, rewind the tape one record, reread the short record, and put the drive back into fixed-block mode before it can continue with the read. (Short blocks on fixed-record tapes may occur because of a non-constant number of records stored per block. For example, if the **rformat** option specifies **fb:8000:80** (record length of 80 bytes, block size of 8000 bytes), each tape block should contain 100 records. Short blocks would result if only 99 records were put on some of the blocks.)

Copyright © 1994 Thinking Machines Corporation

8.5 Tape Transfers Using Serial I/O

The serial I/O programming method presented here performs tape ↔ CMFS or SFS transfers. The host system is a control processor, a CM-IOPG, or a VMEIO host computer.

This programming method uses a loop consisting of a UNIX I/O call and a CMFS serial I/O call: the UNIX I/O call performs the host memory ↔ tape transfer; the CMFS serial I/O call performs the host memory ↔ CMFS or SFS transfer. (In a variation of this method, the loop consists simply of a UNIX I/O call that performs the UNIX file system ↔ tape transfer, and a CMFS copy command performs the UNIX file system ↔ CMFS or SFS transfer. This method is useful if the data will be used by both the CM and the serial machine.) See Figure 9.

This method is useful when the application calls for

- Preliminary massaging of data on the host system. For example, although byte-swapping is usually done on the CM, certain circumstances may dictate that it be done on the tape drive's host.
- Buffering data before sending it to its final destination. Doing so may improve bus performance.
- Processing several chunks of data in succession rather than the entire amount at once. For example, an application that uses an amount of data so large that it will not all fit on a DataVault must process the data in stages:
  - The first stage transfers a chunk of data from tape to a DataVault(s).
  - The second stage transfers the chunk of data from the DataVault(s) to the CM, while the first stage executes again, simultaneously.

  The simultaneous execution of the two stages, after the initial execution of the first stage, allows the CM to remain in continuous computation mode.

Copyright © 1994 Thinking Machines Corporation

Follow these steps when writing a program to read from a tape into a CMFS file:

1. Open the tape device using the UNIX **`open()`** call.
2. Open the CMFS file using the **`CMFS_open`** call:
   - If the file does not yet exist, create it with the appropriate read/write/execute permissions.
   - If the file already exists, verify that you have permission to read, write, and/or execute the file. If you don't have the appropriate permissions, change them.

