BIT ROT

Brian Hayes

A reprint from

American Scientist

the magazine of Sigma Xi, the Scientific Research Society

Volume 86, Number 5
September–October, 1998
pages 410–415

BIT ROT

Brian Hayes

Somewhere in a cobwebby corner of my computer's hard disk are a few manuscripts I wrote 15 years ago on my first PC. The word-processing software I used then was grandly named The Final Word. It was anything but. I've gone through a dozen word processors since then, and nearly as many computers. To keep older documents accessible, I've had to transfer and transform them repeatedly, from one disk to the next and from one file format to another. And still I have yet to find the *Final* Word. Sooner or later I'll be gathering up my digital belongings yet again and converting them to some new format. This time I'll have 12,000 files in tow. I can't wait.

But my personal data-migration problems are puny compared with those of corporations, universities, libraries and publishers. (Imagine the plight of the National Archives, the agency charged with preserving everything the U.S. government deems to be worth keeping.) And the material to be preserved is not just text. Obsolete storage media and file formats are just as vexing when the files hold other kinds of information, such as images, engineering drawings, the numerical results of scientific experiments, digitized audio and video, maps, tax returns, databases.

One cause for worry among archivists is the impermanence of digital storage media. In this respect civilization has been going downhill ever since Mesopotamia. Paper documents cannot match the longevity of the Sumerians' clay tablets, and magnetic media seem to be even more evanescent than paper. That's disturbing news, and yet I suspect that relatively few disks or tapes have yet died of old age. Long before the disk wears out or succumbs to bit rot, the machine that reads the disk has become a museum piece. So the immediate challenge is not preserving the information but preserving the means to get at it.

Occasionally, the rescue of some long-neglected digital resource calls for heroic measures, such as reconstructing an antique tape drive. But most file transfers and translations are routine; utility software handles the conversion, though often with a minor loss of information. Even when the process is easy and successful, however, file conversion is a nuisance. It's a lot like moving your household—more work than you expected, and a few dishes always get broken. As my stockpile of files for the digital U-Haul continues to grow, I dread the prospect more and more. I daydream of hiking into the woods as a cybersurvivalist, refusing ever again to upgrade my hardware and software. If I stock up on spare parts—80-megabyte disk drives, 30-pin SIMMs—I could live out my remaining years in a log cabin with a Macintosh SE/30.

Most likely I would not be alone in the woods, but before I begin hoarding the computer equivalent of canned goods it seems prudent to consider less-extreme alternatives. All I really want is some way of representing digital information that I can stick with for a while. I want a file format for the ages—a single format that will serve many purposes and continue to work with many combinations of hardware and software. Which format is that? I don't have a definitive answer—not even an answer that meets all my own immediate needs. But I think I know where to look for inspiration. Here's a hint: Among all the kinds of things stored in computers, the ones that are hardest to keep up-to-date and hardest to move from one platform to another are programs for the computer itself. Maybe programmers know something the rest of us ought to learn.

The $1.29 *Oeuvre*

Are all those bits and bytes worth saving? Surely not. But if they're not worth saving, they're also not worth throwing out. The cost of magnetic disk storage is roughly 10 cents per megabyte these days. Tapes and CD-ROMs are even cheaper. My entire *oeuvre*—everything I've ever written for publication, as well as all the private and personal ephemera of a lifetime, from school compositions to love letters to grocery lists—all this would fit on a single CD-ROM. The storage cost for a lifetime's worth of words is $1.29. That's not much incentive for cleaning out the attic.

Lately I've found an even better reason for keeping those files. I've discovered that I'm not just a writer anymore—I'm a content provider! And my disk drive is stuffed full of *content*.

In years past an author's final product was the printed page. The computer file was nothing

Brian Hayes is a former editor of American Scientist. *Address: 211 Dacian Avenue, Durham, NC 27701. Internet: bhayes@amsci.org.*

more than an intermediate stage in the process of putting ink on paper. Once the presses were rolling, the disk files had no further value, and they could be discarded just as carbon copies were in the age of the typewriter.

