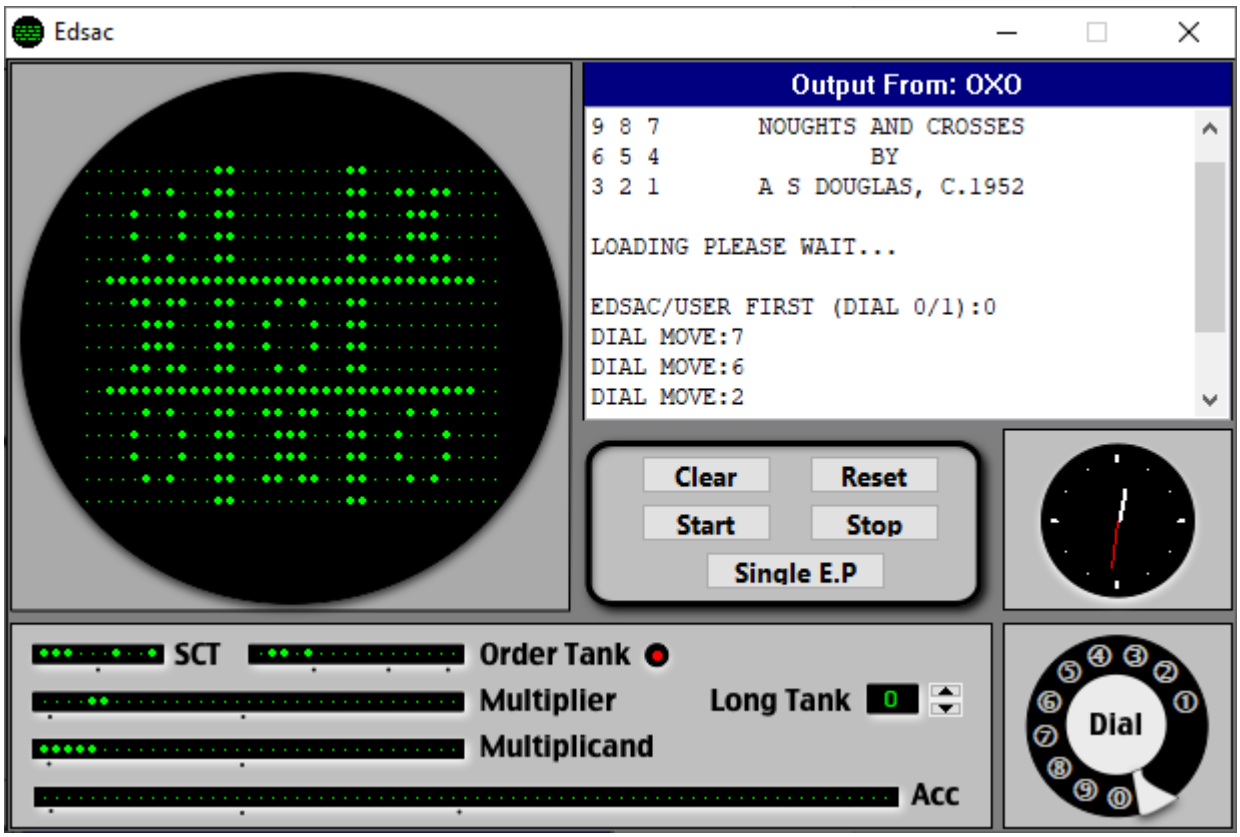
The EDSAC Replica Project

Tutorial Guide to the EDSAC Simulator



December 2021

The EDSAC Replica Project, The National Museum of Computing,
Bletchley Park, Milton Keynes MK3 6EB, United Kingdom
© Martin Campbell-Kelly, 1990-2021

The Cover

The cover shows an interactive computer game of Noughts and Crosses (Tic-Tac-Toe) developed by a PhD student, Sandy Douglas, about 1952.

To play the game, open OXO from the folder of Demonstration Programs and press the Start button. Enter your moves using the telephone dial.

A Tutorial Guide to the EDSAC Simulator
for
Windows, Macintosh, and Linux

Abstract

The EDSAC was the world's first stored-program computer to operate a regular computing service. Designed and built at Cambridge University, the EDSAC performed its first fully automatic calculation on 6 May 1949. The simulator is a faithful emulation of the EDSAC designed to run on a personal computer. The user interface has all the controls and displays of the original machine, and the system includes a library of original programs, subroutines, debugging software, and program documentation. The *Tutorial Guide* includes a description of the EDSAC and an account of the programming techniques developed for it during 1949-51. Several demonstration programs and programming problems are supplied, so that users can gain first-hand experience of what it was like to develop and run a program on a first-generation computer.

Contents

	Before You Begin: What the Papers Said	4
1	Getting Started	5
2	EDSAC Architecture and Arithmetic	13
3	Programming the EDSAC	19
4	Debugging: Getting Programs Right	31
5	Problems from the Summer School and Elsewhere	37
	Bibliography	40
	Appendix of Tables	41

Before You Begin: What the Papers Said

In the late 1940s the EDSAC - and "electronic brains" in general - captured the public imagination and were widely reported in the press. Before you begin using the simulator you might like to read the newspaper headlines and extracts below; while not always accurate or temperate, they do capture the excitement of the period.

A Don Builds a Memory

Short, dapper Dr. M.V. Wilkes, director of the Cambridge mathematical laboratory and ex-wartime radar backroom boy, is in charge of the calculator ... He told me yesterday: "The brain will carry out mathematical research. It may make sensational discoveries in engineering, astronomy, and atomic physics. It may even solve economic and philosophic problems too complicated for the human mind. There are millions of vital questions we wish to put to it."

- *Daily Mail*, October 1947

New Brain Stores Orders

The world's most advanced electronic calculator, one of the so-called mechanical minds, was recently completed at Cambridge University mathematical laboratory. Yesterday the joint designers, Mr. M.V. Wilkes and Mr. W. Renwick, gave me a preview of "Edsac" (electronic delay storage automatic calculator). It has a 3,500-valve "brain" weighing about a ton. ... A team of 10 have been assembling "Edsac's" 120 racks of valves, covering a floor area of about 500 square feet, since early in 1946.

- *Daily Telegraph*, June 1949

Mechanical Brain

On the top floor of a rather drab building in a narrow Cambridge back street is an apparatus which seems to consist chiefly of a vast number of valves set in grey painted racks. ... this weird array of wires and valves is a "mechanical brain." It has just been completed and it is the most advanced in the world. It is probably the major scientific marvel of 1949 and although until now we have lagged behind America in mechanical brains this one puts us streets ahead ...

This is how it works. First Mr Wilkes fed a strip of paper punched with holes into a "ticker-tape" machine. As the paper ticked through ... miniature television screens showed a row of green blobs ... then almost instantaneously a teleprinter nearby began to print rows of figures. That was all. There were no dramatic sparks, no dramatic flashes ...

There are not enough "brains" to go around at the moment, but a dozen would probably be sufficient for the whole country ... The future? The "brain" may one day come down to our level and help with our income-tax and book-keeping calculations. But this is speculation and there is no sign of it so far.

- *The Star*, June 1949

1 GETTING STARTED

The purpose of the EDSAC simulator is to provide an understanding of what programming was like on a first-generation computer. The material in this guide is accessible at several levels. This section, Getting Started, gives a broad overview of the technology of the EDSAC, and enables the first demonstration programs that were designed to put the machine through its paces to be run; this material should be accessible to any computer literate person. Section 2, Architecture and Arithmetic, describes the EDSAC's architecture, the instruction set, data storage, and arithmetic; this material should be accessible to anyone who is familiar with twos-complement arithmetic and basic computer structure. Sections 3 and 4, which cover programming and debugging, should be accessible to anyone familiar with programming. Finally, in Section 5 a number of programming problems are given, which range from elementary to quite difficult.

So that you can explore the EDSAC without recourse to other materials, this guide is designed as a self-contained document; however, you should note that this still leaves quite a lot more you can learn about the EDSAC. Details of the literature on the EDSAC are given in the Bibliography.

You may find the *Tutorial Guide* is of most value if you work through it systematically, run each demonstration program as it is encountered, and attempt at least some of the exercises. This is advisable, not least, because the EDSAC simulator is an accurate representation of a very primitive computer system—there are, deliberately, almost no facilities provided for trouble-shooting, other than those which were originally provided on the EDSAC.

1.1 Display and Controls

The EDSAC simulator runs in a simple Interactive Development Environment (IDE) in which you can either edit program texts or run programs (Fig. 1). However, before examining the simulator in detail it will be useful to see what the original EDSAC environment looked like (Fig. 2). [1]

Fig. 2a shows a general view of the EDSAC taken shortly after its completion in May 1949. Like all stored-program computers, the EDSAC had a processor, a memory, and input-output devices. The processor occupied most of the bulk of the EDSAC—some 3500 electronic tubes in all. The memory cannot be seen in the general view, but Fig. 2b shows a battery of the mercury delay lines from which it was constructed, photographed shortly before the machine was put together. Input-output was achieved on the EDSAC by means of a 5-track paper-tape reader operating at 50 characters per second, and a Creed teleprinter operating at $6^{2}/_{3}$ characters per second. This equipment can be seen on the wooden table at the right of the general view.

A little more about the memory. The main memory was designed to have a total of 32 delay-lines (or "tanks"), each of which stored 32 words of 18 bits. Hence the total memory capacity of the EDSAC was the equivalent of about 2 kilobytes. The same technology of mercury delay lines was also used for the processor registers—although

[1] In this manual "Edsac" applies specifically to the simulator; "EDSAC" is used to refer to the original computer.