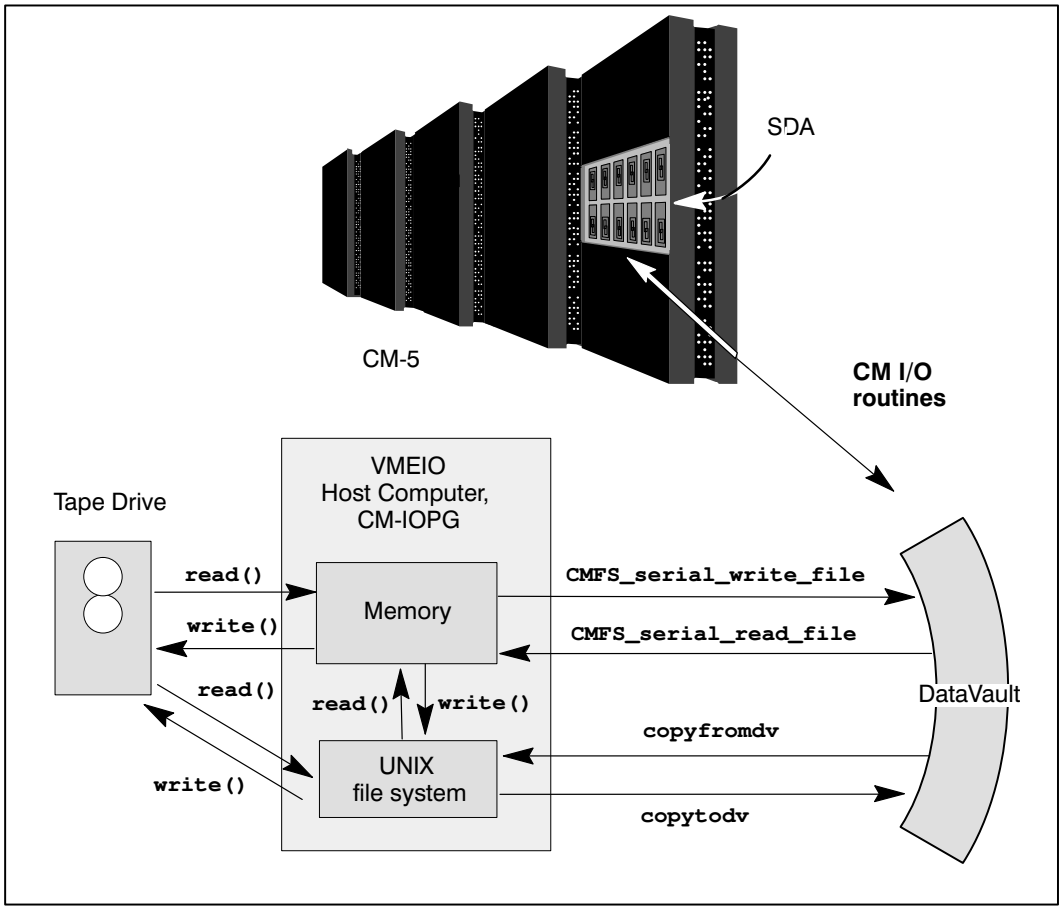

Figure 9. Tape I/O on a serial host.

Copyright © 1994 Thinking Machines Corporation

3. Prepare the CMFS file for writing (see Chapter 4):
   - If necessary, call a CMFS truncate routine to modify the file's size.
   - Set the file pointer appropriately by calling a CMFS **lseek** routine.
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** call. If immediate processing is required, issue CMFS parallel read calls to transfer the data into the CM.
5. Call **CMFS_close** to close the DataVault file. Issue **close()** to close the device file.

8.5.1 Sample Programs

Two sample programs are listed in this section. The first programming example, **createFile**, is a generic program that reads data from a tape drive connected to a VMEIO host computer into a CMFS file on a DataVault. The second programming example, **stageCopy**, illustrates the use of the serial I/O method of tape I/O: the application uses an amount of data so large that it will not all fit on a DataVault.

Sample Program: createFile

The sample program below, **createFile**, performs steps 1−5 (see Section 8.5), but does not send any data into the CM. For best performance, compile and run **createFile** on the host machine of the tape drive. The **createFile** program accepts two command-line arguments — the raw data's pathname, and the name to give to the CMFS file. A sample compilation and the program listing follows.

```
% cc -o createFile createFile.c -lcmfs_cm5
% createFile vme1:/dev/tape dv1:/acct5/imagedata
```


Copyright © 1994 Thinking Machines Corporation

```
/*   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);
```


Copyright © 1994 Thinking Machines Corporation

```
/*                          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)

}
```


Copyright © 1994 Thinking Machines Corporation

Sample Program: stageCopy

The **stageCopy** program illustrates how to use the serial I/O method for an application that uses an amount of taped data too great to fit on a DataVault, as discussed in Section 8.5.

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/ioctl.h>
#include <sys/fcntlcom.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/signal.h>
#include <sys/errno.h>
#include <cm/cm_file.h>
#include <cm/cm_errno.h>
#include <cm/cm_stat.h>
#include "params.h"
#include "shbuf.h"
#include "sysIO/stcreg.h" /* For StorageTek driver ioctl calls */

extern mark_buffer_free ();
extern mark_buffer_full ();
extern get_free_buf ();
extern get_full_buf ();

/*  Copy input file, INFILE,  to output file, OUTFILE, using shared  */
/*  for the buffers.  Files can be on the UNIX host or the Data Vault*/

fast_copy (infile, outfile, flags, physical_width, virtual_width)
     char *infile, *outfile;
     int flags;
     int physical_width;
     int virtual_width;
{
    int shm_id;
    int i;
    int total_bytes_read = 0;
    int total_bytes_written = 0;
    void (*old_sighup)();
    struct shmid_ds sd;
    int unix_input_block_size = DEFAULT_INPUT_BLKSIZE;
    int unix_output_block_size = DEFAULT_OUTPUT_BLKSIZE;
    int input_block_size, output_block_size;
    int verbose = 0;

    if (flags & VERBOSE) verbose++;     /* CMI_vmeio_swab = 0; */
```


Copyright © 1994 Thinking Machines Corporation

