PCT WORLD INTELLECTUAL PROPERTY ORGANIZATION
International Bureau

INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(51) International Patent Classification 5 :		(11) International Publication Number:
G06F 15/40, 15/407, 5/00	A1	WO 93/17393
		(43) International Publication Date: 2 September 1993 (02.09.93)

(21) International Application Number: PCT/US93/01891

(22) International Filing Date: 19 February 1993 (19.02.93)

(30) Priority data:
07/840,869 25 February 1992 (25.02.92) US

(71) Applicant: SALIENT SOFTWARE, INC. [US/US]; 124 University Avenue, Suite 300, Palo Alto, CA 94301 (US).

(72) Inventor: CHAMBERS, Lloyd, Lamont, IV ; 328 Waverley, Apt. 2, Menlo Park, CA 94025 (US).

(74) Agent: FLIESLER, Martin, C.; Fliesler, Dubb, Meyer and Lovejoy, Four Embarcadero Center - Suite 400, San Francisco, CA 94111-4156 (US).

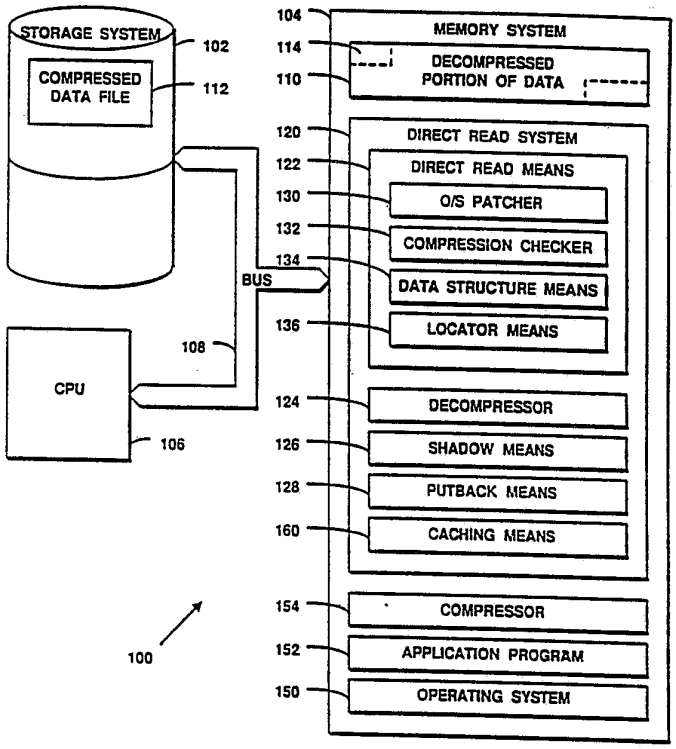
(81) Designated States: AU, CA, JP, European patent (AT, BE, CH, DE, DK, ES, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE).

Published
With international search report.

(54) Title: DIRECT READ OF BLOCK COMPRESSED DATA FILE

(57) Abstract

A method and apparatus for reading and decompressing a subset (114) of a block compressed data file (112) in which three operating modes are used. The three operating modes are selected according to user access privileges. A direct read mode for read only access, a file shadowing mode for read/write access when it is likely there will be no write, and a file putback mode, when it is likely that the file will be written to, are the three access modes. Data catching (160) is employed to improve system throughput and transparency. Access of individual blocks of compressed data is through a table of offsets (136), either stored in the compressed data file or constructed from scratch.



FOR THE PURPOSES OF INFORMATION ONLY

Codes used to identify States party to the PCT on the front pages of pamphlets publishing international applications under the PCT.

AT	Austria	**FR**	France	**MR**	Mauritania
AU	Australia	**GA**	Gabon	**MW**	Malawi
BB	Barbados	**GB**	United Kingdom	**NL**	Netherlands
BE	Belgium	**GN**	Guinea	**NO**	Norway
BF	Burkina Faso	**GR**	Greece	**NZ**	New Zealand
BG	Bulgaria	**HU**	Hungary	**PL**	Poland
BJ	Benin	**IE**	Ireland	**PT**	Portugal
BR	Brazil	**IT**	Italy	**RO**	Romania
CA	Canada	**JP**	Japan	**RU**	Russian Federation
CF	Central African Republic	**KP**	Democratic People's Republic of Korea	**SD**	Sudan
CG	Congo			**SE**	Sweden
CH	Switzerland	**KR**	Republic of Korea	**SK**	Slovak Republic
CI	Côte d'Ivoire	**KZ**	Kazakhstan	**SN**	Senegal
CM	Cameroon	**LI**	Liechtenstein	**SU**	Soviet Union
CS	Czechoslovakia	**LK**	Sri Lanka	**TD**	Chad
CZ	Czech Republic	**LU**	Luxembourg	**TG**	Togo
DE	Germany	**MC**	Monaco	**UA**	Ukraine
DK	Denmark	**MG**	Madagascar	**US**	United States of America
ES	Spain	**ML**	Mali	**VN**	Viet Nam
FI	Finland	**MN**	Mongolia		

"Direct Read of block compressed data file."

LIMITED COPYRIGHT WAIVER

A portion of the disclosure of this patent document contains material which is subject to copyright protection. The copyright owner has no objection to the facsimile reproduction by anyone of the patent document or the patent disclosure, as it appears in the Patent and Trademark Office patent file or records, but otherwise reserves all copyright rights whatsoever.

CROSS-REFERENCE TO RELATED APPLICATIONS

This application is a continuation-in-part of application 07/759,226 entitled "Fast Data Compressor with Direct Lookup Table Indexing into History Buffer", which was filed September 13, 1991 by Lloyd Lamont Chambers IV, which is commonly assigned with the present application, and which is incorporated herein by reference as if fully set forth herein. Another application of particular interest is application _____ entitled "Method and Apparatus for Locating Longest Prior Target String Matching Current String in Buffer", which was filed February 21, 1992 by Lloyd Lamont Chambers IV. These applications are commonly assigned to Salient Software, Inc.

BACKGROUND OF THE INVENTION

Field of the Invention

The present invention relates generally to reading of data from compressed data files, and more specifically to a method and an apparatus for decompressing and reading only a minimum subset of a compressed data file directly into memory.

Description of the Background

Compressed data residing in a data file must generally be decompressed in order to be of any real use. However, if access to only a small portion of the data file is needed, it is wasteful of computational resources to decompress the entire file. CPU time is unnecessarily wasted decompressing the file, and storage space is unnecessarily wasted holding the unneeded portions of the decompressed data.

Presently-available data decompressors appear to suffer both these faults. When a program issues an access request to a compressed file, the data decompressor decompresses the entire file and stores the decompressed version for access by the program. While the data decompressor is operating, the program must wait. If the data decompressor cannot allocate storage for the decompressed file, unknown errors may occur or the results may be otherwise unpredictable.

For example, if the compressed data file is on a disk which has less free space than the decompressed version of the file, the data decompressor may be unable to operate.

Additionally, existing data decompressors do not operate at the same level of logical operation at

which most computer programs operate. Specifically, while computer programs tend to issue access requests to a certain file (i.e. on a file basis), existing data decompressors tend to operate at the much lower level of an operating system (i.e. on a block basis, disk basis, device driver basis, or the like).

This incompatibility of operational levels imposes certain additional inefficiencies on existing data decompressors. For example, existing data decompressors do not take sufficient advantage of the various read/write permission levels involved in data access requests. If a program opens a file "read only", for example, there is no need to save a permanent copy of the decompressed data, because it will ultimately be discarded when the program closes the file.

What is needed, therefore, is an improved method and apparatus for accessing data files on a file basis, and within the file basis, on the more limited basis of only a requested subset of the data content of a specified file. The improved method and apparatus should have a reduced usage of computational resources, by decompressing only a minimum amount of data. The improved method and apparatus should operate regardless of how much free storage is present on the medium on which the compressed file is stored. The method and apparatus should, additionally, take optimum advantage of access permission levels indicated by programs issuing data access requests.

SUMMARY OF THE INVENTION

The present invention provides an apparatus and method for providing compressed data from a compressed data file in response to a data request which identifies given non-compressed data in a non-compressed data file. The method and apparatus identify, within in the compressed data file, given compressed data which correspond to the given non-compressed data identified by the data request, decompress the given compressed data to reconstitute the given non-compressed data, and provide the reconstituted non-compressed data in response to the data request.

The invention provides three operating modes, selected according to access level permission requested by an application seeking access to the file. A direct read mode is used for read-only permission; a file shadowing mode is used for read/write permission when it is likely that the application will only read from the file and never write to it; a file putback mode is used for read/write permission when it is likely that the application will write to the file.

The invention provides data caching for decompressed data blocks from a file, to avoid repetitious reading and decompressing of a data block containing two or more data requested at different times, to improve throughput and transparency of the system.

BRIEF DESCRIPTION OF THE DRAWINGS

Fig. 1 illustrates the present invention as embodied in a computer system.

Fig. 2 suggests the overall operation of the invention in the control hierarchy of various levels of a computer system.

Fig. 3 shows the basic operation of the invention to decompress only requested portions of a data file into memory.

Figs. 4A, 5A, and 6A are flowcharts demonstrating operation of the invention in response to file open requests in direct read, file shadowing, and file putback modes, respectively.

Figs. 4B, 5B, and 6B are flowcharts demonstrating operation of the invention in response to data write requests, in the three modes.

Figs. 4C, 5C, and 6C are flowcharts demonstrating operation of the invention in response to data read requests, in the three modes.

Figs. 4D, 5D, and 6D are flowcharts demonstrating operation of the invention in response to file close requests, in the three modes.

Figs. 7A-F illustrate operation of the invention to create, decompress, and move files in the file shadowing mode.

Figs. 8A-F illustrate operation of the invention to create, decompress, and move files in the file putback mode.

Fig. 9 illustrates one exemplary file format of a file which may be advantageously used with the present invention.

Fig. 10 illustrates the invention as embodied on a recording medium such as a floppy disk.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENTS

I. COMPONENTS OF THE INVENTION

Fig. 1 shows the present invention as embodied in a computer system 100 having a storage system 102, a memory system 104, and a central processing unit (CPU) 106 coupled together by a bus 108. The storage system may, for example, be a hard disk drive. The memory system may, for example, be random access memory (RAM). These may also be characterized as permanent and temporary storage, respectively. The present invention may be practiced using a variety of data processing environments. However, the preferred environment is a data processing computer having a CPU, memory, and a permanent storage device.

Fig. 10 illustrates the invention as embodied in a floppy disk or the like, and may be understood to represent any appropriate recording medium upon which the invention may be embodied and from which the invention may be retrieved and/or executed. Other such media include hard disks, removable hard disks, optical discs, PROMs, and the like.

Fig. 1 shows that, at a highest level of abstraction, the present invention may be characterized as an apparatus and method for decompressing only a requested portion 110 of a compressed data file 112 from the storage system to RAM. In practice, the decompressed portion 110 may include certain excess data 114 which are not requested, but which are nevertheless decompressed owing to the particular compression/decompression scheme employed.

The present invention is a direct read system 120 which may reside in RAM while in use. The basic

components of the direct read system are a direct read means 122, a decompressor 124, a shadow means 126, a putback means 128, and a caching means 160. The direct read means further includes an operating system patcher 130, a compression checker 132, a data structure means 134, and a locator means 136.

Other components of the computer system may include an operating system 150 which provides a controlling interface to the system hardware. The computer system may further include an application program 152 which issues data file access requests. Such requests may include instructions for such file-oriented operations as "open file", "read data", "write data", "get file position", "set file position", "get end-of-file position", set end-of-file position", "close file", and the like. Data read and write requests may take on a variety of forms. For example, a data read request may specify "read X bytes starting at location Y", or it may specify "read bytes from location X to location Y", or the like. Either of these may be considered as directly or indirectly reciting a starting location, and ending location, and an amount of data.

According to known technology, these requests are ordinarily serviced by the operating system. However, the patcher 130 redirects these requests to be handled by the direct read system. Methods are known by which this is accomplished, such as by altering the contents of a "trap table" or by altering the instructions of the operating system itself. The exact means and methods of the patcher will depend on the particular computer system upon which it is desired to practice the invention.

The computer system may further include a data compressor 154 which compresses data files stored in the storage system. While the data compressor is not a necessary part of the present invention as such, it will be understood that the decompressor 124 is capable of utilizing the same compressed file format as the compressor 154. It will be further understood that the compressor may be included in the invention, to provide data re-compression, for example.

The preferred data compressor and decompressor are described in the cross-referenced applications. The block-oriented compression technique employed therein is ideally suited to use by the present invention, to permit decompression of minimized subsets of a compressed data file, to minimize the waste of computational resources otherwise used at each data request. Because the compression algorithm completely restarts itself for each successive history buffer of data (i.e. the data in one block are compressed entirely independently of data in another block), the present invention is able to extract and decompress only those blocks which contain requested data. Fig. 9 shows the general file format of a compressed data file, which is created by the data compressor and which is readable by the decompressor.

II. GENERAL OPERATION OF THE INVENTION

Fig. 2 illustrates how the present invention cooperates with an application program and an operating system, to provide a wholly-transparent, seamless interface between the application program and data files, whereby compressed data files can be

accessed without modification of -- or notice to -- either the application program or the operating system.

When the application program issues a data file access request, such as those listed above, the request is intercepted by the direct read system before it arrives at the operating system. This ability is provided by the patcher. For example, the application program may issue the request "read the next byte from file X". The direct read system alters the request according to information retained by the data structure means and locator means. After modification, the request may be "read block Y from file X". The operating system receives this modified request; both the application and the operating system are unaware that the request has been modified. The operating system performs the indicated request, in this case reading block Y from file X.

