REINVENTING THE COMPUTER

Brian Hayes

A reprint from

American Scientist

the magazine of Sigma Xi, the Scientific Research Society

Volume 85, Number 1
January–February, 1997
pages 16–20

This reprint is provided for personal and noncommercial use. For any other use, please send a request to Permissions, *American Scientist*, P.O. Box 13975, Research Triangle Park, NC, 27709, U.S.A., or by electronic mail to perms@amsci.org. Entire contents © 1996 Brian Hayes.

Reinventing the Computer

Brian Hayes

The computing profession has been celebrating birthdays—the 50th of the ENIAC, the 25th of the microprocessor, the 15th of the IBM PC (not to mention the imminent birth of HAL on January 12, 1997). The anniversaries offer an occasion for looking back and marveling at how swiftly computers have changed. Room-filling racks of vacuum tubes have collapsed into a sliver of silicon. A few dollars at Toys R Us buys more computing power than all the world possessed in 1950. Punch cards, teletype machines and video terminals are all long gone; now we point and click.

But these dazzling transformations in technology, economics and the human interface may blind us to aspects of computing that have changed little over the years—aspects that may in fact be overdue for rethinking. There is much about the modern computer that would have been familiar to the designers of the ENIAC and the EDVAC in the 1940s. Indeed, computers seem to grow steadily more homogeneous as the years go by, crowding into one small corner of a multidimensional "design space." Perhaps there is a good reason for this trend toward uniformity—perhaps this one tight cluster of designs is where all the best ideas are found—but it seems worthwhile making at least a brief survey of some of the unpopulated wilderness areas elsewhere in the computer design space.

Religious Wars

Of course computers are not really all alike. You get to choose: Macintosh or Windows or Unix, or perhaps something slightly exotic, like NeXT, OS/2, Amiga, VMS, BeOS. I cannot claim neutrality in the ongoing hostilities among these factions; I have my own partisan loyalties. And yet I cannot help wondering, when I take a few steps back from the fray, if the present battle for domination of the desktop will not seem to later generations about as compelling as the War of Jenkins's Ear. Looked at from a certain height, most computers are very much alike.

And it's not just that they *look* alike. (Some of them come in gray boxes now instead of beige!) Under the skin, computers are remarkably uniform in their basic architecture—the way they are assembled from logical building blocks such as registers, adders and instruction decoders. Nearly all computers in widespread use today are recognizable descendants of the von Neumann architecture, named for John von Neumann, who first described it in 1945. They have a single processor, a memory that holds both data and instructions for the processor, and facilities for input and output.

The longevity of the von Neumann architecture is surprising and interesting, but I want to consider the computer at a somewhat higher level of abstraction. The ordinary computer user, sitting down to write a memo or recalculate a spreadsheet, does not think of the computer as an assemblage of registers and logic gates. Nor is it perceived as a box full of silicon chips and solder joints. For the typical user, the fundamental elements of the computer are less-tangible entities, such as files, programs, windows, menus, directories, documents. These are the objects we work with when we work with computers. Here I want to focus particularly on files and programs.

A file is the basic unit of information storage in almost all modern computers. For most of its life, a file sits inertly on a mass-storage medium such as a magnetic disk. In this condition, the contents of the file are inaccessible. To get at the information, you have to *open* the file and copy it into random-access memory (RAM). Once the file is open, you can read or display it. You can also alter it, but your changes will be preserved only if you then *save* the file, writing it back onto the magnetic disk. Another important fact about files is that they have names; you can't create a file without naming it, and generally speaking you have to know the name—or at least recognize it when you see it—in order to find the file again.

A computer program, from one point of view, is just a special kind of file. It also resides inertly on disk most of the time, becoming active only when loaded into RAM. And, as with other files, you need to know a program's name in order to invoke it. But a program is *executed*, or *run*, rather than opened; it is a file whose content is "machine code"—instructions for the processor that are generally indecipherable by the human reader.

Programs and other files are usually organized in a treelike hierarchy of directories, or folders. Each

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

folder can contain not only files but also other folders, which can contain still more deeply nested folders, and so on. This "file system" is what is *in* the computer, from the user's point of view.

The concepts of file and program are surely familiar by now to schoolchildren everywhere and to most of their parents. Indeed the ideas are so much taken for granted that it's hard to imagine what a computer would be like without them. But it's worth trying.