```
                                        /* CMI_soft_error_print = 1; */

  /* Get shared memory segment with the current process ID as the key */
  shm_id = shmget (getpid (), sizeof (struct shbuf), IPC_CREAT|IPC_EXCL);
  if (shm_id == -1)
  {
       perror ("shmget");
       return (-1);
  }

/* Make it accessable by other users */
  sd.shm_perm.mode = 0644;
  sd.shm_perm.uid = getuid ();
  sd.shm_perm.gid = getgid ();
  if (shmctl (shm_id, IPC_SET, &sd) < 0)
       perror ("shmctl set permissions");
  shbuf = (struct shbuf *)shmat (shm_id, 0, 0);
  if ((int)shbuf == -1)
  {
       perror ("shmat");
       (void) shmctl (shm_id, IPC_RMID, 0);
       return (-1);
  }
  (void) shmctl (shm_id, IPC_RMID, 0); /* Return shared memory ID */



  /* Mark all buffers as free */
  for (i = 0; i < NBUF; i++)
  {
       shbuf->buf_status[i] = FREE;
       shbuf->buf_buf[i].size = SHBUFSIZE;
  }

  /* No error */
  shbuf->reader_error_flag = shbuf->writer_error_flag = 0;
  expected_sequence = sequence = 0;

  /* If reading/writing to CMFS set input/output block size to the */
  /* size of the shared memory buffer.  Otherwise, use the default */
  /* unix block size
            */
  if (flags & FROMDV)
       input_block_size = SHBUFSIZE;
  else
       input_block_size = unix_input_block_size;
  if (flags & TODV)
       output_block_size = SHBUFSIZE;
  else
       output_block_size = unix_output_block_size;
```


Copyright © 1994 Thinking Machines Corporation

```
reader_pid = getpid ();
old_sighup = signal (SIGHUP, signal_handler);


if ((writer_pid = fork ()))      /* PARENT--READER process */
{
     int infd;
     if (flags & FROMDV)       /* Open input file on Data Vault */
     {
          infd = CMFS_open (infile, CMFS_O_RDONLY);
          if (infd < 0)
          {
               (void) kill (writer_pid, SIGKILL);
               shbuf->reader_error_flag = CMFS_errno;
               goto done;
           }
     }
     else               /* Not from DV, open input file on UNIX */
     {
          infd = open (infile, O_RDONLY);
          if (infd < 0)
          {
               perror (infile);
               (void) kill (writer_pid, SIGKILL);
               shbuf->reader_error_flag = errno;
               goto done;
          }
     }
          else         /* Not from DV, open input file on UNIX */
          {
               infd = open (infile, O_RDONLY);
               if (infd < 0)
               {
                    perror (infile);
                    (void) kill (writer_pid, SIGKILL);
                    shbuf->reader_error_flag = errno;
                    goto done;
               }
          }
     while (1)                 /* Read INFILE */
     {
          register char *p;
          int bytes_available;
          int buf_no;
          int bytes_read;
          int last_bytes_read;
          buf_no = get_free_buf ();

          if (buf_no < 0)
          {
```


Copyright © 1994 Thinking Machines Corporation

```
     /* Writer error */
     break;
}

/* p is assigned to point to buffer number BUF_NO */
p = shbuf->buf_buf[buf_no].data;
bytes_available = shbuf->buf_buf[buf_no].size;

