

UNITED STATES PATENT AND TRADEMARK OFFICE

BEFORE THE PATENT TRIAL AND APPEAL BOARD

NETAPP INC., RACKSPACE US INC.,
Petitioners

v.

REALTIME DATA LLC
Patent Owner.

Patent No. 7,378,992

Inter Partes Review No. IPR2017-_____

DECLARATION OF DANIEL HIRSCHBERG

I, Daniel Hirschberg, make this declaration in connection with the proceeding identified above.

I. Introduction

1. I have been retained by counsel for NetApp Inc. and Rackspace US Inc. as a technical expert in connection with the proceeding identified above. I submit this declaration in support of NetApp's and Rackspace's Petition for *Inter Partes* Review of United States Patent No. 7,378,992 ("the '992 patent").

2. I am being paid an hourly rate for my work on this matter. I do not have any personal or financial stake or interest in the outcome of the present proceeding.

II. Qualifications

3. My resume is attached to this declaration as Exhibit A.

4. I earned my Ph.D. in Computer Science from Princeton University in 1975. I also earned a MSE and MA from Princeton University in 1973. I also earned a BE in Electrical Engineering from City College of New York in 1971.

5. Since 2003, I have been a Professor of Computer Science and EECS at University of California, Irvine (UCI). Prior to that, I was a professor in various departments and held various other positions at UCI. I also held the position of Assistant Professor of Electrical Engineering at Rice University from 1975 through 1981. As a professor at UCI, I have taught courses in computer science topics, including a course covering compression techniques.

6. In addition to my roles with UCI and Rice University, I have also provided various consulting services over the years. For example, I have consulted on the design of compression/decompression techniques. I have also provided technical expert services in intellectual property cases covering various technologies, including compression.

7. I have also extensively published in the area of compression and participated in professional organizations and conferences focused on compression technologies. For example, publications nos. B2, J25, J29, J30, J35, J36, J43, J47, C15, C16, C19, C21, C22, and C31 all relate to lossless data compression.

III. Materials Considered

8. In preparing this declaration, I have reviewed, among other things, the following materials:

- a) the '992 patent;
- b) the prosecution history for the '992 patent, including reexamination prosecution history;
- c) NetApp's and Rackspace's petition for *inter partes* review of the '992 patent (the "Petition") to which my declaration relates (I generally agree with the statements regarding the technical disclosures and characterizations of the '992 patent and prior art contained in the Petition); and

d) the exhibits to the Petition (below, I use the names defined in the Petition's exhibit list to refer to the exhibits) and any other documents cited below.

IV. Legal Standards

A. Claim Construction

9. I have been informed that, when construing claim terms in an unexpired patent, a claim subject to post grant review receives the broadest reasonable construction in light of the specification of the patent in which it appears. I have also been informed that the '992 patent is likely to expire during any IPR proceeding instituted based on the Petition. I understand that under this circumstance, the claims terms are construed according to their plain meaning in light of the intrinsic record.

B. Obviousness

10. I understand that a patent claim may also be invalid if the claimed invention would have been obvious to a person of ordinary skill in the art at the time of the claim's effective filing date. I understand that an invention may be obvious if a person of ordinary skill in the art with knowledge of the prior art would have conceived the claimed invention, even if all of the limitations of the claim cannot be found in a single prior art reference.

11. I understand that, in assessing whether a claimed invention would have been obvious, the following factors are considered.

12. First, I understand that the level of ordinary skill that a person working in the field of the claimed invention would have had at its effective filing date must be considered.

13. Second, I understand that the scope and content of the prior art must be considered. I understand that, to be considered as prior art, a reference must be reasonably related to the claimed invention, which means that the reference is in the same field as the claimed invention or is from another field to which a person of ordinary skill in the art would refer to solve a known problem.

14. Third, I understand that the differences, if any, that existed between the prior art and the claimed invention must be considered. I understand that the determination of such differences should focus on the claimed invention as a whole.

15. I understand that it is not sufficient to prove a patent claim obvious to show that each of its limitations was independently known in the prior art but that there also must have been a reason for a person of ordinary skill in the art to have combined or modified the elements or concepts from the prior art in the same way as in the claimed invention.

16. In assessing whether such a reason existed, I understand that the following may be considered: (i) whether the combination or modification was merely the predictable result of using prior art elements according to their known functions; (ii) whether the claimed invention provided an obvious solution to a known problem in the art; (iii) whether the claimed invention applied a known technique that had been used to improve a similar device or method in a similar way; (iv) whether the prior art teaches or suggests making the combination or modification claimed in the patent; (v) whether the prior art teaches away from combining elements in the claimed invention; (vi) whether it would have been obvious to try the combination or modification, such as when there is a design need or market pressure to solve a problem and there are a finite number of identified, predictable solutions; and (vii) whether the combination or modification resulted from design incentives or other market forces.

17. I understand that, when considering whether a claimed invention was obvious, one should be careful not to use the benefit of hindsight and that, instead, one should put oneself in the position of a person of ordinary skill in the art as of the effective filing date of the claimed invention and should not consider what is known today or what is learned from the teaching of the patent.

V. The '992 Patent

18. The '992 patent, published May 27, 2007, is entitled "Content Independent Data Compression Method and System."

A. Priority Date

19. I understand that page 1 of the '992 patent include a priority chain with an earliest date of December 11, 1998 (for U.S. Patent No. 6,195,024 (the "'024 patent'")).

20. I understand that for the '992 patent to be entitled to a priority date of December 11, 1998, the specification of the application that issued as the '024 patent must have provided sufficient description of the claims of the '992 patent such that a person of ordinary skill in the art would have understood that the named inventors were in possession of the claimed technology.

21. I reviewed the application that issued as the '024 patent. The application only describes content independent data compression. The '992 patent's claims include both content independent data compression (i.e., a default encoder) and content dependent data compression (i.e., encoders associated with specific data types). Accordingly, a person of ordinary skill in the art reviewing the application filed on December 11, 1998 that issued as the '024 patent would not have understood the named inventor to have been in possession of the technology claimed in the '992 patent as of December 11, 1998.

22. The first time that the concept of content dependent data compression appears in any of the patent applications in the priority chain is October 29, 2001 when the application that issued as U.S. Patent No. 6,624,761 was filed. In that application, FIGS. 13-18 and associated text were added, which are the portions of the '992 patent that describe content dependent data compression. Accordingly, this is the earliest possible priority date for the claims of the '992 patent.

B. Generally

23. The '992 patent discloses data compression “using a combination of content independent data compression and content dependent data compression.” '992 patent at Abstract. FIGS. 13A and 13B, reproduced below, depict an example of the '992 patent's system.

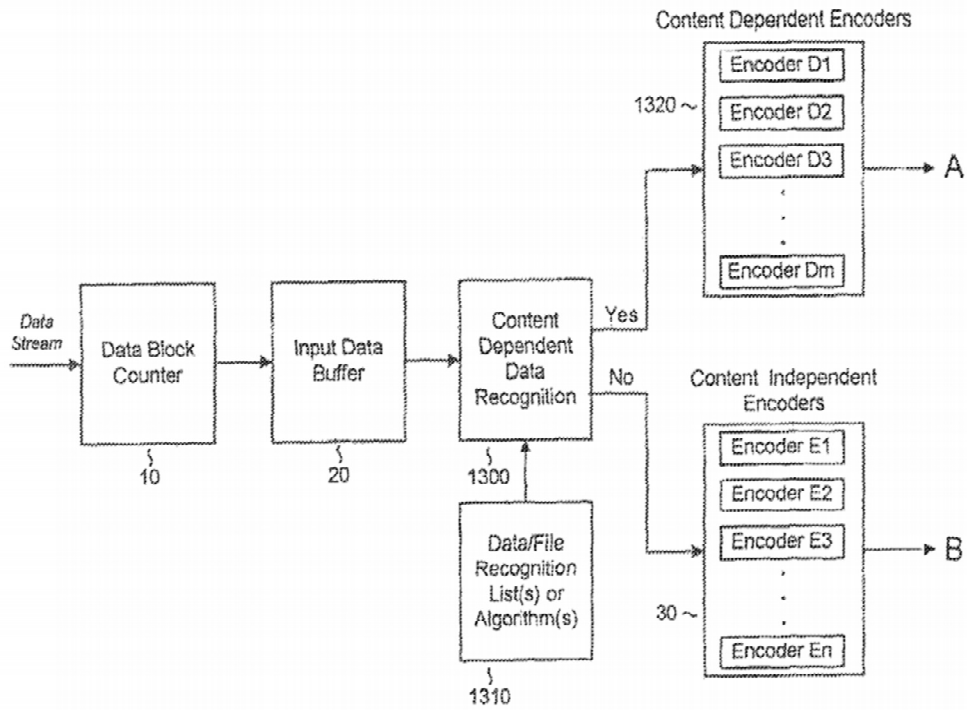


FIGURE 13A

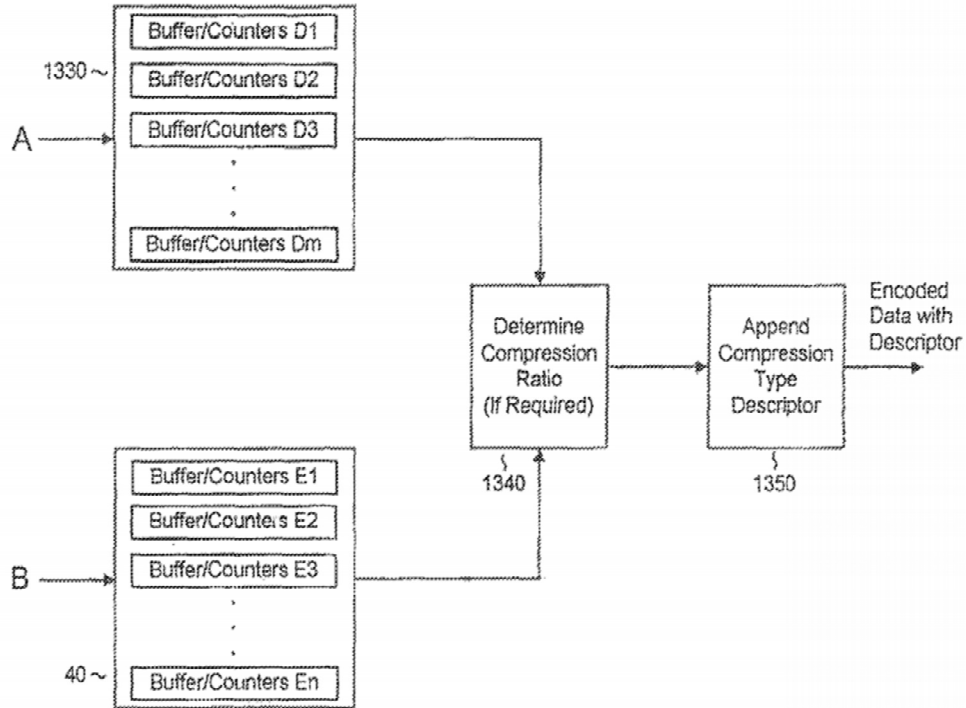


FIGURE 13B

24. The system described in the '992 patent includes an input data buffer that buffers a data stream after passing through a data block counter that counts the sizes of data blocks in the data stream. '992 patent at 16:7-26. The content dependent data recognition module "analyzes the incoming data stream to recognize data types, data structures, data block formats, file substructures, file types, and/or any other parameters that may be indicative of either the data type/content of a given data block or the appropriate data compression algorithm or algorithms (in serial or in parallel) to be applied." '992 patent at 16:27-34.
25. For each data block, if the above analysis recognizes the data block, the data block is routed to a content dependent encoder module. '992 patent at 16:36-38. If the analysis does not recognize the data block, the data block is sent to a content independent encoder module. '992 patent at 16:38-40.
26. In the content dependent encoder module, a data block is compressed using a subset of available encoders $D_1 \dots D_m$ producing compressed data block versions (e.g., $C_1 \dots C_m$). '992 patent at 16:41-56. The compression ratios (e.g., $R_1 \dots R_m$) are calculated for each of the compressed versions, where a ratio is the size of the uncompressed data block divided by the size of the compressed data block (i.e., a higher compression ratio indicates more compression). '992 patent at 17:49-57.
27. The compression encoder that produces the highest compression ratio is chosen. '992 patent at 19:10-27. If that highest compression ratio is less than a

minimum predefined threshold, then the original uncompressed data block is output with an associated descriptor Null (meaning no compression used).

'992 patent at 18:65-19:9. Otherwise, a compressed block with descriptor that identifies the compression technique that was used is output. '992 patent at 19:23-27.

B. State of the Art

28. In general, well before March 1999, the concepts described and claimed in the '992 patent were widely known and implemented in the computer industry.

For example, in D.A. Lelewer and D.S. Hirschberg, "Data compression," Computing Surveys 19:3 (1987) 261-297 ("Lelewer") (Ex. 1017), my co-author and I explain that the benefits of compression were recognized in many areas, such as communications and archival systems. It was also widely recognized that different types of compression techniques were more suitable for certain types of data.

