Brian Hayes

The Discovery of Debugging

On May 6, 1949, a length of punched paper tape was threaded into a machine at the University of Cambridge; a few seconds later a nearby teleprinter began tapping out a list of numbers: 1, 4, 9, 16, 25, ... The tape reader and the teleprinter were the input and output devices of an electronic digital computer, a machine called the EDSAC; calculating the squares of a list of numbers was its first full test. Indeed, it was the first time any full-scale computer, in the modern sense of the term, had successfully run a program.

Computers and computing have changed drastically since 1949—more so, perhaps, than any other element of technology. The EDSAC is long gone; most of its parts were sold for scrap in 1958. No one will ever build another machine like it. Nevertheless, it is still possible to write programs for the EDSAC, to load those programs into the paper-tape reader and then to see (and hear) the results come ticking out of the teleprinter. The time machine that offers this transport into the past is a simulator created by Martin Campbell-Kelly, a historian of computing who teaches at the University of Warwick. The simulator turns a modern Macintosh computer into a surprisingly faithful—sometimes frustratingly faithful—replica of the EDSAC.

It is a commonplace observation that all the computing power of forty years ago—the rooms filled with glowing vacuum tubes, the tons of air conditioning, the squadrons of attending technicians—will now fit comfortably on a corner of your desk or in your briefcase or even in your pocket. But it is one thing to compare performance specifications, memory capacities and program benchmarks; it is quite another actually to see one of the early iron giants of computing encompassed in the little plastic box of a Macintosh. There is something incongruous about it, like the dozen circus clowns in the miniature car.

The EDSAC was part of the first generation of computing machines to emerge in the aftermath of the wartime ENIAC (Electronic Numerical Integrator and Computer) project. The ENIAC itself, built at the University of Pennsylvania, had been churning out ballistics tables for the U.S. Army since 1945, but it was not a computer in the modern sense. It could not be programmed except by setting switches and plugging in patch cords. The idea of controlling a machine by means of a program stored in its own memory made its public debut in John von Neumann's "First Draft of a Report on the EDVAC," written in 1945, just after the end of the Second World War.

At Cambridge a year later, Maurice V. Wilkes, the new director of the Mathematics Laboratory, saw a copy of von Neumann's report. Soon thereafter he was on his way to Philadelphia for a summer school at the University of Pennsylvania, where the stored-program concept was discussed in detail. The Pennsylvania group was then gearing up to begin building the EDVAC (Electronic Discrete Variable Automatic Computer), and Wilkes returned to England with the determination to create a stored-program machine of his own. He called it the Electronic Delay Storage Automatic Calculator. The similarity of the acronyms EDSAC and EDVAC was no coincidence.

Back in the United States the EDVAC

project suffered several changes of leadership and also developed a case of creeping-enhancement disease: every stage of construction was delayed by some bright idea about how to make the machine bigger or better or faster. Indeed, the version of the EDVAC finally constructed at Pennsylvania bore little resemblance to the machine described in von Neumann's "First Draft." Wilkes stuck close to his original design and finished a year earlier.

The EDSAC occupied the upper floor of a building that had once been the anatomy school in Cambridge. The control circuits, as well as the circuits responsible for logical and arithmetical operations, were based on vacuum tubes—some 3,500 of them—arranged in tall steel racks that filled the room like library shelves. The memory elements of the computer were mercury delay lines: tubes five feet long, filled with mercury, with a quartz transducer at each end. Ultrasonic pulses representing binary digits were transmitted into the mercury at one end of the tube and received at the other end; the received signal was then amplified and transmitted again, so that the pulses circulated continuously and could be stored indefinitely. Each such "long tank" in the main memory held 576 bits, organized as thirty-two "words" of eighteen bits each; the design called for thirty-two tanks, and so the total capacity was 1,024 words.