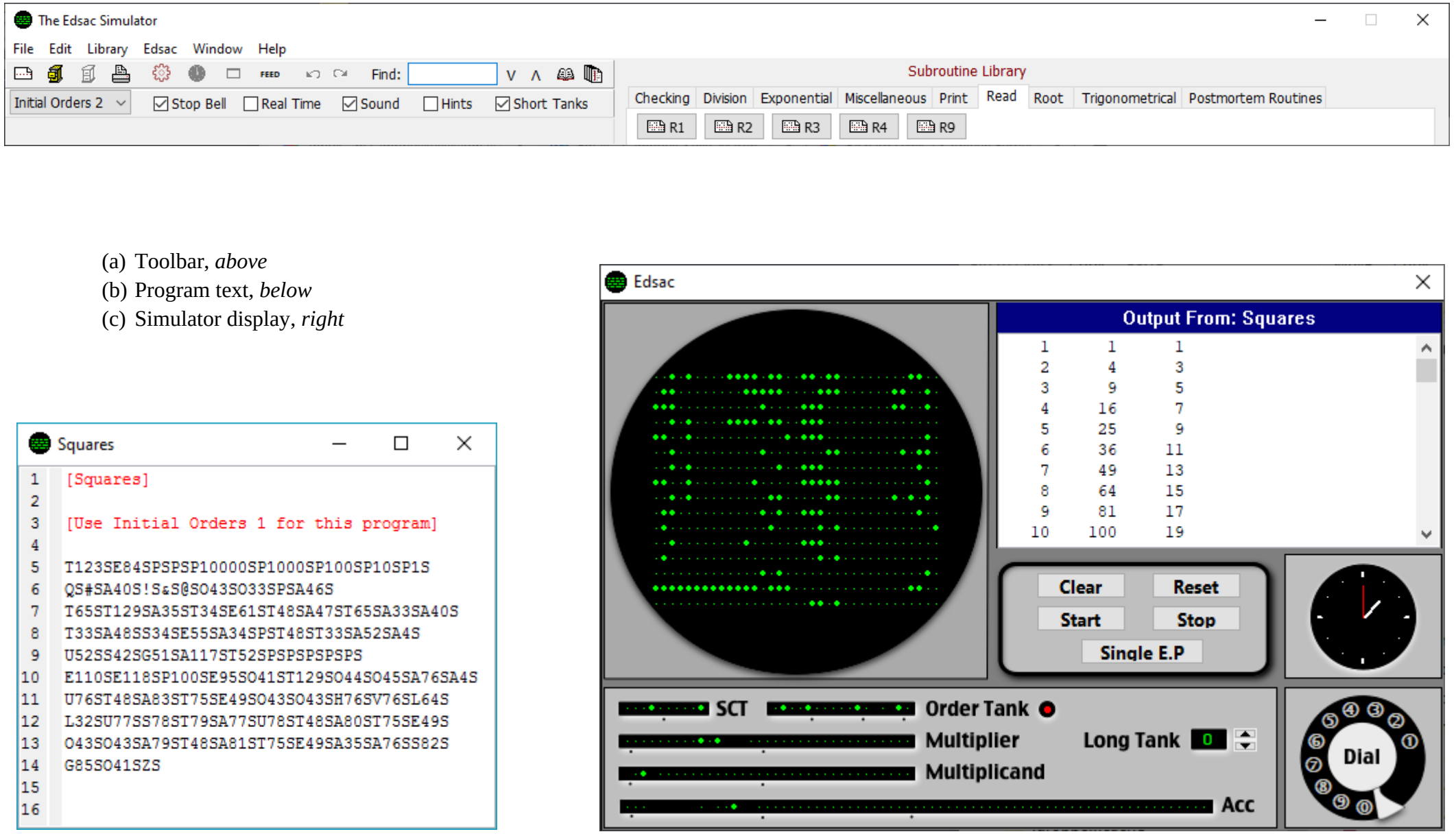
the delay lines were much shorter as they stored only a few bits of information. The two types of delay line were therefore known as long and short tanks. A useful feature of this early serial memory technology was that it was possible to display the contents of the store on Cathode Ray Tube (CRT) monitors. Three of the EDSAC's monitor tubes can be seen in the general view at the back of the photograph, and towards the right; a much better photograph is shown in Fig. 2c. The left monitor tube in this photograph shows the contents of the counter (a kind of internal clock). The right monitor shows the Sequence Control Tank (which contains the address of the current instruction). The centre monitor shows the 32 words in a long tank—just one of the main memory tanks could be displayed at any time, as determined by a rotary switch. Three further monitor tubes, which are not shown here, displayed the other processor registers—the Order Tank (which held the current instruction), the Accumulator, and two multiplication registers. The monitor tubes were a very important way of observing the progress of a program and debugging it—although this was time consuming, so that software debugging aids were soon invented (of which more in Section 4).

The EDSAC was controlled by five push buttons: Start, Stop, Clear, Reset, and Single E.P. The first three are self-explanatory. Reset restarted the machine after a stop operation. The Single E.P. button caused a single instruction to be obeyed, which enabled a program to be stepped through one instruction at a time.

The EDSAC was a research machine rather than a production model, so it tended to be enhanced from time to time. For example, initially only 512 words of memory were provided; this gradually built up to 1024 words as all 32 long tanks were got working. The Clear button was another early addition—at first, the memory had to be cleared by earthing the electrical terminals with a wet finger! Another improvement was the addition of a rotary dial which enabled a single decimal digit to be input by the machine operator. The version of EDSAC provided by the simulator corresponds to the machine which existed during 1949-1951, and it is compatible with all the software developed during that period.

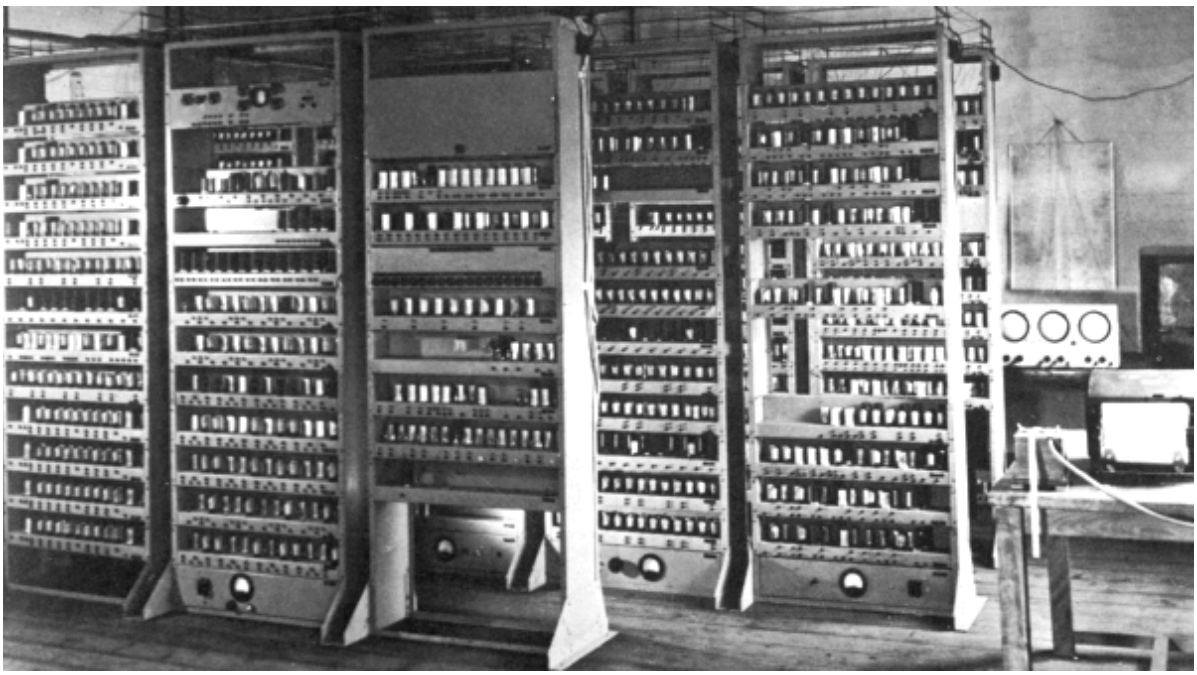
Now we can get back to the Edsac simulator, shown as the main window in Fig. 1. The upper-left of the simulator display represents the main-memory monitor tube. In this display, a binary "one" is represented by a bright spot and a "zero" by a single pixel—the appearance of this display conforms quite closely to that of the original machine. The panel at the lower-left of the display shows in a slightly stylized form the five registers, or short tanks, that were useful to programmers: the Sequence Control Tank, the Order Tank, the Multiplier and Multiplicand registers, and the Accumulator. Next to these, is an indicator light which turns on when a Stop instruction has been obeyed by a program. The control labelled Long Tank is used to select the memory tank displayed on the monitor tube.

At the lower-right is the telephone dial input. Immediately above this is a clock which shows the time in minutes and seconds that Edsac has been running—not necessarily in real time, but the time that the original EDSAC would have taken to do the identical computation. The clock can be used to time programs; and the speed that the hands sweep around the face gives a good feel for the degree to which time has been speeded up or slowed down by the simulator.

