The EDSAC Programming Systems

DAVID J. WHEELER

This article gives the highlights and background of the EDSAC programming system as it developed from 1948 in anticipation of the EDSAC completion to the improvements made for the second Cambridge computer.

My introduction to computers was Douglas R. Hartree's inaugural lecture on November 13, 1946.[1] It discussed the ENIAC and concepts of computers. Thereafter, although still an undergraduate, I followed the work going on in the Mathematical Laboratory, constructing the EDSAC, under the leadership of Maurice V. Wilkes.

To a minor extent I participated in the work going on in the Mathematical Laboratory. One such job was the selection of letters for the five function bits of an instruction. We chose them to be as mnemonic as possible, but although the order decoder of the EDSAC was implemented for each order separately, certain restrictions remained so that, for example, M could not be used for multiplication. Another action was to point out an inconsistency of the overflow provision and preliminary design of the initial orders. I must have been a pest! These initial orders were orders wired on a stepping switch which were automatically loaded into the store when the Start button was pressed. They were then obeyed and loaded the user program, punched on paper tape, into the store. I wrote a replacement for the initial orders which read in programs punched in a mnemonic code and decimal address notation.

I also helped construct a clock frequency adjusting circuit to allow for temperature variations. This was not good enough, so the mercury ultrasonic delay lines were placed in a box whose temperature was thermostatically controlled.

The initial orders and subroutines

My work as a research student began in summer 1948, and my first program of consequence as a research student was the "coordinating orders." This was a supplement to my first set of initial orders and was loaded from paper tape before the rest of a program.

It allowed for the simple building of a program from subroutines held on paper tape. These were automatically adjusted for their destination and listed in a directory so that, for example, subroutine 5 could be entered using the fifth position in a list. I am not sure when they were completed, but I gave a seminar in November 1948 extolling their advantages.

The machine began working in May 1949. It was originally intended that numbers would be held in the range –2 to 2. However, when the initial orders were tried, they failed as it was discovered the numbers were held in the range –1 to 1. We rapidly convinced ourselves of the superiority of this range, adjusted the initial orders (changing two wired bits), and began to run programs. Martin Campbell-Kelly's papers describe those early days.[2,3]

After the EDSAC commenced work there was an opportunity in August 1949 to rewire the initial orders, the capacity of the stepping switches activated by the START button having been increased to 41 orders. The paper-tape codes and function codes were already fixed, so it was a self-contained job to program as many facilities as possible into the restricted space. The facilities of the coordinating orders gave us a good idea what was needed, and we also removed the tedious limitation of the first initial orders that the tape had to be loaded into the reader at exactly the first character. The final program to be wired on the stepping switches as a new set of initial orders had one spare position as any extra facility needed at least two extra instructions. This spare position was used to store a useful constant, which was used in entering what we called closed subroutines.

We had already thought about subroutines, and Douglas R. Hartree had suggested the names *open* and *closed* for what are now called *macros* and *procedures*. It was essential to have a library, and we used "preset parameters" to make the routines more versatile. The preset parameters were written and punched like orders, but were used by the initial orders to tailor the program being read in, so that they used no valuable space or time thereafter. For example, one could set the number of decimals to be printed or the number of differential equations to be solved. More general "program parameters" could be used, but their use entailed a greater use of the rather small store. Space was needed both for the values and for the orders needed to use the values, whereas where they were applicable, preset ones avoided both these space demands. In May 1949 the store was 256 words, each word able to hold a long number (10 decimal digits or 35 bits) or two short numbers or orders. Later that year the store was doubled in size. However, on some days not all the delay lines, each holding 16 words, would be working, and the store would be electrically reconfigured so the addressing of working delay lines was consecutive. Thus the machine was usable for programs which did not use the full store.