All that has changed. Print is no longer the only destiny of the written word. This column, for example, will not only be bound into a printed magazine but will also be posted on the *American Scientist* Web site. Indeed, it will be offered in several electronic forms, including Postscript and PDF files that mimic the appearance of the magazine pages and a version coded in HTML, the native language of the Web. Someday the same column might be made available in still other media, such as CD-ROM. An abstract might be prepared automatically for a bibliographic database. The article might be reprinted in an anthology with a different typographic design, or it could be reproduced through a print-on-demand service for classroom use. Who knows what else might be in prospect? Last year's printout and this year's Web page could be next year's virtual-reality environment. Or tee-shirt.

In this new world of digital content, computer files are most certainly not disposable intermediate forms. The disk version is the master document, from which everything else derives. I might well throw away printouts and proofs, but I keep the disks. Furthermore, I worry about files that might be stranded or orphaned as computer hardware and software evolve.

This is not just my problem (although I may be more compulsive about it than most). In the sciences, almost everyone is becoming a content provider. Papers are submitted to journals and conferences in electronic form, and they may also appear as electronically distributed e-prints. Supporting data, such as genetic sequences, wind up in public databases. Many of these digital documents are considered a permanent part of the scientific literature. Their life expectancy is greater than that of the software that created them.

The Nitty Gritty

Strictly speaking, a computer file doesn't have *a* format; it has many formats, built in layers one atop the other. At the bottom of the hierarchy is the pattern of magnetized stripes on a disk or tape, or the microscopic pits in the reflective surface of a CD-ROM. This physical layer is the domain of hardware; you can't even perceive the recorded information, much less make sense of it, without the right machinery.

At the next level, patterns of bits are interpreted as numbers, characters, images and the like. The meaning of the patterns is always not obvious. Numbers can be stored in a baffling variety of formats. An integer might be represented by 8, 16, 32 or 64 bits. The bits could be read from left to right or from right to left. Negative numbers could be encoded according to either of two conventions, called one's complement and two's complement. Other variations include binary-coded decimal and floating-point numbers.

For text the situation is not much better. "Plain ASCII text" is often considered the lowest common denominator among computer file formats—a rudimentary language that any system ought to understand—but in practice it doesn't always work that way. ASCII stands for American Standard Code for Information Interchange. The "American" part of the name is a tip-off to one problem: ASCII represents only the characters commonly appearing in American English. If a text includes anything else—such as letters with accents or mathematical symbols—it lies beyond the bounds of pure ASCII.

Each ASCII character is represented by a seven-bit binary number, which has room for values in the range from 0 to 127. Most computers store information in bytes of eight bits each, allowing for another 128 characters. Unfortunately, every designer seems to have chosen a different set of extra characters. Not that there aren't standards for the use of the eighth bit. That's just the problem: There are more than a dozen of them. Grown men and women have given up decades of their lives to sit on committees arguing over the proper place of the dollar sign in computer character sets.

Many of ASCII's limitations are addressed in a new standard for character representation called Unicode. By giving each character two bytes instead of one, Unicode can specify more than 65,000 characters, enough for all the world's major alphabetic languages as well as the thousands of symbols in Chinese, Japanese and Korean. Unicode seems to be catching on. It is built into Microsoft Windows NT and the Java programming language, and Apple has announced its plan to support the standard. In the long run, this is good news; Unicode will solve some ticklish problems. On the other hand, it will mean another round of conversions for those 12,000 files I drag around be-

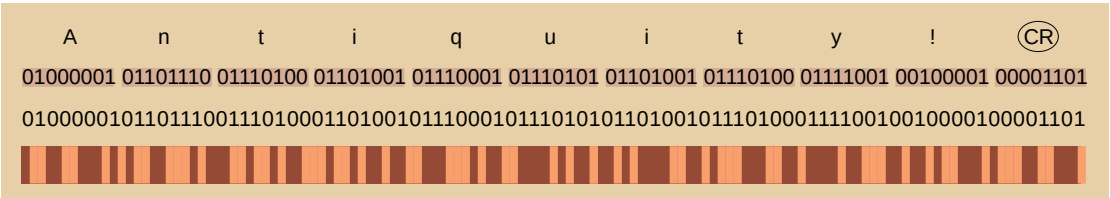

Figure 1. Hierarchy of digital file formats begins with the pattern of magnetized stripes on a disk or tape, which are interpreted as a stream of bits. The bits have higher-level structures imposed upon them; in this case the bits are organized into eight-bit bytes, which represent ASCII text and control codes, such as the carriage return at right.

hind me. Indeed, almost every computer file in existence today may eventually need to be converted.