Then, block Y is decompressed by the decompressor, to form block Y'. The direct read system then extracts the originally-requested byte from the block Y', and returns that byte to the application program. To the application program, the operation of the direct read system is transparent -- the application program never knows that its original request was not simply handled by the operating system.

This permits the storage system to store an increased amount of data, without requiring modification of the application program or the operating system. Perhaps more importantly, it requires no intervention or special knowledge by a person using the application program.

The data compressor and decompressor disclosed in the cross-referenced applications have been selected for use in the present invention because their block-oriented nature allows decompression of less than an entire data file, and further because the decompressor is optimized to decompress at highly improved speeds. This dramatically improves the transparency of the present invention to a user. The user may be entirely unaware that decompression is taking place, even when performing operations with which the user has become intimately familiar in a wholly non-compressed environment.

Fig. 3 illustrates that a requested subset of a data file may be read into RAM in response to a data read request, for example. Blocks which do not include the requested data are not decompressed, and are not read into RAM. This reduces CPU time required for decompression, and reduces bus usage required for transfer from the storage system to RAM.

III. OPERATION OF THE INVENTION IN DIRECT READ MODE

Figs. 4A-D illustrate operation of the present invention in a read-only direct read mode, showing file open, write, read, and close operations, respectively.

A. File Open Request -- Direct Read Mode

Fig. 4A shows operations beginning at node 402 when an application issues a data access request to open a file for read-only permission. Initially, at conditional block 404, the compression checker 132 in the direct read means 122 determines whether the indicated file is compressed, and if so, whether it is compressed using a format which the decompressor

can decompress. According to the decompressor of the cross-referenced applications, this may be performed simply by consulting the global file header of the compressed file.

Throughout the remainder of this application, a file compressed in a format which the decompressor 154 can decompress will be referred to as a "direct read file". If conditional block 404 determines that the file is a direct read file, operation passes to block 406. There, the direct read system issues a request for the operating system to open the file. Upon opening of the file, operation continues at block 408, where the data structure means 134 creates data structures for the open file. These data structures will be described below in Section VII regarding "Access to Individual Blocks".

Then, at block 410, the direct read system returns to the application program that file information which the application program would ordinarily expect to receive directly from the operating system. Such information may include, for example, a file number, file size, current file pointer value, and the like.

If, at conditional block 404, it is determined that the file is not a direct read file, operation passes to block 412, where the direct read system simply hands off the file open request to the operating system. At this point, the direct read system, in essence, "takes itself out of the loop" regarding the particular file. Then, at block 410, the operating system itself returns the proper file information to the application program, entirely bypassing the direct read system.

In either case, the file open operation is complete at node 414.

B. Data Write Request -- Direct Read Mode

Fig. 4B illustrates operation of the invention in response to a write request in read-only mode. Operation begins at node 420, when the application program issues the request. Then, unknown or undefined operations are performed by the operating system at block 422, because of the illogical request for writing to a read-only file. Typically, the application program will receive an error message result from the operating system, and no data will be written.

C. Data Read Request -- Direct Read Mode

Fig. 4C illustrates data read operations beginning at node 430, when the application program issues a data read request. Operation continues at conditional block 432, where the compression checker determines whether the file is a direct read file.

If it is a direct read file, operation continues at block 434, where the locator means 136 identifies which blocks of the compressed data file contain the requested data. Then, at block 436, the decompressor 124 decompresses those blocks into RAM. Then, the specifically-requested bytes of those blocks are copied into the buffer where the application program expects to receive them.

At conditional block 432, if the file is not a direct read file, operation passes to block 438, where the direct read system takes itself out of the loop by handing the request off to the operating system for normal data read operations.

In either case, data read operations are complete at node 440.

D. File Close Request -- Direct Read Mode

Fig. 4D illustrates operation of the direct read system in response to a file close request issued by the application program at node 450. Operation continues at conditional block 452, where the compression checker determines whether the file is a direct read file.

If it is a direct read file, operation continues at block 454, where the data structure means 134 discards any data structures used for the file and the caching means 160 de-allocates any cached portions of the decompressed data. Then, at block 456, the direct read system issues a request for the operating system to close the open direct read file.

At conditional block 452, if the file is not a direct read file, operation passes directly to block 456. In either case, file close operations are complete at node 458.

IV. OPERATION OF THE INVENTION IN FILE SHADOWING MODE

Figs. 5A-D illustrate file open, data write, data read, and file close operations of the direct read system when operating in file shadowing mode. In the interest of clarity, Figs. 5A-D do not show conditional blocks determining whether the request involves a direct read file. It will be understood that, as in Figs. 4A-D, if other than a direct read file is involved, the direct read system will hand off the request for normal operations by the operating system.

Direct read operations in file shadowing mode are performed by the direct read means, just as in normal read-only direct read mode. Additional features of file shadowing mode are provided by the

shadow means, and will be described below, with reference to Figs. 5A-D and Figs. 7A-F.

A. File Open Request -- File Shadowing Mode

Fig. 5A shows operations of the direct read system in file shadowing mode. File shadowing mode is entered at node 502 when an application requests to open a file for read/write permission in circumstances indicating that the application will most likely only read from, and not write to, the file.

At block 504, the direct read system issues a request for the operating system to open the compressed file. Then, at block 506, the shadow means determines the decompressed (expanded) size of the entire file. As indicated in the cross-referenced applications, this information may be obtained by simply consulting the global file header of the compressed file.

Then, at block 508, the shadow means issues a request for the operating system to open a shadow file of this decompressed size. In one mode, this request may include a first request to open a file, and a second request to set the end-of-file marker to the indicated position. Thus, the shadow file will be sufficiently large, but will contain garbage. At this point, the shadow file serves only as a "placeholder" file, ensuring that sufficient storage is reserved to hold the entire file upon a subsequent decompression. The shadow file may, in one mode, be stored in a separate location from the original, compressed data file.

Fig. 7A shows the compressed data file as residing in an original location in the storage system, while the storage system further includes a temporary

storage location. In one mode, the original and temporary locations may be distinct "directories" or "folders" in the storage system. In other modes, they could be separate hard disks, or the like.

Fig. 7B shows the compressed data file residing in its original location, and the shadow file residing in the temporary location.

File open operations are complete at node 510.

B. Data Write Request -- File Shadowing Mode

Fig. 5B illustrates operation of the shadow means in response to a data write request issued by the application program at node 520. Operation continues at conditional block 522, where the shadow means determines whether the shadow file is still only a placeholder, or, in other words, whether the compressed data file has yet been decompressed.

If the file has not been decompressed, operation passes to block 524, where the decompressor decompresses the entire data file into the shadow file. Fig. 7B shows the shadow file as a placeholder, while Fig. 7D shows the shadow file as including decompressed data.

From block 524, operation passes to block 526, where the shadow means issues a request for the operating system to move the shadow file from the temporary location to the same location as the compressed data file, as seen in Fig. 7E. From block 526, operation passes to block 528, where the shadow means issues requests for the operating system to close and delete the original compressed data file, as seen in Fig. 7F.

From block 528, operation passes to block 530, where the data write request is handed off to the operating system, which performs the data write to

the decompressed shadow file. If, at conditional block 522, it is determined that the file had already been decompressed, operation passes directly to block 530. In either case, the write is complete at node 532.

C. Date Read Request -- File Shadowing Mode

Fig. 5C illustrates operation of the direct read means and the shadow means in response to a data read request from an application program at node 540. From there, operation continues at conditional block 542, where the shadow means determines whether the file has been decompressed.

If the file has not yet been decompressed, the file is still eligible for ordinary direct read usage, identical to that of the read-only direct read mode, as seen in Fig. 7A. Thus, operation passes to block 544, where the direct read means locates the appropriate blocks of the compressed data file. Then, operation continues at block 546, where the decompressor decompresses those blocks into RAM and copies the requested bytes into the application program's buffer.

If, at conditional block 542, it is determined that the file has already been decompressed, as seen in Fig. 7F, operation passes to block 548. At block 548, the shadow means issues a request for the operating system to read the indicated bytes from the decompressed shadow file. In either case, read operations are complete at node 550.

D. File Close Request -- File Shadowing Mode

Fig. 5D illustrates operation of the shadow means and the direct read means in response to a file close request issued by the application program at node 560. From there, operation continues at

conditional block 562, where the direct read means determines whether the file has been decompressed.

If it has not been decompressed, operation passes to block 564, where the shadow means issues requests for the operating system to close and delete the shadow file. Note that if the file was never decompressed, this is because it never became "dirty", or, in other words, it was never written to. The only thing which block 564 will delete is a garbage-containing placeholder shadow file.

From block 564, operation passes to block 566, where the direct read means issues a request for the operating system to close the open direct read file. Thus, although the file was opened for read/write permission, it was only used for reading. In this situation, the history of the files is described by the sequence: Fig. 7A, Fig. 7B, Fig. 7C.

If, at conditional block 562, the file was determined to have been decompressed (i.e. written to), operation passes to block 568. At block 568, the shadow means issues a request for the operating system to close the open decompressed shadow file. In this situation, the history of the files is described by the sequence: Fig. 7A, Fig. 7B, Fig. 7D, Fig. 7E, Fig. 7F. Note that the file is left in a decompressed state. The compressor may subsequently be used to re-compress the file, if desired. This process could, optionally, be incorporated into the close operation, by making the compressor an integral part of the direct read system.

In either case, file close operations are complete at node 570.

V. OPERATION OF THE INVENTION IN FILE PUTBACK MODE

Figs. 6A-D illustrate file open, data write, data read, and file close operations of the direct read system when operating in file putback mode. In the interest of clarity, Figs. 6A-D do not show conditional blocks determining whether the request involves a direct read file. It will be understood that, as in Figs. 4A-D, if other than a direct read file is involved, the direct read system will hand off the request for normal operations by the operating system.

Direct read operations in file putback mode are performed by the direct read means, just as in normal read-only direct read mode. Additional features of file putback mode are provided by the putback means, and will be described below, with reference to Figs. 6A-D and Figs. 8A-F.

A. File Open Request -- File Putback Mode

Fig. 6A shows operations of the direct read system in file putback mode. File putback mode is entered at node 602 when an application requests to open a file for read/write permission in circumstances indicating that the application will most likely write to the file.

At block 604, the direct read system issues a request for the operating system to open the compressed file. Then, at block 606, the putback means determines the decompressed (expanded) size of the entire file. Fig. 8A shows the compressed data file in its original location.

Then, at block 608, the putback means issues a request for the operating system to open a putback file of this decompressed size, in the same location. Then, operation continues at block 610,

where the decompressor decompresses the entire data file into the putback file, as seen in Fig. 8B.

After decompression, operation passes to block 612, where the putback means issues requests for the operating system to close the original compressed data file and move it to the temporary location, as in Fig. 8C.

File open operations are complete at node 614.

B. Data Write Request -- File Putback Mode

Fig. 6B illustrates operations of the putback means in response to a data write request issued by the application program at node 620. Operation passes to block 622, where the putback means issues a request for the operating system to write the indicated data to the decompressed putback file, rather than to the compressed data file as originally indicated by the application's request. Write operations are complete at node 624.

C. Date Read Request -- File Putback Mode

Fig. 6C illustrates operations of the putback means in response to a data read request issued by the application program at node 630. Operation passes to block 632, where the putback means issues a request for the operating system to read the indicated data from the decompressed putback file, rather than from the compressed data file. Read operations are complete at node 634.

D. File Close Request -- File Putback Mode

Fig. 6D illustrates operation of the putback means and direct read means in response to a file close request. At node 640, the application program issues the request. Operation continues at conditional block 642, where the putback means determines whether the file has been written to.

If the file has not been written to, operation passes to block 644, where the putback means issues a request for the operating system to move the compressed data file from the temporary location back to its original location, as seen in Fig. 8D. Then, at block 648, the putback means issues a request for the operating system to delete the decompressed putback file, as seen in Fig. 8E.

If the application did not write to the file, the history of the files is described by the sequence: Fig. 8A, Fig. 8B, Fig. 8C, Fig. 8D, Fig. 8E.

If, at conditional block 642, the putback means determines that the application program wrote to the file, operation passes to block 650, where the putback means issues a request for the operating system to close the open decompressed putback file. Then, at block 652, the putback means issues a request for the operating system to delete the compressed data file from the temporary location, as seen in Fig. 8F.

If the application wrote to the file, the history of the files is described by the sequence: Fig. 8A, Fig. 8B, Fig. 8C, Fig. 8F.

Thus, what is "put back" into the original location will be either the original compressed data file or the decompressed data file, depending on whether the application program wrote any data to the file. In either case, file close operations are complete at node 654.

VI. CACHING

It is highly undesirable that the apparatus and method would have to fetch and decompress an entire

block of data each time an application program requested a single byte, for example.

Fig. 1 shows that, in order to prevent this, the direct read system includes caching means 160 for storing one or more decompressed data blocks for each open file.

Fig. 2 illustrates this feature. The first time an application program requests data from a file, the direct read means must rely on the operating system to fetch a compressed block from the permanent storage medium, and must further rely on the decompressor to decompress the block into an output buffer. Then, the requested data are provided to the application program by the direct read system.

On a subsequent request for data in the same block, the direct read system need not invoke the operating system nor the decompressor, because the block will already be stored in a decompressed form in one of the particular file's cache blocks. Thus, the direct read system receives the data read request, and immediately provides the requested data to the application program from the cache block.