The second set of initial orders made it easy to load the computer tape reader as the tape could be positioned anywhere on a blank leader at the tape beginning. Other facilities included the use of preset parameters to make the subroutine library more versatile, and the use of "jiffy tapes" for ease of correction of programs. The basic routine al-

1058-6180/92/1000-0034$03.00 © 1992 IEEE

lowed directives punched on the program tape to set a current marker, to send routines to specified locations while doing the relocation adjustment, or to transfer control to starting points specified absolutely or relative to the current marker. One regretted omission, due to the 41 order limit of the stepping switches, was the easy input of numbers with the program orders. So to read and convert numbers required an "interlude" using a library routine, or else converting them to the order format and inputting in that notation.

Interludes were computations that took place while the program was being constructed. An interlude was read in, obeyed, and then usually overwritten by the remaining program. The interlude might well be part of a library routine. It allowed the use of such routines to be simplified by doing preliminary computations. For example, a program to solve linear equations could have as preset parameters N and N^2 set by the user, or N set by the user and N^2 set by the interlude. Another use of interludes was the unwinding of program loops during input so the code ran faster, although it took more space. The input of sets of numbers often used a number-input library routine, which was overwritten by the remaining program or used as working space for the calculation.

Master program

Location	Order			Notes
m	A	*m*	F	The accumulator is assumed zero, so the A order adds itself into the accumulator. The A order is negative.
m+1	G	*n*	F	Jump to *n* if accumulator is negative.
m+2	. . .			The subroutine jumps here when it ends.
Subroutine				
n	A	3	F	Constructs order E (*m*+2) F from A *m* F by adding (E *m*+2 F – A *m* F) = U 2 F which is held in 3, the "spare" space of the initial orders used to hold a constant.
n+1	T	*p*	F	Stores E *m*+2 F in location *p*.
	. . .			
	. . .			
	. . .			
p	E	(*m*+2)	F	Jumps back to master program if accumulator is positive or zero.

Figure 1. Subroutine calling method on EDSAC. F is used as an order terminator when the operand is an order or short number. The identifiers *m*, *n*, and *p* represent decimal addresses.

The subroutine-call sequence used for EDSAC (Figure 1) was designed for ease of use and shortness of the master program. The basic idea spread elsewhere. However, later machines, including the EDSAC 2, had special instructions for entering and leaving subroutines. The call sequence also shows how the accumulator had to be used for adjusting orders when there were no index registers. These ideas were explained in my paper of 1950.[4]

The initial orders were only part of a complete "operating system" that provided regular testing times for short programs, running of production programs by operators, fault location, and so on. The test-time queues in which users waited while ready to test a program were a valuable means of communication that updated users on new facilities or hazards. There was a positive incentive to provide assistance to the people ahead of you in a queue to help them end their runs satisfactorily as soon as possible. Only on very rare occasions was such help malicious!

The control panel was very simple, having just five push buttons:

- Start loaded the initial orders and by obeying them started reading the paper tape of the program.
- Clear made the store all zeros, which ensured consistency of results and, as the order zero stopped the machine, helped fault finding.
- The Stop button would stop the program running.
- The Single Step button would obey one order while the machine was in the stopped state (or had an unpredictable effect on a running program).
- A Reset button was available for restarting the machine after a manual or program stop. It would not restart the machine after an unimplemented order or one having zero function bits.

The Clear store button was a later addition, and in the early days individual delay lines were cleared by touching an exposed diode on the appropriate chassis with a wet finger. There was a punched-paper-tape reader by the control panel which acted as the input device of the EDSAC. It was program controlled, and an input order could read just one row from the tape giving five bits.