Lost in the Files

Why do files exist, anyway? Look at it from the program's point of view. While a program is running, it can create all kinds of fancy data structures—arrays, lists, sets, trees, queues, etc.—but they all disappear the instant the program exits. The internal data structures are like the thoughts or memories of a living person; they do not survive after death. Actually, the program's predicament is even worse, because a program dies and comes back to life repeatedly; each time it runs, it starts with a blank slate. It is as if each night when you went to sleep, you forgot everything you ever knew. To cope with such recurrent amnesia, you might try leaving yourself a note where you could find it in the morning. That's what a program is doing when it creates a file. A file is a program's way of communicating with its future self.

Historically, the reading and writing of files have been viewed as input and output operations. In the early years, files were stored on reels of magnetic tape, which had to be mounted by hand when needed. Thus the files were clearly external to the computer, just as a written memorandum is external to your own memory. But the boundary between inside and outside is not so sharp today. A disk file may well live inside the same plastic box as the processor, so that writing a file no longer really seems like "output." On the other hand, you may have instant access to files kept on a server down the hall or on a Web site across the continent, so that the whole question of where information is stored no longer seems entirely germane. And if you needn't care whether a file is stored on a local disk or a remote one, why should you have to keep track of whether it is on disk or in RAM?

One answer to this question is that RAM is "volatile": It goes *pffft* when you pull the plug, whereas magnetic disk memory endures until it is erased. But this distinction also gets muddled. In a computer with virtual memory, information that appears to be in RAM may actually reside on disk; conversely, with a "RAM disk," files that seem to be safely on disk are actually held perilously in RAM. No wonder that newcomers to computing often have trouble understanding the difference between permanent and ephemeral memory. They labor to create their first computer document, and then forget to save the file before switching the machine off at the end of the day. This is not to be taken as evidence that users are losers; on the contrary, it is a hint that something is awry with the underlying concepts. Perhaps the flaw is in the desktop metaphor that dominates most modern computer interfaces. After all, if you leave a draft of a letter on a *real* desktop overnight, it doesn't vanish when you turn the lights out.

Files bring further problems. Naming them—even if your computer allows you more than 8 + 3 characters—can be an irksome chore. (In the paper-and-pencil world, you don't have to dream up a name for every letter or memo you write.) You also have to decide where to put a file in the directory structure. Should you organize your correspondence chronologically or alphabetically? A year later, when you've forgotten both what you named the file and where you put it, you discover the final drawback of file systems: the agony of retrieval. Even if you know where you're going in the tree of folders, getting there may require a long, tedious sequence of mouse clicks.

Files, folders and the desktop metaphor were all liberating innovations when they first appeared. The trouble is, they don't scale well. If you have only a few hundred files, organization is easy: You can keep everything on a handful of floppy disks. With a few thousand files, hierarchical folders work well. But after a decade or two of living in close symbiosis with a computer, you accumulate tens of thousands of files, and they become a management challenge. Personal archives of gigabyte size are no longer unusual, and the really big data sets—including those collected by astronomers, geophysicists and others in the sciences—are soaring beyond terabyte territory into the petabyte range. You would not want to explore 10^{15} bytes of files by clicking through nested folders. The nature of the problem is already clearly illustrated by the huge hierarchical file system called the World Wide Web. I am not the only one who sometimes consults a search engine such as Lycos or AltaVista to find a document whose location I already know; the search is simply quicker and easier than clicking my way through a series of linked pages.

Persistence

One response to these problems is to graft a new interface onto the file system. Utility programs that index and search for files have been proliferating, as have other shortcuts that keep recently used or frequently used files ready at hand. The most sophisticated of these techniques can create whole new views of the file system on the fly. For example, even if you routinely organize your correspondence chronologically, you could summon up a "virtual directory" of all your letters sorted by recipient. (Virtual directories are an invention of David K. Gifford and his colleagues at MIT; several other groups have explored related themes.)

The more radical approach is to abolish the file system altogether, and with it the distinction between permanent storage and ephemeral memory.