Many of the methods discussed in this paper are implemented in production systems. The UNIX utilities compact and compress are based on methods discussed in Sections 4 and 5, respectively [UNIX 1984]. Popular file archival systems such as ARC and PKARC use techniques presented in Sections 3 and 5 [ARC 1986; PKARC 1987]. The savings achieved by data compression can be dramatic; reduction as high as 80% is not uncommon [Reghbati 1981]. Typical values of compression provided by compact are text (38%), Pascal source (43%), C source (36%), and binary (19%).

Compress generally achieves better compression (50-60% for text such as source code and English) and takes less time to compute [UNIX 1984]. (Lelewer at 2.)

29. Using different compression methods on different types of data blocks was also widely implemented long before the priority date of the '992 patent. For example, the commonly used and widely distributed program PKZIP supported using different compression techniques for different files. The specification for the PKZIP file format (<https://www.pkware.com/documents/APPNOTE/APPNOTE-1.0.txt> (attached as Exhibit B to this declaration)) describes a “compression method” field in the header that describes the compression method used for that file. As noted in the APPNOTE file for version 1.0, this version was released in 1990.

Overall zipfile format:

[local file header + file data + data_descriptor] . . .

[central directory] end of central directory record

A. Local file header:

local file header signature	4 bytes (0x04034b50)
version needed to extract	2 bytes
general purpose bit flag	2 bytes
compression method	2 bytes
last mod file time	2 bytes
last mod file date	2 bytes
crc-32	4 bytes
compressed size	4 bytes
uncompressed size	4 bytes
filename length	2 bytes
extra field length	2 bytes
filename	(variable size)
extra field	(variable size)

compression method: (2 bytes)

(see accompanying documentation for algorithm descriptions)

- 0 - The file is stored (no compression)
- 1 - The file is Shrunk
- 2 - The file is Reduced with compression factor 1
- 3 - The file is Reduced with compression factor 2
- 4 - The file is Reduced with compression factor 3
- 5 - The file is Reduced with compression factor 4
- 6 - The file is Imploded
- 7 - Reserved for Tokenizing compression algorithm
- 8 - The file is Deflated

30. PKZIP had the ability to choose file-specific compression algorithms. This allowed for the appropriate algorithm to be applied to each file in a directory.

31. The prior art described in more detail below provides more examples showing that the '992 patent's technology was well known as of the priority date of the '992 patent.

C. Person of Ordinary Skill in the Art

32. A person of ordinary skill in the art for the '992 patent in October 2001 would have had an undergraduate degree in computer science, computer engineering, electrical and computer engineering, or equivalent field and one to three years of experience working with data compression or a graduate degree with course work or research in the field of data compression. A person without the undergraduate or graduate degree described above would still qualify as a person

of ordinary skill in the art if they had additional education or industry experience that compensated for the deficiency in the requirement above.

33. I am familiar with the capabilities and skills of a person of ordinary skill in the art. For example, I have supervised and taught graduate students who would qualify as persons of ordinary skill in the art.

VI. CHALLENGED CLAIMS

34. I understand that claim 48 is challenged in the Petition.

35. Claim 48 is:

A computer implemented method comprising:

receiving a data block;

associating at least one encoder to each one of several data types;

analyzing data within the data block to identify a first data type of the data within the data block;

compressing, if said first data type is the same as one of said several data types, said data block with said at least one encoder associated to said one of said several data types that is the same as said first data type to provide a compressed data block; and

compressing, if said first data type is not the same as one of said several data types, said data block with a default encoder to provide said compressed data block;

wherein the analyzing of the data within the data block to identify one or more data types excludes analyzing based only on a descriptor that is indicative of the data type of the data within the data block.

VII. Claim Construction

36. My analysis below is based on meanings of the claim terms according to their plain meaning in light of the intrinsic record. The claim constructions in the Petition are consistent with the plain meaning in light of the intrinsic record.

VII. Prior Art Analysis

A. References

1. “Automatic Synthesis of Compression Techniques for Heterogeneous Files,” by Hsu and Zwarico (“Hsu”) (Ex. 1002)

37. Hsu presents a technique to compress a data file by applying the “most appropriate” compression algorithm from a suite of available algorithms to each block of the file. Hsu’s technique works in two phases.

38. In the first phase, the system first determines for each block its type (one of ten) and its compressibility via what Hsu calls “redundancy metrics.” If the data block has an appropriate combination (i.e., one that has an assigned compression algorithm) of data classification (also called a data type in Hsu) and largest redundancy metric, then the assigned compression algorithm is tagged for the data block by associating the data block with the assigned compression algorithm in a compression plan. Hsu at 1109. On the other hand, if an appropriate combination of data classification and largest redundancy metric is not identified (and no other

redundancy metric that would produce an appropriate combination is above a metric threshold), the data block is tagged with “no compression.” Hsu at 1106.

39. The block type is determined by an extension of the Unix “file” command that examines the first, middle, and last 512 bytes of the block and compares the pattern of data to a collection of known data patterns. There are ten data classifications (sometimes also referred to as “data types” in Hsu) in Hsu’s database depicted in Table I and reproduced below with annotations to show the different rows that represent each data classification. The entries in Table I are chosen to provide for better compression for each combination of data classification and redundancy metric.

Table I. Database of compression algorithms[†]

	M_{AD}	M_{RL}	M_{SR}
ANSI	arithmetic coding *	run-length encoding byte-wise encoding	Lempel-Ziv freeze
hexadecimal	arithmetic coding *	run-length encoding n -bit run count	Lempel-Ziv freeze
natural language	arithmetic coding *	* *	Lempel-Ziv freeze
source code	arithmetic coding *	run-length encoding n -bit run count	Lempel-Ziv freeze
low redundancy binary	* *	run-length encoding n -bit run count	Lempel-Ziv *
audio	* *	run-length encoding byte-wise encoding	Lempel-Ziv freeze
low resolution graphic	* *	run-length encoding n -bit run count	Lempel-Ziv freeze
high resolution color graphic	JPEG improved Huffman	run-length encoding n -bit run count	JPEG improved Huffman
high redundancy binary	arithmetic coding *	run-length encoding n -bit run count	Lempel-Ziv freeze
object	arithmetic coding *	run-length encoding byte-wise encoding	Lempel-Ziv freeze

[†] Note: the first line of each entry is the basic algorithm and the second line is the heuristic. An * as the heuristic indicates that no heuristic is used. Two * indicates no entry.

40. A person of ordinary skill in the art would have recognized nine of these data classifications (i.e., ANSI, hexadecimal, natural language, source code, audio, low resolution graphic, high resolution graphic, high redundancy binary executable, and object) are data classifications that have recognizable, specific structures. *See* Hsu at 1103-04 for a description of the data classifications. On the other hand, low redundancy binary does not have a specific structure. A person of ordinary skill in the art would have recognized that Hsu's low redundancy binary data is a default data classification that is used when none of the other nine data classifications are identified using the "new-file" program.

41. This understanding of how the “new-file” program works is consistent with how the “file” program worked before the priority date of the ’992 patent. Specifically, according to the manual page for the “file” program at that time, “file” would output the type “data” to mean “anything else” that could not be matched to another data type. *See Manual Page for “file” Command* (attached as Exhibit C this declaration).

The *first* test that succeeds causes the file type to be printed.

The type printed will usually contain one of the words **text** (the file contains only ASCII characters and is probably safe to read on an ASCII terminal), **executable** (the file contains the result of compiling a program in a form understandable to some UNIX kernel or another), or **data** meaning anything else (data is usually ‘binary’ or nonprintable).

(Bold and italics in original.) A person of ordinary skill in the art would have understood the “low redundancy binary” data classification in Hsu to be the equivalent of the “data” type in the unaltered version of “file.” It represents any other binary data classification that did not match one of the other nine well-recognized data classifications.

42. Returning to the operation of Hsu’s first phase, the compressibility of the data block is determined by the values of three redundancy metrics representing the degree of variation in character frequency (“M_{AD}”), average run length

("M_{RL}"), and string repetition ratio ("M_{SR}") in the block. Each redundancy metric is calculated by a separate statistical sampling routine and normalized using a gamma distribution function to be a number between 0 and 10. If the metrics are all below a threshold then the block is not compressed in order to save the overhead of attempting to compress the data block when the redundancy metric indicates low potential for significant compression. Otherwise, using the block type and largest metric, the assigned compression algorithm is chosen from the compression algorithm database. The data block is then tagged with the assigned compression algorithm in a compression plan. If the block classification and largest metric are not an appropriate combination in the database and none of the other redundancy metrics are above a metric threshold, then the data block is tagged for no compression in the compression plan in order to save on the overhead of trying to compress the data block when the data block produced an unexpected combination of data classification and largest redundancy metric.

43. In the second phase, Hsu's system compresses the data block according to the compression plan. Prior to applying the chosen compression algorithm, adjacent blocks are merged if they are to be compressed with the same algorithm. (Hsu at 1109.) After a data block or merged group of data blocks is compressed with the identified compression algorithm, the compressed data block is checked for negative compression (e.g., making sure that the compression ratio is greater

than 1). (Hsu at 1109.) If negative compression is detected, the uncompressed data block is used and “no compression” is recorded in the compression history. (Hsu at 1109.) Otherwise, the compressed data block is used and the appropriate compression algorithm is recorded in the compression history. Once all of the data blocks are processed, the compression history is prepended to the resulting data blocks to produce the output file that can be stored.

2. U.S. Patent No. 6,253,264 (“Sebastian”) (Ex. 1005)

44. Sebastian discloses compressing different types of data sources with different compression encoders (called “filters” in Sebastian) to produce a “format-specific compression” system. When the type of a data source is not supported and an associated encoder cannot be identified, a “generic” compression encoder is used.

At the highest level, a preferred compression system in accordance with the invention uses an architecture called a Base-Filter-Resource (BFR) system. This approach integrates the advantages of format-specific compression into a general-purpose compression tool serving a wide range of data formats. The system includes filters which each support a specific data format, such as for Excel XLS worksheets or Word DOC files. The base includes the system control modules and a library used by all the filters. The resources include routines which are used by more than one filter, but which are not part of the base. *If a filter is installed which matches the format of the data to be encoded, the advantages of format-specific compression can be realized for that data. Otherwise, a “generic” filter is used which achieves performance*

similar to other non-specific data compression systems (such as PKZip, Stacker, etc.). (Sebastian at 1:36-60 (emphasis added).)

FIG. 2 is a block diagram of a preferred encoder using a Base-Filter-Resource (BFR) network in accordance with the invention. The encoder 3` is based on the use of a plurality of filters 10a, . . . , 10x, . . . , 10z which serve specific file formats. For example, one filter 10a might support several versions of the DBF database format, while another filter 10z might support several versions of the DOC format used by the Microsoft Word software program. The individual filters provide respective selection criteria 12 to a filter selection system 22.

The filter selection system 22 receives the source data 2 and checks the selection criteria 12a, . . . , 12x, . . . , 12z of all filters 10a, . . . , 10x, . . . , 10z installed in the system to see if any of them support the source data's format. *If not, a "generic" filter is used which provides compression performance similar to other generic compression systems, such as Lempel-Ziv (LZ) engines.* In a particular preferred embodiment of the invention, the generic compression system employs an SZIP engine as described by Mr. Schindler in U.S. application Ser. No. 08/970,220 filed Nov. 14, 1997, the teachings of which are incorporated herein by reference in their entirety. The descriptions of the network will primarily cover the situations in which a filter to support the data format is successfully found. (Sebastian at 3:66-4:22 (emphasis added).)

3. U.S. Patent No. 5,870,036 ("Fanaszek") (Ex. 1003)

45. Franaszek describes a data compression system that applies different data compression techniques to different data blocks depending on the data type of the data blocks. In Franaszek's system, a data block to be compressed is checked to

determine whether a “data type” field identifies the data type of the data block. If the “data type” field identifies a data type, a list of compression methods corresponding to the data type (that was identified by the contents of the “data type” field) is assigned to the data block. Franaszek at Fig. 2, 4:25-35, 5:49-54, 6:1-11. If the “data type” field does not identify the data type, a list of default compression methods is assigned to the data block. In other words, Franaszek teaches automatically applying a predefined default encoder from the list of compression methods to the data block when a data type specific encoder is not identified from the data type. Nothing in Franaszek describes how the “data type” information for the “data type” field is determined or where it came from.

46. The system determines the “best” compression technique available by compressing a portion of the block with each of the list of compression methods and selecting the technique that results in the best compression ratio. This is similar to a process used to select compression techniques in the ’992 patent. ’992 patent at 4:11-26.

47. Nothing in Franaszek limits how many compression methods are present in the lists of compression methods (i.e., the lists of compression methods associated with specific data types or the list of default compression methods).