The EDSAC, like most of the other electronic computers conceived in the 1940s, was a "bit-serial" machine. The data paths threading throughout the processor were just one bit wide, and so communicating an eighteen-bit machine word from one place to another required eighteen steps. Bit-serial architectures are rare today; most of the common microprocessors have data paths that are sixteen or thirty-two bits wide. The serial arrangement was slower, but it had the important advantages of simplicity and economy.

The EDSAC's clock speed, which set the pace of all operations, was 0.5 megahertz, or 500,000 cycles a second. That was a conservative specification even in 1949 (modern computers run at 25 or even 50 megahertz), but it was consistent with Wilkes's emphasis on getting the machine built quickly and reliably rather than on seeking the ultimate in performance. The EDSAC could typically execute about 600 instructions a second (for present-day computers the corresponding number is a few million). A crude figure of merit for computer performance is the product of speed and memory capacity. By that measure a modern desktop machine might score 100 or 200 megahertz-megabytes. The EDSAC checks in at 0.0006 megahertz-megabytes. In one respect, however, the EDSAC exceeded the fondest dreams of the modern "power user." It drew about twelve kilowatts from the municipal electric supply. (A Macintosh today draws about a tenth of a kilowatt.)

Comparing the EDSAC with the very latest computer technology may not be the best way of conveying a sense of its capabilities. The EDSAC reminds me strongly not of the computers I see on every desktop today but of the first machine on which I had any experience of programming, not quite twenty years ago. That machine was closely matched to the EDSAC in memory capacity and execution speed, as well as in its Spartan facilities for input and output. It was a programmable hand-held calculator made by Hewlett-Packard.

A few weeks after the EDSAC ran its first program, a conference on "high speed automatic calculating machines" was convened at Cambridge. Here was a chance for Wilkes and the rest of the EDSAC crew to show off their accomplishments and to compare notes with representatives of other laboratories in Great Britain and in Europe and the U.S. The record of the conference has been reprinted as Volume 14 in the Charles Babbage Institute Reprint Series for the History of Computing; reading it offers a glimpse into the small world of computer engineering at midcentury.

What struck me first when I read the conference record was the remarkably advanced state of thinking on computer architecture and hardware design at a moment when there was little practical experience to guide that thinking. Much of the "design space" for digital computers had already been explored or at least roughly mapped out. The basic circuits for performing various arithmetical and logical operations had been devised. The relative merits of binary notation and of other options such as binary-coded-decimal were well understood. There was extensive debate over what set of elementary machine instructions (or orders, as they were then known in England) would yield the best performance. One speaker introduced the idea of a storage hierarchy, in which a fast but small memory is supplemented by a slower but larger auxiliary storage.

In contrast to all the sophisticated analyses of hardware, the idea of software could hardly be said to exist in 1949. To be sure, there was much talk about the "coding" of problems for specific machines: how to multiply two numbers efficiently with a certain set of instructions; which algorithm would prove best for numerical integration; and so on. But those exercises in problem solving were far from the modern conception of software as the "personality" of a computer—as the component that allows a single machine to be a word processor one moment, an artist's sketch pad the next and a simulator of an antique computer the next.

The disparity between the advanced state of hardware design and the primitive state of software development can be expressed as follows. Most of the abstractions that serve as the fundamental building blocks of computer hardware had already been recognized in 1949: Boolean logic gates, registers, clock circuits, counters, adders, shifters, and perhaps most important the division of the machine into subunits for memory, control, arithmetic and logic, input and output. There is a corresponding set of abstractions for software: sequences, branches, loops, iteration, recursion, procedures, arrays, sets, lists, queues, stacks and so forth. Few of those concepts had yet appeared, except in the most embryonic form. Even more conspicuously absent was the idea of a programming language. No higher-level programming languages existed (FORTRAN did not come along until 1954), and the programming notations then in use, namely the instruction sets of the various computers, were not viewed as having a linguistic aspect. No one had set forth the grammar of an instruction set, and no one was ready to attribute meaning to a program as an independent entity, separate from the computer that would execute it.