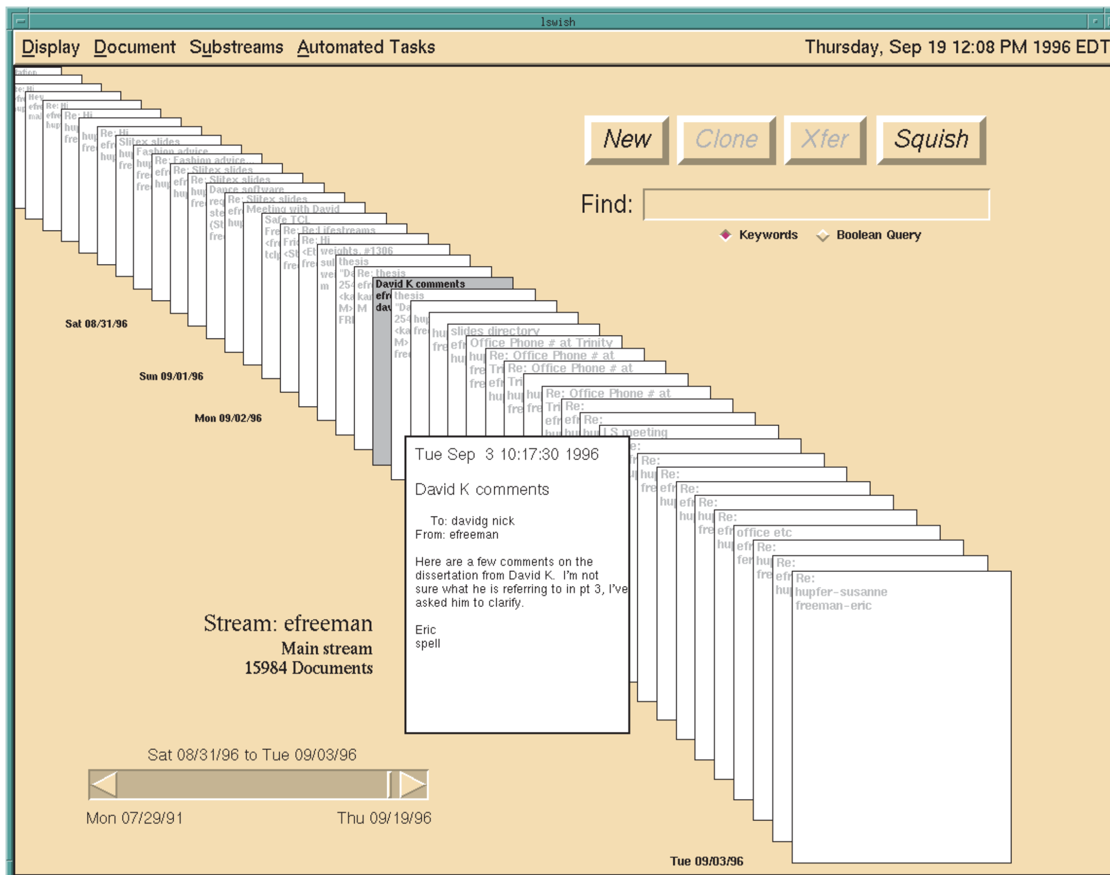

Figure 1. "Lifestreams" organize personal documents chronologically. (Image courtesy of Eric Freeman.)

Why should files alone have eternal life? Why can't you jot down an address or a phone number and have the note survive, without having to name it and put it someplace in particular? Why should every reminder, doodle or Web clipping require its own named slot in the storage bin?

Looked at from the programmer's point of view rather than the user's, the issue here is the persistence of data structures. The alternative to a file system is to give every program the ability to create objects—a bit of text, say, or a record representing your rent check—that remain intact after the program has finished running, or even after the computer has been turned off. When the program starts up again, the objects are present in memory just as if there had been no interruption.

Several programming languages have long offered a form of persistent data. Smalltalk and APL, for example, allow you to save a "world" or a "workspace" and then later reload it, restoring all variables and other data to their previous state. But this is an all-or-nothing approach.

More flexible techniques for working with persistent data have been a research topic in computer science for more than a decade. Research has been particularly active in Scotland, where persistent-programming systems have been developed by groups at the universities of Edinburgh, Glasgow and St. Andrews. Their work has drawn on ideas not only from the theory of programming languages but also from database theory. This connection is not surprising, since a database is, in one view, a large collection of persistent, unnamed objects.

The key idea in a persistent-programming system is to make longevity an "orthogonal" property of data objects; in other words, the same rules for determining lifetime apply to all types of data, from the simplest numeric variables to the most complex record structures. Every object is potentially immortal; its actual lifetime is determined by an algorithm that throws things away when it can prove they will not be needed again.

To make persistent programming work, long-lived data have to be moved at some point to a nonvolatile storage medium, and obsolete data must be purged from permanent storage. But neither the programmer nor the user of the system need be aware of these movements. It is rather like the constant shuttling between disk and RAM in a virtual-memory system, which also takes place behind the scenes.