In step 401, if a data type (e.g. text, image, etc.) for a given uncompressed block B is available, in step 404 the Compression Method List (CML) is set to a list of

compression methods that have been preselected for that data type. Otherwise, if no data type is available, in step 407 the CML is set to a default list of compression methods. (Franaszek at 5:49-54.)

In step 414, it is determined if a data type is available (i.e. the block includes a “data type” entry in the type field 205). If a data type is available, in steps 417, 421, 424, and 427, the CML is expanded by replacing E with the list (M,D1), (M,D2), . . . , (M,Dj), where (D1, . . . , Dj) is a list of dictionary block identifiers that have been preselected for the data type when using compression method M. Otherwise, if no data type is available), steps 419, 421, 424, and 427, replace E with the list (M,D1'), (M,D2'), . . . , (M,Dk'), where (D1', . . . , Dk') is a default list of dictionary block identifiers for compression method M. (Franaszek at 6:1-11.)

Accordingly, a person of ordinary skill in the art would understand Franaszek's lists to contain any number of compression methods (i.e., one or more compression methods). This is similar to the disclosure in the '992 patent that describes encoding a data block using “a set of encoders D1, D2, D3 . . . Dm . . . [that] may include any number ‘n’ of those lossless or lossy encoding techniques currently well known with the art.” '992 patent at 16:45-47.

48. Franaszek's compression works by using a sample from a block that is to be compressed. Each of a set of compressors is applied to the sample and the compressor that results in the best compression of the sample is selected. For example, a block of plain text might be compressed with an LZ compression technique while a block of image data might be compressed with a run-length

compression technique. If, however, the best compression is not sufficient, then a NOOP (i.e., no compressor) may be used for the block. If a “best” compressor is identified, then that compressor is applied to the data block to produce a compressed data block (i.e., a block with fewer bytes than the original data block) that can be stored in memory. A compression method description (CMD) stored with the compressed data block identifies the compressor used for a particular data block.

49. Franaszek’s decompression sequence is the opposite sequence of the compression sequence. The compressed data block, which includes the CMD and compressed data, is retrieved. The CMD is used to determine the identity of the particular compressor (or NOOP) that was used to produce compressed data in the compressed data block. This information will enable the system to identify the decompressor (or NOOP) that should be used to decompress the data. The identified decompressor is then used to generate the uncompressed data.

50. The Franaszek system stores and retrieves the compressed data blocks on a memory device. For example, Franaszek discloses memory devices such as “semiconductor memories, magnetic storage (such as a disk or tape), optical storage or any other suitable type of information storage media.” Franaszek at 4:10-12.

51. Franaszek discloses many of these same compression techniques that the '992 patent describes as default compression encoders and compression encoders associated with data types.

For simplicity, assume that all uncompressed data blocks are a given fixed size, say 4096 bytes. Each such block can be considered to consist of 8 512-byte sub-blocks, for example. In some cases it may happen that using the first such sub-block as a dictionary may yield better compression than any of the fixed static dictionaries. This method is shown as 602 in the compression method table 240. Alternatively, take the first 64 bytes of each sub-block, and concatenating these into a 512-byte dictionary, could be the best dictionary for some blocks having certain phrase patterns. This method is indicated by 603 in the figure. The other three methods indicated in the figure are arithmetic coding 600, run-length coding 601, and LZ1 using one of the fixed set of static dictionaries 602. (Franaszek at 7:6-19.)

52. Franaszek outputs a descriptor that indicates the compression method that was used to compress a particular block of data.

Each block of data includes a coding identifier which is indicative of the method or mechanism used to compress the block. The coding identifier is examined to select an appropriate one of the data decompression mechanisms to apply to the block. The block is then decompressed using the selected one of the mechanisms. Franaszek at 3:42-45.

The compressor outputs compressed data blocks 230, with an index (M) 232 identifying the selected compression method, and for dictionary-based methods, dictionary block identifier (D), encoded in a compression

method description (CMD) area 235 in the compressed block. Franaszek at 4:55-59.

53. The “CMD” area is stored with each compressed data block in the second memory of FIG. 1 and identifies the compression method that was used to compress the compressed data block. For example, CMD area 235 in FIG. 2, which is described at 4:55-59, is stored with each compressed data block in the second memory of FIG. 1 and contains an identifier of the compression method that was used to produce the compressed data block. The CMD area is used during decompression to identify the correct decompression technique to use for the compressed data block.

Compressed data blocks 230, with the compression method identifier M and for dictionary-based methods dictionary block identifier D encoded in the CMD area 235 are input to the de-compressor 270. The de-compressor 270 de-compresses the block using the specified method found in the compression method table 240 (using the compression method identifier as an index), and for dictionary-based methods, specified dictionary block found in the dictionary block memory 250, and outputs uncompressed data blocks 280. Franaszek at 4:65-5:7.

4. U.S. Patent No. 5,467,087 (“Chu”) (Ex. 1015)

54. Chu describes a compression system that determines a data type of a data stream and then chooses an appropriate compression algorithm based on the identified data type.

A data compression process and system that identifies the data type of an input data stream and then selects in response to the identified data type at least one data compression method from a set of data compression methods that provides an optimal compression ratio for that particular data type, thus maximizing the compression ratio for that input data stream. Moreover, the data compression process also provides means to alter the rate of compression during data compression for added flexibility and data compression efficiency. (Chu at Abstract.)

55. Chu discloses using the contents of the data stream itself to determine the type of data. For example, claims 3, 7, and 8, reproduced below, disclose “analyzing data” (e.g., looking for data bytes representing values above a threshold or having identical contents) to determine a data type.

3. An electronic data compression process for compressing at least one set of input data, the at least one set of input data being of a specific data type of a plurality of data types, the electronic data compression process comprises the steps of:

identifying the specific data type of the set of input data;

selecting at least one data compression method in response to the identified data type;

compressing the set of input data with the selected at least one data compression method; and

allocating an amount of memory to perform the step of compressing, the amount of memory being an amount substantially equal to the greater of an initial amount of memory and an amount of memory necessary to compress the set of input data. (Chu at 8:28-43 (emphasis added).)

7. The electronic data compression process of claim 3 wherein the set of input data includes a series of bytes, wherein the step of *identifying the specific data type of the set of input data includes the step of determining whether each byte in said series of bytes represents a value greater than a predetermined value.*

8. The electronic data compression process of claim 3 wherein the set of input data includes a series of bytes, wherein the step of *identifying the specific data type of the set of input data includes the step of determining whether selected bytes in said series of bytes are identical.* (Chu at 8:56-66 (emphasis added).)

B. MODIFYING REFERENCES

1. Modifying Hsu to Include a Default Encoder

56. An implementation of Hsu with “a default encoder” is consistent with the teaching of other prior art references. For example, as I described above, Sebastian and Franaszek each disclose compression systems that select compression algorithms that are most appropriate for the data type being compressed. When the data type is not identified or not recognized, Sebastian’s system and Franaszek’s system each use a default compression algorithm. Sebastian and Franaszek thus each provide teachings that a default compression algorithm can be used when no data type is identified.

57. Implementing Hsu to use “a default encoder,” as recited in claim 48 of the ’992 patent, would have required nothing more than using known technologies for their intended purposes to produce predictable results. Such an implementation

also would have been well within the ability of a person of ordinary skill in the art. Hsu states that the described compression system was implemented using the C programming language. A person of ordinary skill in the art would have been well versed in the C programming language.

58. A person of ordinary skill in the art would have found it obvious to implement Hsu to use a default encoder as recited in claim 48 of the '992 patent. For example, Hsu acknowledges that there are four data classification/redundancy metric combinations that have no entry in the database represented by Table I (the locations of double “*” in Table I). Hsu states that occurrence of these combinations is “very unusual.” Hsu at 1106. When such a combination occurs, the next highest metric is used instead unless the other metrics are below the threshold. Hsu discloses one solution when this occurs, which is to not perform any compression at all in order to save the overhead of compressing a data block when there is poor potential for compression. Hsu at 1106.

59. A person of ordinary skill in the art would have recognized another potential solution to the problem of a combination of a data classification and largest redundancy metric without an associated encoder that would have been obvious to try. Specifically, it would have been obvious to try implementing Hsu’s system to just use a default encoder instead of tagging the data block for no compression at all. Such a modification would have been well within the skill of a person of

ordinary skill in the art for similar reasons as I explained above. Such a modification would have simply used known technologies (i.e., Hsu's compression system and Sebastian's or Franaszek's default encoder) to produce predictable results. Additionally, such a modification could be done with simple C programming features, such as "if . . . then" statements with minimal changes to the compression lookup function to the compression database.

60. In fact, not only would it have been obvious to try implementing a default encoder in Hsu's system, as taught in Sebastian and Franaszek, but a person of ordinary skill in the art would have been motivated to make such a modification for a variety of reasons. Hsu discloses that one of its goals is to achieve a compression system with "better space savings." Hsu at Abstract. A person of ordinary skill in the art implementing Hsu with this goal in mind would have recognized the potential for improved space savings by not tagging data classification/redundancy metric combinations for no compression just because the expectation is that compressing would result in no space savings. Instead, a default compression algorithm could be used to see if any space savings could be achieved. This modification would have been simple to implement. It would only require tagging a data block with the default compression algorithm when such data classification/redundancy metric combinations were identified. The rest of Hsu's system could remain unmodified.

61. A person of ordinary skill in the art would have found such a modification to be useful in several situations. For example, while Hsu explains that data blocks with certain combinations of data classifications and largest redundancy metric are tagged for no compression because they are “interpreted as a decision that the (poor) potential for compression is outweighed by the overhead of executing the compression algorithm” (Hsu at 1106), in situations where computing resources are not limited and/or compression ratio is more important, it would have been obvious to try compressing these data blocks anyway just in case further space savings could still be achieved. In other words, in Hsu, there is a tradeoff between potential space savings and computing resources and Hsu’s system is configured to save computing resources in this situation. A person of ordinary skill in the art that weighs these factors differently would have found it obvious to choose space savings over computing resources.

62. Space savings could be achieved even with the very unusual combinations of data classification and largest redundancy metric in several circumstances. For example, Hsu acknowledges that modifications to the system would be needed to accommodate the continuing development of new data classifications in the future, such as “high-resolution animation and three-dimensional images.” Hsu at 1114. Prior to the filing of the ’992 patent, it was well known the number of data formats and data types was increasing and it was common to encounter unknown data types

in common applications such as data storage and data communications. *See, e.g.*, International Patent Application Publication Nos. WO 2001/063772 (Ex. 1018) at 5 and WO 2001/050325 (Ex. 1019) at 10-11 (describing how to handle unknown data types when compressing data).

63. If Hsu's system were to encounter the new data classifications before "new-file" was expanded to properly handle them, the data blocks with the new data classifications could be misclassified as one of Hsu's "very unusual" combinations and tagged for no compression even though compressing the data block might result in space savings. Hsu at 1106. Using a default encoder would address this issue until "new-file" could be properly updated. As another example, Hsu's data classification identification module and redundancy metric modules are based on sampling a data block. That means if a very unusual data classification/redundancy metric combination is identified, it could just mean that the sampling was unlucky and produced an incorrect combination. Again, using a default encoder would address this issue while maximizing space savings.

2. Modifying Franaszek to Analyze Data without a Descriptor

64. Franaszek's system relies on a "data type" field, the contents of which can be used by the system to identify the data type of a data block. Franaszek, however, does not specify how the "data type" field came to contain its contents. For example, it is possible that the data blocks that were received as input to

Franaszek's compression system contained the "data type" field content. On the other hand, it is also possible that the data blocks did not have "data type" field content. In this case, the "data type" content would need to have been determined using a separate process before feeding the data blocks to the data compressor (see Figure 2). Accordingly, a person of ordinary skill in the art would have looked to see how others had addressed the same problem. Hsu and Chu both teach determining a data type for data can be accomplished by analyzing data directly without relying on a data descriptor to indicate the data type. Based on the teaching in Hsu and Chu with respect to systems that are similar to Franaszek's, a person of ordinary skill in the art would have found it obvious to adapt Franaszek's system to work with data streams that do not have "data type" field contents and instead to analyze the data bytes of the data block to determine the data type without relying on a descriptor that indicates the data type.

65. Analyzing data bytes themselves, as taught in Hsu and Chu, to determine the contents for Franaszek's "data type" field would have been well within the abilities of a person of ordinary skill in the art. Such an implementation of Franaszek's system would have only required using two known technologies (e.g., Franaszek's compression system with a "data type" field and Hsu's or Chu's analysis of the data itself to determine the data type) to produce predictable results. Each of the technologies would be used exactly for its respective intended purpose.

I declare that all statements made herein of my own knowledge are true and that all statements made on information and belief are believed to be true, and that these statements were made with knowledge that willful false statements and the like so made are punishable by fine or imprisonment, or both, under section 1001 of Title 18 of the United States Code.