Another valued part of the system was the careful printed description of the library routines available for use. These descriptions were excellently written by Eric Mutch. The quality of such a library is determined by the rejected routines. We did not fall into the common trap of accepting any submitted routine! The program guides that were available developed into the book titled *The Preparation of Programs for an Electronic Digital Computer*, the first published book on programming.[5] It was published in 1951, with a second edition in 1957, and it has been reprinted as Vol-

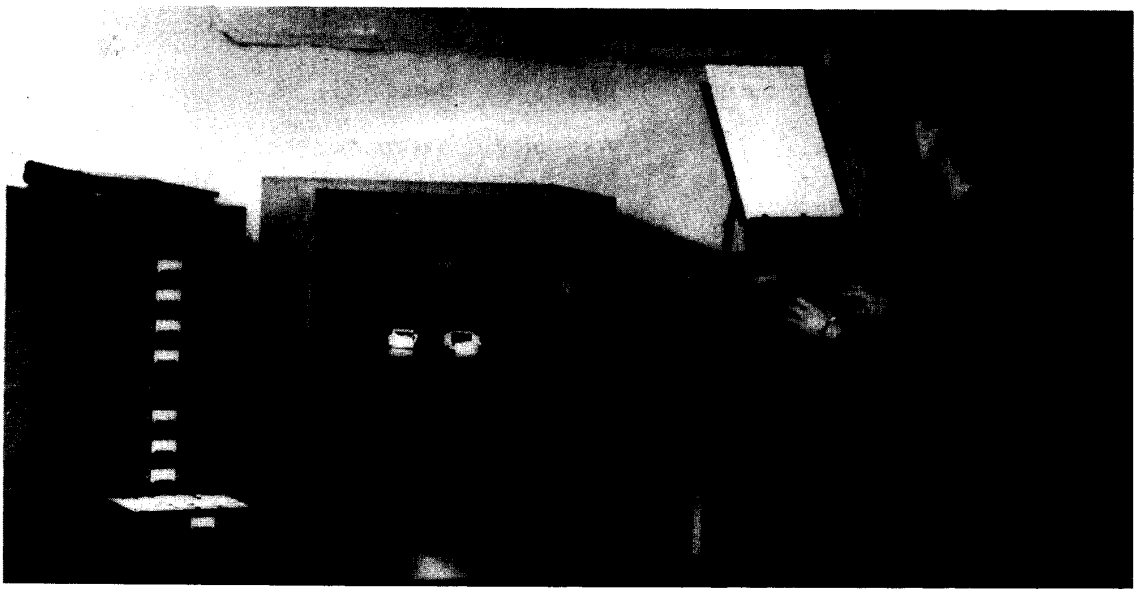


Figure 2. Punched tape preparation: Cara Mumford typing a program from a program sheet. The cabinet on the left held library tapes. The tape reader in the center was used to copy library and other tapes onto the user's tape.

ume 1 in the Charles Babbage Institute Reprint Series for the History of Computing. The latter has an excellent introduction written by Martin Campbell-Kelly.

By the end of its life in 1958, the EDSAC had grown in power and reliability. The introduction of an index register in 1953 eased programming and about doubled its effective speed. Experimental magnetic tapes became available rather late in its life, and the original programming system remained unchanged — an example of the power of large numbers of users with working programs.

Running a program

Let us consider how a calculation might be tackled. After the job was formulated, the first task was to break the calculation into parts. This was usually done by having a master program controlling a set of subroutines. Some of these would be library subroutines or subroutines from earlier programs; others would have to be written and tested before being incorporated in the program.

The programs for EDSAC were written on printed program pads. Program sections usually began with the control combination G K. When this was read by the input routine, it recorded the location into which the next order was to be put. The orders following, which were terminated by Θ, had their written addresses increased by that value. Thus an order having an address written as 5Θ would be converted to $5 + m$, where m was the location recorded by G K.

Other order terminators such as H or N allowed preset parameters to be added to an order as it was read from the paper tape and put in the store. They were set by different directives. The use of program pads and the breaking down of a program into subroutines meant that flowcharts were never used as programming aids for EDSAC.