I can suggest two reasons software was so much the stunted sibling of hardware in the earliest years of computing. First, most of the computing pioneers still viewed the computer as a strictly numerical calculator. Its function was to solve mathematical problems. The idea that it might operate not just on numbers but also on symbols more generally—including its own programs—had not yet caught on. Programs were seen not as interesting objects in themselves but merely as tools for manipulating numbers. Second, no one yet appreciated that programming a digital computer was going to be an intellectual challenge, one that might reward the interest of a mathematician or an engineer. It appeared then that programming might be tedious but not fundamentally difficult.

A crucial discovery was yet to be made: the discovery of debugging.

I find it quite remarkable that the EDSAC's first program ran without errors. Moreover, the next two programs—which printed slightly more elaborate lists of squares as well as prime numbers—also worked the first time. It is true they were extremely simple programs, and they had been carefully checked and rechecked in the weeks before the machine was ready for them. But still I am full of admiration. Three bug-free programs in a row: it is a record that may never have been equaled. In any event, the streak stopped at three; with the next program, which was more ambitious, Wilkes glimpsed the awful truth.

Campbell-Kelly tells the story of that program in the *Annals of the History of Computing* (Volume 14, Number 4, 1992). After the June conference at Cambridge, Wilkes began work on the first nontrivial EDSAC program, which calculated a table of values for Airy's integral (the solution of a differential equation that turns up in areas such as the theory of the rainbow). Wilkes has given no detailed account of how he wrote the program, but there is documentary evidence that it did not run successfully on the first try. In 1979, while cleaning out his office, Wilkes came upon an old punched tape, which turned out to be an early version of the Airy program. He gave a copy to Campbell-Kelly, who has since teased out its meaning.

The program is 126 lines long and has twenty errors. Most of the errors are mere typographical slips and small lapses in syntax or logic. For example, Wilkes got two conditional-branch instructions backward, causing the program to jump to a new location when it should continue sequentially, and vice versa. The program calculates an intermediate result but then neglects to store it for future use. In two other cases a value is stored in the wrong place. All of these minor blunders will be maddeningly familiar to anyone who has done some programming. The human reader passes over such errors without even seeing them, unconsciously filling in the intended meaning, but the machine is resolutely literal. It does only what it is told to do.

One of the bugs in the program was more subtle. Campbell-Kelly writes:

> When Wilkes first sent me the tape in 1979, it did not take very long to coax some numbers from the program, but although the results were correct to four decimal places, there was an error of as much as four units in the fifth place.... Since everything else in the program looked perfect—and since I had spent more time trying to debug the program than I really care to admit—I was forced to concede defeat and put it to one side. During the intervening years between then and now I looked at the program again two or three times but the bug remained obstinately hidden. Finally, one morning in early 1990, the penny finally dropped: The error was caused by the fact that the constant $(\delta x)^2/12$ in location 45 was stored only to single precision instead of double precision.

The prevalence and the stubbornness of such errors took the early programmers by surprise. The speakers at the 1949 Cambridge meeting were gravely concerned about hardware faults and how to detect them. Several early machines, including the EDVAC (but not the EDSAC), were built with two complete arithmetic and logic units just so that all operations would be continuously double-checked. But the corresponding problem of software errors got little attention. After all, programs would be written by mathematicians and others skilled in the manipulation of complex formulas; surely they could get it right the first time.

Having spent a little time writing simple EDSAC programs and struggling to get them to run on Campbell-Kelly's simulator, I am impressed that any useful work at all was ever done on the computer. The pioneer programmers who mastered the stern and unyielding machine were obviously prodigies of concentration and patience.

The simulation is unrelentingly realistic. The only aids to writing, running and debugging programs are the ones that were available to Wilkes and his colleagues in 1949. Moreover, the only input and output facilities are the ones available to the original machine. You punch a program tape by typing the orders in a long list. Pressing the Start button reads the tape into memory and starts the computation. Other buttons execute a single program step, clear the memory and reset the processor, or stop the machine. Output from the program appears in a scrolling window, which mimics the continuous roll of paper in the teleprinter of the original EDSAC; the simulated printer clicks quietly and at the end of every line produces the *whump-clunk* of a carriage return. The only facility for interacting with a running program is a telephone dial, which allows single digits to be dialed in; even that device was not added until 1951.