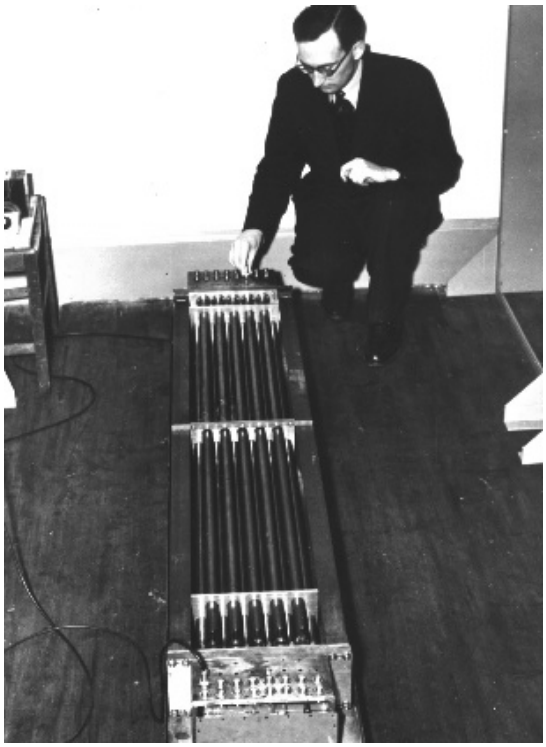


(a) Toolbar, *above*
(b) Program text, *below*
(c) Simulator display, *right*

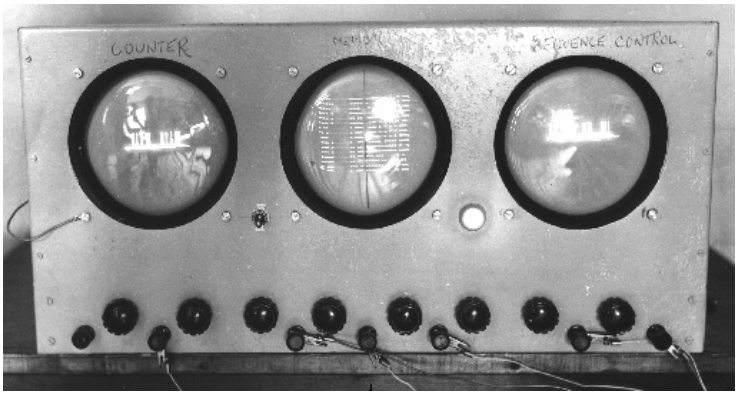
Fig 1. The Edsac Simulator



(a) A photograph of the EDSAC taken shortly after its completion in May 1949. The left three-quarters of the picture shows the main racks of the arithmetic unit, control and memory. The input-output equipment (a paper-tape reader and teleprinter) can be seen on the table towards the right. Three of the monitor tubes can be seen to the rear and right of the picture. The EDSAC operated at a speed of approximately 600 operations per second.



(b) Mercury delay lines or "long tanks" for the main memory, with M.V. Wilkes looking on. The battery of 16 tanks shown here had a capacity of 512 words - the equivalent of a little over 1 Kilobyte.



(c) EDSAC monitor tubes showing: *left*, the Counter; *centre*, the 32 words in a long tank; *right*, the Sequence Control Register.

Fig. 2 The EDSAC environment

On a personal computer the simulator will normally run significantly faster than the original EDSAC. To run the simulator at the original speed, the Real Time option on the Edsac menu or toolbar should be checked. If this item is left unchecked, the simulator will run as fast as the hardware permits. Generating the bit-by-bit display produces a massive computational overhead, so Edsac can be made to run faster by turning off the register displays by unchecking the Short Tanks checkbox on the toolbar. (Note that the Long Tank display is updated relatively infrequently compared with the registers, so that it will not normally significantly affect the performance of the simulator.)

The teleprinter output produced during the running of a program is shown in the text window at the upper-right of the display. Although only the last few lines printed are visible in the window, when the simulator is not running the scroll bar can be used to examine the full output produced. The FEED button on the toolbar can be used to advance paper manually, a line at a time.

Finally, in the very centre of the display are the five main control buttons of the EDSAC: Start, Stop, Clear, Reset, and Single E.P.

1.2 The June 1949 Programs

In June 1949, the EDSAC was demonstrated in public for the first time to the delegates of a conference on *High-Speed Automatic Calculating Machines* organized by the Cambridge University Mathematical Laboratory. For the purpose of this demonstration two programs were run: one printed a table of squares and first differences, and the other printed a table of prime numbers. It should be emphasized that these programs—like most of the routines supplied with the Edsac simulator—have not been rewritten, but are historical artifacts. They have been sitting in the original conference proceedings since 1949, only awaiting a simulator or a replica machine to bring them back to life. To get familiar with the simulator we will run the Squares program now; you can explore the Primes program later.

The Squares and Primes programs used a loading program known as Initial Orders 1—this was a short routine that read the user's program from paper tape and placed it in the main memory. To select these initial orders, chose "Initial Orders 1" from the Edsac menu, or directly from the toolbar. Next, ensure that "Real Time" is checked on the Edsac menu or toolbar. Open the Squares program, either by selecting Open... from the File menu, or using the filing cabinet icon on the toolbar. You will find the Squares program in the Demonstration Programs folder in the Edsac Tapes folder. Note that when the program has been selected, its name is displayed in the title bar of the output window of the simulator confirming your choice.

Press Clear. Ensure Long Tank number 0 is selected. Press the Start button. You will now see the Initial Orders being loaded into words 0-31—this takes about 5 seconds. The display will come to life as the instructions of the Squares program are read in.

Now, use the Long Tank control to display tank number 1. You will see the instructions of the Squares program being loaded one-by-one into locations 32 upwards. When tank 1 is full, look at tank 2 filling, and so on. Also, have a look at tank 1 again, and observe the data words in the main memory being changed.

Eventually, the Squares program will have been completely loaded and will start printing out a table of squares and come to a stop (or you can press the Stop button when you have seen enough).

We will now examine the Squares program. It should look the same as Fig. 3b. You can have as many text windows open as you like, each one of which will correspond to a program "tape". Of course only one tape can be mounted on the Edsac tape-reader at any time, as indicated by the program name in the title bar of the Edsac output window; this will normally be the front-most text window—if you want to change tapes you can do this by bringing the appropriate window to the front.

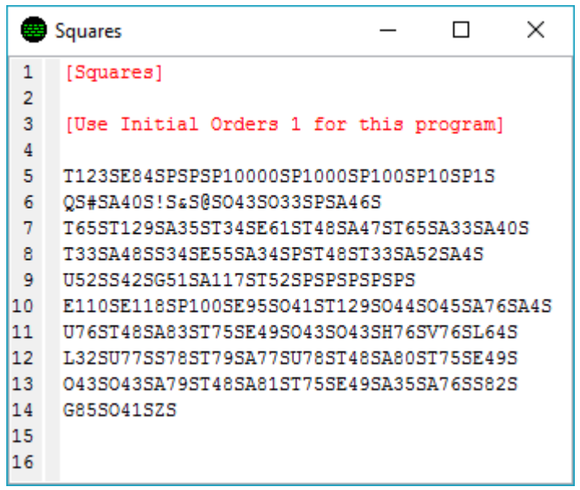
Fig. 3a shows the manuscript for the Squares program, which includes comments and layout characters. (You will find manuscripts corresponding to most of the demonstration programs and library subroutines in the *Program Documentation*—see below.) On the program tape comments were omitted, and no layout characters whatever were used. This meant that tapes were physically very short; for example, the Squares program would have been only about a metre long, with a few inches of leader tape at either end. On the simulator, new lines and spaces are ignored and can be used freely to layout programs—this is advisable even though it is not quite authentic. In addition comments, which are completely ignored by the simulator, can be inserted in the program text in square brackets.

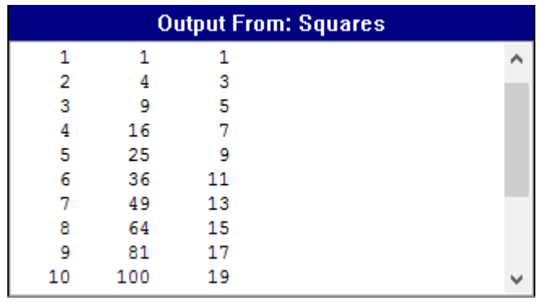
1.3 The Simulator Environment

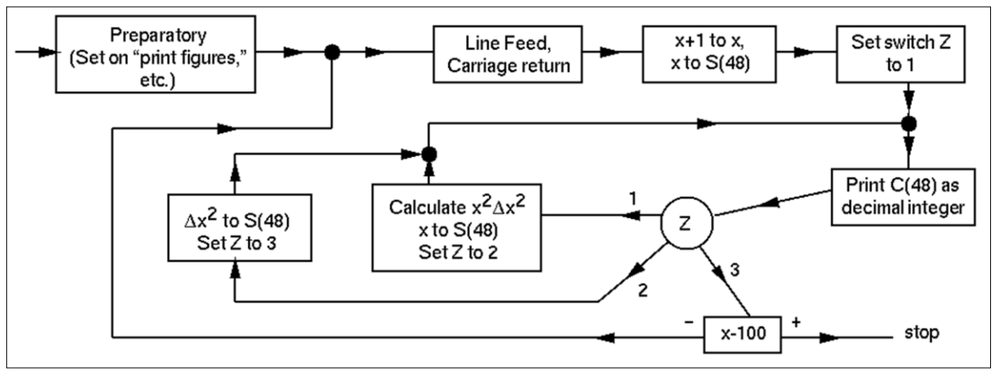
The simulator environment includes an integral text editor so that you can create, amend, and print program files. You can edit programs using the usual cut-and-paste conventions and there is also a search facility.

You can print the teleprinter output by choosing "Print Edsac Output" from the Edsac menu, or you can save it permanently by selecting "Save Edsac Output As...". (There is also a "Literal Output" choice on the File menu. This produces Edsac output as if it had been punched onto paper tape rather than printed on the teleprinter. This is useful if you want the output from one program to become the input of another.)