To test a subroutine, a tape was made which ran a few tests of the subroutine and usually printed results from test data. Such a test involved making a program tape by punching on a keyboard perforator the new tape parts. This was often done twice, and then the tapes were tested against each other to check for errors. This eliminated most of the punching errors. In the tape preparation room there was a comparator into which two tapes could be fed and which would stop when a discrepancy occurred. Then the full test tape would be made, using a tape reproducer, from the new program tape and library tapes held in cabinets nearby. This copying was again tested in the comparator. All this tape-room preparation of program tapes was very tedious and gave great incentive for care in all aspects of program preparation (Figure 2).

Another technique was to punch a new program complete with all its library routines copied in. Jiffy tapes could adjust the program so that individual parts could be tested. A jiffy tape was a short length of tape with one or two directives and a few orders that modified a program already in the memory by overwriting a few orders by their replacement, or else planting a jump to a set of additional orders. Thus a program could be patched up.

The use of jiffy tapes saved tape preparation effort at the cost of extra input time. Very short jiffy tapes were punched on a keyboard perforator in the EDSAC room. The tape preparation room was two flights of stairs away! There was no provision for changing instructions in the store directly by means of manual keys or switches. All small changes were made by reloading the initial orders and reading in a jiffy tape. Because such a tape could be keyboarded in direct notation and if desired punched again and compared by eye, finger trouble was minimized.

The EDSAC was run by a schedule which included test-run sessions a few times per day and production runs of longer duration for the rest of the day. At these test sessions, your time was limited to a few minutes. Operators were used to run programs in your absence at test-run times and for the longer production runs during the rest of a day. The "operating system" was a queue of tapes ready to run.

At night the machine was given over to various program groups who would use the machine collectively until it broke down. There was no night maintenance by engineering staff in the early years. The groups would allow other users short test runs in "their" time. Some people became adept at bypassing the effect of minor hardware faults. There were many potentiometers that adjusted the gain of aging electronic tubes. Some faults could be cured by adjusting them, but it was easy to adjust the wrong one and make the machine worse.

It is worth noting that the only replaceable plug-in units were vacuum tubes, and intermittent faults were common. The average time between machine errors was about half an hour. However, this is deceptive as the were "good" and "bad" days, and most faults were intermittent.

Finding program faults

Program faults showed in a variety of ways. If an undefined order was tried, the machine stopped and showed the faulty order in the order register display. If the program went into a loop, it could be manually stopped. In both these cases the Start button could be pressed to read in a "postmortem" program. This would then print designated parts of the store as decimal numbers or instructions (not in hexadecimal as in core dumps of later years). The portion of store was selected by the precise point at which the postmortem tape was loaded. Later developments allowed the designation by the use of a telephone dial. The origin of the term "postmortem" is unknown, but some hold it was due to the fact that the Mathematical Laboratory was housed in the old Anatomy Building which included a corpse lift!

The noise made by a running program helped diagnose program faults. The noise made by an unchanging loop or the silence of a stopped machine were particularly useful. The noise was simply generated by connecting the accumulator bit stream directly to a loudspeaker. Loops had characteristic notes, and the rhythm of a running program was memorized subconsciously and discrepancies were noted without conscious effort. Some of the other fault-finding methods in common use were looking at selected registers visually while the machine was running, using checkpoints, and using trace routines.

It was possible to select one delay line and show the 32 orders or short numbers on a monitor cathode-ray tube, while the program was running. The display was in binary and users noted activity rather than looking for precise values. Some users put important variables in a single delay line so that the display reflected their activity. Needless to say, the display was subverted for games such as ticktacktoe, and occasionally used to display such figures as a dancing kilted Highlander.

```
     Order        Notes
 1   P 10 F       a constant = 10 · 2^-15 written as a
                  pseudo-order
 .
 .
 .
12   S  1 Θ       subtract 10 · 2^-15
13   H 30 D       set mult register
14   T 12 F       store count value
15   V    D       multiply by 0D
16   T    D       product to 0D
17   A 12 F       add counter
18   A  2 F       unit of count
19   G 14 Θ       jump to 14 if accumulator is negative
20   H    D

Trace output
SHTVTAAG
TVTAAG
TVTAAG
  repeated six more times
TVTAAGH...
```

Figure 3. Example of Gill's checking (trace) routine. The code fragment calculates $x^{10} \cdot y$ by a simple multiply count loop. The result is placed in location 0D; x is in 30D, and y in 0D.