In one mode, a block is cached only if both: it is the last block involved in a data request, and it contains data beyond that requested. In other words, a block is cached if at least one byte in the block's latter portion was not yet requested. This mode is selected based upon the likelihood that once an application program has accessed an entire data block, the application program is less likely to make subsequent requests for the same data again. It is further selected based upon the likelihood that, once an application program has requested

given data in a block, there is a high likelihood that immediately subsequent data in the block will later be requested by the application program.

In one mode, up to twelve cache blocks are maintained for each file, as needed. It is believed unlikely that an application program will read partial blocks from more than twelve locations in a file, in a manner such that caching of additional blocks would be advantageous.

VII. ACCESS TO INDIVIDUAL BLOCKS

A. Use of Block Tables

As shown in Fig. 9, and according to the file format taught in the cross-referenced applications, a compressed data file includes a block table which provides access to the individual blocks of the compressed data file.

Although each compressed block *represents* the same quantity of non-compressed data, such as eight kilobytes, each compressed block does not necessarily *include* the same quantity of compressed data.

Therefore, the block table entries may contain offsets from to the beginnings of the respective compressed data blocks. The offsets may be measured from the beginning of the file as a whole, for example. Thus, in response to a data read request specifying "read X bytes starting at location Y in file Z", the data structure means 134 and the locator means 136 may access the block table of file Z to gain access to the particular blocks effectively containing bytes Y through Y+X-1 of the file.

In one mode, the data structure means 134 and the locator means 136 may sequentially scan the block table entries until it finds the block containing byte Y, and the block containing byte Y+X-1. The data structure means and the locator means must, of course, take into account the size of the global file header, if the block table entries indicate offsets from the absolute beginning of the file, rather than from the beginning of block 0.

It will be understood that, if the data access request specifies less than the entire data file, the locator means may identify less than all blocks in the compressed data file. The decompressor will then selectively decompress only those identified blocks, without decompressing other, non-specified blocks.

In one mode, the block table merely points to the beginning locations of the respective blocks. In this mode, the data structure means and the locator means must further consult the block headers to determine how much non-compressed data each block effectively holds, in order to determine what blocks are involved in a given data access request. In another mode, the block table entries may each further include a copy of the expanded data size field from the corresponding block header. In this mode, the data structure means and the locator means need not consult the block headers in order to build data structures providing access to the respective blocks.

As is explained in the cross-referenced applications, the block table can be reconstructed if that portion of the file is damaged. The global file header is a known, fixed size. Therefore, the

block header for block 0 begins at a known, fixed position in the file. Block header 0 contains a field indicating the compressed size of block 0. Therefore, the location of the beginning of block header 1 can readily be determined, and so forth. Therefore, the locator means may include means for reconstructing a damaged block table from the block headers.

As is further indicated in the cross-referenced applications, each block header includes checksums generated over both the block and the block header itself. This information may be utilized to reconstruct at least some portions of a block table which has been damaged, even if one or more of the blocks and block headers have been damaged. Therefore, the locator means may include means for reconstructing some or most of a damaged block table from damaged blocks and block headers.

B. Data Structures

Those skilled in the art will appreciate that the following description of the source code and of the data structures is for teaching purposes only, and that the source code speaks for itself.

As is seen in the attached Software Source Code Appendix, each direct read file will have an associated "FileStruct" which contains the data structure entries for the particular file. Each "FileStruct" includes "DRCommonItems", and may optionally include other items as well.

The FileStruct structures for read-only direct read files and shadow files are stored in a doubly-linked list, as indicated by the items "*next" and "*prev" in "DRCommonItems". Putback files and their associated compressed data files are

handled within a simple array (not shown), which is kept separate from the direct read and shadow files for purposes of speed. Other modes are, of course, feasible within the scope of this invention. When a direct read file is opened, the data structure means creates for it a "FileStruct" and adds it to the doubly-linked list.

The item "algorithm" indicates whether the file is compressed, and if so, by what method. When a direct read file is opened, the data structure means sets the "algorithm" item according to what compression method was used, as indicated by the file's global file header.

The items "logicalEOF" and "logicalFPos" are used to keep track of where the application program thinks it is in a non-compressed data file. These values are used and updated by the direct read means and the shadow means.

The item "shadowFileRef" ties a direct read file to a shadow file. The item "compFileRef" ties a shadow file back to its compressed data file. These values are used and set by the shadow means.

The item "ddHeader" is filled by the data structure means according to a global file header in a given direct read file, and can be understood from the cross-referenced applications. The shadow means uses a "expSize" field from the "ddHeader" to determine how large a shadow file the operating system should be requested to create.

The item "**offsets" is a handle to a "PairOffsetsStruct", and is filled by the data structure means according to the offsets table of the particular direct read file. Specifically, the item "offsets[]" is a dynamically-allocated array of

offsets to the various data blocks. The locator means uses the "offsets[]" to identify which compressed block(s) in a file are involved in a given data access request.

The item "cacheList" points to a doubly-linked list of cache blocks for the file, and is maintained and used by the caching means. Each cache block is contained in a "CacheBlock" as shown. The cache blocks for a given file are stored in a doubly-linked list by the items "**next" and "**prev". The cached block's bytes are stored in the dynamically-allocated item "bytes[]".

The usage of the other items will be understood from the source code itself.

VIII. CONCLUSION

While the invention has been described with reference to the embodiment of a data compression system, its use in any suitable embodiment is contemplated. Various aspects of the invention's cooperation with an operating system may be incorporated directly into the direct read system itself. For example, the direct read system could directly move or delete files, rather than using the more conventional method of using an available operating system. It will also be understood that the terms "application program" and "operating system" may be used somewhat interchangeably.

These and various other changes in the form and details discussed herein are possible within the scope of this disclosure, including but not limited to those discussed above. However, the invention is to be afforded protection according to the scope of the following Claims.

METHOD AND APPARATUS FOR PERFORMING DIRECT READ OF COMPRESSED DATA FILE

Inventor:
Lloyd Lamont Chambers IV

SOFTWARE SOURCE CODE APPENDIX

Copyright 1991 Salient Software, Inc.
All Rights Reserved Except as Otherwise Disclaimed

```
/*    ©1991  Salient,
**    All  Rights  Reserved
**
**    Salient  Software,  Inc.
**    124  University  Ave.,  Suite  300
**    Palo  Alto,  CA   94301
**     (415)-321-5375
*/


/* The structure that describes the number and offsets to blocks.
** Keep this structure a multiple of 4 bytes! */
typedef struct
{ ulong  numBlocks;
  ulong  numBlocks2;
  ulong  blockSize;    /* or 0 for default */
  uchar  offsetsChecksum;
  uchar  unused[3];
  long  offsets[];
} PairOffsetsStruct;


/* The header that precedes compressed data.
** IMPORTANT: Keep the size of this structure a multiple of 4 bytes */
typedef struct
{ ushort  compSize;
  ushort  expSize;
  ulong  futureUse;
  uchar  compDataChecksum;
  uchar  reserved : 2;
  uchar  repaired : 1;  /* was this block repaired? */
  uchar  compressorUsed : 2;  /* if Pairs, which compressor was used */
  uchar  compSizePad : 2; /* true compressed size is 'compSize' - 'compSizePad' */
  uchar  copied : 1; /* set if the data was copied instead of compressed */
  uchar  expDataChecksum; /* optional */
  uchar  headerChecksum;
} BlockHeader;


enum
{ DirectReadKind = 3,
  ShadowKind
};


#define DRCommonItems \
  union _FileStruct_  *next;\
  union _FileStruct_  *prev;\
  short  fileRef;\
  short  kind;\
  Boolean  closed;\
  Boolean  naughty; \
  Boolean  allowAsync;\
  Boolean  shadowHosed;\
  OSType  appType;\
  uchar  appName[32];\
  short  appRefNum;


typedef struct _CacheBlock_
{ struct _CacheBlock_  **next;
  struct _CacheBlock_  **prev;
  long  index;  /* < 0 if invalid */
  uchar  bytes[];  /* of appropriate size */
} CacheBlock, **CacheBlockHandle;
```

© 1991 Salient Software, Inc.

```
/* Direct Read files */
typedef struct
{ DRCommonItems
  ushort  algorithm : 8;  /* NoCompression, PairsCompression, PairsSortCompression */
  ushort  resourceFork : 1;  /* TRUE if this is a resource fork */
  ushort  unused : 7;
  ulong  noCompressionOffset;  /* where to start reading if raw data */
  ulong  logicalEOF;
  ulong  logicalFPos;
  long  blockSize;
  short  shadowFileRef;  /* file ref num of shadow file */
  ulong  lastAccessTime;
  DDHeader  ddHeader;
  PairOffsetsStruct  **offsets;
  ulong  cacheSize;
  short  numCacheBlocks;
  CacheBlockHandle  cacheList;
} DRFileStruct, *DRFilePtr;

typedef struct
{ DRCommonItems
  short  dataForkRef;
  short  resForkRef;
  Boolean  ourDataFork;  /* TRUE if it's only us who owns fork */
  Boolean  ourResFork;    /* TRUE if it's only us who owns fork */
  short  compFileRef;
} ShadowFileStruct, *ShadowFilePtr;

typedef struct
{ DRCommonItems
} CommonFileStruct;

/* generic type shared between OpenFileStruct, ShadowFileStruct */
typedef union _FileStruct_
{ CommonFileStruct  common;
  DRFileStruct  drFile;
  ShadowFileStruct  shadowFile;
} FileStruct, *FilePtr;

OSErr
DirectReadInit()
{ OSErr  err;
  ulong  inSize, outSize;
  HaveFlushTrap = TrapAvailable(_HWPriv);
  err = DLInit();  /* allocate this first because it will never move! */
  if (err != noErr)
    return(err);
  inSize = PAIRS_MAX_COMP_SIZE(PAIRS_MAX_BLOCK_SIZE);
  outSize = PAIRS_MAX_BLOCK_SIZE + PAIRS_MAX_OVERWRITE;
  ReserveMem(outSize);  /* enough to accomodate a whole block + spillover */
  TempOutPtr = NewPtr(outSize);
  err = TempInInit(inSize);
  if (err != noErr)
    return(err);
  return(noErr);
}  /* routine DirectReadInit */

OSErr
ReadAtOffset(short fileRef, long *count, uchar *buffer, long offset)
{ OSErr  err;
  err = SetFPosAD(fileRef, fsFromStart, offset);
  if (err != noErr)
    return(err);
  err = FSReadAD(fileRef, count, buffer);
  return(err);
}  /* routine ReadAtOffset */
```

© 1991 Salient Software, Inc.

```
/* Things to check: (1) file already open writable or shared (2) volume locked */
OSErr
DirectSetupForFileRef(short fileRef, Boolean resourceFork, DRFilePtr *result)
{ DRFilePtr  info;
  OSErr  err;
  DDHeader  header;
  ulong  count;
  BlockOffsetsStruct  **offsets;
  ulong  offsetsSize;
  ulong  pos;
  ulong  dataLength;
  short  alg;
  *result = NIL;
  /* read the main file header */
  count = sizeof(header);
  err = FSReadAD(fileRef, (long *)&count, &header);
  if (err != noErr)
    return(err);
  /* Now check to see if this file is appropriate or not */
  if ((err = DirectCanRead(&header)) != noErr)
    return(err);
  if (resourceFork)
  { dataLength = header.resForkLength;
    alg = header.resAlgorithm;
  }
  else
  { dataLength = header.dataForkLength;
    alg = header.dataAlgorithm;
  }
  if (dataLength == 0)
    alg = NoCompression;
  info = (void *)DLAllocFile();
  if (info == NIL)
  { MakeMoreMemory(sizeof(FileStruct));
    info = (void *)DLAllocFile();
    if (info == NIL)
      return(mFulErr); /* File Managers' mFulErr, not mem mgr's memFullErr */
  }
  (*info).kind = DirectReadKind;
  (*info).algorithm = alg;
  (*info).resourceFork = resourceFork;
  (*info).ddHeader = header;
  (*info).fileRef = fileRef;
  /* set default block sizes */
  if (alg == PairsCompression)
  { (*info).blockSize = PAIRS_DLT_BLOCK_SIZE;
    (*info).cacheSize = PAIRS_DLT_BLOCK_SIZE + PAIRS_MAX_OVERWRITE;
  }
  else if (alg == PairsSortCompression)
  { (*info).blockSize = PAIRS_SORT_BLOCK_SIZE;
    (*info).cacheSize = PAIRS_SORT_BLOCK_SIZE + PAIRS_MAX_OVERWRITE;
  }
  (*info).logicalEOF = dataLength;
  if ((*info).logicalEOF == 0)  /* an empty fork */
  { pos = 0;
    offsetsSize = 0;
  }
  else if ((*info).algorithm == NoCompression)
  { pos = sizeof(DDHeader);  /* read the length */
    if (resourceFork)
      pos += header.compDataForkLength;
    offsetsSize = 0;  /* one long for start of data */
  }
  else  /* regular case */
  { pos = sizeof(header) + header.compDataForkLength -
            sizeof(long); /* read length */
```