Dated: June 22, 2017

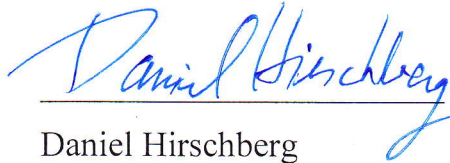

Daniel Hirschberg

Exhibit List for Declaration of Daniel Hirschberg

Exhibit Description	Exhibit
CV of Daniel Hirschberg	A
PKZIP APPNOTE Version 1.0	B
Unix man page for “file” command, available from https://web.archive.org/web/20010810082609/http://unixhelp.ed.ac.uk:80/CGI/man-cgi?file	C

Daniel S. Hirschberg

Department of Computer Science
University of California, Irvine
Irvine, CA 92697-3435

(949) 824-6480
dan@ics.uci.edu
<http://www.ics.uci.edu/~dan>

EDUCATION

1975 Ph.D. (Computer Science), Princeton University
1973 MSE, MA, Princeton University
1971 BE(EE), City College of New York

ACADEMIC APPOINTMENTS

2003- Professor of Computer Science and EECS
1994-2003 Professor of Information and Computer Science and ECE
1987-94 Professor of Information and Computer Science
1992-93, 96-98 Associate Chair of Undergraduate Studies, ICS
1984-90 Associate Chair of Graduate Studies, ICS
1981-87 Associate Professor of Information and Computer Science
1975-81 Assistant Professor of Electrical Engineering (*Rice University*)

CONSULTING ACTIVITIES

1998-2017 several law firms
Consulting expert for intellectual property cases
Provide expert testimony in judicial proceedings
1984-94 Manufacturing and Consulting Services, Inc. (Scottsdale, AZ)
Design and analysis of database structures for CAD/CAM
1989 A-Chip Co., Inc. (Santa Ana, CA)
Design of data compression/decompression techniques
1984-89 Pick Systems, Inc. (Irvine, CA)
Design of operating system data structures
1986 Computer Cognition, Inc. (San Diego CA)
Design of data structures for AI applications
1978-81 University of Texas Health Science Center (Houston, TX)
Database development for genetics research
1976 Argonne National Laboratories (Argonne, IL)
System simulation, Numerical analysis

PROFESSIONAL SERVICE

Referee of grant proposals for Army Research Office, National Science Foundation,
Israel Science Foundation, Research Grants Council (Hong Kong)
Reviewer of textbooks for several publishers
Referee of technical papers for numerous journals
Associate Editor, ACM Trans. on Mathematical Software (1988–90)
Associate Editor, Discrete Mathematics, Algorithms and Applications (2009–present)
Member of the Program Committee
IEEE Data Compression Conference (1991,1992,1993,1994)
Combinatorial Pattern Matching (1992,1993,1994,1996,1997,2009) [co-chair 1996]
String Processing and Information Retrieval (1998)
Combinatorial Optimization and Applications (2010)

PUBLICATIONS

A. Books and Book Chapters

- B1 D.S. Hirschberg, “Recent results on the complexity of common subsequence problems,” in *Time Warps, String Edits, and Macromolecules*, D. Sankoff and J.B. Kruskal (Eds.), Addison-Wesley (1983) 323–328.
- B2 D.S. Hirschberg and D.A. Lelewer, “Context modeling for text compression,” in *Image and Text Compression*, J. A. Storer, Ed., Kluwer Academic Publishers, Boston, Mass. (1992) 113–145.
- B3 D.S. Hirschberg, M.J. Pazzani and K. Ali, “Average case analysis of k-CNF and k-DNF learning algorithms,” in *Computational Learning Theory and Natural Learning Systems: Constraints and Prospects*, S. Hanson, M. Kearns, T. Petsche and R. Rivest (Eds.), MIT Press, Cambridge, Mass. (1994) 15–28.
- B4 D. Hirschberg and G. Myers (Eds.), *Combinatorial Pattern Matching, Proceedings 1996, Lecture Notes in Computer Science*, vol. 1075, Springer-Verlag, Berlin (1996) 392 pp.
- B5 D.S. Hirschberg, “Serial computations of Levenshtein distances,” in *Pattern Matching Algorithms*, Apostolico, A. and Galil, Z. (Eds.), Oxford University Press (1997) 123–141.

B. Articles in Refereed Journals

- J1 D.S. Hirschberg, “A class of dynamic memory allocation algorithms,” *Communications ACM* **16**:10 (1973) 615–618.
- J2 D.S. Hirschberg, “A linear space algorithm for computing maximal common subsequences,” *Communications ACM* **18**:6 (1975) 341–343.
- J3 A.V. Aho, D.S. Hirschberg and J.D. Ullman, “Bounds on the complexity of the longest common subsequence problem,” *Journal ACM* **23**:1 (1976) 1–12.
- J4 D.S. Hirschberg and C.K. Wong, “A polynomial-time algorithm for the knapsack problem with two variables,” *Journal ACM* **23**:1 (1976) 147–154.
- J5 D.S. Hirschberg, “An insertion technique for one-sided height-balanced trees,” *Communications ACM* **19**:8 (1976) 471–473.
- J6 A.K. Chandra, D.S. Hirschberg and C.K. Wong, “Approximate algorithms for some generalized knapsack problems,” *Theoretical Computer Science* **3**:3 (1976) 293–304.
- J7 D.S. Hirschberg, “Algorithms for the longest common subsequence problem,” *Journal ACM* **24**:4 (1977) 664–675.
- J8 D.S. Hirschberg, “An information theoretic lower bound for the longest common subsequence problem,” *Information Processing Letters* **7**:1 (1978) 40–41.
- J9 D.S. Hirschberg, “Fast parallel sorting algorithms,” *Communications ACM* **21**:8 (1978) 657–661.
- J10 A.K. Chandra, D.S. Hirschberg and C.K. Wong, “Bin packing with geometric constraints in computer network design,” *Operations Research* **26**:5 (1978) 760–772.
- J11 D.S. Hirschberg and C.K. Wong, “Upper and lower bounds for graph-diameter problems,” *Journal of Comb. Theory (B)* **26**:1 (1979) 66–74.
- J12 D.S. Hirschberg, A.K. Chandra and D.V. Sarwate, “Computing connected components on parallel computers,” *Communications ACM* **22**:8 (1979) 461–464.

- J13 D.S. Hirschberg, “On the complexity of searching a set of vectors,” *SIAM Journal on Computing* **9**:1 (1980) 126–129.
- J14 D.S. Hirschberg and J.B. Sinclair, “Decentralized extrema-finding in circular configurations of processors,” *Communications ACM* **23**:11 (1980) 627–628.
- J15 M. Kumar and D.S. Hirschberg, “An efficient implementation of Batchers’ odd-even merge algorithm and its application in parallel sorting schemes,” *IEEE Trans. on Computers* **C-32**:3 (1983) 254–264.
- J16 L.L. Larmore and D.S. Hirschberg, “Efficient optimal pagination of scrolls,” *Communications ACM* **28**:8 (1985) 854–856.
- J17 J. Hester and D.S. Hirschberg, “Self-organizing linear search,” *Computing Surveys* **17**:3 (1985) 295–311.
- J18 J.H. Hester, D.S. Hirschberg, S.-H.S. Huang and C.K. Wong, “Faster construction of optimal binary split trees,” *Journal of Algorithms* **7**:3 (1986) 412–424.
- J19 D.S. Hirschberg and L.L. Larmore, “Average case analysis of marking algorithms,” *SIAM Journal on Computing* **15**:4 (1986) 1069–1074.
- J20 D.S. Hirschberg and L.L. Larmore, “The Set LCS problem,” *Algorithmica* **2** (1987) 91–95.
- J21 D.S. Hirschberg and D.J. Volper, “Improved update/query algorithms for the interval valuation problem,” *Information Processing Letters* **24** (1987) 307–310.
- J22 D.S. Hirschberg and L.L. Larmore, “New applications of failure functions,” *Journal ACM* **34**:3 (1987) 616–625.
- J23 D.S. Hirschberg and L.L. Larmore, “The least weight subsequence problem,” *SIAM J. on Computing* **16**,4 (1987) 628–638.
- J24 J. Hester and D.S. Hirschberg, “Self-organizing search lists using probabilistic backpointers,” *Communications ACM* **30**:12 (1987) 1074–1079.
- J25 D.A. Lelewer and D.S. Hirschberg, “Data compression,” *Computing Surveys* **19**:3 (1987) 261–297. Reprinted in *Japanese BIT Special issue in Computer Science* (1989) 165–195.
- J26 J.H. Hester, D.S. Hirschberg, and L.L. Larmore, “Construction of optimal binary split trees in the presence of bounded access probabilities,” *Journal of Algorithms* **9**:2 (1988) 245–253.
- J27 D.S. Hirschberg and L.L. Larmore, “The Set-Set LCS problem,” *Algorithmica* **4**:4 (1989) 503–510.
- J28 C. Ng and D.S. Hirschberg, “Lower bounds for the stable marriage problem and its variants,” *SIAM J. on Computing* **19**:1 (1990) 71–77.
- J29 D.S. Hirschberg and D.A. Lelewer, “Efficient decoding of prefix codes,” *Communications ACM* **33**:4 (1990) 449–459.
- J30 L.L. Larmore and D.S. Hirschberg, “A fast algorithm for optimal length-limited codes,” *Journal ACM* **37**:3 (1990) 464–473.
- J31 C. Ng and D.S. Hirschberg, “Three-dimensional stable matching problems,” *SIAM J. Discr. Math.* **4**:2 (1991) 245–252.
- J32 D.S. Hirschberg and L.L. Larmore, “The traveler’s problem,” *Journal of Algorithms* **13** (1992) 148–160.
- J33 D.S. Hirschberg and S.S. Seiden, “A bounded-space tree traversal algorithm,” *Information Processing Letters* **47** (1993) 215–219.

- J34 S.S. Seiden and D.S. Hirschberg, "Finding succinct minimal perfect hashing functions," *Information Processing Letters* **51** (1994) 283–288.
- J35 L.M. Stauffer and D.S. Hirschberg, "Systolic self-organizing lists under transpose," *IEEE Trans. on Parallel and Distributed Systems* **6**:1 (1995) 102–105.
- J36 D.S. Hirschberg and L.M. Stauffer, "Dictionary compression on the PRAM," *Parallel Processing Letters* **7**:3 (1997) 297–308.
- J37 D. Eppstein and D.S. Hirschberg, "Choosing subsets with maximum weighted average," *Journal of Algorithms* **24** (1997) 177–193.
- J38 M. Dillencourt, D. Eppstein, and D. S. Hirschberg. "Geometric thickness of complete graphs," *J. Graph Algorithms and Applications* **4**:3 (2000) 5–17. Reprinted in *Graph Algorithms and Applications 2*, Giuseppe Liotta, Robert Tamassia, and Ioannis G Tollis, ed., (2004).
- J39 D.S. Hirschberg and M. Regnier, "Tight bounds on the number of string subsequences," *Journal of Discrete Algorithms* **1**:1 (2000) 123–132.
- J40 M. Mamidipaka, D. Hirschberg, and N. Dutt, "Adaptive low power address encoding techniques using self-organizing lists," *IEEE Trans. on Very Large Scale Integration Systems* **11**:5 (2003) 827–834.
- J41 D. Eppstein, M.T. Goodrich, and D.S. Hirschberg, "Improved combinatorial group testing algorithms for real-world problem sizes," *SIAM J. on Computing* **36**:5 (2007) 1360–1375.
- J42 G.I. Bell, D.S. Hirschberg, and P. Guerrero-Garcia, "The minimum size required of a solitaire army," *Integers: Electronic Journal of Combinatorial Number Theory* **7** (2007), #G07 <http://www.integers-ejcnt.org/vol7.html> (22 pages).
- J43 P. Baldi, R. Benz, D.S. Hirschberg, and S. Swamidass, "Lossless compression of chemical fingerprints using integer entropy codes improves storage and retrieval," *Journal of Chemical Information and Modeling* **47**:6 (2007) 2098–2109.
- J44 M.T. Goodrich and D.S. Hirschberg, "Improved adaptive group testing algorithms with applications to multiple access channels and dead sensor diagnosis," *Journal of Combinatorial Optimization* **15**:1 (2008) 95–121.
- J45 P. Baldi, D.S. Hirschberg, and R. Nasr, "Speeding up chemical database searches using a proximity filter based on the logical exclusive-or," *Journal of Chemical Information and Modeling* **48**:7 (2008) 1367–1378.
- J46 P. Baldi and D.S. Hirschberg, "An intersection inequality sharper than the Tanimoto triangle inequality for efficiently searching large databases," *Journal of Chemical Information and Modeling* **49**:8 (2009) 1866–1870.
- J47 R. Nasr, D.S. Hirschberg, and P. Baldi, "Hashing algorithms and data structures for rapid searches of fingerprint vectors," *Journal of Chemical Information and Modeling* **50**:8 (2010) 1358–1368.
- J48 D. Eppstein and D. Hirschberg, "From discrepancy to majority," *Algorithmica* (2017) to appear.