The trace routine was invented by Stan Gill[6] and was a powerful method of locating stubborn faults. It obeyed the original program interpretively order by order and printed just the function letter of each order obeyed. A new line of printing was started at each program jump. Although this sounds like a strange form of trace, it did match the way programmers tended to think about programs and so was much more effective than it may appear. It corresponds to running down the left-hand letter written on a program sheet. (When a program was created, often just the left letter was written first and the addresses filled in later.) An example is shown in Figure 3.

Unlike conditions today, some faults were computer hardware faults and caused symptoms similar to a program fault. The hardware faults were usually intermittent, and a simple repeat could detect such a fault. Human nature caused a number of unnecessary repeats as the hope "it was not my fault" was strong.

Errors were usually corrected by jiffy tapes. These short tapes were read in after the program was loaded and would correct an order by replacing it with a new one or by putting a jump order in the program and jumping to a patch of a few orders, which would correct the program and then jump back to the original program. The directives obeyed by the initial input orders made this easy, and the length of such a jiffy varied from a few inches to a few feet. It was made of directives and orders to overwrite what was there, followed finally by a start directive. For example, to replace order

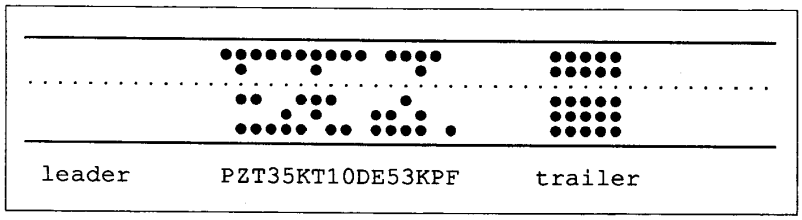


Figure 4. Example of a jiffy tape. P and 0, T and 5, and so on, are on the same key, so their binary equivalent is the same.

T 12 D by T 10 D at location 135 and start the program at 53, one would punch a tape as follows:

```
P      Z : causes blank tape to be ignored
T 135  K : directive: causes the following orders on tape
           to be put at 135, 136, etc.
T  10  D : replacement order
E  53  K : directive: enters program at 53
P      F : with zero accumulator (P gives zero function
           bits in an order)
```

The characters punched on the jiffy tape were those before the colons, with blanks omitted. The 17 punched characters occupied 1.7 inches, short enough that the tape could usually be punched without error. A useful convention was to start a tape with a few inches of blank tape and to end it with a few rows of blank tape followed by a few rows of all holes. This distinguished the ends of a tape and was particularly important for short jiffy tapes (Figure 4). The jiffy tapes grew longer as more corrections were included. Then, when the program was thought to be correct or the jiffy tape so long that to correct it was about as much trouble as changing the original program tape, the original program was corrected and the whole process started again.

The printing of results from the EDSAC was tedious as the teleprinter could only print at less than seven characters per second. Even when a punch was connected in 1951 instead of the teleprinter, it would only punch about 30 characters per second. Even so, there was a leader of tape between the punch and printer so the delay until printed results appeared was actually a little greater — but the next program could be entered while this was done. Some long output tapes were taken to a teleprinter in the tape room to print.

A considerable effort was made to facilitate the output of readable results, so there were printing routines of considerable scope. Nearly all print routines had digit or page layout. This encouraged users to print their results legibly, and not in a crude simple column. The very first programs printed legible decimal results. On many other computers they were printed in octal or hexadecimal, as if this was a desirable notation to learn!

It is to be noted that programs were never thought of or written in binary, octal, or such awkward formats. The library tapes were in the standard mnemonic decimal format, and the initial input routine acted both as assembler and as loader. The postmortem tapes produced decimal or order format, and there was no postmortem tape to print out in binary or octal.