The EDSAC had several oscilloscope displays whereby the contents of memory tanks and various registers could be monitored. Those displays are also present in the simulator. They show binary 1's and 0's as large and small dots, which dance and flicker appealingly when a program is running. The largest of the displays has sixteen rows of thirty-five dots each, representing the entire contents of a long tank. Among the demonstration programs included with the simulator is a virtuoso performance by A. S. Douglas, who was then a student and later became a president of the British Computer Society. In Douglas's program the contents of memory are manipulated in such a way that the long-tank oscilloscope displays the playing field for a game of tic-tac-toe.

On first glance at the EDSAC instruction set, programming the machine seems straightforward enough. The instruction A 45 adds the contents of memory location 45 to the contents of a special register called the accumulator. Similarly S 46 subtracts the contents of location 46 from the accumulator. T 47 transfers the contents of the accumulator to location 47 and resets the accumulator to 0. There are eighteen instructions altogether, each specified by a one-letter code. It is easy to see how the designers of the machine could have believed that those instructions would meet all their needs, and indeed there is nothing wrong with the individual instructions. The trouble begins when you try to string instructions together in a meaningful sequence.

Suppose you want to add the numbers 5 and 8. Doing the addition is quite easy—all it takes is an A instruction—but almost every other step of the program has its own little snag. If you want to add 8 to 5, you must first get a 5 into the accumulator, but there is no instruction for directly setting the accumulator to a specified value. All you can do is make sure the accumulator is initially 0 and then add 5 to it. The only way to zero the accumulator, however, is to execute a T instruction, which also has the effect of storing the old value of the accumulator at some memory address. Hence some location must be available to serve as a "rubbish heap."

On the basis of those ideas, a sequence of instructions for adding two numbers might look like this:

```
065    T    71    F
066    A    69    F
067    A    70    F
068    Z          F
```

Here the numbers in the leftmost column are not part of the program; they merely identify the memory locations at which the instructions are stored. The next column holds the instruction codes themselves, T for transfer and A for add. The numbers following each code are addresses: location 71 is serving as a rubbish heap, whereas location 69 is assumed to hold the value 5, and location 70 holds the value 8. The F that follows each instruction indicates that the instruction operates on "short," or single-precision, quantities of seventeen bits; a D would signal "long," or double-precision, values of thirty-five bits. The Z instruction halts the machine when the addition is done.

The program is already a fairly intricate piece of work just for adding a couple of numbers, but it gets worse. Storing the constant values 5 and 8 in their assigned places might seem a trivial task. In fact, it is the hairiest part of the program. The EDSAC had no special provision for specifying numeric values, and so constants had to be entered on the tape by typing out the instruction code whose binary pattern matched the appropriate numeric value. The integer 5, for example, is represented by the instruction P 2 D, which just happens to have the seventeen-bit binary encoding 00000000000000101, equal to 5 in decimal notation. The code for decimal 8 is P 4 F, which translates to 00000000000001000. Hence the full addition program would consist of these instructions:

```
065   T   71   F
066   A   69   F
067   A   70   F
068   Z        F
069   P    2   D
070   P    4   F
071   P        F
```

The statements in locations 69 through 71 that specify constants are called pseudoinstructions, because although they look like instructions, they are not meant to be executed. The final pseudoinstruction, which stores a 0 in location 71 to reserve that address as the rubbish heap, illustrates one more little peculiarity for the programmer: a 0 in the address field of an instruction is represented not by a 0 but by a blank.

All those tricks and foibles of the instruction code put a formidable obstacle in the way of the beginning programmer, but in the long run they are probably not the major source of difficulty in working with the EDSAC. Such minor arcana can be mastered with a few days' practice. The snares that catch even experienced programmers lie elsewhere.