C. Papers in Conference Proceedings and Workshops

- C1 A.V. Aho, D.S. Hirschberg and J.D. Ullman, "Bounds on the complexity of the longest common subsequence problem," *Proc. 15th IEEE Symp. on Switching and Automata Theory*, New Orleans, LA (1974) 104–109.

- C2 D.S. Hirschberg, "A slightly better bound for the vertex connectivity problem," *Proc. Conf. of Info. Sci. and Systems*, Baltimore MD, Johns Hopkins Univ. (1975) 257–258.
- C3 D.S. Hirschberg, "Parallel algorithms for the transitive closure and the connected component problems," *Proc. 8th ACM Symp. on Theory of Computing*, Hershey PA (1976) 55–57.
- C4 D.S. Hirschberg, "Complexity of common subsequence problems," *Fundamentals of Computation Theory*, Poznan Poland, *Lecture Notes in Computer Science*, vol. 56, Springer-Verlag, Berlin (1977) 393–398.
- C5 D.S. Hirschberg, "A lower worst-case complexity for searching a dictionary," *Proc. 16th Allerton Conf. on Communications, Control, and Computing*, Monticello IL, Univ. of Ill. (1978) 50–53.
- C6 D.S. Hirschberg, "Election processes in distributed systems", *Proc. 18th Allerton Conf. on Communications, Control, and Computing*, Monticello IL, Univ. of Ill. (1980) 823.
- C7 M. Kumar and D.S. Hirschberg, "An efficient implementation of Batchers' odd-even merge algorithm and its application in parallel sorting schemes," *Proc. Conf. of Info. Sci. and Systems*, Baltimore MD, Johns Hopkins Univ. (1981).
- C8 D.S. Hirschberg, "Parallel graph algorithms without memory conflicts," *Proc. 20th Allerton Conf. on Communications, Control, and Computing*, Monticello IL, Univ. of Ill. (1982) 257–263.
- C9 D.S. Hirschberg and D.J. Volper, "A parallel solution for the minimum spanning tree problem," *Proc. Conf. of Info. Sci. and Systems*, Baltimore MD, Johns Hopkins Univ. (1983) 680–684.
- C10 D.S. Hirschberg and L.L. Larmore, "Average case analysis of marking algorithms," *Proc. 22nd Allerton Conf. on Communications, Control, and Computing*, Monticello IL, Univ. of Ill. (1984) 508–509.
- C11 L.L. Larmore and D.S. Hirschberg, "Breaking a paragraph into lines in linear time," *Proc. 22nd Allerton Conf. on Communications, Control, and Computing*, Monticello IL, Univ. of Ill. (1984) 478–487.
- C12 D.S. Hirschberg and L.L. Larmore, "The Least Weight Subsequence Problem," *Proc. 26th IEEE Symp. on Foundations of Computer Science*, Portland Oregon (1985) 137–143.
- C13 R.R. Razouk and D.S. Hirschberg, "Tools for efficient analysis of concurrent software systems," *Proc. of SOFTFAIR II, A Second Conference on Software Development Tools, Techniques, and Alternatives*, San Francisco CA (1985).
- C14 J.H. Hester and D.S. Hirschberg, "Generation of optimal binary split trees," *Proc. 24th Allerton Conf. on Communications, Control, and Computing*, Monticello IL, Univ. of Ill. (1986) 308–313.
- C15 L.L. Larmore and D.S. Hirschberg, "Length-limited coding," *Proc. First ACM-SIAM Symposium on Discrete Algorithms*, San Francisco (1990) 310–318.
- C16 D.A. Lelewer and D.S. Hirschberg, "Streamlining context models for data compression," *Proc. IEEE Data Compression Conference*, Snowbird UT (1991) 313–322.
- C17 D.S. Hirschberg, M.J. Pazzani and K. Ali, "Average case analysis of k-CNF and k-DNF learning algorithms," *Second Annual Workshop on Computational Learning Theory and Natural Learning Systems: Constraints and Prospects*, Berkeley CA (1991).
- C18 S. Bhatia, D.S. Hirschberg and I.D. Scherson, "Shortest paths in orthogonal graphs," *Proc. 29th Allerton Conf. on Communications, Control, and Computing*, Monticello IL, Univ. of Ill. (1991) 488–497.

- C19 L.M. Stauffer and D.S. Hirschberg, "Transpose coding on the systolic array," *Proc. IEEE Data Compression Conference*, Snowbird UT (1992) 162–171.
- C20 D.S. Hirschberg and M.J. Pazzani, "Average case analysis of learning k-CNF concepts," *Proc. Ninth International Machine Learning Conference*, Aberdeen Scotland, Morgan Kaufmann, San Mateo (1992) 206–211.
- C21 D.S. Hirschberg and L.M. Stauffer, "Parsing algorithms for dictionary compression on the PRAM," *Proc. IEEE Data Compression Conference*, Snowbird UT (1994) 136–145.
- C22 L.M. Stauffer and D.S. Hirschberg, "PRAM algorithms for static dictionary compression," *Proc. IEEE 8th International Parallel Processing Symposium*, Cancun Mexico (1994) 344–348.
- C23 D. Eppstein and D.S. Hirschberg, "Choosing subsets with maximum weighted average," *Proc. 5th MSI Workshop on Computational Geometry*, Stonybrook NY (1995) 7–8.
- C24 J.K. Martin and D.S. Hirschberg, "On the complexity of learning decision trees," *Proc. 4th Int. Symp. on Artif. Intell. and Math.*, Fort Lauderdale FL (1996) 112–115. [Expanded version in "The time complexity of decision tree induction," Tech. Rpt. 95–27, ICS Dept., UC Irvine (August, 1995).]
- C25 M.B. Dillencourt, D.E. Eppstein and D.S. Hirschberg, "Geometric thickness of complete graphs," *Graph Drawing: 6th Int'l Symp.*, Montreal Canada, *Lecture Notes in Computer Science*, vol. 1547, Springer-Verlag, Berlin (1998) 102–110.
- C26 D.S. Hirschberg, "Bounds on the number of string subsequences," *Proc. Symp. on Combinatorial Pattern Matching*, Warwick UK, *Lecture Notes in Computer Science*, Springer-Verlag, Berlin (1999) 115–122.
- C27 M. Mamidipaka, D. Hirschberg, and N. Dutt, "Low power address encoding using self-organizing lists," *Proc. ACM/IEEE Int'l Symp. on Low Power Electronics and Design*, Huntington Beach CA (2001) 188–193.
- C28 M. Mamidipaka, D. Hirschberg, and N. Dutt, "Efficient power reduction techniques for time multiplexed address buses," *Proc. 15th ACM Int'l Symp. on System Synthesis*, Kyoto (2002) 207–212.
- C29 D. Eppstein, M.T. Goodrich, and D.S. Hirschberg, "Improved combinatorial group testing algorithms for real-world problem sizes," 9th Workshop Algorithms and Data Structures (WADS), Waterloo, 2005. *Lecture Notes in Comp. Sci.* **3608** (2005) 86–98.
- C30 M.T. Goodrich, and D.S. Hirschberg, "Efficient parallel algorithms for dead sensor diagnosis and multiple access channels," *Proc. 18th ACM Symposium on Parallelism in Algorithms and Architectures* (SPAA), Cambridge MA (2006) 118–127.
- C31 D.S. Hirschberg and P. Baldi, "Effective compression of monotone and quasi-monotone sequences of integers," *Proc. IEEE Data Compression Conference*, Snowbird UT (2008) 520.
- C32 D.S. Hirschberg, "Constructing problems of geometric combinatorics," *Gathering for Gardner Conference* (G4G9), Atlanta GA (2010), 8 pp.
- C33 M.T. Goodrich, D.S. Hirschberg, M. Mitzenmacher, and J. Thaler, "Cache-oblivious dictionaries and multimaps with negligible failure probability," *Proc. Mediterranean Conference on Algorithms*, Kibbutz Ein-Gedi Israel (2012), *Lecture Notes in Computer Science*, vol. 7659, Springer-Verlag, Berlin (2012) 203–218.
- C34 D. Eppstein, M. Goodrich, and D. Hirschberg, "Combinatorial pair testing: distinguishing workers from slackers," *Algorithms and Data Structures Symposium* (WADS) 2013, *Lecture Notes in Computer Science*, vol. 8037, Springer-Verlag, Berlin (2013) 316–327.

C35 D. Eppstein and D. Hirschberg, “From discrepancy to majority,” *Proc. 12th Latin American Theoretical Informatics Symposium (LATIN’16)*, Ensenada, Mexico, (2016) 390–402.

Although PKWARE will attempt to supply current and accurate information relating to its file formats, algorithms, and the subject programs, the possibility of error can not be eliminated. PKWARE therefore expressly disclaims any warranty that the information contained in the associated materials relating to the subject programs and/or the format of the files created or accessed by the subject programs and/or the algorithms used by the subject programs, or any other matter, is current, correct or accurate as delivered. Any risk of damage due to any possible inaccurate information is assumed by the user of the information. Furthermore, the information relating to the subject programs and/or the file formats created or accessed by the subject programs and/or the algorithms used by the subject programs is subject to change without notice.

General Format of a ZIP file -----

Files stored in arbitrary order. Large zipfiles can span multiple diskette media.

Overall zipfile format:

[local file header+file data] . . .
[central directory] end of central directory record

A. Local file header:

local file header signature	4 bytes	(0x04034b50)
version needed to extract	2 bytes	
general purpose bit flag	2 bytes	
compression method	2 bytes	
last mod file time	2 bytes	
last mod file date	2 bytes	
crc-32	4 bytes	
compressed size	4 bytes	
uncompressed size	4 bytes	
filename length	2 bytes	
extra field length	2 bytes	
filename (variable size)		
extra field (variable size)		

B. Central directory structure:

[file header] . . . end of central dir record

File header:

central file header signature	4 bytes	(0x02014b50)
version made by	2 bytes	
version needed to extract	2 bytes	
general purpose bit flag	2 bytes	
compression method	2 bytes	
last mod file time	2 bytes	
last mod file date	2 bytes	
crc-32	4 bytes	
compressed size	4 bytes	
uncompressed size	4 bytes	
filename length	2 bytes	
extra field length	2 bytes	

	APPNOTE-1.0.txt
file comment length	2 bytes
disk number start	2 bytes
internal file attributes	2 bytes
external file attributes	4 bytes
relative offset of local header	4 bytes

filename (variable size)
extra field (variable size)
file comment (variable size)

End of central dir record:

end of central dir signature	4 bytes	(0x06054b50)
number of this disk	2 bytes	
number of the disk with the start of the central directory	2 bytes	
total number of entries in the central dir on this disk	2 bytes	
total number of entries in the central dir	2 bytes	
size of the central directory	4 bytes	
offset of start of central directory with respect to the starting disk number	4 bytes	
zipfile comment length	2 bytes	
zipfile comment (variable size)		

C. Explanation of fields:

version made by

The upper byte indicates the host system (OS) for the file. Software can use this information to determine the line record format for text files etc. The current mappings are:

0 - MS-DOS and OS/2 (F.A.T. file systems)	
1 - Amiga	2 - VMS
3 - *nix	4 - VM/CMS
5 - Atari ST	6 - OS/2 H.P.F.S.
7 - Macintosh	8 - Z-System
9 - CP/M	10 thru 255 - unused

The lower byte indicates the version number of the software used to encode the file. The value/10 indicates the major version number, and the value mod 10 is the minor version number.

version needed to extract

The minimum software version needed to extract the file, mapped as above.

general purpose bit flag:

bit 0: If set, indicates that the file is encrypted.
bit 1: If the compression method used was type 6, Imploding, then this bit, if set, indicates an 8K sliding dictionary was used. If clear, then a 4K sliding dictionary was used.
bit 2: If the compression method used was type 6, Imploding, then this bit, if set, indicates an 3 Shannon-Fano trees were used to encode the sliding dictionary output. If clear, then 2 Shannon-Fano trees were used.

APPNOTE-1.0.txt

Note: Bits 1 and 2 are undefined if the compression method is other than type 6 (Imploding).

The upper three bits are reserved and used internally by the software when processing the zipfile. The remaining bits are unused in version 1.0.

compression method:

(see accompanying documentation for algorithm descriptions)

- 0 - The file is stored (no compression)
- 1 - The file is Shrunk
- 2 - The file is Reduced with compression factor 1
- 3 - The file is Reduced with compression factor 2
- 4 - The file is Reduced with compression factor 3
- 5 - The file is Reduced with compression factor 4
- 6 - The file is Imploded

date and time fields:

The date and time are encoded in standard MS-DOS format.