The Higher Formatting

How computers represent numbers and characters is not something that most of us spend time brooding over. In any event, we don't have much choice about these low-level formats. Once you choose a computer, those decisions are made for you. But further layers in the hierarchy of file formats are not rooted so deeply in the silicon and the operating system.

When you write with a word-processing program, the information stored in the document includes more than just a sequence of alphabetic characters and punctuation marks. There are also formatting codes that indicate how the words are to look on the page. The codes specify typefaces (Times, Caslon, Palatino), type sizes (10 point, 12 point), stylistic variations (italic, boldface, superscript), the alignment of text (centered, flush-left, justified) and dozens of other properties. Sometimes the codes are explicitly entered into the text; in The Final Word, which was inspired by a famous text-editing program called Emacs, I would italicize a word by typing "@i<Titanic>." More recent software generally hides the formatting codes and shows only the results of applying them. When you select the command to italicize a word, the word appears in italics on the display screen, as if the text were simply stored inside the computer in italic type. This is an illusion. A computer's memory has neither italic bits nor roman ones. Hidden somewhere in the document file are explicit markers indicating the change in type style.

Some of the formatting information can be complicated. Particularly troublesome are nonlocal constructs, such as footnotes, cross-references and markers for index and table-of-contents entries. Consider a manuscript with consecutively numbered notes printed at the end. When a note is entered in the middle of the text, the number that appears there depends on how many notes precede it, while the output of the note itself has to wait until the rest of the manuscript has been processed. Thus the document cannot be viewed as a strictly linear text, with characters arranged in sequence; there are data structures that span the entire file. Features of this kind tend to be the hardest to translate when you convert a file to a new computer system or to a new purpose such as presentation on the Web.

Some files include formatting at an even higher level of abstraction, with labels that indicate the function of various parts of the document, rather than instructions about how they are to appear. The most familiar examples come from HTML, the Hypertext Markup Language of the Web. Headings and subheadings in an HTML document can be labeled with tags such as <h1> and <h2>, which indicate the relative importance of the headings but don't say directly how they should look; decisions about visual formatting are deferred until the document is displayed. Similarly, a phrase can be marked with the tag to indicate it bears emphasis, or with the tag for strong emphasis. The emphatic text is usually displayed in italic or boldface type, but those are not the only possibilities; an old-fashioned printer might underline the phrase, and a text-to-speech system might make a change of intonation.

This more abstract style of formatting is known variously as generic or descriptive markup, in contrast to visual or presentational markup. In this context "markup" refers to anything included in the file that's not part of the text or data. Descriptive markup has important advantages for the forward-looking content provider. As a document goes through its various transformations from inked paper to Web to CD-ROM to synthetic speech to whatever's next, a heading might be displayed in many different ways, but it always remains a heading.

Some word processors and other programs offer a rudimentary form of descriptive markup by means of style sheets. You define a style called "Heading," and assign it a set of visual formats; then if you change your mind or adapt the document to some other purpose, revising the definition will alter all text that has the Heading style.

No standards for higher-level visual or abstract markup have the universality of ASCII. Every program goes its own way. And, unfortunately, abstract markup seldom survives translation between file formats. When you convert a document, the heading style is replaced by the corresponding visual attributes, such as 12-point bold type. This transformation is irreversible and entails a loss of information: You cannot subsequently convert all instances of 12-point bold type back into headings, because nonheading text may have the same attributes.