There are additional controls for the simulator on the Edsac menu and toolbar. You can clear the output window with the Discard Edsac Output command, zeroize the clock, and select the Initial Orders you wish to use. You can turn the sound on or off according to taste. If you wish to set any of these options by default, choose the Edsac|Options menu item or the gearwheel icon on the toolbar. The Hints option enables you to display the contents of a store word or a register as a symbolic instruction or a decimal number by pointing at it on the simulator display. (Of course, there was no such feature on the original EDSAC, but this facility will put you on a par with contemporary programmers who developed over time the skill of reading binary numbers straight from the monitors.) All of the simulator controls can be accessed from the Edsac menu or by using a keystroke equivalent.







(a) Program manuscript, *right* (b) Program tape, *above top*

(c) Printout, *above middle* (d) Flow-diagram, *above*

Fig. 3 Squares program (June 1949)

PRINT SQUARES

	31	T	123	S	As required by initial input
enter →	32	E	84	S	Jump to 84
	33	P		S	Used to keep count of subtractions
	34	P		S	Power of 10 being subtracted
	35	P	10000	S	For use in the decimal binary conversion
	36	P	1000	S	
	37	P	100	S	
	38	P	10	S	
	39	P	1	S	
	40	Q		S	
	41	π		S	Figures
	42	A	40	S	
	43	ϕ		S	Space
	44	Δ		S	Line feed
	45	θ		S	Carriage return
	46	O	43	S	
	47	O	33	S	
	48	P		S	Becomes number to be printed
94 →	49	A	46	S	Put O 43 S in 65S
	50	T	65	S	
72 →	51	T	129	S	Clear 129S
	52	(A	35	S)	Put power of 10 in 34S
	53	T	34	S	
	54	E	61	S	Jump to 61
63 →	55	T	48	S	
	56	A	47	S	
	57	T	65	S	To control printing
	58	A	33	S	
	59	A	40	S	
	60	T	33	S	
54 →	61	A	48	S	
	62	S	34	S	Print contents of 48S
	63	E	55	S	
	64	A	34	S	
	65	P		S	
	66	T	48	S	
	67	T	33	S	
	68	A	52	S	
	69	A	4	S	
	70	U	52	S	
	71	S	42	S	
	72	G	51	S	
	73	A	117	S	
	74	T	52	S	
	75	(P		S)	End print [link]

	76	P		S	Becomes x
	77	P		S	Becomes x^2
	78	P		S	Becomes x^2
	79	P		S	Becomes Δx^2
	80	E	110	S	
	81	E	118	S	
	82	P	100	S	
	83	E	95	S	
32 →	84	O	41	S	Set on print figures
120 →	85	T	129	S	Clear 129S
	86	O	44	S	
	87	O	45	S	
	88	A	76	S	x+1 to 76S and 48S
	89	A	4	S	
	90	U	76	S	
	91	T	48	S	
	92	A	83	S	Set switch Z
	93	T	75	S	
	94	E	49	S	
1: 75 →	95	O	43	S	Double space
	96	O	43	S	
	97	H	76	S	$x^2 \cdot 2^{15}$ to 77S
	98	V	76	S	
	99	L	64	S	
	100	L	32	S	
	101	U	77	S	
	102	S	78	S	Δx^2 to 79S
	103	T	79	S	
	104	A	77	S	x^2 to 48S and print
	105	U	78	S	
	106	T	48	S	
	107	A	80	S	
	108	T	5	S	
	109	E	49	S	
2: 75→	110	O	43	S	Double space
	111	O	43	S	
	112	A	79	S	Δx^2 to 48S and print
	113	T	48	S	
	114	A	81	S	
	115	T	75	S	
	116	E	49	S	
	117	A	35	S	
3: 75→	118	A	76	S	Test for finish:
	119	S	82	S	
	120	G	85	S	
	121	O	41	S	
	122	Z		S	

You can use the Edsac|Options to change the size of simulator display, by rescaling it to be smaller or larger. You can also change to font size. Clicking the Save button will preserve your settings between sessions.

The Library menu and the tabbed control at the right of the toolbar give access to the subroutine library—this will be covered in full in sections 4 and 5.

Documentation for the Edsac simulator is supplied as two pdf files: the *Tutorial Guide* (EdsacTG.pdf) and *Program Documentation* (EdsacDoc.pdf). The pdf files are stored in the Resources folder or they can be accessed directly from the Help menu or the toolbar.

The *Tutorial Guide* can be read online or printed. The document is designed as a doubled-sided printed booklet that opens flat, for study at or away from your computer. Pages in the *Tutorial Guide* use a mixture of paper sizes and orientations, so select the "Shrink oversized pages" and "Select Auto portrait/landscape" print options. The pages in the *Program Documentation* pdf file are, for the most part, exact transcriptions of the original "programme sheets" now preserved in the Cambridge University archives.

Finally, inside the Resources folder the document WWG1951.pdf is a copy of the classic 1951 textbook on programming the EDSAC: *The Preparation of Program for an Electronic Digital Computer*, of which more in chapter 3.

Exercises

1 Edit the Squares program so that it prints out the squares of 1 to 10 instead of 1 to 100. (*Hint:* Change the constant P 100 S in location 82 to P 10 S.)

2 Load the Primes program. Run the program and note how the output slows down as successively larger integers are tested for primality.

2 EDSAC ARCHITECTURE AND ARITHMETIC

The demonstration programs used in this section, and in the rest of the Tutorial Guide, make use of the second form of the initial orders which were introduced in September 1949. These replaced the much less sophisticated Initial Orders 1 which were only in service for about three months. Choose “Initial Orders 2” from the Edsac menu and close any text windows that are open.

2.1 Architecture and Instruction Set

One of the nicer features of the EDSAC is that it is conceptually a very simple machine. The reason for this simplicity is that when Wilkes and his team were designing the machine, they chose to keep things as simple as possible: this was partly to minimize the engineering difficulty, but also so that they could start developing programs for a real computer as soon as possible, instead of just dreaming them up for an imaginary machine. Besides the EDSAC’s historical importance, its simple design makes the machine a worthwhile one to study today as a particularly clean example of what has come to be called the “von Neumann architecture”. (Although, of course, like all real machines, the EDSAC does have some annoying features that one wishes were not there.)

The original design of the EDSAC was based on that of the EDVAC, the computer designed during 1944-45 at the Moore School of Electrical Engineering, University of Pennsylvania, by a group that included John von Neumann, J. Presper Eckert and John W. Mauchly. The design of the EDVAC was described in von Neumann’s classic *First Draft of a Report on the EDVAC* in June 1945. This is the foundation on which almost all computers have been based for the last seventy years. The EDSAC consisted of the classical arrangement of five functional parts: a control unit, an arithmetic-logic unit (ALU), a memory (or store), input, and output (Fig. 4). The combined control unit and ALU is now usually known as the processor. In the EDSAC processor there were five principal registers: the Sequence Control Tank, the Order Tank, the Multiplicand and Multiplier registers, and the Accumulator. The

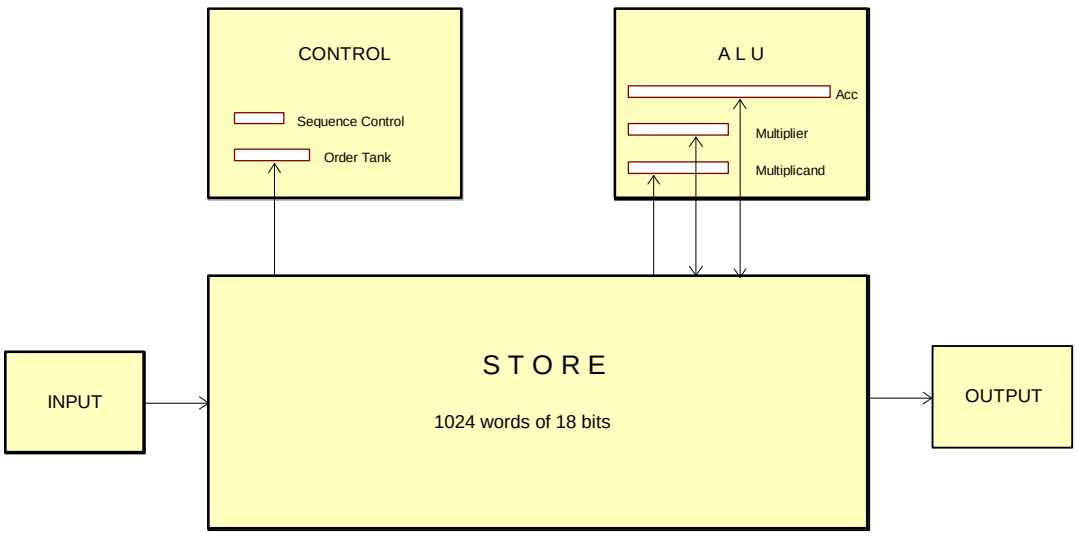


Fig. 4 EDSAC architecture

Sequence Control Tank contains the address of the current instruction and the Order Tank stores the current instruction.

The EDSAC used a single-address instruction format, shown in Fig. 5. Although the EDSAC was based on an 18-bit word, only 17 bits were used, the leading bit being unusable for reasons connected with circuit set-up time. The opcode (or "function") was specified in 5 bits, and the address in 10 bits. A further bit specified the operand length: most instructions could operate on either a 17-bit short word, or a 35-bit long word; the length indicator specified which. (If you look carefully at the register panel on the Edsac display, you will notice some black dots beneath the Order Tank: these indicate the different fields of the instruction.)