Suppose you have written a program to compute the sum 5+8, and you need to modify it to calculate 5+8+21. Inserting a third A instruction is easy, and working out a way of representing 21 (it's P 10 D) is merely a bizarre exercise, but when the new instructions have been written, the job is not yet finished. Interpolating a third add instruction pushes all the subsequent data down in memory, which means that the addresses embedded in the earlier instructions must be adjusted. Indeed, address adjustments are needed whenever instructions are inserted or removed, with the consequence that a change anywhere in the program text can have repercussions arbitrarily far away.

The tedious and exacting chore of keeping addresses current was quickly recognized as an invitation to error. An ingenious partial solution was contributed by David J. Wheeler, a Cambridge student who joined the EDSAC project early on and who is now professor of computer science at Cambridge. Wheeler wrote the "initial orders," which were manually loaded into the machine's first few memory locations and which then loaded other programs through the tape reader. The initial orders included aspects of what would now be called an operating system (for loading and executing programs) and aspects resembling those of an assembler (for converting symbolic instructions into their binary form). In the second version of the initial orders, completed in September 1949, Wheeler included a facility for "relocatable" subroutines. The facility would adjust addresses automatically according to where the routines were loaded into memory.

In spite of the sparse instruction set, the weird notation for numeric constants and the difficulty of working with absolute addresses, it needs to be said that the EDSAC was the friendliest of all the first-generation computers. At least it accepted input in the form of alphabetic characters and decimal numbers. Workers on other machines had to manually translate their programs into raw binary or hexadecimal (base 16) notation. And at least EDSAC instructions had only one address to go wrong; in the EDVAC every instruction had four address fields, each of which might have to be adjusted after every program change.

It did not take long for the Cambridge group and others to recognize that getting programs right was going to be a long and laborious undertaking that would demand a major investment of intellectual resources. Wilkes wrote in his autobiography, *Memoirs of a Computer Pioneer*, of the moment, during his work on the Airy tape, that the nature of the problem became clear to him:

The EDSAC was on the top floor of the building and the tape-punching and editing equipment one floor below. . . . It was on one of my journeys between the EDSAC room and the punching equipment that, "hesitating at the angles of stairs," the realization came over me with full force that a good part of the remainder of my life was going to be spent in finding errors in my own programs.

By the time of the next British computing conference, in July 1951, the problems of programming and debugging were being taken more seriously. There was even mention of "programme-translating programmes," and Alan M. Turing discussed the idea of an interpreter, a program that would enable the computer to execute programs written in a higher-level language. By then, too, the EDSAC had a suite of debugging routines ready for programs that went awry. They included "postmortem" routines, which printed out the state of the machine after a program had stopped, and "trace" routines, which printed out information as each instruction was executed.

At about the time of the second conference, Wilkes, Wheeler and their colleague Stanley Gill published the first textbook on computer programming, *The Preparation of Programs for an Electronic Digital Computer*. The three collaborators included a chapter on "pitfalls," which began by acknowledging that even a competent programmer will sometimes make a mistake. "Experience has shown that such mistakes are much more difficult to avoid than might be expected," they wrote. "It is, in fact, rare for a program to work correctly the first time it is tried, and often several attempts must be made before all errors are eliminated." The copy of Wilkes, Wheeler and Gill in my local university library was evidently donated by someone who had had first-hand experience of programming the EDSAC or one of the machines modeled on it. The margins are full of penciled notes and comments. The two quoted sentences are underlined in red. ●

BRIAN HAYES is a contributing editor of THE SCIENCES.

For readers of *The Sciences* who want to experiment with the EDSAC simulator, Martin Campbell-Kelly has agreed to make the program and a tutorial guide available for $20, to cover the cost of printing and shipping. The package includes a library of subroutines and a variety of demonstration programs, so that it is not necessary to master the EDSAC instruction code to run the simulator. The programs come on one three-and-a-half-inch floppy disk. Checks for the $20 should be made out to the New York Academy of Sciences and sent to Department EDSAC, *The Sciences*, 622 Broadway, New York, New York 10012.