Design of EDSAC 2

The detailed design of EDSAC 2 was aided by the use of EDSAC 1, which was used to generate wiring schedules for both the microprogram and the reserved store, which held permanently available routines. EDSAC 1 also simulated the EDSAC 1½, which was a test version of the EDSAC 2 using only 64 microprogram steps rather than the full 1,024 and not implementing floating-point operations. An interesting design program was one which positioned the micro-orders to minimize cross talk in the core matrix and thus enhance the working tolerances. It was my first program (1956) in which I used hashing to minimize search time.

In the system design of the EDSAC 2 programming system, one had available a microprogram of 1,024 steps including a complete order decoder of 128 steps, so that unlisted orders produced a "report stop." There was a read-only memory of 768 words with its own working store of 64 words. This was much better than the 41 initial orders of the EDSAC.

Thus it was practical to provide many facilities and ensure that they were provided in a simple and direct manner. The report stop is one example. A report stop was performed after a program failure such as using an undefined order, using an invalid operand, or ignoring an overflow detected by the microprogram. It printed the contents of the accumulator, whether an overflow had occurred, the order location, the order itself, and the values of the two index registers, each in appropriate decimal format. Thus pinpointed, the cause of an error was usually easy to locate.

Although the computer was built before the interrupt concept was known, and a primitive form was added toward the end of its life, the microprogram essentially had an internal interrupt that allowed a program to move a magnetic tape forward or backward a given number of blocks while the computation was proceeding. As in most machines of that era when logic was costly, transfers from magnetic tape were assembled into 40-bit words in the accumulator from 5-bit characters from the magnetic tape. A single order initiated the transfer of a number of words to or from the selected magnetic tape, and the microprogram attended to the details of the transfer.

Although programming pads were issued for the EDSAC 2, they were not used very much. The labels and parameter system made these less useful, and as program tapes could be printed by teleprinter, the printed version was a better reference, the comments, if any, being copied by hand. This facility lengthened the input tape because a double space or carriage return and line feed were needed between program items. Tape length still mattered. Although the computer could read the tapes at 1,000 characters per second, the teleprinters still only printed at less than seven characters per second. Later a line printer was connected, which could print about two lines per second. However, it was restricted to printing the decimal digits, decimal point, and a minus sign.

The complete programming system and details of how to use the computer were contained in a printed booklet of 64 pages. EDSAC 1 users could convert to the new computer and programming system by just reading the booklet, while new users needed a few days to become accustomed to it.

Programming EDSAC 2

The EDSAC 2 was an easy machine to use. It did not use a high-level language but provided most of the language facilities in the reserved store except for the notation. The code and assembler were designed as a single-pass system; the input tape was read at full speed with no time needed for a second pass or compiler time. The integration of the computer notation, input and output notation, built-in facilities, and powerful order code greatly reduced the preparation time — while the report and trace system facilitated the location of errors. Most numbers were dealt with in a floating-point format, so the considerable scaling problems of fixed-point machines such as the EDSAC 1 were eliminated.

Care had been taken so that orders did what was expected and faults were detected. A division overflow, for example, would stop the machine before it ran away to disguise the origin of a fault. Most operations would round correctly rather than truncate the result. The right and left shift instructions behaved as expected but also worked for zero or negative addresses.

The Appendix on page 40 gives an example of reading in a set of numbers, doing a calculation, and printing the result. This is the famous TPK example devised by Knuth and Trabb Pardo[7] as a test of programming languages. The comments are extensive to avoid listing the order code.

The calculation is to read in a list of 11 numbers, and compute for each list entry x, $F(x)$, where

$$F(x) = \sqrt{|x|} + 5x^3$$

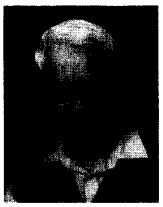
and print the list position n and $F(x)$ for each value read, in reverse order. If $F(x) > 400$ then "999" is to be printed instead.