CRC-32:

The CRC-32 algorithm was generously contributed by David Schwaderer and can be found in his excellent book "C Programmers Guide to NetBIOS" published by Howard W. Sams & Co. Inc. The 'magic number' for the CRC is 0xdeb20e3. The proper CRC pre and post conditioning is used, meaning that the CRC register is pre-conditioned with all ones (a starting value of 0xffffffff) and the value is post-conditioned by taking the one's complement of the CRC residual.

compressed size:

uncompressed size:

The size of the file compressed and uncompressed, respectively.

filename length:

extra field length:

file comment length:

The length of the filename, extra field, and comment fields respectively. The combined length of any directory record and these three fields should not generally exceed 65,535 bytes.

disk number start:

The number of the disk on which this file begins.

internal file attributes:

The lowest bit of this field indicates, if set, that the file is apparently an ASCII or text file. If not set, that the file apparently contains binary data. The remaining bits are unused in version 1.0.

external file attributes:

The mapping of the external attributes is host-system dependent (see 'version made by'). For MS-DOS, the low order byte is the MS-DOS directory

attribute byte.

relative offset of local header:

This is the offset from the start of the first disk on which this file appears, to where the local header should be found.

filename:

The name of the file, with optional relative path. The path stored should not contain a drive or device letter, or a leading slash. All slashes should be forward slashes '/' as opposed to backwards slashes '\' for compatibility with Amiga and Unix file systems etc.

extra field:

This is for future expansion. If additional information needs to be stored in the future, it should be stored here. Earlier versions of the software can then safely skip this field, and find the next file or header. This field will be 0 length in version 1.0.

In order to allow different programs and different types of information to be stored in the 'extra' field in .ZIP files, the following structure should be used for all programs storing data in this field:

header1+data1 + header2+data2 . . .

Each header should consist of:

Header ID - 2 bytes
Data Size - 2 bytes

Note: all fields stored in Intel low-byte/high-byte order.

The Header ID field indicates the type of data that is in the following data block.

Header ID's of 0 thru 31 are reserved for use by PKWARE. The remaining ID's can be used by third party vendors for proprietary usage.

The Data Size field indicates the size of the following data block. Programs can use this value to skip to the next header block, passing over any data blocks that are not of interest.

Note: As stated above, the size of the entire .ZIP file header, including the filename, comment, and extra field should not exceed 64K in size.

In case two different programs should appropriate the same Header ID value, it is strongly recommended that each program place a unique signature of at least two bytes in size (and preferably 4 bytes or bigger) at the start of each data area. Every program should verify that it's unique signature is present, in addition to the Header ID value being correct, before assuming that it is a block of known type.

file comment:

The comment for this file.

number of this disk:

The number of this disk, which contains central directory end record.

number of the disk with the start of the central directory:

The number of the disk on which the central directory starts.

total number of entries in the central dir on this disk:

The number of central directory entries on this disk.

total number of entries in the central dir:

The total number of files in the zipfile.

size of the central directory:

The size (in bytes) of the entire central directory.

offset of start of central directory with respect to the starting disk number:

Offset of the start of the central directory on the disk on which the central directory starts.

zipfile comment length:

The length of the comment for this zipfile.

zipfile comment:

The comment for this zipfile.

D. General notes:

- 1) All fields unless otherwise noted are unsigned and stored in Intel low-byte:high-byte, low-word:high-word order.
- 2) String fields are not null terminated, since the length is given explicitly.
- 3) Local headers should not span disk boundaries. Also, even though the central directory can span disk boundaries, no single record in the central directory should be split across disks.
- 4) The entries in the central directory may not necessarily be in the same order that files appear in the zipfile.

♀
UnShrinking

Shrinking is a Dynamic Ziv-Lempel-Welch compression algorithm with partial clearing. The initial code size is 9 bits, and the maximum code size is 13 bits. Shrinking differs from conventional Dynamic Ziv-Lempel-Welch implementations in several respects:

- 1) The code size is controlled by the compressor, and is not automatically increased when codes larger than the current code size are created (but not necessarily used). When the decompressor encounters the code sequence 256 (decimal) followed by 1, it should increase the code size

read from the input stream to the next bit size. No blocking of the codes is performed, so the next code at the increased size should be read from the input stream immediately after where the previous code at the smaller bit size was read. Again, the decompressor should not increase the code size used until the sequence 256,1 is encountered.

- 2) When the table becomes full, total clearing is not performed. Rather, when the compressor emits the code sequence 256,2 (decimal), the decompressor should clear all leaf nodes from the Ziv-Lempel tree, and continue to use the current code size. The nodes that are cleared from the Ziv-Lempel tree are then re-used, with the lowest code value re-used first, and the highest code value re-used last. The compressor can emit the sequence 256,2 at any time.

♀

Expanding

The Reducing algorithm is actually a combination of two distinct algorithms. The first algorithm compresses repeated byte sequences, and the second algorithm takes the compressed stream from the first algorithm and applies a probabilistic compression method.

The probabilistic compression stores an array of 'follower sets' $S(j)$, for $j=0$ to 255, corresponding to each possible ASCII character. Each set contains between 0 and 32 characters, to be denoted as $S(j)[0], \dots, S(j)[m]$, where $m < 32$. The sets are stored at the beginning of the data area for a Reduced file, in reverse order, with $S(255)$ first, and $S(0)$ last.

The sets are encoded as $\{ N(j), S(j)[0], \dots, S(j)[N(j)-1] \}$, where $N(j)$ is the size of set $S(j)$. $N(j)$ can be 0, in which case the follower set for $S(j)$ is empty. Each $N(j)$ value is encoded in 6 bits, followed by $N(j)$ eight bit character values corresponding to $S(j)[0]$ to $S(j)[N(j)-1]$ respectively. If $N(j)$ is 0, then no values for $S(j)$ are stored, and the value for $N(j-1)$ immediately follows.

Immediately after the follower sets, is the compressed data stream. The compressed data stream can be interpreted for the probabilistic decompression as follows:

```

let Last-Character <- 0.
loop until done
  if the follower set S(Last-Character) is empty then
    read 8 bits from the input stream, and copy this
    value to the output stream.
  otherwise if the follower set S(Last-Character) is non-empty then
    read 1 bit from the input stream.
    if this bit is not zero then
      read 8 bits from the input stream, and copy this
      value to the output stream.
    otherwise if this bit is zero then
      read B(N(Last-Character)) bits from the input
      stream, and assign this value to I.
      Copy the value of S(Last-Character)[I] to the
      output stream.

```

assign the last value placed on the output stream to Last-Character.

end loop

$B(N(j))$ is defined as the minimal number of bits required to encode the value $N(j)-1$.

The decompressed stream from above can then be expanded to re-create the original file as follows:

let State <- 0.

loop until done

 read 8 bits from the input stream into C.

 case State of

 0: if C is not equal to DLE (144 decimal) then
 copy C to the output stream.
 otherwise if C is equal to DLE then
 let State <- 1.

 1: if C is non-zero then
 let V <- C.
 let Len <- L(V)
 let State <- F(Len).
 otherwise if C is zero then
 copy the value 144 (decimal) to the output stream.
 let State <- 0

 2: let Len <- Len + C
 let State <- 3.

 3: move backwards D(V,C) bytes in the output stream
 (if this position is before the start of the output
 stream, then assume that all the data before the
 start of the output stream is filled with zeros).
 copy Len+3 bytes from this position to the output stream.
 let State <- 0.

 end case

end loop

The functions F, L, and D are dependent on the 'compression factor', 1 through 4, and are defined as follows:

For compression factor 1:

 L(X) equals the lower 7 bits of X.

 F(X) equals 2 if X equals 127 otherwise F(X) equals 3.

 D(X,Y) equals the (upper 1 bit of X) * 256 + Y + 1.

For compression factor 2:

 L(X) equals the lower 6 bits of X.

 F(X) equals 2 if X equals 63 otherwise F(X) equals 3.

 D(X,Y) equals the (upper 2 bits of X) * 256 + Y + 1.

For compression factor 3:

 L(X) equals the lower 5 bits of X.

 F(X) equals 2 if X equals 31 otherwise F(X) equals 3.

 D(X,Y) equals the (upper 3 bits of X) * 256 + Y + 1.

For compression factor 4:

 L(X) equals the lower 4 bits of X.

 F(X) equals 2 if X equals 15 otherwise F(X) equals 3.

 D(X,Y) equals the (upper 4 bits of X) * 256 + Y + 1.

♀

Imploding

The Imploding algorithm is actually a combination of two distinct algorithms. The first algorithm compresses repeated byte

sequences using a sliding dictionary. The second algorithm is used to compress the encoding of the sliding dictionary output, using multiple Shannon-Fano trees.

The Imploding algorithm can use a 4K or 8K sliding dictionary size. The dictionary size used can be determined by bit 1 in the general purpose flag word, a 0 bit indicates a 4K dictionary while a 1 bit indicates an 8K dictionary.

The Shannon-Fano trees are stored at the start of the compressed file. The number of trees stored is defined by bit 2 in the general purpose flag word, a 0 bit indicates two trees stored, a 1 bit indicates three trees are stored. If 3 trees are stored, the first Shannon-Fano tree represents the encoding of the Literal characters, the second tree represents the encoding of the Length information, the third represents the encoding of the Distance information. When 2 Shannon-Fano trees are stored, the Length tree is stored first, followed by the Distance tree.

The Literal Shannon-Fano tree, if present is used to represent the entire ASCII character set, and contains 256 values. This tree is used to compress any data not compressed by the sliding dictionary algorithm. When this tree is present, the Minimum Match Length for the sliding dictionary is 3. If this tree is not present, the Minimum Match Length is 2.

The Length Shannon-Fano tree is used to compress the Length part of the (length,distance) pairs from the sliding dictionary output. The Length tree contains 64 values, ranging from the Minimum Match Length, to 63 plus the Minimum Match Length.

The Distance Shannon-Fano tree is used to compress the Distance part of the (length,distance) pairs from the sliding dictionary output. The Distance tree contains 64 values, ranging from 0 to 63, representing the upper 6 bits of the distance value. The distance values themselves will be between 0 and the sliding dictionary size, either 4K or 8K.

The Shannon-Fano trees themselves are stored in a compressed format. The first byte of the tree data represents the number of bytes of data representing the (compressed) Shannon-Fano tree minus 1. The remaining bytes represent the Shannon-Fano tree data encoded as:

High 4 bits: Number of values at this bit length + 1. (1 - 16)
Low 4 bits: Bit Length needed to represent value + 1. (1 - 16)

The Shannon-Fano codes can be constructed from the bit lengths using the following algorithm:

- 1) Sort the Bit Lengths in ascending order, while retaining the order of the original lengths stored in the file.
- 2) Generate the Shannon-Fano trees:

```
Code <- 0
CodeIncrement <- 0
LastBitLength <- 0
i <- number of Shannon-Fano codes - 1 (either 255 or 63)

loop while i >= 0
  Code = Code + CodeIncrement
  if BitLength(i) <> LastBitLength then
    LastBitLength=BitLength(i)
    CodeIncrement = 1 shifted left (16 - LastBitLength)
  ShannonCode(i) = Code
  i <- i - 1
end loop
```

- 3) Reverse the order of all the bits in the above ShannonCode() vector, so that the most significant bit becomes the least significant bit. For example, the value 0x1234 (hex) would become 0x2C48 (hex).
- 4) Restore the order of Shannon-Fano codes as originally stored within the file.

Example:

This example will show the encoding of a Shannon-Fano tree of size 8. Notice that the actual Shannon-Fano trees used for Imploding are either 64 or 256 entries in size.

Example: 0x02, 0x42, 0x01, 0x13

The first byte indicates 3 values in this table. Decoding the bytes:

0x42 = 5 codes of 3 bits long
 0x01 = 1 code of 2 bits long
 0x13 = 2 codes of 4 bits long

This would generate the original bit length array of:
 (3, 3, 3, 3, 3, 2, 4, 4)

There are 8 codes in this table for the values 0 thru 7. Using the algorithm to obtain the Shannon-Fano codes produces:

Val	Sorted	Constructed Code	Reversed Value	Order Restored	Original Length
0:	2	1100000000000000	11	101	3
1:	3	1010000000000000	101	001	3
2:	3	1000000000000000	001	110	3
3:	3	0110000000000000	110	010	3
4:	3	0100000000000000	010	100	3
5:	3	0010000000000000	100	11	2
6:	4	0001000000000000	1000	1000	4
7:	4	0000000000000000	0000	0000	4

The values in the Val, Order Restored and Original Length columns now represent the Shannon-Fano encoding tree that can be used for decoding the Shannon-Fano encoded data. How to parse the variable length Shannon-Fano values from the data stream is beyond the scope of this document. (See the references listed at the end of this document for more information.) However, traditional decoding schemes used for Huffman variable length decoding, such as the Greenlaw algorithm, can be successfully applied.