© 1991 Salient Software, Inc.

```
    if (resourceFork)
      pos += header.compResForkLength;
    count = sizeof(offsetsSize);
    err = ReadAtOffset((*info).fileRef, (long *)&count,
                       (void *)&offsetsSize, pos);
    if (err != noErr)
    { DLFreeFile((FilePtr)info);
      return(err);
    }
  }
  count = offsetsSize;
  offsets = (void *)NewHandle(offsetsSize);
  if (offsets == NIL)
  { MakeMoreMemory(offsetsSize);
    offsets = (void *)NewHandle(offsetsSize);
  }
  (*info).offsets = offsets;
  if ((err != noErr) || (offsets == NIL))
  { DLFreeFile((FilePtr)info);
    return(memFullErr);
  }
  if(dataLength != 0 && ((*info).algorithm == NoCompression))
    (*info).noCompressionOffset = pos;
  else if (dataLength != 0)  /* normal compressed case */
  { uchar offsetsChecksum;
    pos -= offsetsSize;  /* reposition to read the offsets */
    err = ReadAtOffset((*info).fileRef, (long *)&count, (void *)*offsets, pos);
    if (err != noErr)
    { DisposHandle(offsets);
      DLFreeFile((FilePtr)info);
      return(err);
    }
    /* check to see if offsets are valid */
    offsetsChecksum = 0;
    ComputeXORChecksum((void *)&(**offsets).offsets[0],
          offsetsSize - sizeof(**offsets), &offsetsChecksum);
    if (offsetsChecksum != (**offsets).offsetsChecksum)
    { DisposHandle(offsets);
      DLFreeFile((FilePtr)info);
      return(checksumError);
    }
  }  /* else */
  /* If there is a block size specified, use it instead of default. */
  if ((**offsets).blockSize != 0)
  { Boolean badBlockSize = FALSE;
    (*info).blockSize = (**offsets).blockSize;
    if (alg == PairsCompression)
    { if ((**offsets).blockSize != PAIRS_DLT_BLOCK_SIZE)
        badBlockSize = TRUE;
      (*info).cacheSize = (**offsets).blockSize + PAIRS_MAX_OVERWRITE;
    }
    else if (alg == PairsSortCompression)
    { if ((**offsets).blockSize > PAIRS_MAX_BLOCK_SIZE)
        badBlockSize = TRUE;
      (*info).cacheSize = (**offsets).blockSize + PAIRS_MAX_OVERWRITE;
    }
    if (badBlockSize)
    { DirectClose(info);
      return(internalError);
    }
  }
  TempInRealloc(PAIRS_MAX_BLOCK_SIZE*2L); /* try to leave enough for about 2 files */
  *result = info;
  return(noErr);
}  /* routine DirectSetupForFileRef */
```

© 1991 Salient Software, Inc.

```
OSErr
DirectClose(DRFilePtr file) /* caller should take care of closing file */
{ if ((*file).kind != DirectReadKind)
    return(-1);
  TempInFlush(file);
  if ((*file).offsets != NIL)
    DisposHandle((*file).offsets);
  DisposeCacheBlocks(file);
  Nullify(file, sizeof(*file));
  DLFreeFile((FilePtr)file);
  return(noErr);
} /* routine DirectClose */


/* Read bytes from the file. Shouldn't be called directly, because it does not
** account for newline mode or other variations in parameter block. */
OSErr _DirectRead(DRFilePtr file, long *count, uchar *destBuffer);

  OSErr DirectRead(DRFilePtr file, long *count, uchar *destBuffer)
  { OSErr err;
    err = _DirectRead(file, count, destBuffer);
    if (HaveFlushTrap)
      FlushInstructionCache(); /* flushes both caches on an MC68040 */
    return(err);
  } /* nested routine DirectRead */


static OSErr
_DirectRead(DRFilePtr file, long *count, uchar *destBuffer)
{ ulong  startLogPos;
  long  blockIndex, firstBlock, lastBlock;
  long  destBufferAvail;
  long  destBufferUsed;
  OSErr  err;
  Boolean  mustReturnEOF = FALSE;
  if (*count == 0)  /* handle stupid case */
    return(noErr);
  if ((*file).logicalEOF == 0)  /* empty file */
  { *count = 0;
    return(eofErr);
  }
  startLogPos = DirectGetFPos(file);
  /* If the read would go beyond the EOF, read as much as we can,
  ** but return an eofErr. */
  if (startLogPos + *count > DirectGetEOF(file))
  { *count = 0;
    if (startLogPos < DirectGetEOF(file))
      *count = DirectGetEOF(file) - startLogPos;
    mustReturnEOF = TRUE;
  }
  if ((*file).algorithm == NoCompression)  /* read raw uncompressed data */
  { err = ReadAtOffset((*file).fileRef, count, destBuffer,
            startLogPos + (*file).noCompressionOffset);
    DirectSetFPos(file, startLogPos + *count);
    if (mustReturnEOF)
      return(eofErr);
    return(err);
  }
  firstBlock = startLogPos / (*file).blockSize;
  lastBlock = (startLogPos + (*count) -1) / (*file).blockSize;
  destBufferAvail = *count;
  destBufferUsed = 0;
  (*file).lastAccessTime = AccessTime++;
  /* read all the bytes in all the blocks */
  for (blockIndex = firstBlock; blockIndex <= lastBlock; ++blockIndex)
```

© 1991 Salient Software, Inc.

```
{ uchar  *dest;
  long  fromOffset;
  long  advanceCount;
  uchar  *data;
  Boolean  moveData;
  CacheBlockHandle  cacheBlock;
  fromOffset = (startLogPos + destBufferUsed) % (*file).blockSize;
  advanceCount = (*file).blockSize - fromOffset;
  if (advanceCount > destBufferAvail)
    advanceCount = destBufferAvail;
  moveData = TRUE;
  /* fetch the data in this block */
  if ((cacheBlock = CacheValid(file, blockIndex)) != NIL)
    data = &(**cacheBlock).bytes[0];
  else
  { data = TempOutPtr;
    /* if a partial block and it's the last one, put it into the cache */
    if ((blockIndex == lastBlock) && (advanceCount != (*file).blockSize))
    { cacheBlock = DirectReallocCache(file);
      if (cacheBlock != NIL)
      { (**cacheBlock).index = blockIndex;
        data = &(**cacheBlock).bytes[0];
      }
    }
    err = TempInGetBlock(file, blockIndex, lastBlock, data);
    if (err != noErr)
      return(err);
  }  /* else */
  if (moveData)
    MyBlockMove(data + fromOffset, destBuffer + destBufferUsed,
          advanceCount);
  destBufferAvail -= advanceCount;
  destBufferUsed += advanceCount;
  DirectSetFPos(file, startLogPos + destBufferUsed);
 }  /* for */
 if (mustReturnEOF)
  return(eofErr);
 return(noErr);
}  /* routine DirectRead */


OSErr
DirectSetFPos(DRFilePtr file, ulong newPos)
{ if (newPos > DirectGetEOF(file))
  { (*file).logicalFPos = DirectGetEOF(file);
    return(eofErr);
  }
  (*file).logicalFPos = newPos;
  return(noErr);
}  /* routine DirectSetFPos */


ulong
DirectGetFPos(DRFilePtr  file)
{ return((*file).logicalFPos);
}  /* routine DirectGetFPos */


ulong
DirectGetEOF(DRFilePtr  file)
{ return((*file).logicalEOF);
}  /* routine DirectGetEOF */
```

© 1991 Salient Software, Inc.

```
static DRFilePtr
FindOldestFileWithCache(ulong  minCacheSize)
{ DRFilePtr  oldestFile = NIL;
  DRFilePtr  cur;
  cur = (void *)DLGetList();
  while (cur != NIL)
  { DRFilePtr  file;
    file = (void *)cur;
    if ((*cur).kind == DirectReadKind &&
        (*file).numCacheBlocks != 0 &&
        (*file).cacheSize >= minCacheSize)
    { if (oldestFile == NIL)
        oldestFile = file;
      else if ((*file).lastAccessTime < (*oldestFile).lastAccessTime)
        oldestFile = file;
    }
    cur = (void *)cur->next;
  }  /* while */
  return(oldestFile);
}  /* routine FindOldestFileWithCache */


/* Memory is needed. Purge cache blocks until enough is available.
** This is a slow, expensive operation and should be called only if necessary. */
void
DirectPurgeCaches(ulong  requiredBlock)
{ while (MaxBlock() < requiredBlock)
  { DRFilePtr  oldestFile;
    CacheBlockHandle  cacheBlock;
    oldestFile = FindOldestFileWithCache(0);
    if (oldestFile == NIL)
      break;
    DisposeCacheBlocks(oldestFile); /* ahhh...trash them all for speed */
  }
}  /* routine DirectPurgeCaches */


/* find oldest (last) cache block in the list */
static CacheBlockHandle
GetOldestCacheBlock(DRFilePtr file, Bcolean remove)
{ CacheBlockHandle  cur, last;
  cur = last = (*file).cacheList;
  while (cur != NIL)
  { last = cur;
    cur = (**cur).next;
  }
  if (last != NIL && remove)
  { CacheBlockHandle  prev;
    prev = (**last).prev;
    if (prev != NIL)
      (**prev).next = NIL;
    else
      (*file).cacheList = NIL;
    (*file).numCacheBlocks--;
  }
  return(last);
}  /* routine GetOldestCacheBlock */
```

© 1991 Salient Software, Inc.

```
/* Allocate another cache block for this file if possible.
** Return its index, or -1 if not possible.
*/
static CacheBlockHandle
DirectReallocCache(DRFilePtr  file)
{ CacheBlockHandle  newBlock;
  if ((*file).numCacheBlocks == MAX_CACHE_BLOCKS)
    /* If we already have the max num of cache blocks, use the oldest one */
    newBlock = GetOldestCacheBlock(file, TRUE);
  else
  { ulong allocSize = (*file).cacheSize + sizeof(CacheBlock);
    newBlock = NewHandle(allocSize);
    if (newBlock == NIL)
    { TempInRealloc(allocSize);
      newBlock = NewHandle(allocSize);
    }
    if (newBlock == NIL)  /* take cache away from other file */
    { DirectPurgeCaches(allocSize);
      newBlock = NewHandle(allocSize);
    }
  }  /* else */
  if (newBlock == NIL)
    return(NIL);
  /* insert new cache block at front of list */
  if ((*file).cacheList != NIL)
    (**(*file).cacheList).prev = newBlock;
  (**newBlock).next = (*file).cacheList;
  (**newBlock).prev = NIL;
  (*file).cacheList = newBlock;
  (*file).numCacheBlocks++;
  return(newBlock);
}  /* routine DirectReallocCache */


static CacheBlockHandle
CacheValid(DRFilePtr  file, long blockIndex)
{ register CacheBlockHandle  cur;
  cur = (*file).cacheList;
  while (cur != NIL)
  { if ((**cur).index == blockIndex)  /* move this one to front of list */
    { if (cur != (*file).cacheList)  /* remove cur from list */
      { (**(**cur).prev).next = (**cur).next;
        if ((**cur).next != NIL)
          (**(**cur).next).prev = (**cur).prev;
        (**cur).next = (*file).cacheList;  /* insert cur at front... */
        (**(*file).cacheList).prev = cur;
        (**cur).prev = NIL;
        (*file).cacheList = cur;
      }
      return(cur);
    }
    cur = (**cur).next;
  }  /* while */
  return(NIL);
}  /* routine CacheValid */
```

© 1991 Salient Software, Inc.

```
static void
DisposeCacheBlocks(DRFilePtr  file)
{ register CacheBlockHandle  cur;
  cur = (*file).cacheList;
  while (cur != NIL)
  { CacheBlockHandle  temp;
    temp = (**cur).next;
    DisposHandle(cur);
    cur = temp;
  }
  (*file).cacheList = NIL;
  (*file).numCacheBlocks = 0;
}  /* routine DisposeCacheBlocks */


void
MakeMoreMemory(ulong   requiredSpace)
{ /* next, get rid of excessive size input buffer */
  if (MaxBlock() < requiredSpace)
  { TempInForceToMin();
    /* finally, get rid of caches until we have enough space */
    if (MaxBlock() < requiredSpace)
      DirectPurgeCaches(requiredSpace);
  }
}  /* routine MakeMoreMemory */


typedef struct
{ Boolean  busy;
  short  unused;  /* to keep 4-byte aligned */
  ulong  size;
  ulong  minSize;
  long  startBlockIndex;
  long  stopBlockIndex;
  DRFilePtr  file;
  uchar  **bytes;
} TempInStruct;
static TempInStruct  TempIn;
static Boolean  Have68020;

void  _TempInFlush(DRFilePtr file);

 OSErr
  TempInInit(ulong   minSize)
  { long  result;
    Have68020 = FALSE;
    if (Gestalt(gestaltProcessorType, &result) == noErr)
      Have68020 = (result >= gestalt68020);
    TempIn.size = minSize;
    TempIn.minSize = minSize;
    ReserveMem(minSize);
    TempIn.bytes = NewHandle(minSize);
    if (TempIn.bytes != NIL)
      HLock(TempIn.bytes);
    return((TempIn.bytes != NIL) ? noErr:memFullErr);
  }  /* nested routine TempInInit */
```