It should be noted that the TPK program was written in "one pass." For the EDSAC 1 a common way of working was to write a program in two or three passes. On the first pass, function letters were inserted on the programming pad and some addresses filled in. The second pass enabled most of the unknown addresses to be filled in, while a third pass was often needed to fill in the remaining references.

Programming for the EDSAC 2 evolved from the EDSAC but was very much easier to do. The provision of labels, a structured order code, and a means to let orders have arbitrary constants instead of addresses enabled simple programs to be written directly. The permanent availability of the standard mathematical functions such as square root or exponential, together with routines for matrix division and solution of differential equations, ensured that most calculations were easy to program with short program tapes, and a language was not needed to supply them. The built-in trace routine, report facilities, and standard postmortem routines made debugging easy. The input and printing routines were superior to those of most languages of the time. The algebraic notation of languages catered to only a small part of the programming task, while the interchangeability benefit did not exist till much later. ■

References

1. D.R. Hartree, *Calculating Machines: Recent and Prospective Developments and Their Impact on Mathematical Physics*, Cambridge Univ. Press, Cambridge, UK, 1947; reprinted in Charles Babbage Inst. Reprint Series for the History of Computing, Vol. 6, MIT Press, Cambridge, Mass., and Tomash Publishers, Los Angeles, 1986.
2. M. Campbell-Kelly, "Programming the EDSAC: Early Programming Activity at the University of Cambridge," *Annals of the History of Computing*, Vol. 2, No. 1, Jan. 1980, pp. 7-36.
3. M. Campbell-Kelly, "The Airy Tape: An Early Chapter in the History of Debugging," Research Report 153, Dept. of Computer Science, Univ. of Warwick, Coventry, UK, 1990.
4. D.J. Wheeler "Program Organisation and Initial Orders for the EDSAC," *Proc. Royal Society*, Series A, Vol. 202, 1950, pp. 573-589.
5. M.V. Wilkes, D.J. Wheeler, and S. Gill, *The Preparation of Programs for an Electronic Computer*, Addison-Wesley, Cambridge, Mass., 1951 (first edition), 1957 (second edition). First edition reprinted as Vol. 1 of the Charles Babbage Inst. Reprint Series for the History of Computing, MIT Press, Cambridge, Mass., and Tomash Publishers, Los Angeles, 1982.
6. S. Gill, "The Diagnosis of Mistakes in Programmes on the EDSAC," *Proc. Royal Society*, Series A, Vol. 206, 1951, pp. 538-554.
7. D.E. Knuth and L. Trabb Pardo, "The Early Development of Programming Systems," in *A History of Computing in the Twentieth Century*, N. Metropolis et al., eds., Academic Press, New York, 1980, pp. 197-213.



David J.Wheeler is professor of computer science at Cambridge University, where he has spent most of his career. He started computer work as an undergraduate in 1947, and his PhD, titled *Automatic Computing with the EDSAC*, was granted in 1951. This led to a fellowship at Trinity College, Cambridge, in 1951, but he spent the next two years at the University of Illinois, helping design the programming systems for the ORDVAC and the ILLIAC. Returning to Cambridge in 1953, he designed extensions to the EDSAC such as the index register and the programming system for EDSAC 2. Since then he has worked on the Cambridge Titan computer, extensions for on-line working, the CAP computer, and the Cambridge Ring.

Wheeler has spent time at the Universities of Illinois, California, and Sydney, Australia, and has acted as consultant to various companies including Bell Labs at Murray Hill, N.J., and DEC at the Western Research Laboratory.

Wheeler was elected a fellow of the British Computing Society in 1970 and a fellow of the Royal Society in 1981. He received the Pioneers Award of the IEEE in 1985.

Wheeler can be reached at the Computer Laboratory, University of Cambridge, Pembroke Street, Cambridge CB2 3QG, UK.