Self-Documenting Documents

A useful exercise in thinking about data preservation and conversion is to imagine yourself a paleographer in the distant future, long after the collapse of civilization (brought on, no doubt, not by a wayward asteroid but by the year 2000 bug). Your job is to recover the wisdom of the ancients from the disks and tapes they left behind. This situation may seem contrived, but it really isn't that different from rediscovering a forgotten carton of eight-inch floppy disks full of Wordstar and Visicalc files.

What characteristics of a file format would help you recover the contents when the program that created the file is defunct? One obvious help is documentation. It's always easier to find your way if you have a map, and if the streets have signs. Archivists call it metadata: all the information *about* the information, starting with the handwritten label stuck on a floppy disk. The ideal is a self-documenting file—one that explains its own structure. If you want to be a fundamentalist about self-documentation, it be-

comes a game like communicating with extraterrestrials. Every disk has to include instructions for building a machine to read it, and instructions for reading the instructions, and so on. But in practice it's possible to supply a lot of metadata without getting caught in a bottomless regress. For example, an image file might consist of 307,200 eight-bit bytes; interpreting this block of data is easier with the clue that the bytes represent the colors of pixels arranged in a rectangular array of 480 rows and 640 columns.

If the file can't fully document itself, then at least it can be documented elsewhere. The Postscript page-description language would not be easy to fathom without help, but it is thoroughly described in a series of fat books. If those manuals survive the millennium, future generations should be well equipped to read Postscript. The TEX typesetting system and its nephew LATEX are also meticulously documented. But with a few notable and laudable exceptions, the file formats of commercial software are closed and proprietary. If you want to figure them out, you're on your own.

Finally, the job of recovery and reconstruction is a great deal easier for files that employ abstract markup. The nature of abstract markup is to tell you *what* is in the file, rather than *how* to present it. That's the ultimate in metadata, and just what you need to maximize your chances of correctly understanding the information.

Most people don't choose their computer software by evaluating the qualities of file formats. They are swayed instead by lists of features, and by the sensuous experience of clicking on tool palettes or dragging-and-dropping. This situation is unlikely to change, and so the file formats of popular commercial programs are the ones that future antiquarians will have to deal with. In this respect an intriguing development is Microsoft's recent decision to make HTML a "companion" file format for all the programs of the Microsoft Office suite, including Word and the Excel spreadsheet. The ability to save files in HTML format is nothing unusual; what's important about the Microsoft initiative is that HTML files can also be *read* by the applications. A Microsoft press release promises "seamless round-tripping" from HTML to other formats. In principle, then, HTML could become the primary medium for much digital information. Regrettably, the HTML generated by the Office programs is heavily laden with visual markup.