/* Fill up shared buffer by INPUT_BLOCK_SIZE byte chunks */
while (bytes_available >= input_block_size)
{
     if (flags & FROMDV)
          bytes_read = CMFS_serial_read_file (infd, p,
                         input_block_size);
     else
          bytes_read = read (infd, p, input_block_size);
     if (bytes_read < 0)
     { /* Handle read error */
          shbuf->reader_error_flag = ((flags & FROMDV) ?
                                      CMFS_errno : errno);
          break;
     }
     if (bytes_read == 0)
     {   /* handle EOF */
          shbuf->reader_error_flag = -1; /* EOF */
          break;
     }


     if (bytes_read < THRESH)
     {    /* Throw away data by ignoring the fact that it */
          /* was ever read at all... */
          printf ("P::Skipping %d byte record\n", bytes_read);
          fflush (stdout);
          if ((last_bytes_read == 3200) && (bytes_read ==
                                                  400))
          {
               int sixty_two_forty = 6240;
               if (ioctl (infd, STCIOC_SET_BLKSIZE,
                    &sixty_two_forty) == 0)
               {
                    printf ("P::Toggling input_block_size to
                                   6240");
                    printf ("...and setting STC to fixed block
                                   mode\n");
                    fflush (stdout);
                    input_block_size = SHBUFSIZE;
               }
               else
               {
```


Copyright © 1994 Thinking Machines Corporation

```
                                printf ("P::Error setting blocksize to
                                                6240\n");
                                fflush (stdout);
                           }
                      }
                      last_bytes_read = bytes_read;
                      continue;
                 }
                 bytes_available -= bytes_read;
                 p += bytes_read;
                 total_bytes_read += bytes_read;
            } /* end WHILE */
            shbuf->buf_buf[buf_no].bytes_valid =
                                shbuf->buf_buf[buf_no].size -
                                     bytes_available;
            mark_buffer_full (buf_no);
            if (bytes_read <= 0)
                 break;
            if (verbose)
            {
                 printf (""P::%d    \r", total_bytes_read);
                 fflush (stdout);
            }
       } /*end while -- Done reading INFILE */

       if (flags & FROMDV)       /* Close input file INFILE */
            CMFS_close (infd);
       else
            close (infd);
  done:
       /* Wait for writer to exit */
       while (wait (0) != writer_pid)
            ;


       /* Restore SIGHUP signal handler */
       (void) signal (SIGHUP, old_sighup);

       /* Report error messages before exiting */
    {
            char *reader_status, *writer_status;
            if (shbuf->reader_error_flag == 0)
                 reader_status = NULL;
            else if (shbuf->reader_error_flag == -1)
                 reader_status = NULL;
            else if (flags & FROMDV)
                 reader_status = CMFS_sys_errlist
                                          [shbuf->reader_error_flag];
            else
                 reader_status = sys_errlist[shbuf->reader_error_flag];
```


Copyright © 1994 Thinking Machines Corporation

```
            if (shbuf->writer_error_flag == 0)
                 writer_status = NULL;
            else if (shbuf->writer_error_flag == -1)
                 writer_status = NULL;
            else if (flags & TODV)
                 writer_status = CMFS_sys_errlist
                                              [shbuf->writer_error_flag];
            else
                 writer_status = sys_errlist[shbuf->writer_error_flag];
            if (reader_status)
                 fprintf (stderr, "Error reading %s: %s\n",infile,
                                                        reader_status);
            if (writer_status)
                 fprintf (stderr, "Error writing %s: %s\n",outfile,
                                                        writer_status);
            if (reader_status || writer_status)
                 total_bytes_read = -1;
       }

       return (total_bytes_read);
  }


  else      /* CHILD--WRITER process */
  {
       int outfd;
       int outfile_current_size;
       struct cm_stat outfile_initial_stat;

       /* Set the write flags for both Unix and CMFS */
       int cmfs_write_flags = CMFS_O_WRONLY|CMFS_O_CREAT|CMFS_O_RECORDWISE;
       int unix_write_flags = O_WRONLY|O_CREAT;

       /* Add the append option if specified */
       cmfs_write_flags |= (flags & APPEND) ? CMFS_O_APPEND : CMFS_O_TRUNC;
       unix_write_flags |= (flags & APPEND) ? O_APPEND : O_TRUNC;



       if (flags & TODV)    /* Open output file on DataVault */
       {
            outfd = CMFS_open (outfile, cmfs_write_flags, 0666,
                                        physical_width, virtual_width);
            if (outfd < 0)
            {
                 shbuf->writer_error_flag = CMFS_errno;
                 exit (1);
            }