Appendix: The TPK calculation for EDSAC 2

This EDSAC 2 program performs the TPK algorithm, as it would have been written about 1958. The program is punched as the left-hand column. Either a carriage return and line feed, or a double space, separated program items. Items were allowed single spaces in their format so that, for example, 1.2345 6732 would be read as a single number. Thus formatted, output could easily be read in again. No comment facility was provided, and the comments below were not punched. (This saved both preparation and program loading time.)

```
p1=100          The program is put in location 100 and onward.
                The constant list starts at 2000 by default. This
                saves the line of code p2=2000.
3/22  4/2       Set the layout for printing: 22 items in a block
                of 2 columns.

1/-.29405905    Set layout constant for routine 21 to
                print 9 figures with a point after first four. Zero
                suppression to point. This can also be done
                during program.
f               A directive to convert numbers to floating
                format.
t     n  F(x)
                The title facility causes the rest of a line started
                by t to be copied to the output. The above are
                all directives and so did not use or change the
                program store.
```

Subroutine for *F(x)*

```
19f2            Save x by storing the accumulator in 2.
20f2            Load the absolute value of x.
59f11           Call the permanent square root routine.
                Permanent routine entry number 11.
19f4            Save result in 4.
26f5            Set accumulator to value 5 in floating-point form.
14f2            Multiply by x.
14f2            Multiply by x.
14f2            Multiply by x.
12f4            Add root saved in 4.
60f0            Exit from subroutine with result in accumulator.
```

Master program

```
71s22(10        Set s modifier to –22 to count 11 times in steps
                of 2. Also set p10 to current position to act as a
                label. Note the label is on the right of the item.
59f10           Read a number in floating format using
                permanent routine 10.
19s42           Store result in location 42 + s.
74sr-2          Jump back two orders ("r" indicates relative).
                Increase s by 2 and jump till zero is reached.
                Labels were often not used for jumps of 1 or 2.
70s22           Set s = 22.
26s-2(11        Set s – 2 in accumulator in floating format.
14*.5           (s – 2)/2 = i for printing. * caused the assembler
                to insert .5 in location p2, the address to
                become the value of p2, and p2 to be increased
                by 2. Thus the accumulator is multiplied by .5
                at execution time.
59f24           Print value as a signed four-decimal integer.
10s18           Load argument from s + 18 initially 40.
58f100          Call subroutine to calculate F(x).
19f2            Save F(x).
13*400          Subtract 400; * causes it to be listed as a constant.
55p12           Jump if -ve to label (12.
26f999          Load literal address in floating format.
19f2            Replace by 999 if too large.
10f2(12         Load F(x) or 999.
14*1_10 5       Multiply by listed constant.
59f21           Print in layout determined by the preset
                parameter.
75sp11          Decrease s by 2, jump to order labeled
                by (11 unless s zero.
101f0           Stop and signal operator.
sp10            Directive: start program at label 10.
```

```
1.5  8  –6  9.5  2.3  9.9          test data
2.1  –2.1  6  0.001  –0.002
```

The results would have been printed as below:

n	F(x)
10	0.44472
9	0.03162
8	999.00000
7	–44.85586
6	47.75414
5	999.00000
4	62.35158
3	999.00000
2	–1077.55051
1	999.00000
0	18.09974

Notes

As part of the general design, the characters f, r, s, t, p, n, $_{10}$, and $_2$ were on figure shift for ease of input and printing. Thus a number such as 3.123 456 $789_{10}3_2$-39 could be input in floating or fixed format. The $_{10}$ symbol was punched and printed as a single character.

The digit-layout constant allowed for the output of floating- or fixed-point numbers with a decimal point, spaces, and a decimal or a binary exponent. Orders could also be printed in their own format. There were a number of default settings for ease of use; for example, the routine call "59f24" printed a floating-point number as an integer of up to four decimal places.

A more natural way of doing the TPK calculation would have been to read a string of numbers terminated by the end marker "/". The program would have been shorter and in general more useful. The input routine recognized / and would then jump to index *t*.