© 1991 Salient Software, Inc.

```
/* Increase size of TempIn, leave at least 'leaveFreeAmount' bytes. */
OSErr
TempInRealloc(long  leaveFreeAmount)
{ long  maxBlock;
  long  maxSize;
  long  newSize;
  if (TempIn.busy)
    return(fBsyErr);
  /* don't bother if it's already as small as possible */
  if (TempIn.minSize == TempIn.size)
    return(noErr);
  TempIn.busy = TRUE;
  _TempInFlush(TempIn.file);
  /* get rid of excess data */
  HUnlock(TempIn.bytes);
  SetHandleSize(TempIn.bytes, TempIn.minSize);
  TempIn.size = GetHandleSize(TempIn.bytes);
  newSize = TempIn.minSize;
  maxBlock = MaxBlock();
  if (maxBlock > leaveFreeAmount)
    newSize = maxBlock - leaveFreeAmount;
  if (newSize > 128L * 1024L)
    newSize = 128L * 1024L;
  if (newSize > TempIn.minSize)
  { SetHandleSize(TempIn.bytes, newSize);
    TempIn.size = GetHandleSize(TempIn.bytes);
  }
  HLock(TempIn.bytes);
  TempIn.busy = FALSE;
  return(noErr);
}  /* nested routine TempInRealloc */


OSErr
TempInForceToMin()
{ return(TempInRealloc(1024L * 1024L));
}


/* flush without checking busy flag */
static void
_TempInFlush(DRFilePtr   file)
{ if (file == NIL || TempIn.file == file)
  { TempIn.startBlockIndex = -1;
    TempIn.stopBlockIndex = -1;
    TempIn.file = NIL;
  }
}  /* routine _TempInFlush */


OSErr
TempInFlush(DRFilePtr  file)
{ if (TempIn.file == file)
  { if (TempIn.busy)
      return(fBsyErr);
    TempIn.busy = TRUE;
    _TempInFlush(file);
    TempIn.busy = FALSE;
  }
  return(noErr);
}  /* routine TempInFlush */
```

© 1991 Salient Software, Inc.

```
OSErr
TempInGetBlock(DRFilePtr file, long desiredIndex, long lastDesiredIndex, uchar *block)
{ long  readStartPos;
  long  count;
  OSErr  err;
  long  index;
  uchar  *inBuf;
  if (TempIn.busy)
    return(fBsyErr);
  TempIn.busy = TRUE;
  readStartPos = (**(*file).offsets).offsets[desiredIndex];
  if (TempIn.file == file &&
      desiredIndex >= TempIn.startBlockIndex &&
      desiredIndex <= TempIn.stopBlockIndex) /* block already in our cache */
  { ulong  offset;
    offset = readStartPos - (**(*file).offsets).offsets[TempIn.startBlockIndex];
    inBuf = &(*TempIn.bytes)[offset];
  }
  else  /* must read fresh data */
  { uchar checksum;
    BlockHeader  *blockHeader;
    _TempInFlush(TempIn.file);  /* reset now in case of failure */
    /* figure out how many blocks we can read into buffer and still fit */
    index = desiredIndex;
    count = 0;
    while (index <= lastDesiredIndex)
    { long  newCount;
      newCount = (**(*file).offsets).offsets[index+1] - readStartPos;
      if (newCount > TempIn.size)
        break;
      count = newCount;
      ++index;
    }
    --index;
    err = ReadAtOffset((*file).fileRef, &count, *TempIn.bytes, readStartPos);
    if (err != noErr)
    { TempIn.busy = FALSE;
      return(err);
    }
    TempIn.startBlockIndex = desiredIndex;
    TempIn.stopBlockIndex = index;
    TempIn.file = file;
    inBuf = &(*TempIn.bytes)[0];
    checksum = 0;  /* check to see if header is valid */
    blockHeader = (void *)inBuf;
    ComputeXORChecksum((void *)blockHeader,
        sizeof(BlockHeader) - sizeof(blockHeader->headerChecksum), &checksum);
    if (checksum != blockHeader->headerChecksum)
    { TempIn.busy = FALSE;
      return(checksumError);
    }
  }  /* else */
  if ((*file).algorithm == PairsCompression ||  /* decompress the block */
      (*file).algorithm == PairsSortCompression)
  { if (Have68020)
      err = PairsDecompressBlock68020((PairsBlockHeader *)inBuf,
            inBuf + sizeof(PairsBlockHeader), block);
    else
      err = PairsDecompressBlock68000((PairsBlockHeader *)inBuf,
            inBuf + sizeof(PairsBlockHeader), block);
  }
  TempIn.busy = FALSE;
  if (err != noErr)
    return(err);
  return(noErr);
}  /* routine TempInGetBlock */
```

© 1991 Salient Software, Inc.

CLAIMS

I Claim:

1. A method of providing compressed data from a compressed data file in response to a data request, the data request identifying given non-compressed data in a non-compressed data file, the method comprising the steps of:

identifying, within in the compressed data file, given compressed data which correspond to the given non-compressed data identified by the data request, the given compressed data comprising less than all data in the compressed data file;

decompressing the given compressed data to reconstitute the given non-compressed data; and

providing the reconstituted non-compressed data in response to the data request.

2. The method of Claim 1, further comprising the steps of:

caching the reconstituted non-compressed data; and

in response to a second, subsequent data request, skipping the decompressing step.

3. The method of Claim 1, wherein the method is performed by a computer system having random access memory (RAM) and having permanent storage storing the compressed data file, and wherein the decompressing step comprises the step of:

decompressing the given compressed data into the RAM.

4. The method of Claim 1, wherein the compressed data file includes a plurality of independently-compressed blocks of compressed data, the blocks including one or more first blocks which include compressed data corresponding to the non-compressed data specified by the request and which further include one or more second blocks, and wherein:

the step of identifying includes identifying the one or more first blocks from among the plurality of blocks;

the step of decompressing includes decompressing the one or more first blocks; and

the step of providing includes providing one or more bytes from the decompressed first blocks.

5. The method of Claim 4, wherein the compressed data file further includes a table of entries indicating locations at which the blocks begin in the compressed data file, and wherein the step of identifying includes the step of:

accessing the table of entries to identify an initial one of the first blocks and a final one of the first blocks.

6. The method of Claim 1, further comprising the steps of:

in response to a request to open the compressed data file, creating a shadow file;

in response to a request to read from the compressed data file,

if the shadow file has not been written to, reading from the compressed data file, and

if the shadow file has been written to, reading from the shadow file;

in response to a request to write to the compressed data file,

if the shadow file has not been written to, decompressing the compressed data file into the shadow file and writing to the shadow file and deleting the compressed data file, and

if the shadow file has been written to, writing to the shadow file; and

in response to a request to close the compressed data file,

if the shadow file has not been written to, deleting the shadow file and closing the compressed data file, and

if the shadow file has been written to, closing the shadow file.

7. The method of Claim 1, further comprising the steps of:

in response to a request to open the compressed data file, creating a putback file and decompressing the compressed data file into the putback file;

in response to a request to read from the compressed data file, reading from the putback file;

in response to a request to write to the compressed data file, writing to the putback file and,

if the putback file has not been written to, deleting the compressed data file; and

in response to a request to close the compressed data file,

if the putback file has not been written to, deleting the putback file, and

if the putback file has been written to, closing the putback file.

8. A method of reading data from a compressed data file in response to a data request, the compressed data file including blocks of compressed data each corresponding to respective non-compressed data in a non-compressed data file, the data request identifying given non-compressed data in the non-compressed data file, the blocks including one or more first blocks containing compressed data corresponding to the given data and further including one or more second blocks, the method comprising the steps of:

identifying, within the compressed data file, the one or more first blocks;

responsive to the identifying step, selectively decompressing the one or more first blocks; and

responsive to the selectively decompressing step, providing decompressed data from the decompressed first blocks as the given non-compressed data.

9. The method of Claim 8, further comprising the steps of:

caching the reconstituted non-compressed data; and

in response to a second, subsequent data request, skipping the decompressing step.

10. The method of Claim 8, wherein the method is performed by a computer system having random access memory (RAM) and having permanent storage storing the compressed data file, and wherein the decompressing step comprises the step of:

decompressing the given compressed data into the RAM.

11. The method of Claim 8, wherein the compressed data file further includes a table of entries indicating locations at which the blocks begin in the compressed data file, and wherein the step of identifying includes the step of:

accessing the table of entries to identify an initial one of the first blocks and a final one of the first blocks.

12. The method of Claim 11, wherein the data request specifies a starting location and an ending location of requested data, and wherein one or more entries in the table each identifies an offset from the beginning of the compressed data file, at which offset a corresponding block begins, and wherein the identifying step comprises the steps of:

determining a first entry identifying an offset corresponding to a location less than or equal to the starting location of the requested data;

determining a second entry identifying an offset corresponding to a location greater than or equal to the ending location; and

identifying all blocks corresponding to entries from the first entry to the second entry.

13. The method of Claim 8, further comprising the steps of:

creating a shadow file;

decompressing the compressed file into the shadow file and discarding the compressed data file in

response to a request for writing to the non-compressed data file; and

closing the shadow file in response to a request for closing the non-compressed data file.

14. The method of Claim 8, further comprising the steps of:

creating a shadow file; and

discarding the shadow file and closing the compressed data file in response to a request for closing the non-compressed data file.

15. The method of Claim 8, further comprising the steps of:

creating a putback file;

decompressing the compressed data file into the putback file; and

discarding the putback file and closing the compressed data file in response to a request for closing the non-compressed data file.

16. The method of Claim 8, further comprising the steps of:

creating a putback file;

decompressing the compressed data file into the putback file; and

writing to the putback file and discarding the compressed data file in response to a request for writing to the non-compressed data file.

17. A method of reading data from a compressed data file in response to a data request, the compressed data file residing on a permanent storage medium and including independently-compressed blocks

of compressed data, the data request specifying a starting location and an ending location of non-compressed data to be read from the permanent storage medium into random access memory (RAM), the method comprising the steps of:

locating a first block of the independently-compressed blocks, which first block contains compressed data corresponding to non-compressed data at the starting location;

decompressing the first block from the permanent storage medium into RAM;

if the first block does not contain compressed data corresponding to non-compressed data at the ending location,

locating one or more additional blocks of the independently-compressed blocks up to a second block, the additional blocks containing compressed data corresponding to non-compressed data at locations up to the ending location, the second block including non-compressed data at the ending location, and

decompressing the additional blocks from the permanent storage medium into RAM; and

providing, as the non-compressed data, decompressed data from a first location in RAM to a second location in RAM.

18. The method of Claim 17, further comprising the steps of:

caching the decompressed blocks in RAM; and

in response to a second, subsequent data request, skipping the decompressing steps.

19. The method of Claim 17, wherein the compressed data file further includes a table of entries indicating locations at which the blocks begin in the compressed data file, and wherein the locating steps includes the step of:

accessing the table of entries to identify the first block and the additional blocks.

20. The method of Claim 19, wherein the data request specifies a starting location and an ending location of requested data, and wherein one or more entries in the table each identifies an offset from the beginning of the compressed data file, at which offset a corresponding block begins, and wherein the locating steps comprise the steps of:

determining a first entry identifying an offset corresponding to a location less than or equal to the starting location of the requested data;

determining a second entry identifying an offset corresponding to a location greater than or equal to the ending location; and

identifying all blocks corresponding to entries from the first entry to the second entry.

21. The method of Claim 17, further comprising the step of:

performing file shadowing.

22. The method of Claim 17, further comprising the step of:

performing file putback.

23. An apparatus for performing direct read operations on a compressed data file, the compressed

data file, the apparatus receiving a data access request for the compressed data file, the apparatus comprising:

compression checker means for determining whether the compressed data file includes a file format having independently-compressed data blocks;

data structure means, responsive to the compression checker means determining that the file includes the file format, for constructing a data structure providing individual access to the data blocks;

locator means, responsive to the data access request, for accessing the data structure to identify one or more of the blocks as including data locations involved in the data access request; and

decompressor means for decompressing the identified one or more blocks, and for providing access to data in the decompressed blocks in response to the data access request.

24. The apparatus of Claim 23, further comprising:

file shadowing means for shadowing the compressed data file during read access requests.

25. The apparatus of Claim 23, further comprising:

file putback means for restoring the compressed data file after read access requests.

26. A data decompressor providing an application program transparent access to given data in a compressed data file such that the application program may issue data access requests as though the

given data were in a non-compressed data file, the data compressor comprising:

means for identifying a proper subset of the compressed data file containing given compressed data corresponding to the given data; and

means for decompressing the proper subset of the compressed data.

27. The data decompressor of Claim 26, wherein the compressed data file includes a plurality of independently-compressed blocks, the blocks including one or more first blocks containing the given compressed data and further including one or more second blocks, and wherein the means for identifying further comprises:

means for determining which of the plurality of blocks are the first blocks.

28. The data decompressor of Claim 27, wherein:

the means for decompressing decompresses the first blocks to constitute a portion of decompressed data, wherein the portion includes the given data and other data; and

the means for decompressing includes means for providing only the given data from the portion.

29. The data decompressor of Claim 28, further comprising:

means for caching the portion in random access memory; and

means, responsive to a subsequent request for second data in the portion, for providing the second data directly from the cached portion.

30. A recording medium having recorded thereon a computer program providing an application program transparent access to given data in a compressed data file such that the application program may issue data access requests as though the given data were in a non-compressed data file, the recording medium comprising:

instructions for identifying a proper subset of the compressed data file containing given compressed data corresponding to the given data; and

instructions for decompressing the proper subset of the compressed data.

31. The recording medium of Claim 30, wherein the compressed data file includes a plurality of independently-compressed blocks, the blocks including one or more first blocks containing the given compressed data and further including one or more second blocks, and wherein the instructions for identifying comprise:

instructions for determining which of the plurality of blocks are the first blocks.

32. The recording medium of Claim 31, wherein the instructions for decompressing comprise:

instructions for decompressing the first blocks to constitute a portion of decompressed data, the portion including the given data and other data; and

instructions for providing only the given data from the portion.