            /* In case of APPEND flag, stat the file to get byte size */
```


Copyright © 1994 Thinking Machines Corporation

```
            if (flags & APPEND)
            {
                 if (CMFS_fstat (outfd, &outfile_initial_stat) < 0)
                      CMFS_perror (outfile);
            }
        }
        else             /* Not to DV, open output file on UNIX */
        {
            outfd = open (outfile, unix_write_flags, 0666);
            if (outfd < 0)
            {
                 shbuf->writer_error_flag = errno;
                 exit (1);
            }
#ifdef notdef
            if (ioctl (outfd, STCIOC_SET_BLKSIZE, &unix_output_block_size) == 0)
                 output_block_size = (SHBUFSIZE/unix_output_block_size
                                              * unix_output_block_size);
#endif
        }

        while (1)                 /* Write OUTFILE */
        {
            int buf_no;
            char *p;
            int bytes_written;
            int bytes_available;
            int bytes_to_write;
            buf_no = get_full_buf ();

            if (buf_no == -1)
            { /* EOF or error on read */
                 if (flags & TODV)
                 {
                      printf ("C::total_bytes_written = %d\n",
                      total_bytes_written);
                      fflush (stdout);
                      /* Calculate the current size of the file to  */
                      /* truncate--this kludges around a bug in    */
                      /* CMFS_serial_write_file                    */
                      if (flags & APPEND)
                           outfile_current_size =
                                outfile_initial_stat.cmst_size +
                                total_bytes_written;
                      else    /* Not APPEND:  Created file from scratch */
                           outfile_current_size = total_bytes_written;

                      /* ftruncate to current size of the output file*/
                      if (CMFS_serial_ftruncate_file (outfd,
                                                 outfile_current_size)
```


Copyright © 1994 Thinking Machines Corporation

```
                              < 0)
                              CMFS_perror (outfile);
                         CMFS_close (outfd);
                    }
                    else
                         close (outfd);
                    exit (0);
               }

               /* p points to data space in full buffer */
               p = shbuf->buf_buf[buf_no].data;
               bytes_available = shbuf->buf_buf[buf_no].bytes_valid;
               while (bytes_available > 0)
               {
                    bytes_to_write = min (bytes_available,
                                                      output_block_size);
                    if (bytes_to_write < output_block_size)
                    {
                         int zero = 0;
#ifdef notdef
                         (void) ioctl (outfd, STCIOC_SET_BLKSIZE, &zero);
#endif
                    }
                    if (flags & TODV)
                    {
                    /* bytes_written = BOGUS_write (outfd, p, bytes_to_write); */
                         bytes_written = CMFS_serial_write_file (outfd, p,
                                                        bytes_to_write);
                    }
                    else
                    {
                         bytes_written = write (outfd, p, bytes_to_write);
                    /* bytes_written = BOGUS_write (outfd, p, bytes_to_write); */
                    }
                    if (bytes_written < 0)
                    { /* Handle write error */
                         shbuf->writer_error_flag = ((flags & TODV) ?
                                                  CMFS_errno : errno);
                         if (flags & TODV)
                              CMFS_close (outfd);
                         else
                              close (outfd);

                         exit (1);
                    }
                    bytes_available -= bytes_written;
                    p += bytes_written;
                    total_bytes_written += bytes_written;
               }  /* end while */
               mark_buffer_free (buf_no);
```


Copyright © 1994 Thinking Machines Corporation

```
        }  /* end while (1) */
    /* NOTREACHED */
    } /* end CHILD--WRITER process */
    /* NOTREACHED */
}



int BOGUS_write (a, b, c)
     int a;
     char *b;
     int c;
{
  printf ("BOGUS writing %d bytes\n", c);
  fflush (stdout);
  return (c);
}
```


Copyright © 1994 Thinking Machines Corporation