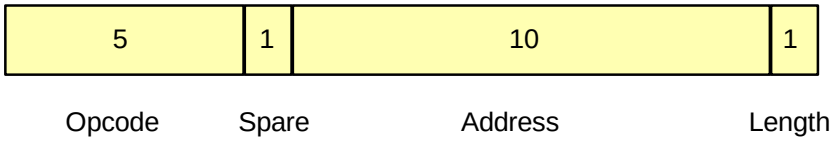


Fig. 5 Instruction format

Table 1 (Appendix, p. 41) shows the EDSAC instruction set as it existed in 1949. Operations were represented by letters of the alphabet, some of which suggested the function they denoted (eg. A for Add, S for Subtract, etc). The binary representation of the opcode was in fact the same as the character code of the corresponding character - see Table 2; this simplified the translation of the symbolic program considerably. Average instruction times were 1.5 ms, although multiplication was longer and took 6 ms; input-output times were determined by the speeds of the peripheral equipment.

Instructions were always written in a symbolic form such as A 56 F, or S 128 D; these meant respectively, "Add the short number in location 56 into the accumulator", and "Subtract the long number in location 128 from the accumulator". Note the use of the length indicators F or D to specify a short or long operand in the instruction. Also, when an address was zero it was omitted altogether; for example, T F meant "Store the short number in the accumulator in location 0".

2.2 Numbers and Arithmetic

In this section we will examine the details of number storage and arithmetic on the EDSAC. It is not important that you follow everything in the description that follows - you can always come back later if you need to.

Modern software systems tend to shield the user from the arithmetic instructions of a computer, and often from the format in which numbers are stored - other than the basic word length and the type of data (integer, floating-point, etc). However, on the EDSAC it was necessary to have an understanding of the formats of numbers, and the instructions which operated on them. An additional complication was that floating point numbers were not used; instead, real numbers were stored as fractions in the range $-1 \leq x < 1$. (If a real number of modulus greater than unity was needed, then scaling had to be used - more on this later.)

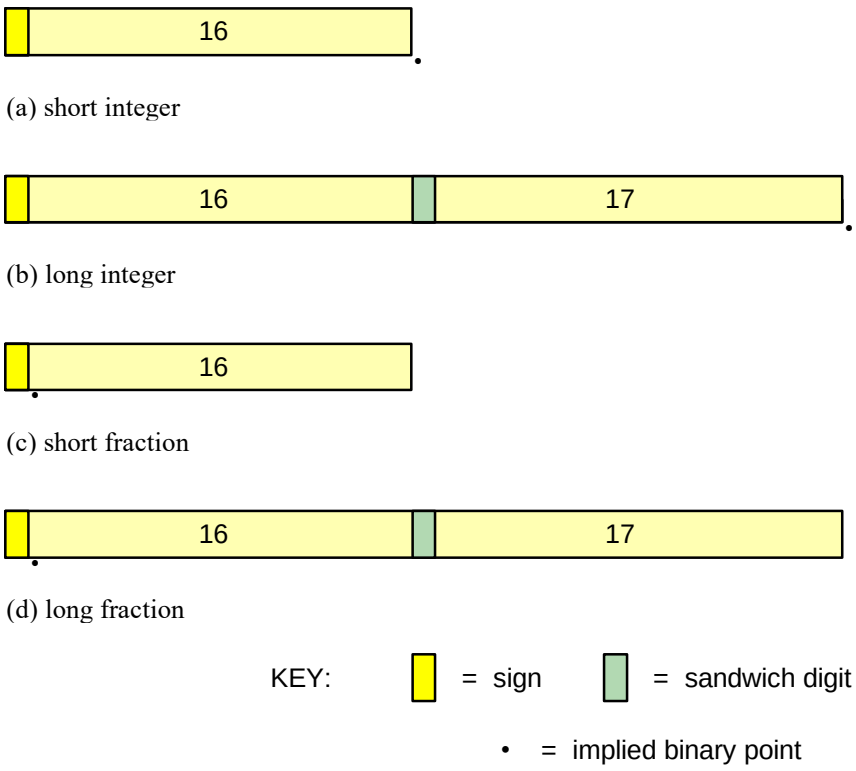


Fig. 6 Number formats

Fig. 6 shows the four number formats used in the EDSAC: short and long integers, and short and long fractions. Short numbers were 17-bits in length and long numbers were 35-bits. (Remember that the basic word length of the EDSAC was 18 bits, but the first bit was never used.) Within the processor, the multiplication registers each had a capacity of 35 bits; and the accumulator had a capacity of 71 bits - sufficient to develop the full product of a pair of long numbers. When using short numbers only the leftmost half of the registers would be used. (The black dots beneath the arithmetic register displays indicate the boundaries of short and long numbers and their signs.)

The Arithmetic program (Fig. 7) does not do anything useful, but it is designed to illustrate EDSAC number storage and arithmetic. Open... the program from the Tutorial Guide Examples folder; press Clear and then Start to load the program into the store. The first instruction of the program (in location 64) is a stop-order, so that when the program has been loaded, you can turn on the registers display, and then work through it by single stepping using the Single E.P. button. (If the program failed to load, check that you selected Initial Orders 2.) Points to note in the program are as follows.

Integers were normally stored in a 17-bit word, in twos-complement form, with the leftmost bit for the sign, and the implied binary point at the rightmost end. Thus:

```
 33 = 00000000000100001
-17 = 11111111111101111
```

Although it was possible to have long integers, these will not be used here.

Short fractions were stored in a 17-bit word, in twos-complement form, with the leftmost bit for the sign, and the implied binary point between the sign bit and the most significant numerical bit. Thus:

$$0.1875 = {}^{3}/_{16} = 00011000000000000$$
$$-0.5 = -{}^{1}/_{2} = 11000000000000000$$

Easy constants like the above were set up in programs by symbolic orders which caused the appropriate bit pattern to be assembled (there were no constant defining operations). For example: P 16 D for the integer 33, and E F for the fraction $^{3}/_{16}$.

Addition and subtraction work exactly as you would expect, except for overflow. Overflow in the accumulator is not detected, and the program will just go on running, working with whatever number the accumulator happens to contain. Multiplication is more complicated. The multiplier was designed to give the correct product with fractions. Thus the product of two short 17-bit fractions is a long 35-bit fraction. Depending on the precision required, either the top 17 bits or the top 35 bits of the accumulator are stored (using a T n F or a T n D order respectively). When integers are multiplied, the multiplier behaves in the same way it would with fractions. For example, since the integer 5 (say) is equivalent to the fraction 5×2^{-16}, the product of 5 x 5 would be 25×2^{-32}. Hence to obtain the result in the correct place in the accumulator, it would have to be left-shifted 16 places.

In the memory, long numbers are stored in an adjacent pair of odd-even locations. The word length is 35 bits, not 34 bits: the extra bit between the two half-words is the so-called “sandwich digit”, which caused some confusion with EDSAC users, but the existence of the subroutine library meant that most of the time people did not need to trouble about it. Long constants are set up by a pair of orders, such as H 682 D, T 682 D for $^{1}/_{3}$. These constants were messy to work out, and useful ones were published from time to time in the EDSAC Programming Bulletins. A selection of useful constants is given in Table 4 of the Appendix. When referring to a storage location, the notation 24D (say) means the long number in locations 25 and 24. Similarly the notation nF means the short number in location n.

Exercise

1 Reload the Arithmetic program by pressing Clear and then Start. Display long tank 3 (locations 96-127) and turn on the Hints control. Verify the values of the constants stored in locations 96 upwards.

2 Single step through the Arithmetic program, observing the contents of the accumulator as each instruction is obeyed and comparing it with the program listing in Fig. 7. This will familiarise you with the various number formats used in the EDSAC.