33. The recording medium of Claim 32, further comprising:

instructions for caching the portion in random access memory; and

instructions, responsive to a subsequent request for second data in the portion, for providing the second data directly from the cached portion.

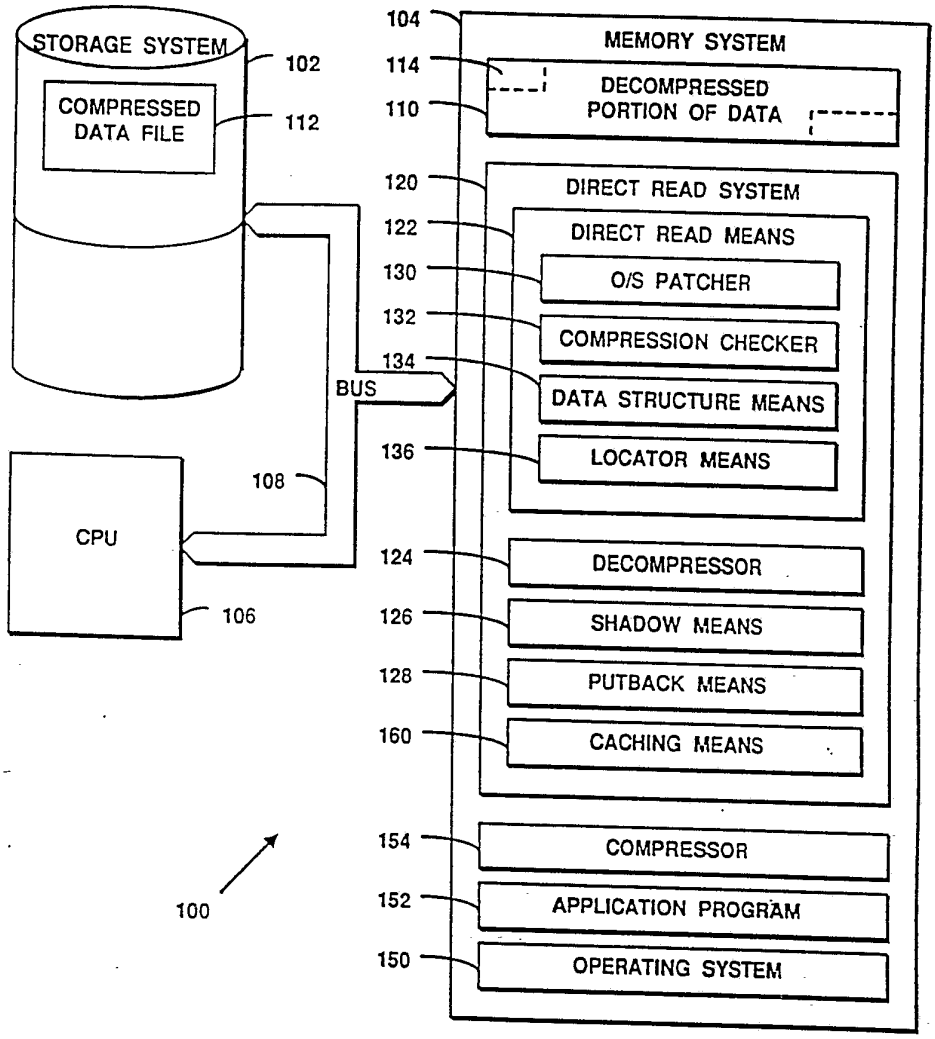

STORAGE SYSTEM
102
COMPRESSED DATA FILE
112
BUS
108
CPU
106
100
104
MEMORY SYSTEM
114
DECOMPRESSED PORTION OF DATA
110
120
DIRECT READ SYSTEM
122
DIRECT READ MEANS
130
O/S PATCHER
132
COMPRESSION CHECKER
134
DATA STRUCTURE MEANS
136
LOCATOR MEANS
124
DECOMPRESSOR
126
SHADOW MEANS
128
PUTBACK MEANS
160
CACHING MEANS
154
COMPRESSOR
152
APPLICATION PROGRAM
150
OPERATING SYSTEM

Fig. 1

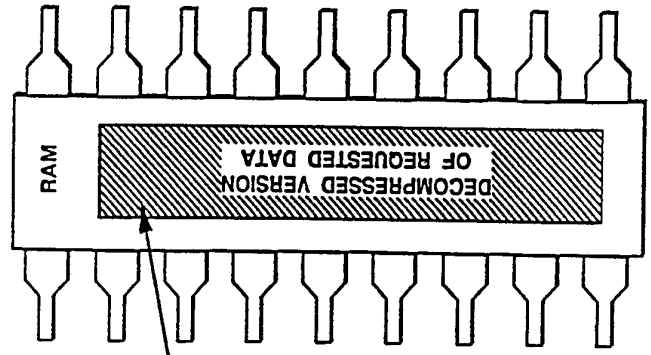


Fig. 3A

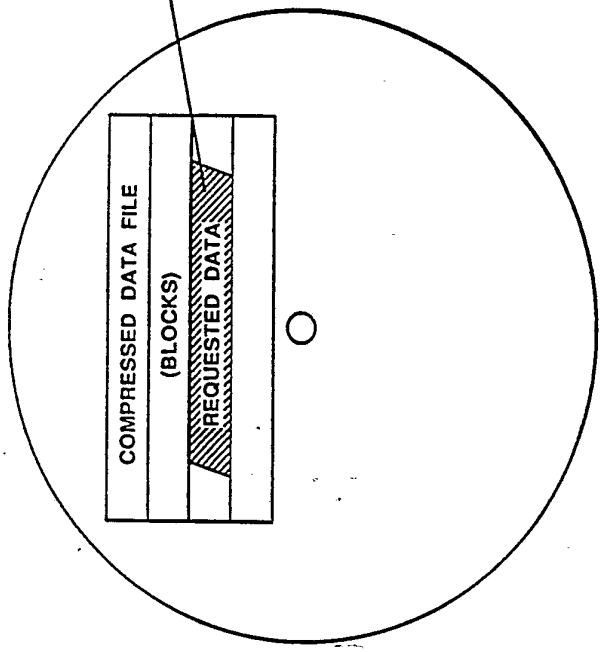


Fig. 3B

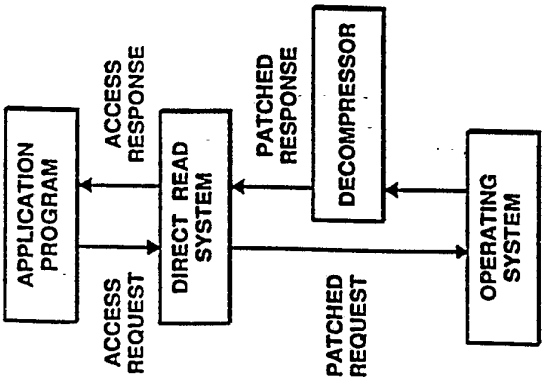


Fig. 2

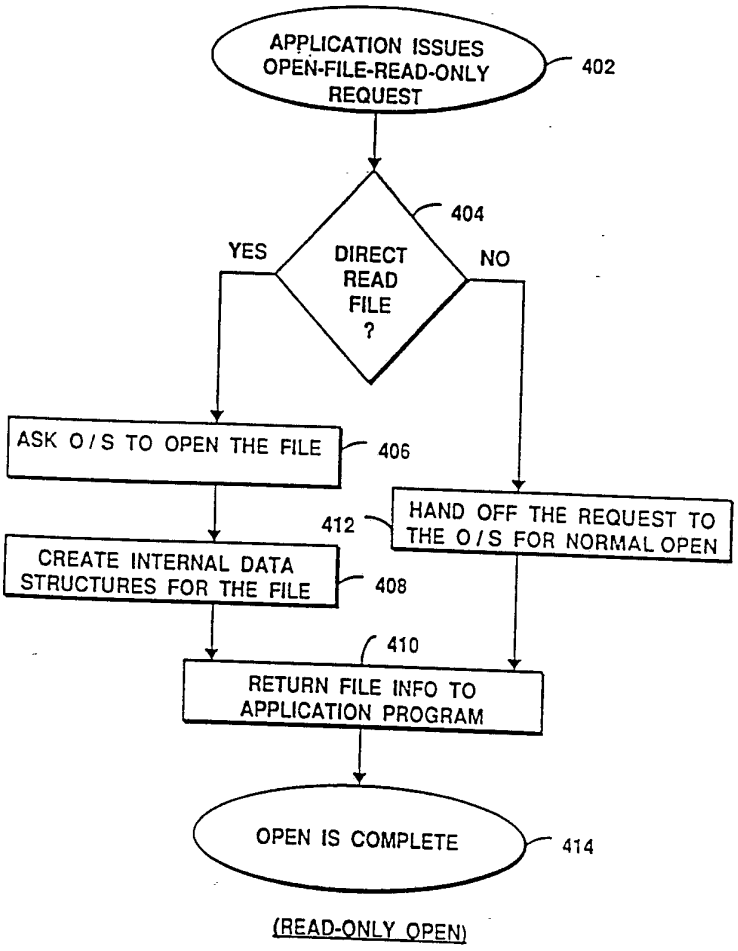

APPLICATION ISSUES OPEN-FILE-READ-ONLY REQUEST
402
404
DIRECT READ FILE ?
YES
NO
ASK O/S TO OPEN THE FILE
406
412
HAND OFF THE REQUEST TO THE O/S FOR NORMAL OPEN
CREATE INTERNAL DATA STRUCTURES FOR THE FILE
408
410
RETURN FILE INFO TO APPLICATION PROGRAM
OPEN IS COMPLETE
414
(READ-ONLY OPEN)

Fig. 4A

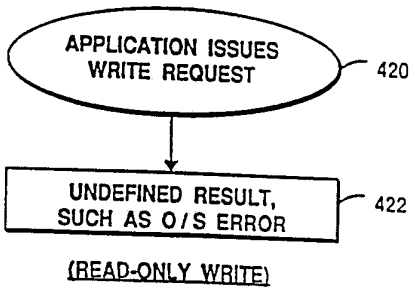

APPLICATION ISSUES WRITE REQUEST
420
UNDEFINED RESULT, SUCH AS O/S ERROR
422
(READ-ONLY WRITE)

Fig. 4B

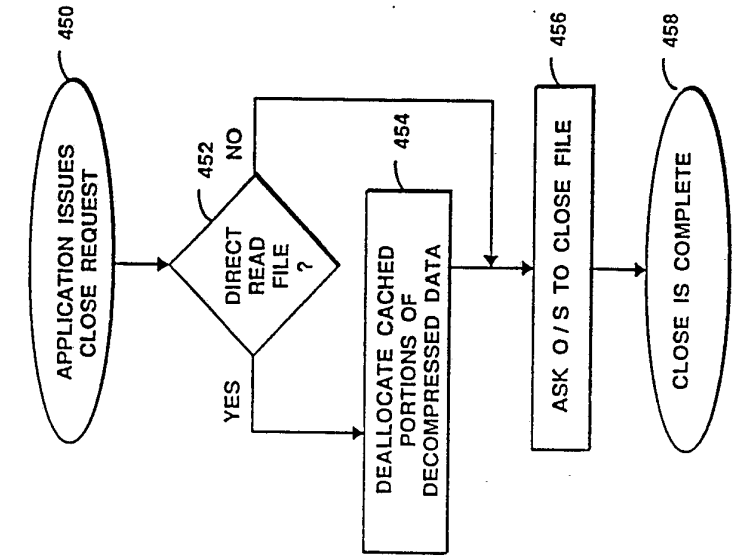

APPLICATION ISSUES CLOSE REQUEST
450
DIRECT READ FILE ?
452
NO
YES
DEALLOCATE CACHED PORTIONS OF DECOMPRESSED DATA
454
ASK O/S TO CLOSE FILE
456
CLOSE IS COMPLETE
458
(READ-ONLY CLOSE)

Fig. 4D

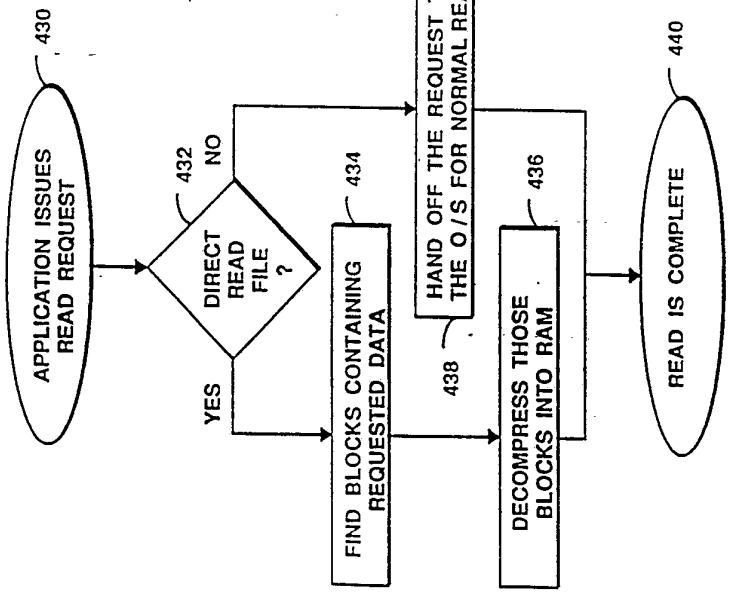

APPLICATION ISSUES READ REQUEST
430
DIRECT READ FILE ?
432
NO
YES
FIND BLOCKS CONTAINING REQUESTED DATA
434
HAND OFF THE REQUEST TO THE O/S FOR NORMAL READ
438
DECOMPRESS THOSE BLOCKS INTO RAM
436
READ IS COMPLETE
440
(READ-ONLY READ)