Filelessness

At least one mass-produced computer incorporates a persistent-object system. It is the Newton MessagePad, a hand-held "personal digital assistant" made by Apple Computer. Data objects in the Newton live in a "soup" (that's the technical term), where they do not have to be named or

deposited at a specific place in a hierarchy of documents. For example, when you scribble notes on a Newton—and "scribble" is the right word, since you write with a stylus instead of a keyboard—every page goes directly into the soup. Later you retrieve a note by searching for any word or phrase it includes. Documents can be filed away if you wish, but there is no compulsion to do so. Nothing is forgotten unless you explicitly delete it; everything in the soup is equally accessible at all times.

Another vision of fileless computing comes from the work of Eric Freeman and David Gelernter of Yale University, who have invented a software system they call Lifestreams. In the Lifestreams model, documents are arranged in the simplest way possible: a one-dimensional, chronological sequence. Everything you might ever look at on the computer screen—incoming and outgoing e-mail messages, manuscripts, pictures, video clips, perhaps programs too—is stacked up from oldest to newest in one giant heap. At the source of the stream are your earliest records, going back perhaps to your "digital birth certificate." Current documents are at the front. The stream can even extend into the future to hold documents that you *will* need someday, such as reminders.

At first, the Lifestreams model looks like an exceptionally primitive file system, without even the amenity of hierarchical directories. But the documents in the Lifestream are not files. You don't have to name them (although you can if you wish). You never have to save them; they are retained as persistent data. You needn't find a place for them in a tree of directories; they are sorted automatically according to creation date. But what about the problem of retrieval? If you wish, you can browse forward and backward through the chronological stack of documents, but the main means of access is through indexing and searching. You find documents by issuing a query, rather like a command in Go Fish!: "Give me all your letters to Bill written after November 5." In response to the query, the system creates a substream consisting of just those documents that satisfy the stated criteria. In this respect the Lifestreams system is much like a database, but there is a further refinement. The substream itself is a persistent object, which remains active indefinitely. If you later write another letter to Bill, it will automatically appear at the head of the substream as well as on the main Lifestream.

Freeman has built Lifestreams prototypes for Unix workstations and also, significantly, for the Apple Newton.

The idea of a computer without files is not as big a departure as it might seem. In the first place, files are an abstraction, or even an illusion, in modern computers. We may imagine that a file occupies a definite place on a disk, adjacent to other files in the same folder. In reality, the disk sectors that comprise any given file can be scattered randomly over the surface of the disk. For the lower-level software that actually communicates with the disk drive, files do not exist.

Furthermore, fileless computers are all around us; we just don't notice them much. They are the "embedded" computers that run appliances, automobiles and even some of the peripheral devices (such as modems and keyboards) attached to other computers. Few embedded computers have any need for a file system.

Deprogramming

Perhaps we can hope to abolish files, but programs are not to be swept away so categorically. A computer without programs is like a car without fuel: at best an object of quiet contemplation. Nevertheless, there may well be opportunities to change the way programs are built and distributed, and the way they cooperate with one another. Under present practice, a program is an indivisible, monolithic, opaque hunk of machine code, which must be swallowed whole or not at all. It is a black box, which accepts inputs and produces outputs, but which cannot be opened up to see (or change) how it works.

A new software technology offers hope of greater flexibility: It may not pry the lid open, but it could divide the one big box into several smaller boxes we can rearrange as needed. The governing metaphor is the component audio system. Just as you hook up a tuner from one manufacturer with an amplifier from another and speakers from a third, you should be able to link together software components from different sources, and have them all work on the same document.

This basic idea has been known under a variety of names, but recent attention has focused mainly on two schemes called OpenDoc and Object Linking and Embedding (OLE), introduced by Apple and Microsoft respectively. What they offer amounts to a change of perspective. Today the program is the central fixture around which the computing universe revolves; you launch a program and then use it to create or edit various satellite documents. OpenDoc and OLE are meant to put the document rather than the program in the middle; you create a document and operate on it with whatever programs are needed, calling on different software components for working with words, pictures, calculations, etc.

The idea of combining many small programs to accomplish a task is hardly new. The Unix community has long favored "little" programs called filters, which can be linked in sequence. But Unix filters are largely confined to working on text files, with a very simple interface between one program and the next. The new schemes are meant to allow tighter integration of programs and a more interactive style of computing.

Even if OpenDoc and OLE live up to their promise, something vital remains missing. What distinguishes the computer from other machines is its programmability, but few users ever write a program. No doubt the main reason is that few

are inclined to learn the arcana of the programmer's art, but it's also fair to say that most current computer systems do not encourage tinkering. A word processor or a spreadsheet comes with the software equivalent of a sticker that reads: "No user-serviceable parts inside." If you don't like some detail about how the program works, your only option is to write a new one from scratch.