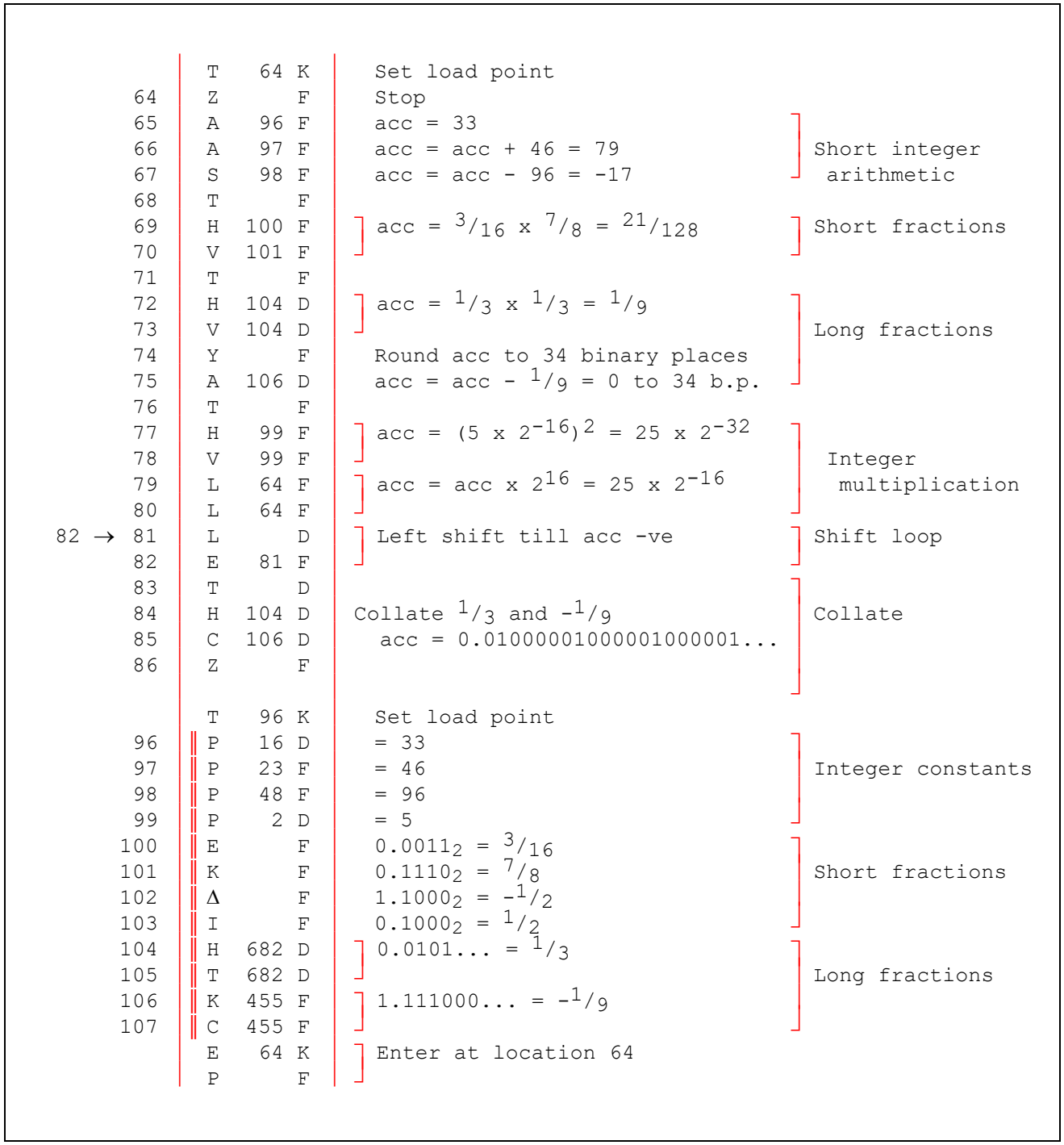


Fig. 7 Arithmetic program

2.3 Miscellaneous

The EDSAC instruction set in Table 1 (p. 41) is fairly self-evident to anyone with a reasonable computer understanding, but a few pointers may be in order.

One of the compromises made to keep the EDSAC simple was to have only two branch instructions, the E- and G-orders. There was no unconditional branch instruction, so that it was always necessary to know the sign of the accumulator when taking a branch (or else to use both an E-order *and* a G-order). The same limitation meant that it took 8 instructions (!) to determine the equality of two numbers - so this was avoided if at all possible. In 1952, an unconditional branch order was added to overcome these problems. (Unfortunately this change also meant that many programs and library subroutines had to be rewritten - this happened whenever the instruction set was significantly changed. This is why it was earlier stated that the simulator models the EDSAC as it existed in 1949-51.)

Another economy in the EDSAC was that it had no hardware divider. Hence division had to be done by a subroutine (see the Reciprocals program in Section 3.3, for an example). The logic operations on the EDSAC were particularly sparse. Only logical AND (the Collate order) was provided. Likewise, there were no instructions for character handling. This is really a reflection of the fact that machines of the EDSAC's era were designed as "mathematical instruments". It was only in the late 1950s that powerful logic and character-handling instructions became available on most computers.

The shift instructions probably gave more trouble to users than any others. This was because, to simplify the engineering, the number of shift positions was given not by the *value* of the address field of the instruction but by the *position* of the rightmost bit in the instruction word. Thus the instruction L 8 F caused the contents of the accumulator to be shifted 5 places left, and not 8 places, as you might expect.

Finally, an interesting feature of the EDSAC and many of its contemporaries was that they had no index registers - not least because the index register was not invented until 1950, and then the idea took a little time to catch on. To perform arithmetic on the elements of an array on the EDSAC it was necessary for a program to modify the addresses in its own instructions, so that in an instruction loop successive elements of the array would be accessed. The ability of an electronic computer to modify its own instructions was one of the key features of the stored-program concept, although we now tend to frown on such "impure" code.

Exercises

1 The instructions R D and L D shift the accumulator one place right and one place left respectively. The instructions R F and L F cause a right shift of 15 places and a left shift of 13 places, respectively. Why?

3 PROGRAMMING THE EDSAC

In this section we will examine three programs which will progressively illustrate the important features of programming for the EDSAC. It is recommended that you "punch" and run the first two programs so that you get familiar with using the system before attempting to write the programs in Section 5. Working copies of the programs are provided in the Demonstration Programs folder in case you get stuck.

3.1 Hello World

This is not exactly an original idea, but as a confidence builder, our first example is a tiny program to print a message. However, as printing "Hello World" would make the program rather longer than necessary, the program will just type "HI". The complete program is shown in Fig. 8.

Fig. 8a shows the program text. In the program, there are two types of entity: actual machine instructions, which are numbered 0 to 7; and "control combinations" at the beginning and end of the program. Control combinations correspond to what we would now call "assembly directives": they are pseudo-instructions for the Initial Orders so that they can load the program and enter it.

This is the place to say a little more about Initial Orders 2. Once the Cambridge group began programming using the first form of the initial orders in the spring of 1949, their limitations soon became apparent. The worst feature by far was that addresses in instructions had to be coded in absolute form: this meant, for example, that if an extra instruction had to be inserted in a program then the addresses in many of the branch instructions would need to be altered. This made program debugging very tedious. Another problem was that the lack of a relocation facility meant it was difficult to organize a subroutine library effectively.

The task of devising a new set of initial orders was given by Wilkes to David Wheeler, then a research student and later Professor of Computer Science at Cambridge. What he produced was the forerunner of the modern assembler. The new programming system was later described in the classic textbook *The Preparation of Programs for an Electronic Digital Computer* (Wilkes, Wheeler and Gill, 1951). This famous book - usually known as *Wilkes, Wheeler and Gill*, often abbreviated as "WWG" - established the programming culture of the early 1950s, which is still to some extent embodied in the assembly systems and subroutine libraries of today's computers. When designing the new initial orders, one of the constraints that Wheeler had was that, for engineering reasons, the initial orders were limited to being just 42 instructions long. But even so, their power was quite astonishing and at the time they were justly celebrated as "the leading example of programming virtuosity". (If you are interested, you can find copies of the program manuscripts for both Initial Orders 1 and Initial Orders 2 in the *Program Documentation.*)

In the *Tutorial Guide* we will by no means exhaust the possibilities of Initial Orders 2, which are fully described in *Wilkes, Wheeler and Gill*; we will just use the basic control combinations below:

T m K	Set the load point to m
G K	Set the θ-parameter to the current load point
T Z	Restore the θ-parameter
E m K P F	Enter the program at location m
E Z P F	Enter the program at location θ

The Hello World program in Fig. 8a uses three of these control combination. The program begins with T 64 K, which causes instructions to be loaded into location 64 upwards. (This would correspond to something like "ORG 64" in a modern assembler. It is true that the latter is a more helpful notation than T 64 K, but this shortcoming was entirely due to the space constraints of the initial orders.) On the next line, G K sets the θ-parameter - which is used for relocation - to the current load point of 64. From this point on, the address in any instruction with the code-letter θ will have the value of the θ-parameter (ie. 64) added to it. This is how relocation is achieved. Finally, the last control combination E Z P F causes the program to be entered at location θ (ie. 64). Note how the program is completely relocatable: just changing the number 64 in the first line of the program will allow it to be placed anywhere in the store.

Let's now turn to the instructions themselves. Notice that the first instruction is a stop-order; and that the program has been located from word 64 onwards, ie. at the edge of a memory tank boundary. This was a common practice when developing programs so that it was possible to check visually in the monitor tube that the program had loaded correctly before running it; locating the program on a page boundary made it easy to find (by comparison, word 56, say, was quite difficult to locate on the monitor, other than by counting up the rows of the display).

Fig. 8b shows the program "tape" exactly as it should be typed - no comments or extraneous characters other than white space characters. The EDSAC tape punch used four Greek characters: theta, phi, delta, and pi. These characters are typed as below:

EDSAC Character		*Type As*
Theta	θ	@
Phi	ϕ	!
Delta	Δ	&
Pi	π	#

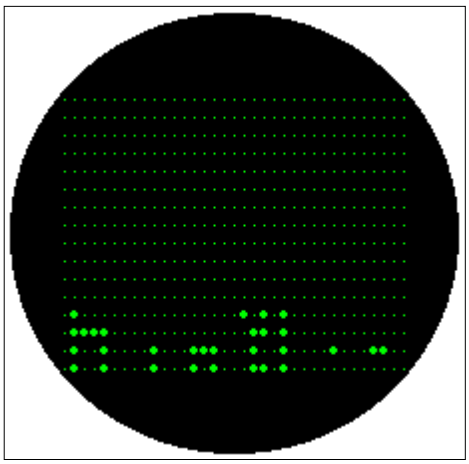
We will now type the Hello World program. First, select New from the File menu or the toolbar to create a new text window. Type the program, exactly as in Fig. 8b. If you wish, you can save the program with a name such as "My_Hello". You can now run the program. Turn on the Short Tanks display, and press Clear followed by Start. After a second or two the simulator will stop - ringing the warning bell as it does so. Examine Long Tank 2 to verify that the program has loaded correctly. It should look exactly as in Fig. 8c. Press Reset. The program should print "HI", and then stop - again ringing the warning bell.

```
                T    64 K    Load from location 64
                G       K    Set θ parameter
Start → 0       Z       F    Stop
        1       O     5 θ    Letter shift
        2       O     6 θ    Print "H"
        3       O     7 θ    Print "I"
        4       Z       F    Stop
        5       *       F    Letters
        6       H       F    "H"
        7       I       F    "I"
                E       Z    Enter at location 0θ
                P       F
```

(a) Program text

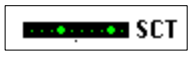
```
T64K
GK
ZF
O5@
O6@
O7@
ZF
*F
HF
IF
EZPF
```

(b) Program tape

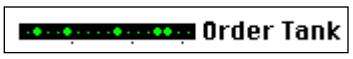


loc'n	order	loc'n	order
71	I F	70	H F
69	* F	68	Z F
67	O 71 F	66	O 71 F
65	O 69 F	64	Z F

(c) Long Tank 2



= 00010.010_2
= tank2.word2 = 66_{10}



= 01001.00001000110.0
= O 70 F

(d) Sequence Control Register and Order Tank

Fig. 8 Hello World program

If your program fails to load correctly, and you get the alert "End of input tape encountered", check that you have selected Initial Orders 2 on the Edsac menu. If your program fails to run, it is probably because you mis-typed something. EDSAC was very unforgiving of typos - particularly Ohs punched as Zeroes, so check very carefully. If you still can't get the program to run, there is a working version in the Demonstration Programs folder.