The compressed data stream begins immediately after the compressed Shannon-Fano data. The compressed data stream can be interpreted as follows:

```

loop until done
  read 1 bit from input stream.

  if this bit is non-zero then      (encoded data is literal data)
    if Literal Shannon-Fano tree is present
      read and decode character using Literal Shannon-Fano tree.
    otherwise
      read 8 bits from input stream.
      copy character to the output stream.
  otherwise                        (encoded data is sliding dictionary match)
    if 8K dictionary size
      read 7 bits for offset Distance (lower 7 bits of offset).
```

```

otherwise
    read 6 bits for offset Distance (lower 6 bits of offset).

using the Distance Shannon-Fano tree, read and decode the
    upper 6 bits of the Distance value.

using the Length Shannon-Fano tree, read and decode
    the Length value.

Length <- Length + Minimum Match Length

if Length = 63 + Minimum Match Length
    read 8 bits from the input stream,
    add this value to Length.

move backwards Distance+1 bytes in the output stream, and
copy Length characters from this position to the output
stream. (if this position is before the start of the output
stream, then assume that all the data before the start of
the output stream is filled with zeros).

```

```
end loop
```

```
♀
```

```
Decryption
```

```
-----
```

The encryption used in PKZIP was generously supplied by Roger Schlafly. PKWARE is grateful to Mr. Schlafly for his expert help and advice in the field of data encryption.

PKZIP encrypts the compressed data stream. Encrypted files must be decrypted before they can be extracted.

Each encrypted file has an extra 12 bytes stored at the start of the data area defining the encryption header for that file. The encryption header is originally set to random values, and then itself encrypted, using 3, 32-bit keys. The key values are initialized using the supplied encryption password. After each byte is encrypted, the keys are then updated using psuedo-random number generation techniques in combination with the same CRC-32 algorithm used in PKZIP and described elsewhere in this document.

The following is the basic steps required to decrypt a file:

- 1) Initialize the three 32-bit keys with the password.
- 2) Read and decrypt the 12-byte encryption header, further initializing the encryption keys.
- 3) Read and decrypt the compressed data stream using the encryption keys.

Step 1 - Initializing the encryption keys

```
-----
```

```

Key(0) <- 305419896
Key(1) <- 591751049
Key(2) <- 878082192

```

```

loop for i <- 0 to length(password)-1
    update_keys(password(i))
end loop

```

Where update_keys() is defined as:

```

update_keys(char):
    Key(0) <- crc32(key(0), char)
    Key(1) <- Key(1) + (Key(0) & 000000ffH)

```

```

Key(1) <- Key(1) * 134775813 + 1
Key(2) <- crc32(key(2), key(1) >> 24)
end update_keys

```

Where `crc32(old_crc, char)` is a routine that given a CRC value and a character, returns an updated CRC value after applying the CRC-32 algorithm described elsewhere in this document.

Step 2 - Decrypting the encryption header

The purpose of this step is to further initialize the encryption keys, based on random data, to render a plaintext attack on the data ineffective.

Read the 12-byte encryption header into Buffer, in locations Buffer(0) thru Buffer(11).

```

loop for i <- 0 to 11
  C <- buffer(i) ^ decrypt_byte()
  update_keys(C)
  buffer(i) <- C
end loop

```

Where `decrypt_byte()` is defined as:

```

unsigned char decrypt_byte()
  local unsigned short temp
  temp <- Key(2) | 2
  decrypt_byte <- (temp * (temp ^ 1)) >> 8
end decrypt_byte

```

After the header is decrypted, the last two bytes in Buffer should be the high-order word of the CRC for the file being decrypted, stored in Intel low-byte/high-byte order. This can be used to test if the password supplied is correct or not.

Step 3 - Decrypting the compressed data stream

The compressed data stream can be decrypted as follows:

```

loop until done
  read a character into C
  Temp <- C ^ decrypt_byte()
  update_keys(temp)
  output Temp
end loop
⌘

```

In addition to the above mentioned contributors to PKZIP and PKUNZIP, I would like to extend special thanks to Robert Mahoney for suggesting the extension .ZIP for this software.

References:

Storer, James A. "Data Compression, Methods and Theory",
Computer Science Press, 1988

APPNOTE-1.0.txt

Held, Gilbert "Data Compression, Techniques and Applications,
Hardware and Software Considerations"
John Wiley & Sons, 1987

0
→

NOTE: click [here](#) if you get an empty page.

FILE(1)

FILE(1)

NAME

`file` - determine file type

SYNOPSIS

```
file [ -bcnsvzl ] [ -f namefile ] [ -m magicfiles ] file ...
```

DESCRIPTION

This manual page documents version 3.27 of the `file` command. `File` tests each argument in an attempt to classify it. There are three sets of tests, performed in this order: filesystem tests, magic number tests, and language tests. The *first* test that succeeds causes the file type to be printed.

The type printed will usually contain one of the words `text` (the file contains only ASCII characters and is probably safe to read on an ASCII terminal), `executable` (the file contains the result of compiling a program in a form understandable to some UNIX kernel or another), or `data` meaning anything else (data is usually `'binary'` or non-printable). Exceptions are well-known file formats (core files, tar archives) that are known to contain binary data. When modifying the file `/usr/share/magic` or the program itself, **preserve these keywords**. People depend on knowing that all the readable files in a directory have the word `'text'` printed. Don't do as Berkeley did - change `'shell commands text'` to `'shell script'`.

The filesystem tests are based on examining the return from a `stat(2)` system call. The program checks to see if the file is empty, or if it's some sort of special file. Any known file types appropriate to the system you are running on (sockets, symbolic links, or named pipes (FIFOs) on those systems that implement them) are intuited if they are defined in the system header file `sys/stat.h`.

The magic number tests are used to check for files with data in particular fixed formats. The canonical example of this is a binary executable (compiled program) `a.out` file, whose format is defined in `a.out.h` and possibly `exec.h` in the standard include directory. These files have a `'magic number'` stored in a particular place near the beginning of the file that tells the UNIX operating system that the file is a binary executable, and which of several types thereof. The concept of `'magic number'` has been applied by extension to data files. Any file with some invariant identifier at a small fixed offset into the file can usually be described in this way. The information in these files is read from the magic file `/usr/share/magic`.

If an argument appears to be an ASCII file, `file` attempts

to guess its language. The language tests look for

Copyright but distributable 1

FILE(1)

FILE(1)

particular strings (cf *names.h*) that can appear anywhere in the first few blocks of a file. For example, the key word **.br** indicates that the file is most likely a **troff(1)** input file, just as the keyword **struct** indicates a C program. These tests are less reliable than the previous two groups, so they are performed last. The language test routines also test for some miscellany (such as **tar(1)** archives) and determine whether an unknown file should be labelled as 'ascii text' or 'data'.

OPTIONS

- b Do not prepend filenames to output lines (brief mode).
- c Cause a checking printout of the parsed form of the magic file. This is usually used in conjunction with **-m** to debug a new magic file before installing it.
- f **namefile**
Read the names of the files to be examined from *namefile* (one per line) before the argument list. Either *namefile* or at least one filename argument must be present; to test the standard input, use '-' as a filename argument.
- m **list** Specify an alternate list of files containing magic numbers. This can be a single file, or a colon-separated list of files.
- n Force stdout to be flushed after check a file. This is only useful if checking a list of files. It is intended to be used by programs want file type output from a pipe.
- v Print the version of the program and exit.
- z Try to look inside compressed files.
- L option causes symlinks to be followed, as the like-named option in **ls(1)**. (on systems that support symbolic links).
- s Normally, **file** only attempts to read and determine the type of argument files which **stat(2)** reports are ordinary files. This prevents problems, because reading special files may have peculiar consequences. Specifying the **-s** option causes **file** to also read argument files which are block or character special files. This is useful for determining the filesystem types of the data in raw disk partitions, which are block special files. This option also causes **file** to disregard the file size as reported by **stat(2)** since on some

Copyright but distributable 2

FILE(1)

FILE(1)

systems it reports a zero size for raw disk parti

tions.

FILES

`/usr/share/magic` - default list of magic numbers

ENVIRONMENT

The environment variable **MAGIC** can be used to set the default magic number files.

SEE ALSO

`magic(4)` - description of magic file format.
`strings(1)`, `od(1)`, [hexdump\(1\)](#) - tools for examining non-textfiles.

STANDARDS CONFORMANCE

This program is believed to exceed the System V Interface Definition of FILE(CMD), as near as one can determine from the vague language contained therein. Its behaviour is mostly compatible with the System V program of the same name. This version knows more magic, however, so it will produce different (albeit more accurate) output in many cases.

The one significant difference between this version and System V is that this version treats any white space as a delimiter, so that spaces in pattern strings must be escaped. For example,

```
>10 string language impress (imPRESS data)
in an existing magic file would have to be changed to
>10 string language\ impress (imPRESS data)
In addition, in this version, if a pattern string contains
a backslash, it must be escaped. For example
0 string \begindata Andrew Toolkit document
in an existing magic file would have to be changed to
0 string \\begindata Andrew Toolkit document
```

SunOS releases 3.2 and later from Sun Microsystems include a `file(1)` command derived from the System V one, but with some extensions. My version differs from Sun's only in minor ways. It includes the extension of the ``&'` operator, used as, for example,

```
>16 long&0x7fffffff >0 not stripped
```

MAGIC DIRECTORY

The magic file entries have been collected from various sources, mainly USENET, and contributed by various authors. Christos Zoulas (address below) will collect additional or corrected magic file entries. A consolidation of magic file entries will be distributed periodically.

The order of entries in the magic file is significant.

Depending on what system you are using, the order that

Copyright but distributable 3

FILE(1)

FILE(1)

they are put together may be incorrect. If your old `file` command uses a magic file, keep the old magic file around for comparison purposes (rename it to `/usr/share/magic.orig`).

EXAMPLES

```
$ file file.c file /dev/hda
file.c:  C program text
file:    ELF 32-bit LSB executable, Intel 80386, version 1,
        dynamically linked, not stripped
/dev/hda: block special

$ file -s /dev/hda{,1,2,3,4,5,6,7,8,9,10}
/dev/hda:  x86 boot sector
/dev/hda1: Linux/i386 ext2 filesystem
/dev/hda2: x86 boot sector
/dev/hda3: x86 boot sector, extended partition table
/dev/hda4: Linux/i386 ext2 filesystem
/dev/hda5: Linux/i386 swap file
/dev/hda6: Linux/i386 swap file
/dev/hda7: Linux/i386 swap file
/dev/hda8: Linux/i386 swap file
/dev/hda9: empty
/dev/hda10: empty
```

HISTORY

There has been a `file` command in every UNIX since at least Research Version 6 (man page dated January, 1975). The System V version introduced one significant major change: the external list of magic number types. This slowed the program down slightly but made it a lot more flexible.

This program, based on the System V version, was written by Ian Darwin without looking at anybody else's source code.

John Gilmore revised the code extensively, making it better than the first version. Geoff Collyer found several inadequacies and provided some magic file entries. The program has undergone continued evolution since.

AUTHOR

Written by Ian F. Darwin, UUCP address {utzoo | ihnp4}!darwin!ian, Internet address ian@sq.com, postal address: P.O. Box 603, Station F, Toronto, Ontario, CANADA M4Y 2L8.

Altered by Rob McMahon, cudcv@warwick.ac.uk, 1989, to extend the ``&'` operator from simple ``x&y; != 0'` to ``x&y; op z'`.

Altered by Guy Harris, guy@netapp.com, 1993, to:

put the ```old-style''`&'` operator back the way it

Copyright but distributable

4

FILE(1)

FILE(1)

was, because 1) Rob McMahon's change broke the previous style of usage, 2) the SunOS ``new-style'' `&' operator, which this version of **file** supports, also handles `x&y; op z', and 3) Rob's change wasn't documented in any case;

put in multiple levels of `>';

put in ``beshort'', ``leshort'', etc. keywords to look at numbers in the file in a specific byte order, rather than in the native byte order of the process running **file**.

Changes by Ian Darwin and various authors including Christos Zoulas (christos@astron.com), 1990-1999.

LEGAL NOTICE

Copyright (c) Ian F. Darwin, Toronto, Canada, 1986, 1987, 1988, 1989, 1990, 1991, 1992, 1993.

This software is not subject to and may not be made subject to any license of the American Telephone and Telegraph Company, Sun Microsystems Inc., Digital Equipment Inc., Lotus Development Inc., the Regents of the University of California, The X Consortium or MIT, or The Free Software Foundation.

This software is not subject to any export provision of the United States Department of Commerce, and may be exported to any country or planet.

Permission is granted to anyone to use this software for any purpose on any computer system, and to alter it and redistribute it freely, subject to the following restrictions:

1. The author is not responsible for the consequences of use of this software, no matter how awful, even if they arise from flaws in it.
2. The origin of this software must not be misrepresented, either by explicit claim or by omission. Since few users ever read sources, credits must appear in the documentation.
3. Altered versions must be plainly marked as such, and must not be misrepresented as being the original software. Since few users ever read sources, credits must appear in the documentation.
4. This notice may not be removed or altered.

A few support files (*getopt*, *strtok*) distributed with this package are by Henry Spencer and are subject to the same

Copyright but distributable

5

FILE(1)

FILE(1)

terms as above.

A few simple support files (*strtol*, *strchr*) distributed

with this package are in the public domain; they are so marked.

The files *tar.h* and *is_tar.c* were written by John Gilmore from his public-domain *tar* program, and are not covered by the above restrictions.