OpenDoc and OLE will not change this situation substantially for Macintosh and Windows users. Writing an OpenDoc or OLE component will not be much easier than writing a standalone program. The Unix world has traditionally been friendlier to the inquisitive or meddlesome programmer. At one time every Unix system came with complete source code, as well as the compilers and other tools needed to rebuild or modify the system. But today most commercial Unix systems do not include source code, and programming tools are an extra-cost option. (The remarkable Linux operating system, which began as the project of a single young programmer, Linus Torvalds, probably owes part of its popularity to its distribution with full source code.)

The computers that most clearly invited tinkering were the "Lisp machines" of the 1970s and 80s. Virtually all of the software for these computers was written in Lisp and was open to inspection and modification. The rule was: If you don't like the way it works, shut up and fix it. For a while two companies (Symbolics and LMI) were making rival Lisp machines; both companies failed, although Symbolics has lately come back from the dead. There have been a few other attempts to build a language-centered computer. The Xerox Palo Alto Research Center created several machines programmed in Smalltalk; the Lilith computer, created by Niklaus Wirth, was based on Modula-2.

The commercial failure of Lisp machines is generally attributed to an economic squeeze: The generic hardware of mainstream computers soon offered better performance for less money, even for programs written in Lisp. And yet the dream of a single language for all computers will not go away. The latest embodiment of this ideal is the Java computer. Java was conceived as a language for programming "set-top boxes," the devices that are meant to bring interactive network computing to your television set; the language has since experienced a tremendous surge of interest as a medium for distributing "applets," or small application programs, over the Internet. Several computer operating systems already have a Java interpreter built in, and specialized hardware for running Java programs is under development.

The Secret Lives of Computers

Most likely, none of these radical new ideas—nor the radical old ones!—will catch on. It's a good bet that computers will grow more and more alike, distinguished by minutiae that only a marketing manager could get excited about. This seems to be the way that technologies evolve and mature.

Look back at the early history of the automobile. At the outset there were cars propelled by steam, by electricity, by diesel and gasoline engines; there were two-wheelers, three-wheelers and four-wheelers; some were steered with a tiller and some with a wheel; some had handbrakes and some footbrakes; the engine might be in front or in back or in the middle. That diversity has disappeared. Apart from rare industry upheavals—such as the switch from rear-wheel to front-wheel drive some years back—cars don't change much. Today there are many makes of automobiles, but few choices in technology.

Radio has gone through a similar loss of species diversity. Radio enthusiasts of the 1920s could choose among the regenerative, the super-regenerative and the superheterodyne circuits. But one of those designs (the superheterodyne) drove the others to extinction. What is more to the point, no one today takes the least interest in how a radio works. (If you go to a store to buy one, don't bother asking the clerk if it has a superheterodyne circuit.)

Will the computer meet the same fate? Will it become an appliance or a commodity? Will it, in other words, become *boring*? That's the direction the industry is heading in, but I can see two glimmers of hope. The first glimmer is institutional: Computer science, as a discipline, is gleefully undisciplined; there is still plenty of willingness to explore the occasional screwball idea. The second glimmer is theoretical: The computer has a protean quality not shared by any other machine, certainly not by automobiles and radios. Just about any computer can put on a convincing impersonation of any other computer. This talent for emulation commonly allows a fancy Macintosh or Unix workstation to masquerade as a plain-vanilla Windows machine, but it can work the other way around as well. Perhaps we will all end up with the most prosaic of computers, but some of them may lead secret lives of adventure and romance.

Bibliography

Adler, Richard M. 1995. Emerging standards for component software. *IEEE Computer*, March 1995, pp. 68–77.

Atkinson, M. P., P. J. Bailey, K. J. Chisholm, P. W. Cockshut and R. Morrison. 1983. An approach to persistent programming. *The Computer Journal* 26:360–365.

Freeman, Eric. 1995. Developers corner: Lifestreams for the Newton. *Mobilis: The Mobile Computing Lifestyle Magazine*. <http://www.volksware.com/mobilis/october.95/ develop1.htm>

Freeman, Eric and David Gelernter. 1995. Lifestreams: A storage model for personal data. *ACM SigMOD Bulletin* 25(1):80–86. <http://www.cs.yale.edu/homes/freeman/papers/SIGMOD/paper.ps>

Gifford, David K., Pierre Jouvelot, Mark A. Sheldon and James W. O'Toole, Jr. 1991. Semantic file systems. In *13th ACM Symposium on Operating System Principles*, October 1991. <http://www.psrg.lcs.mit.edu/ftpdir/pub/papers/sfs.ps>