Press Clear and Start to reload the program. Now, instead of clicking Reset, press Single E.P. repeatedly to step through the program one instruction at a time. Notice (Fig. 8d) how the Sequence Control Tank (SCT) steps through 64, 65, 66, ... The last instruction of the program is in 68. Of course if you carry on pressing Single E.P. the machine will execute nonsense instructions - but since the EDSAC was designed so that non-existent opcodes behaved as stop instructions, nothing very exciting usually happens. Note that only legal stop instructions (using the Z-order) ring the bell.

Exercises

1 Modify the Hello World program so that it is loaded into location 56 upwards, and verify on the monitor tube. If you try to load the program from 32 upwards, strange things happen. Why is this?

2 Modify the Hello World program so that it does indeed print "HELLO WORLD".

3.2 Cubes

The next program is one that calculates and prints the cubes of the integers using the well known formula of Nichomachus:

$1^3 = 1$
$2^3 = 3 + 5$
$3^3 = 7 + 9 + 11$
$4^3 = 13 + 15 + 17 + 19$
etc.

The coding to compute the cubes themselves is fairly trivial; the difficulty lies in actually printing them out. The easy way to do this is to use the library subroutine P6, which prints a short positive integer (Fig. 9).

The EDSAC subroutine library began to take shape from autumn 1949 onwards. Subroutines were classified by a letter indicating the group to which they belonged (eg. D for division, P for printing, etc.) Within a group, subroutines were given a serial number (eg. P1, P2, P3, etc.), which mainly indicated the chronological order in which routines had been placed in the library. Eventually the library grew to contain nearly a hundred subroutines. However, only about two dozen are provided for the Edsac simulator - they are kept in the Subroutine Library folder, and brief specifications for the more popular ones are given in Table 3 of the Appendix.

P6 Print short positive integer.

Closed; 32 storage locations; working positions 1, 4, and 5; time = about 900 msecs.

Prints $2^{-16} \cdot C(0)$ with suppression of nonsignificant zeros but without layout.

		G	K		
	0	A	3 F	Plant link	
	1	T	25 θ		
	2	H	29 θ	Multiply by $2^{16}/10^5$	
	3	V	F		
	4	T	4 D		
	5	A	3 θ	V F = -1/16 to S(0)	
	6	T	F		
	7	H	30 θ	Set multiplier	
	8	S	6 θ	Set digit count	
24 →	9	T	1 F	Digit count	Digit cycle
	10	V	4 D	Multiply	
	11	U	4 D		
	12	A	F	Test for first	
	13	G	26 θ	non-zero digit	
	14	T	F	Clear Acc. and S(0)*	
	15	T	F		
	16	O	5 F	Print	
	17	A	4 D	Check and remove	
	18	F	4 F		
	19	S	4 F		
28 →	20	L	4 F	Shift	
	21	T	4 D		
	22	A	1 F	Count digits	
	23	S	3 θ		
	24	G	9 θ		
	25	(E	F)	Link	
13 →	26	S	F	Add 1/16	Suppress zero
	27	O	31 θ	Space	
	28	E	20 θ		
	29	5 J	995 F	= $2^{16}/10^5$	
	30	5 J	F	= 10/16	
	31	5 ϕ	F	Space	

* S(0) becomes cleared when the first non-zero digit is encountered, thus preventing the suppression of later zeros.

P6

```
[P6]
GKA3FT25@H29@VFT4DA3@TFH30@S6@T1F
V4DU4DAFG26@TFTFO5FA4DF4FS4F
L4FT4DA1FS3@G9@EFSFO31@E20@J995FJF!F
```

(a) Program text, *above*

(b) Program tape, *left*

Fig. 9 Library subroutine P6

Original documentation for all the subroutines is reproduced in the *Program Documentation* pdf file. These subroutines will suffice for all the examples given in the *Tutorial Guide*, and for most of the programs you are likely to think of. If you decide to explore the EDSAC in more depth, you may need more subroutines; many of these are readily accessible in Part III of *Wilkes, Wheeler and Gill* (1951).

Calling a subroutine on the EDSAC used the technique of the "Wheeler jump", shown below. Here, the instruction A m F loads itself into the accumulator (this will be used to form the return link); and then the instruction G n F transfers control to the first instruction of the subroutine in location n. In the subroutine, the instructions A 3 F and T p F manufacture the return link and plant it as the last instruction of the subroutine in location p. (The instruction A 3 F actually uses a constant permanently kept in location 3 to produce the return link.) If all this went over your head on a first reading, don't worry; it is only really important when you want to write subroutines. If you are just going to use library subroutines, all you need to remember is the A m F, G n F calling sequence.

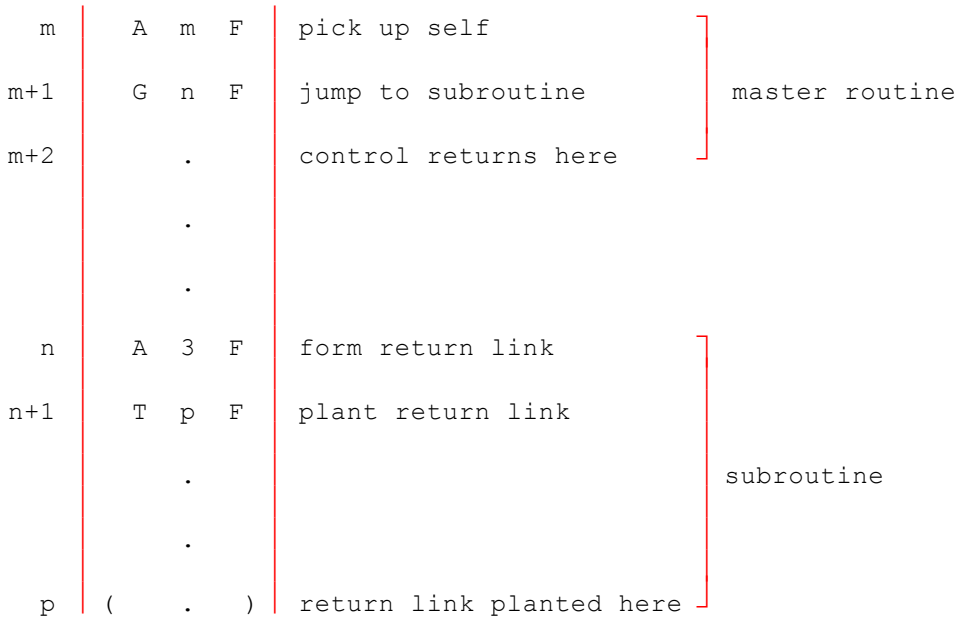


Fig. 10 shows the Cubes program. It consists of two routines: the master routine written by the programmer (Fig. 10a), and the library subroutine P6. The first job is to allocate storage for the program; this is done in Fig. 10b. The convention was to load the program into location 56 upwards, placing all the subroutines and the master routine end-to-end without leaving any gaps. The lengths of subroutines are given in their specifications. Fig. 10c shows the make-up of the complete program tape.

On the original EDSAC, the procedure for punching a program was as follows (Fig. 11). First, the key-punch operator (who was usually the same person as the programmer) would punch the master routine. Then the subroutine library tapes - which were kept in small cardboard boxes in a steel filing cabinet - would be copied onto the program tape, together with the master routine, and interspersed with control combinations. When the subroutine tapes had been copied, they were rewound and returned to the library cabinet.

```
          G      K    Set θ-parameter
Enter→ 0  Z      F    Stop
       1  O   34 θ    Figure shift
 22 →  2  O   35 θ  ]  New line
       3  O   36 θ  ]
       4  T      F  ]
       5  A   27 θ  ]  k to 0F
       6  T      F  ]
       7  A    7 θ  ]  Print 0F using P6
       8  G   56 F  ]
      ---------------
 P6 →  9  T   27 θ    Zero to k
      10  A   28 θ  ]
      11  A   31 θ  ]  n+1 to n
      12  T   28 θ  ]
      13  S   28 θ  ]  -n to count
21 →  14  T   30 θ  ]
      15  A   29 θ  ]
      16  A   32 θ  ]  m+2 to m
      17  U   29 θ  ]
      18  A   27 θ  ]  k+m to k
      19  T   27 θ  ]
      20  A   30 θ  ]
      21  A   31 θ  ]  Increment count
      22  G   14 θ    Jump to 13 if count ≤ 0
      ---------------
      23  A   33 θ  ]  Repeat main cycle
      24  S   28 θ  ]   while n ≤ 10
      25  E    2 θ  ]
      ---------------
      26  Z      F    Stop
      27  P      D    k (n^3; =1 initially)
      28  P      D    n (=1 initially)
      29  P      D    m (=1 initially)
      30  P      F    count
      31  P      D    =1
      32  P    1 F    =2
      33  P    5 F    =10
      34  π      F    figs
      35  θ      F    cr
      36  Δ      F    lf
```