Fig. 4C

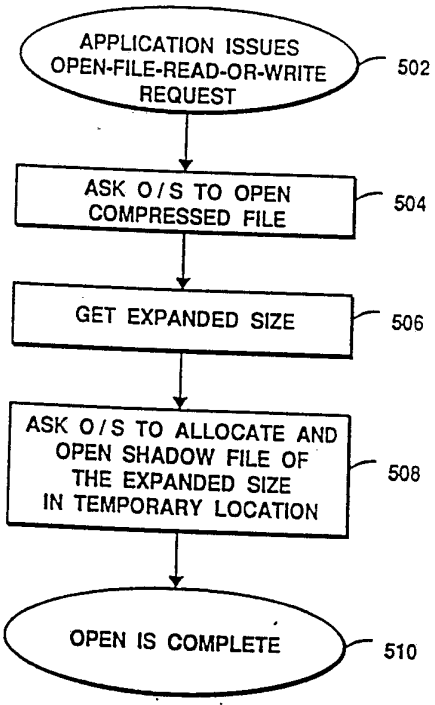

APPLICATION ISSUES OPEN-FILE-READ-OR-WRITE REQUEST
502
ASK O/S TO OPEN COMPRESSED FILE
504
GET EXPANDED SIZE
506
ASK O/S TO ALLOCATE AND OPEN SHADOW FILE OF THE EXPANDED SIZE IN TEMPORARY LOCATION
508
OPEN IS COMPLETE
510
(SHADOW OPEN)

Fig. 5A

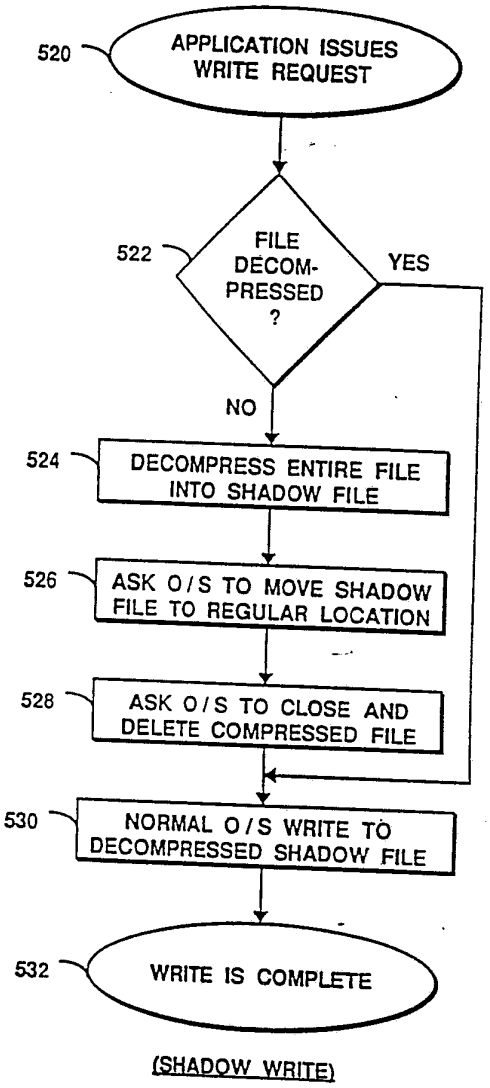

520
APPLICATION ISSUES WRITE REQUEST
522
FILE DECOM-PRESSED ?
YES
NO
524
DECOMPRESS ENTIRE FILE INTO SHADOW FILE
526
ASK O/S TO MOVE SHADOW FILE TO REGULAR LOCATION
528
ASK O/S TO CLOSE AND DELETE COMPRESSED FILE
530
NORMAL O/S WRITE TO DECOMPRESSED SHADOW FILE
532
WRITE IS COMPLETE
(SHADOW WRITE)

Fig. 5B

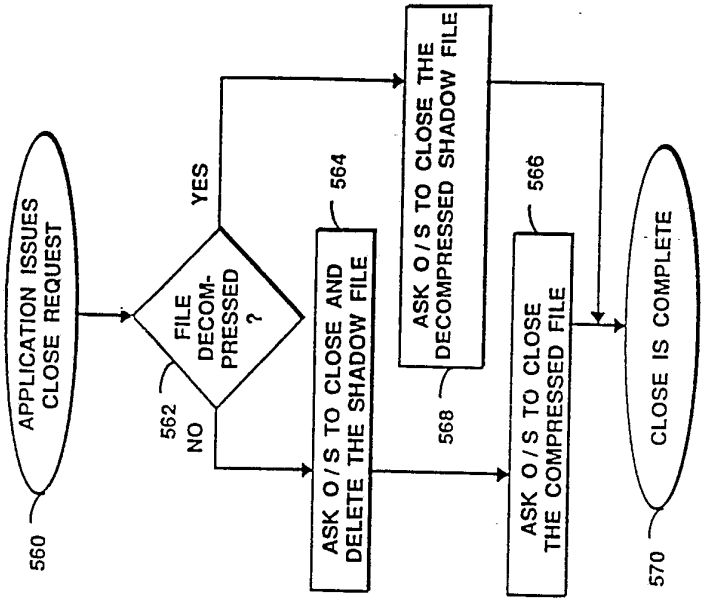

APPLICATION ISSUES CLOSE REQUEST
560
FILE DECOM-PRESSED ?
562
YES
NO
ASK O/S TO CLOSE AND DELETE THE SHADOW FILE
564
ASK O/S TO CLOSE THE DECOMPRESSED SHADOW FILE
568
ASK O/S TO CLOSE THE COMPRESSED FILE
566
CLOSE IS COMPLETE
570
(SHADOW CLOSE)

Fig. 5D

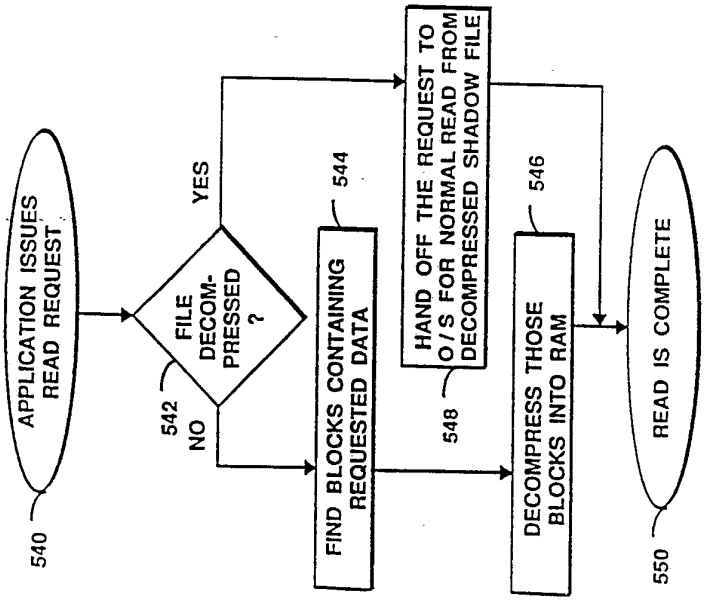

APPLICATION ISSUES READ REQUEST
540
FILE DECOM-PRESSED ?
542
YES
NO
FIND BLOCKS CONTAINING REQUESTED DATA
544
HAND OFF THE REQUEST TO O/S FOR NORMAL READ FROM DECOMPRESSED SHADOW FILE
548
DECOMPRESS THOSE BLOCKS INTO RAM
546
READ IS COMPLETE
550
(SHADOW READ)

Fig. 5C

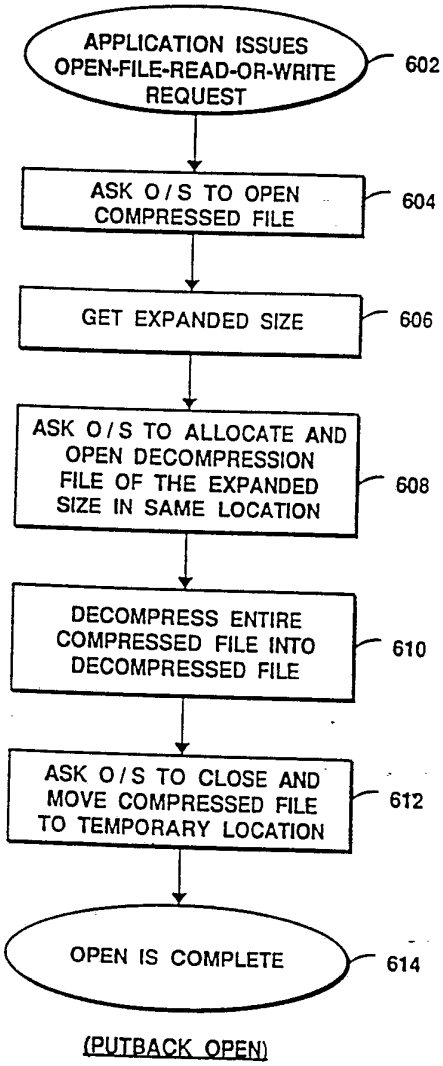


Fig. 6A

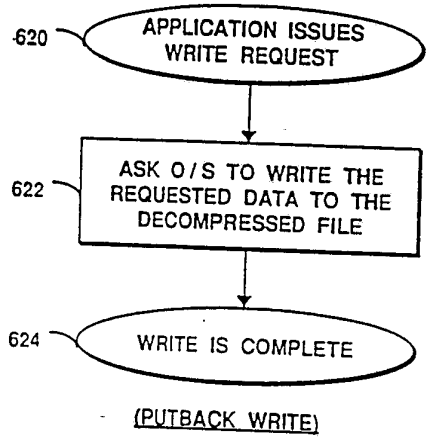


Fig. 6B

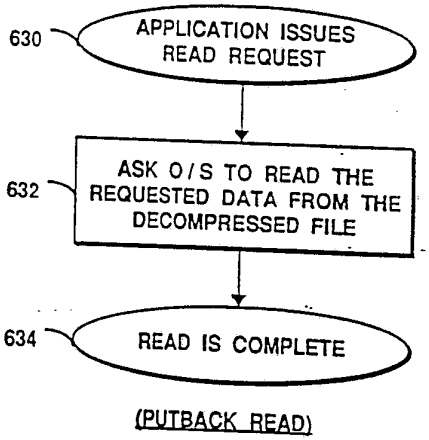


Fig. 6C

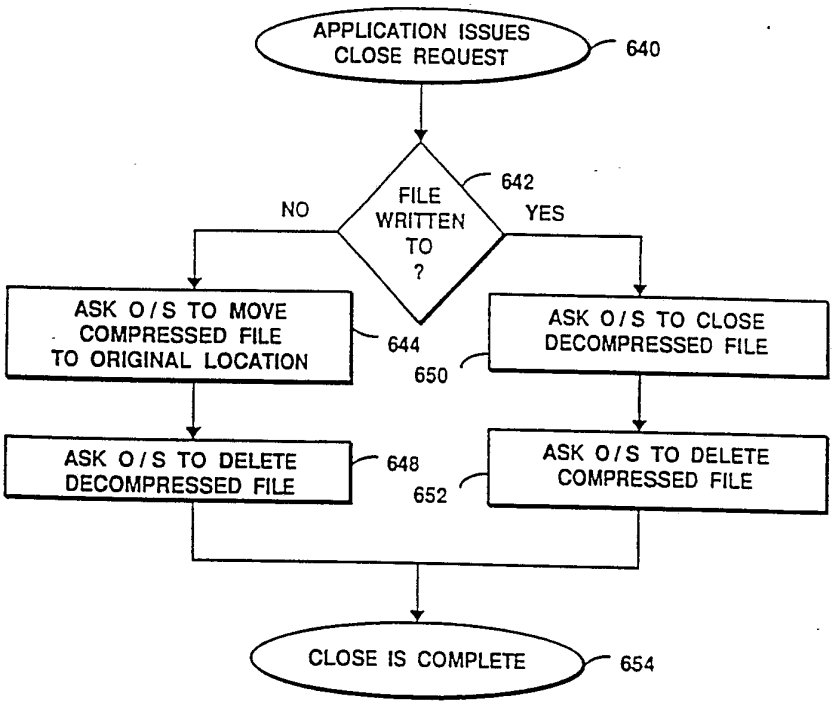

APPLICATION ISSUES CLOSE REQUEST
640
FILE WRITTEN TO ?
642
NO
YES
ASK O/S TO MOVE COMPRESSED FILE TO ORIGINAL LOCATION
644
ASK O/S TO DELETE DECOMPRESSED FILE
648
ASK O/S TO CLOSE DECOMPRESSED FILE
650
ASK O/S TO DELETE COMPRESSED FILE
652
CLOSE IS COMPLETE
654
(PUTBACK CLOSE)