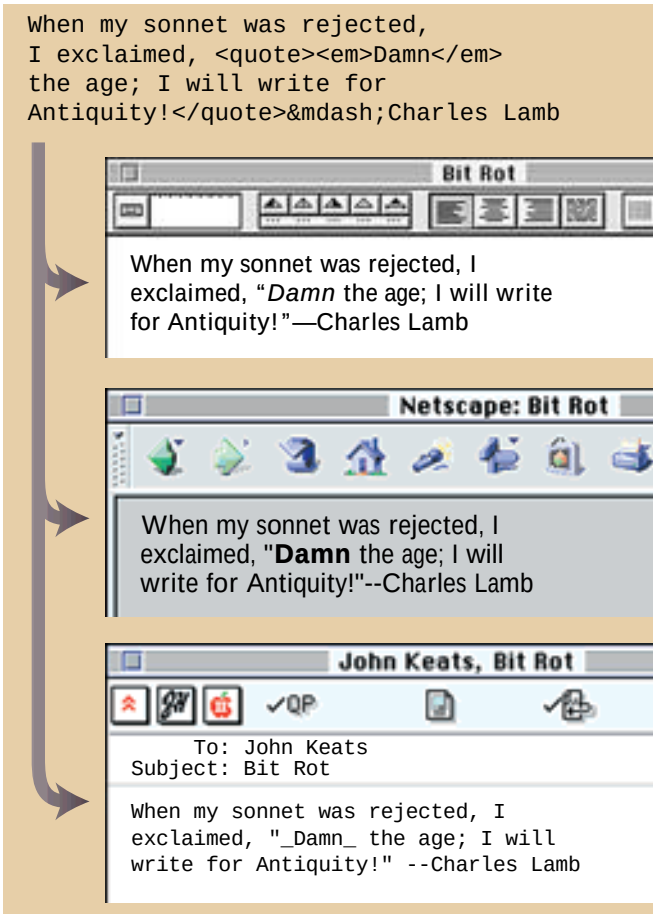

Figure 2. Abstract markup describes structure or function rather than appearance, so that a single file can be readily adapted to several purposes. In print, Web and e-mail versions note the differing treatment of quotation marks, emphasized text and the dash.

Standardized and Generalized

Do any existing file formats offer versatility and the plausible hope of longevity?

Postscript and TEX have already been mentioned as highly readable formats with open standards. On the other hand they are not well-suited to abstract markup. Postscript is essentially a write-only language: Almost anything can be turned into Postscript, but going the other way is difficult. TEX, the creation of Donald Knuth of Stanford University, is the preferred formatting language in the physical sciences and mathematics. With a vast literature already committed to the format, TEX has excellent prospects for long-term survival. But it too is fundamentally a visual formatting language; abstract markup is possible, but it takes discipline.

The leading candidate for a file format for the ages is SGML, the Standard Generalized Markup Language, developed in the early 1980s by Charles F. Goldfarb of IBM and now an international standard. SGML is actually a language for defining markup languages. In the SGML formalism, all elements of a document are labeled by descriptive tags embedded in the text. A title, for example, might be surrounded by the tags <title> and </title>. Visual presentation is deferred until these tags are processed in a later stage of formatting. New tags, and whole new classes of tags, can be defined as needed.

SGML has fervent adherents. It has been received with particular enthusiasm by organizations that produce quantities of highly structured documents, such as technical manuals. It has also been adopted by several journal publishers, including the American Institute of Physics and the IEEE Computer Society. And yet SGML has not won the hearts of content providers everywhere.

Maybe it's just that the name is so forbiddingly standardized and general, but many consider SGML unwieldy and cumbersome. The levels of indirection annoy people. First you define a tag for a title, then you enter the tag and the title into the document, then you define how the tagged title is to be formatted in the finished output. That's a lot of rigmarole when a word processor allows you to just click on the title and apply the formatting directly.

A decade ago James H. Coombs, Allen H. Renear and Steven J. DeRose of Brown University wrote an eloquent plea for SGML and other abstract markup languages. They argued that we all mark up our texts anyway—even conventional punctuation is a markup language—so we might as well choose the style of markup that captures the most important and long-lived information. They also argued that inserting abstract tags requires less cognitive effort than doing typographic formatting; it's easier to remember "This is a <title>" than "Titles are set in 28-point Bodoni Bold." I find the arguments persuasive, and yet I note that the users of Microsoft Word still outnumber the users of SGML .

For a time it seemed that SGML might finally catch on through the reflected glitter of HTML, which began as a kind of SGML-lite—smaller, simpler and lacking the facility to define new tags. The hope was that people would learn the advantages of abstract markup, yearn for something more powerful, and move up to the mother tongue. What's happened instead is that people have gone to extreme lengths to turn HTML into a visual formatting language. They embed text in tables so that they can control margins and columns; they sprinkle pages with hundreds of invisible images to control the placement of text; in preference to abstract tags such as <h1> they use a <font size> tag.

Abstract markup is going to get one more chance. A new language called XML, for Extensible Markup Language, is simpler than SGML but still retains the ability to define new tags (which is what makes it extensible). In principle, anyone can devise a private set of XML tags, but the main interest is in dialects defined for entire communities or disciplines. For example, there are groups creating XML variants for mathematics, chemistry, biological sequence data, astronomy and meteorology. Farther afield, other XML dialects describe real-estate listings, financial data, classified ads, legal documents and genealogies.