BUGS

There must be a better way to automate the construction of the Magic file from all the glop in Magdir. What is it? Better yet, the magic file should be compiled into binary (say, *ndbm(3)* or, better yet, fixed-length ASCII strings for use in heterogenous network environments) for faster startup. Then the program would run as fast as the Version 7 program of the same name, with the flexibility of the System V version.

File uses several algorithms that favor speed over accuracy, thus it can be misled about the contents of ASCII files.

The support for ASCII files (primarily for programming languages) is simplistic, inefficient and requires recompilation to update.

There should be an ```else''` clause to follow a series of continuation lines.

The magic file and keywords should have regular expression support. Their use of ASCII TAB as a field delimiter is ugly and makes it hard to edit the files, but is entrenched.

It might be advisable to allow upper-case letters in keywords for e.g., *troff(1)* commands vs man page macros. Regular expression support would make this easy.

The program doesn't grok FORTRAN. It should be able to figure FORTRAN by seeing some keywords which appear indented at the start of line. Regular expression support would make this easy.

The list of keywords in *ascmagic* probably belongs in the Magic file. This could be done by using some keyword like ```*''` for the offset value.

Another optimisation would be to sort the magic file so that we can just run down all the tests for the first byte, first word, first long, etc, once we have fetched it. Complain about conflicts in the magic file entries. Make a rule that the magic entries sort based on file

Copyright but distributable

6

FILE(1)

FILE(1)

offset rather than position within the magic file?

The program should provide a way to give an estimate of ```how good''` a guess is. We end up removing guesses (e.g. ```From ''` as first 5 chars of file) because they are not as good as other guesses (e.g. ```Newsgroups:''` versus `"Return-Path:"`). Still, if the others don't pan out, it should be possible to use the first guess.

This program is slower than some vendors' file commands.

This manual page, and particularly this section, is too long.

AVAILABILITY

You can obtain the original author's latest version by anonymous FTP on **ftp.astron.com** in the directory */pub/file/file-X.YY.tar.gz*

Copyright but distributable

7

Š 1994 [Man-cgi 1.15](#), Panagiotis Christias <christia@theseas.ntua.gr>

NOTE: click [here](#) if you get an empty page.

FILE(1)

FILE(1)

NAME

`file` - determine file type

SYNOPSIS

```
file [ -bcnsvzl ] [ -f namefile ] [ -m magicfiles ] file ...
```

DESCRIPTION

This manual page documents version 3.27 of the `file` command. `File` tests each argument in an attempt to classify it. There are three sets of tests, performed in this order: filesystem tests, magic number tests, and language tests. The *first* test that succeeds causes the file type to be printed.

The type printed will usually contain one of the words **text** (the file contains only ASCII characters and is probably safe to read on an ASCII terminal), **executable** (the file contains the result of compiling a program in a form understandable to some UNIX kernel or another), or **data** meaning anything else (data is usually 'binary' or non-printable). Exceptions are well-known file formats (core files, tar archives) that are known to contain binary data. When modifying the file `/usr/share/magic` or the program itself, **preserve these keywords**. People depend on knowing that all the readable files in a directory have the word 'text' printed. Don't do as Berkeley did - change 'shell commands text' to 'shell script'.

The filesystem tests are based on examining the return from a `stat(2)` system call. The program checks to see if the file is empty, or if it's some sort of special file. Any known file types appropriate to the system you are running on (sockets, symbolic links, or named pipes (FIFOs) on those systems that implement them) are intuited if they are defined in the system header file `sys/stat.h`.

The magic number tests are used to check for files with data in particular fixed formats. The canonical example of this is a binary executable (compiled program) `a.out` file, whose format is defined in `a.out.h` and possibly `exec.h` in the standard include directory. These files have a 'magic number' stored in a particular place near the beginning of the file that tells the UNIX operating system that the file is a binary executable, and which of several types thereof. The concept of 'magic number' has been applied by extension to data files. Any file with

If an argument appears to be an ASCII file, **file** attempts to guess its language. The language tests look for

Copyright but distributable 1

FILE(1)

FILE(1)

particular strings (cf *names.h*) that can appear anywhere in the first few blocks of a file. For example, the key word **.br** indicates that the file is most likely a **troff(1)** input file, just as the keyword **struct** indicates a C program. These tests are less reliable than the previous two groups, so they are performed last. The language test routines also test for some miscellany (such as **tar(1)** archives) and determine whether an unknown file should be labelled as 'ascii text' or 'data'.

OPTIONS

- b Do not prepend filenames to output lines (brief mode).
- c Cause a checking printout of the parsed form of the magic file. This is usually used in conjunction with **-m** to debug a new magic file before installing it.
- f *namefile*
Read the names of the files to be examined from *namefile* (one per line) before the argument list. Either *namefile* or at least one filename argument must be present; to test the standard input, use **``-''** as a filename argument.
- m *list* Specify an alternate list of files containing magic numbers. This can be a single file, or a colon-separated list of files.
- n Force stdout to be flushed after check a file. This is only useful if checking a list of files. It is intended to be used by programs want file type output from a pipe.
- v Print the version of the program and exit.
- z Try to look inside compressed files.
- L option causes symlinks to be followed, as the like-named option in **ls(1)**. (on systems that support symbolic links).
- s Normally, **file** only attempts to read and determine the type of argument files which **stat(2)** reports are ordinary files. This prevents problems, because reading special files may have peculiar consequences. Specifying the **-s** option causes **file** to also read argument files which are block or character special files. This is useful for determining the filesystem types of the data in raw disk partitions, which are block special files. This option also causes **file** to disregard the file size as reported by **stat(2)** since on some

systems it reports a zero size for raw disk partitions.

FILES

`/usr/share/magic` - default list of magic numbers

ENVIRONMENT

The environment variable **MAGIC** can be used to set the default magic number files.

SEE ALSO

`magic(4)` - description of magic file format.
`strings(1)`, `od(1)`, [hexdump\(1\)](#) - tools for examining non-textfiles.

STANDARDS CONFORMANCE

This program is believed to exceed the System V Interface Definition of FILE(CMD), as near as one can determine from the vague language contained therein. Its behaviour is mostly compatible with the System V program of the same name. This version knows more magic, however, so it will produce different (albeit more accurate) output in many cases.

The one significant difference between this version and System V is that this version treats any white space as a delimiter, so that spaces in pattern strings must be escaped. For example,

```
>10 string language impress (imPRESS data)
```

in an existing magic file would have to be changed to

```
>10 string language\ impress (imPRESS data)
```

In addition, in this version, if a pattern string contains a backslash, it must be escaped. For example

```
0 string \begindata Andrew Toolkit document
```

in an existing magic file would have to be changed to

```
0 string \\begindata Andrew Toolkit document
```

SunOS releases 3.2 and later from Sun Microsystems include a `file(1)` command derived from the System V one, but with some extensions. My version differs from Sun's only in minor ways. It includes the extension of the ``&'` operator, used as, for example,

```
>16 long&0x7fffffff >0 not stripped
```

MAGIC DIRECTORY

The magic file entries have been collected from various sources, mainly USENET, and contributed by various authors. Christos Zoulas (address below) will collect

The order of entries in the magic file is significant.
Depending on what system you are using, the order that

Copyright but distributable 3

FILE(1)

FILE(1)

they are put together may be incorrect. If your old **file** command uses a magic file, keep the old magic file around for comparison purposes (rename it to `/usr/share/magic.orig`).

EXAMPLES

```
$ file file.c file /dev/hda
file.c:  C program text
file:    ELF 32-bit LSB executable, Intel 80386, version 1,
         dynamically linked, not stripped
/dev/hda: block special

$ file -s /dev/hda{1,2,3,4,5,6,7,8,9,10}
/dev/hda:  x86 boot sector
/dev/hda1: Linux/i386 ext2 filesystem
/dev/hda2: x86 boot sector
/dev/hda3: x86 boot sector, extended partition table
/dev/hda4: Linux/i386 ext2 filesystem
/dev/hda5: Linux/i386 swap file
/dev/hda6: Linux/i386 swap file
/dev/hda7: Linux/i386 swap file
/dev/hda8: Linux/i386 swap file
/dev/hda9: empty
/dev/hda10: empty
```

HISTORY

There has been a **file** command in every UNIX since at least Research Version 6 (man page dated January, 1975). The System V version introduced one significant major change: the external list of magic number types. This slowed the program down slightly but made it a lot more flexible.

This program, based on the System V version, was written by Ian Darwin without looking at anybody else's source code.

John Gilmore revised the code extensively, making it better than the first version. Geoff Collyer found several inadequacies and provided some magic file entries. The program has undergone continued evolution since.

AUTHOR

Written by Ian F. Darwin, UUCP address {utzoo | ihnp4}!darwin!ian, Internet address ian@sq.com, postal address: P.O. Box 603, Station F, Toronto, Ontario, CANADA M4Y 2L8.

Altered by Rob McMahon, cudcv@warwick.ac.uk, 1989, to extend the ``&'` operator from simple ``x&y; != 0'` to ``x&y; op`

put the old-style & operator back the way it

Copyright but distributable 4

FILE(1)

FILE(1)

was, because 1) Rob McMahon's change broke the previous style of usage, 2) the SunOS ``new-style'` `&' operator, which this version of `file` supports, also handles `x&y; op z', and 3) Rob's change wasn't documented in any case;

put in multiple levels of `>';

put in ``beshort'', ``leshort'', etc. keywords to look at numbers in the file in a specific byte order, rather than in the native byte order of the process running `file`.

Changes by Ian Darwin and various authors including Christos Zoulas (christos@astron.com), 1990-1999.

LEGAL NOTICE

Copyright (c) Ian F. Darwin, Toronto, Canada, 1986, 1987, 1988, 1989, 1990, 1991, 1992, 1993.

This software is not subject to and may not be made subject to any license of the American Telephone and Telegraph Company, Sun Microsystems Inc., Digital Equipment Inc., Lotus Development Inc., the Regents of the University of California, The X Consortium or MIT, or The Free Software Foundation.

This software is not subject to any export provision of the United States Department of Commerce, and may be exported to any country or planet.

Permission is granted to anyone to use this software for any purpose on any computer system, and to alter it and redistribute it freely, subject to the following restrictions:

1. The author is not responsible for the consequences of use of this software, no matter how awful, even if they arise from flaws in it.
2. The origin of this software must not be misrepresented, either by explicit claim or by omission. Since few users ever read sources, credits must appear in the documentation.
3. Altered versions must be plainly marked as such, and must not be misrepresented as being the original software. Since few users ever read sources, credits must appear in the documentation.
4. This notice may not be removed or altered.

A few support files (*getopt*, *strtok*) distributed with this package are by Henry Spencer and are subject to the same

Copyright but distributable 5

A few simple support files (*strtbl*, *strchr*) distributed with this package are in the public domain; they are so marked.

The files *tar.h* and *is_tar.c* were written by John Gilmore from his public-domain *tar* program, and are not covered by the above restrictions.

BUGS

There must be a better way to automate the construction of the Magic file from all the glop in Magdir. What is it? Better yet, the magic file should be compiled into binary (say, *ndbm(3)* or, better yet, fixed-length ASCII strings for use in heterogenous network environments) for faster startup. Then the program would run as fast as the Version 7 program of the same name, with the flexibility of the System V version.

File uses several algorithms that favor speed over accuracy, thus it can be misled about the contents of ASCII files.

The support for ASCII files (primarily for programming languages) is simplistic, inefficient and requires recompilation to update.

There should be an ```else''` clause to follow a series of continuation lines.

The magic file and keywords should have regular expression support. Their use of ASCII TAB as a field delimiter is ugly and makes it hard to edit the files, but is entrenched.

It might be advisable to allow upper-case letters in keywords for e.g., *troff(1)* commands vs man page macros. Regular expression support would make this easy.

The program doesn't grok FORTRAN. It should be able to figure FORTRAN by seeing some keywords which appear indented at the start of line. Regular expression support would make this easy.

The list of keywords in *ascmagic* probably belongs in the Magic file. This could be done by using some keyword like ```*''` for the offset value.

Another optimisation would be to sort the magic file so that we can just run down all the tests for the first byte, first word, first long, etc, once we have fetched it. Complain about conflicts in the magic file entries. Make a rule that the magic entries sort based on file

Copyright but distributable

6

FILE(1)

FILE(1)

offset rather than position within the magic file?

The program should provide a way to give an estimate of ```how good''` a guess is. We end up removing guesses (e.g. ```From ''` as first 5 chars of file) because they are not as good as other guesses (e.g. ```Newsgroups:''` versus

This manual page, and particularly this section, is too long.

AVAILABILITY

You can obtain the original author's latest version by anonymous FTP on **ftp.astron.com** in the directory */pub/file/file-X.YY.tar.gz*

Copyright but distributable 7

Š 1994 [Man-cgi 1.15](#), Panagiotis Christias <christia@theseas.ntua.gr>