Fig. 6D

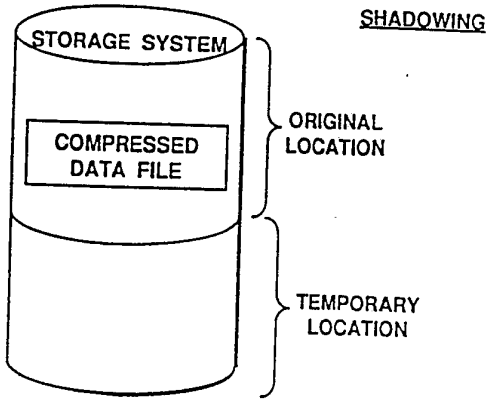


Fig. 7A

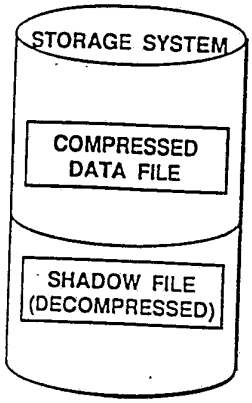


Fig. 7D

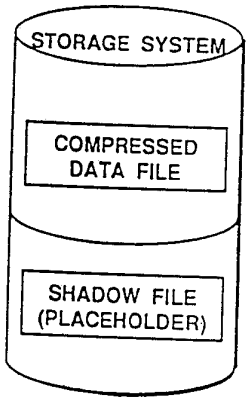


Fig. 7B

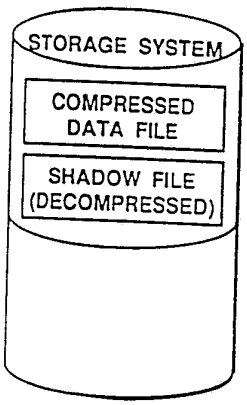


Fig. 7E

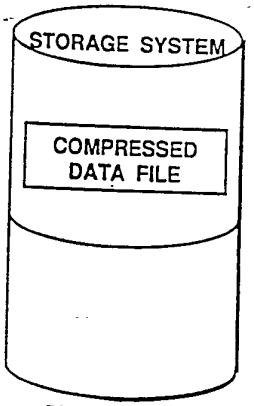


Fig. 7C

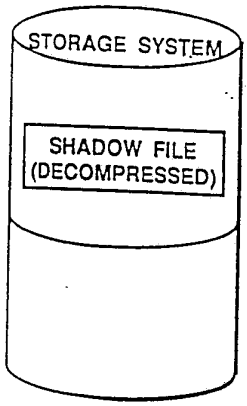


Fig. 7F

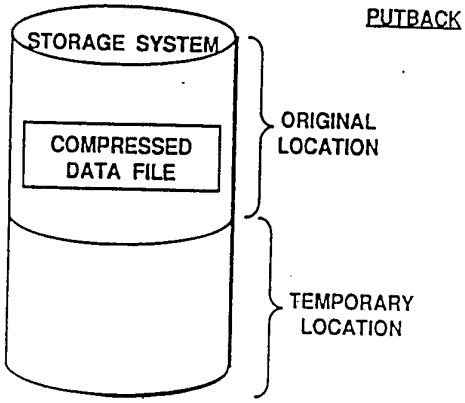


Fig. 8A

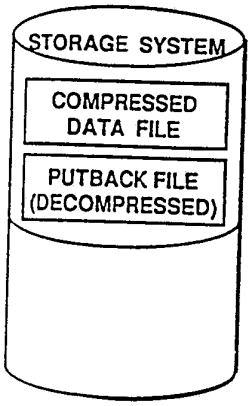


Fig. 8D

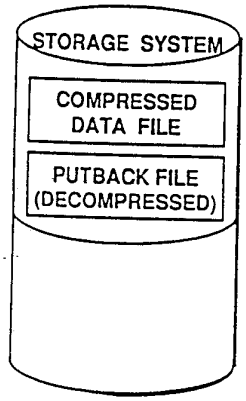


Fig. 8B

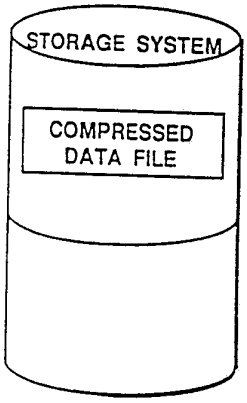


Fig. 8E

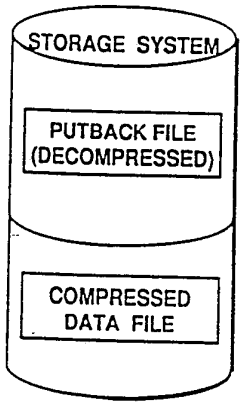


Fig. 8C

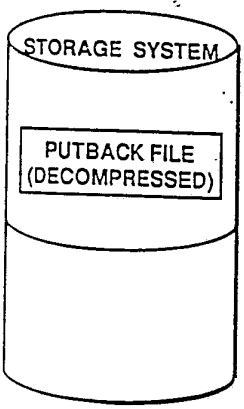


Fig. 8F

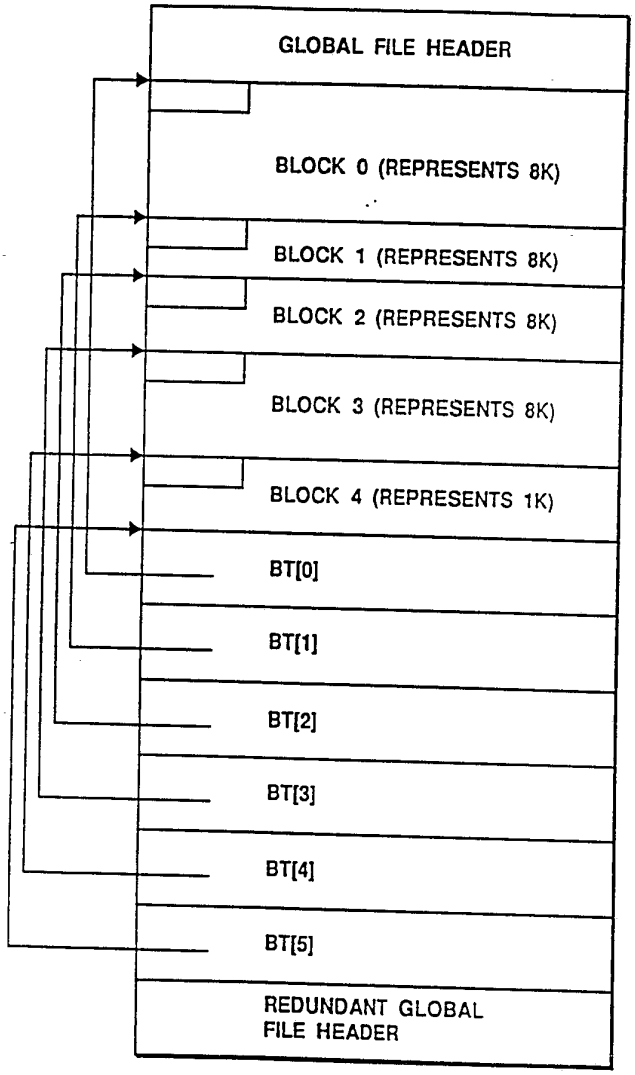

GLOBAL FILE HEADER
BLOCK 0 (REPRESENTS 8K)
BLOCK 1 (REPRESENTS 8K)
BLOCK 2 (REPRESENTS 8K)
BLOCK 3 (REPRESENTS 8K)
BLOCK 4 (REPRESENTS 1K)
BT[0]
BT[1]
BT[2]
BT[3]
BT[4]
BT[5]
REDUNDANT GLOBAL FILE HEADER

Fig. 9

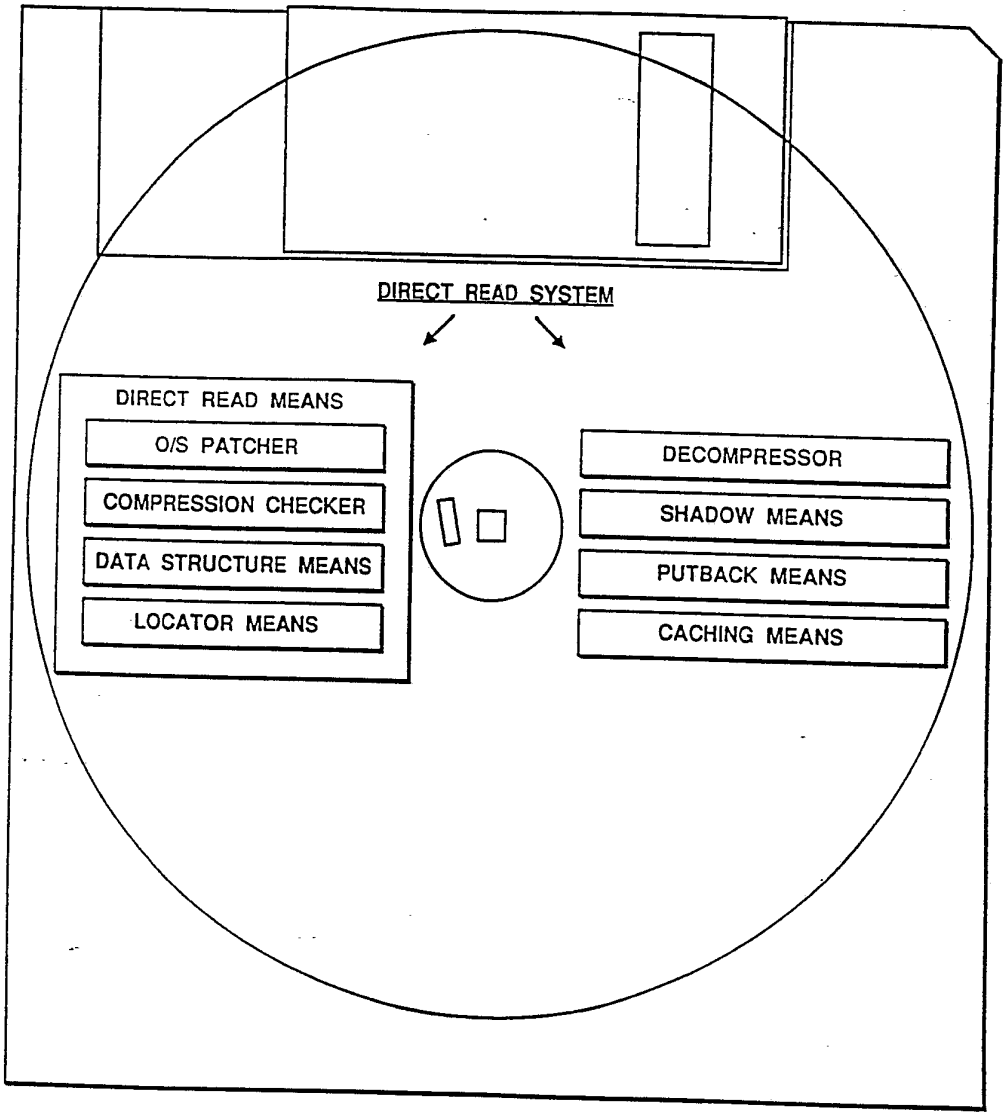

DIRECT READ SYSTEM
DIRECT READ MEANS
O/S PATCHER
COMPRESSION CHECKER
DATA STRUCTURE MEANS
LOCATOR MEANS
DECOMPRESSOR
SHADOW MEANS
PUTBACK MEANS
CACHING MEANS

Fig. 10

INTERNATIONAL SEARCH REPORT

International application No.
PCT/US93/01891

A. CLASSIFICATION OF SUBJECT MATTER

IPC(5) :G06F 15/40, 15/407, 5/00
US CL :345/600; 341/55;
According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)

U.S. : 395/2;395/250;

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Electronic data base consulted during the international search (name of data base and, where practicable, search terms used)

Dialog, 12 Data Bases, Decompress### Compress###; Record# or Block# or Subset# or Segment#; Data "Aivding" Some or all Terms.

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
Y	US,A 4,956,808 (Aakre et al.) 11 September 1990 Abstract, Figs 1, 2b; Col. 2, Col. 7-10.	1-3,8-10, 17-18
Y	US,A, 4,897,717 (Hamilton et al.) 30 January 1990 Abstract; Col. 4-6, 58-59.	1-3,8-10, 17-18
Y	US,A, 4,408,301 (Iida) 04 October 1983 Col. 11-12.	1-3,8-10, 17-18
Y	US,A, 4,467,421 (White) 21 August 1984 Abstract, col. 1; Fig. 3, 10; Claims 5, 24-25.	1-3,8-10, 17-18

[X] Further documents are listed in the continuation of Box C. [ ] See patent family annex.

*	Special categories of cited documents:	"T"	later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention
"A"	document defining the general state of the art which is not considered to be part of particular relevance	"X"	document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone
"E"	earlier document published on or after the international filing date	"Y"	document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art
"L"	document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)	"&"	document member of the same patent family
"O"	document referring to an oral disclosure, use, exhibition or other means		
"P"	document published prior to the international filing date but later than the priority date claimed		

Date of the actual completion of the international search 09 APRIL 1993	Date of mailing of the international search report 07 JUN 1993
Name and mailing address of the ISA/US Commissioner of Patents and Trademarks Box PCT Washington, D.C. 20231 Facsimile No. NOT APPLICABLE	Authorized officer PAUL R. LINTZ Telephone No. (703) 305-3832

Form PCT/ISA/210 (second sheet)(July 1992)*

INTERNATIONAL SEARCH REPORT

International application No.
PCT/US93/01891

C (Continuation). DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
T	US,A, 5,179,651 (Taaffee et al.) 12 January 1993 Abstract.	1-3,8-10, 17-18
T	US,A, 5,155,484 (Chambers IV) 13 October 1992 Abstract, Fig. 1,6; Cols, 30, 31.	1-5,8-12, 17-20
A	US,A, 4,782,325 (Jeppsson et al.) 01 November 1988 Abstract, Col. 1,2,21,22,28.	1,8,17