The mathematical dialect, MathML, illustrates the potential of XML. At one level, MathML addresses the messy problem of how to display mathematical expressions on the Web. In HTML this is often done by embedding scads of miniature images in the page—a solution that works, more or less, but offends the finer sensibilities. MathML puts mathematical notation directly into the markup language. But there's more to it than that. MathML can capture meaning as well as appearance. An expression such as x^2 can be written in such a way that it will be recognized not as "*x* superscript 2" but as "the square of *x*." The dream is copying an equation from a published paper directly into a program such as Mathematica or Maple, where is can be solved or graphed or manipulated algebraically. Whether the dream comes true depends on whether authors accept the discipline of abstract tagging or turn XML into yet another visual formatting language.

Writing Source Code

The problems I'm whining about here—the problems of keeping a grip on digital text and other kinds of data as computer technology evolves—must seem quaint to professional programmers and software engineers. Moving a data file from one machine to another is easy compared with "porting" software. You don't just take a Windows .EXE file and try running it on a Macintosh. And programs are notoriously brittle. A data file may lose something in translation—such as footnotes—and yet still be usable. When a program fails, it fails totally.

Living on such thin ice, programmers learn to tread carefully. It's part of their education and culture. Writing code for keeps is a major theme of software engineering. Tricks and shortcuts that solve the problem of the moment are not much admired if they fail on another platform on in the next version of the operating system. The emphasis is on portability—on program constructs that work in any computer environment.

Programmers observe a distinction between "source code," which is what the programmer writes and revises, and "object code," which is the final product. The Central Dogma of software development holds that information flows only from source code to object code, never the other way around. Programmers also put a high value on abstraction—on expressing concepts in the most generic way possible. And they are wary of the dangers of duplication and repetition: They know that if information is written down twice, the two copies will eventually become inconsistent.

The same principles and attitudes may apply just as well to other kinds of computer work. Indeed, it's not much of a stretch to see writing with a word processor or drawing with illustration software as a kind of programming. The manuscript I am typing at this moment is not a magazine article but the source code of a program that can be compiled to produce a magazine article. If I compile and run the program through a laser printer, the output is a paper printout. Later the same source code will be compiled again and run through a larger machine that produces photographic film for printing plates. Compiling with still another set of options yields Postscript files for distribution via the Web. It all goes smoothly (most of the time) because the same source code is the input to all the transformations. If I make a change to the source, it automatically shows up in all three object-code versions.

Several years ago a book title proclaimed: *The Mac Is Not a Typewriter*. Neither is the PC. It's not a typewriter; it's also not a sketchpad; it's not a ledger book. It's a computer. When you sit at the keyboard, you may think you're writing or drawing or balancing the budget, but what you're doing is creating computer programs, which have to be compiled and run before they yield their output of text or art or spreadsheet. You may think you're just a content provider, but you're really a programmer.

Bibliography

Coombs, James H., Allen H. Renear and Steven J. DeRose. 1987. Markup systems and the future of scholarly text processing. *Communications of the ACM* 30:933–947.

Cover, Robin. The SGML/XML Web Page. http://www.sil.org/sgml/sgml.html

Mohlhenrich, Janice (editor). 1993. *Preservation of Electronic Formats & Electronic Formats for Preservation*. Fort Atkinson, Wis.: Highsmith Press.

National Research Council, Steering Committee for the Study on the Long-term Retention of Selected Scientific and Technical Records of the Federal Government. 1995. *Preserving Scientific Data on Our Physical Universe: A New Strategy for Archiving the Nation's Scientific Information Resources.* Washington, D.C.: National Academy Press.

Rothenberg, Jeff. 1995. "Ensuring the Longevity of Digital Documents." *Scientific American* 272(1):42–47.

Task Force on Archiving of Digital Information (Donald Water and John Garrett, Co-Chairs). 1996. *Preserving Digital Information: Report of the Task Force on Archiving of Digital Information*. Washington, DC: The Commission on Preservation and Access and the Research Libraries Group, Inc. Also available at http://www.rlg.org.