(a) Master routine

Routine	Location of first order	Number of storage locations occupied
P6 (print)	56	32
Master	88	-

(b) Table of routines

space P K

T 56 K

P6

space P Z

Master

E Z P F

(c) Make-up of program tape

```
  1
  8
 27
 64
125
216
343
512
  .
  .
  .
```

(e) Printout

```
[Cubes]
..PK
T56K
[P6]
GKA3FT25@H29@VFT4DA3@TFH30@S6@T1F
V4DU4DAFG26@TFTFO5FA4DF4FS4F
L4FT4DA1FS3@G9@EFSFO31@E20@J995FJF!F
..PZ
[Cubes Master]
GK
ZF
O34@
O35@
O36@
TF
A27@
TF
A7@
G56F
T27@
A28@
A31@
T28@
S28@
T30@
A29@
A32@
U29@
A27@
T27@
A30@
A31@
G14@
A33@
S28@
E2@
ZF
PD
PD
PD
PF
PD
P1F
P5F
#F
@F
&F
EZPF
```

(d) Program tape

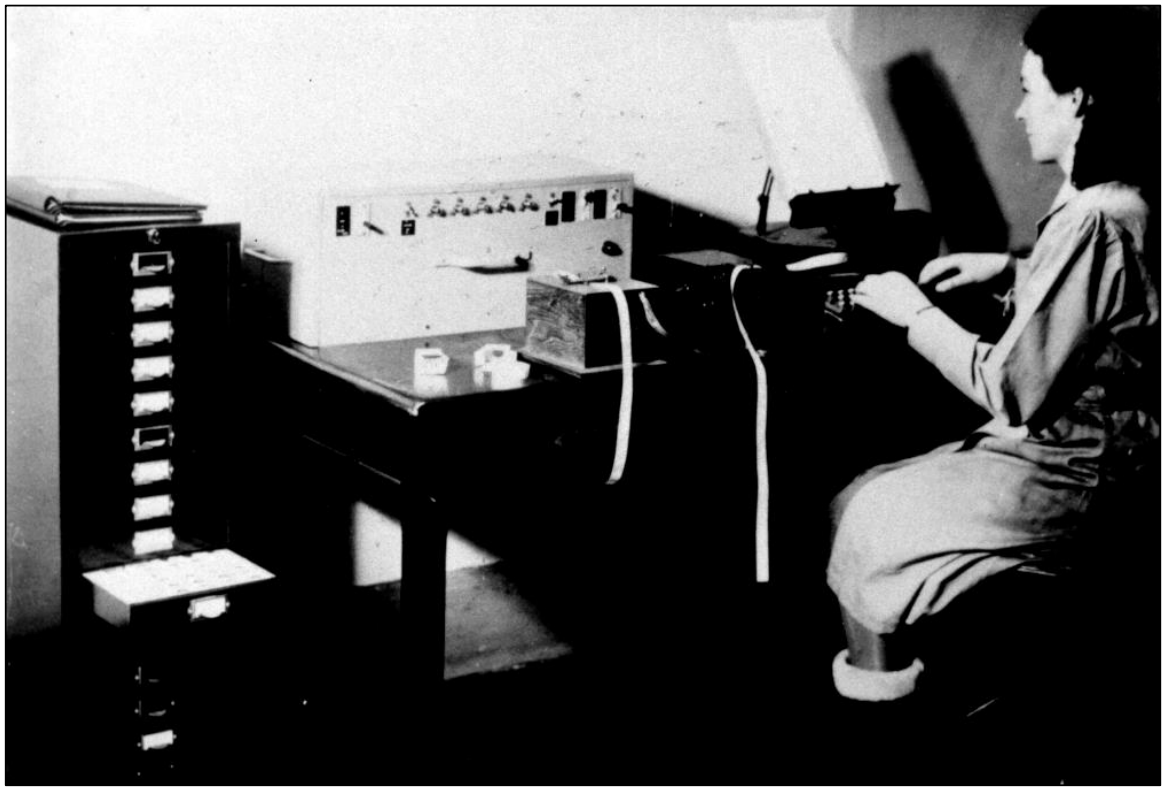
Fig. 10 Cubes program

On the program tape the individual routines were normally separated by a few rows of blank tape; this was useful in spotting how far the program had got if it suddenly stopped loading - the machine operator would mark the tape with a pencil where it had stopped in the paper-tape reader. This blank tape is indicated by “space” in the notation for the make-up of program tapes (Fig. 10c). The blank tape has to be terminated with the control combination P K or P Z.

Much the same logic is used for preparing programs for the simulator. On the Edsac simulator, because an application program such as Cubes is composed from two or more files, and will likely exist in a number of versions, it is advisable to create a new folder for it. A folder for Cubes has already been set up in the Tutorial Guide Examples folder. Normally the first job would be to punch the master routine, but this has also been done for you; it is in the file “Cubes Master” in the Cubes folder.

We now have to create the complete program from the library subroutine and the master routine. First, create a New text window in which to prepare the program. Now, referring to Fig. 10c, we first need to type the control combinations:

space PK
T 56 K



Program tapes for the EDSAC were blind punched using a keyboard perforator. Library subroutines were kept in the steel cabinet (*left*) and were copied mechanically onto the program tape using the tape-reader in the centre of the photograph.

Fig. 11 EDSAC tape preparation facilities

We require at least two rows of blank tape for the "space"; on the Edsac simulator a row of blank tape is represented by a period, so we can represent "space" by "..". The rest of the characters (PKT56K) are typed as written. We then need to copy subroutine P6. Click the "Print" tab on the toolbar, which shows the available print subroutines. Click the button for P6. P6 will now be copied into your program at the current insertion point. Now type the control combination "space P Z". Next, use File|Insert… to locate Cubes Master and copy it at the current insertion point. Finally type the control combination E Z P F. Save the program as "Cubes". (If you prefer you can construct your program by opening windows for the various components and cutting and pasting from one to another. This is messier but the end result will be the same.)

Your program should look exactly as in Fig. 10d - except possibly for white space characters and comments. A few points to note. First, the simulator allows you to put comments in the program between square brackets. The convention adopted is to label all program tapes and library subroutines with their name at the beginning, eg. [P6]. This roughly corresponds to the practice adopted on the original EDSAC of labelling a tape by writing the name of the program on it in pencil. Secondly, notice that the master routine is typed one instruction per line, while the subroutines have been previously typed with ten instructions per line. This convention is adopted in the simulator to keep program listings short - when the program is actually run there is no difference so far as the simulator is concerned. The master routine is typed one order per line to make it easy to correct while it is being debugged; but library subroutines can be assumed to be correct and you should never need to modify them, so they are typed ten orders per line.

You should now be able to run the program. Again, if it fails to run, it may be because you mis-typed something, or perhaps you composed the program incorrectly. Alternatively, perhaps you forgot that the first instruction of the master routine was a stop order and you need to press Reset to make it continue. If you still have problems, there is a working version of the program in the Demonstration Programs folder.

Exercise

1 Modify the Cubes program so that it prints out n followed by n^3 on each line.

3.3 Reciprocals

This section illustrates the remaining important concepts in EDSAC programming: code-letters, subroutine parameters, and scaling. To illustrate these ideas, we will refer to the Reciprocals program which prints the reciprocals of the numbers 2-10 (Fig. 12).

Code-letters

If you did the last exercise, which involved modifying the Cubes program, you will have discovered an awkward problem: Namely that inserting extra instructions in the master routine required you not only to change the addresses in some branch instructions (which you might have expected), but also because the locations of the data and constants changed, the addresses in many of the arithmetic instructions had

to be altered too. This problem can be overcome by the use of code-letters. We have already encountered three code-letters (F, D, and θ), but there are 15 altogether as shown below.

Code-letter	*Location*	*Value*
F	41	0
θ	42	Origin of current routine
D	43	1
ϕ, H, N, M … V	44, 45, 46 … 55	For use by programmer

As the initial orders load each instruction, the value corresponding to its code-letter is added to the instruction before it is placed in the memory. Because the code-letters F and D contain the integers 0 and 1 respectively, this has the effect of setting the length indicator bit. Similarly, the code-letter θ has the effect of adding the origin of the current routine to the address of the instruction - this is how relocation is achieved. You should not normally change F, θ or D directly, for obvious reasons. All the remaining code-letters can be used by the programmer. They occupy locations 41 to 55, and that is why the normal place to begin loading a program is location 56. (The 15 code-letters also correspond to characters 17-31 in the collating sequence - see Table 2 in the Appendix.)

In the master routine of the Reciprocals program, the code-letters θ and M have been used so that there are two separate regions in store: one region for the instructions and another for the data. Now, if it subsequently proved necessary to remove or add an instruction in the master routine it would only be necessary to adjust the value of the M-parameter. Regionalizing the instructions and data in this way also makes programming easier because it is not necessary to know the length of the program before allocating storage for the data.

Subroutine Parameters

Library subroutines had a number of ways of specifying their parameters or arguments. The easiest way was to use a dedicated storage location. This is used, for example, in the print subroutine P6, which prints the integer placed in location 0F before the subroutine is entered; similarly, the division subroutine D6 sets 0D to the value of 0D/4D. A more flexible, though more complicated, arrangement was what the Cambridge group called *program parameters*. Here one or more parameters were specified in the calling sequence itself. For example, in the Reciprocals program the library subroutine P1 prints the long fraction in 0D to *n* decimal places, where *n* is specified as a program parameter (see lines 11 to 13 of the master routine in Fig. 12a).

Program parameters are the way that most software systems still parameterize library subroutines. Incidentally, in *Wilkes, Wheeler and Gill*, there is another technique known as “preset parameters” - this method has since fallen into disuse and we will not discuss it here. But it is one of several now-forgotten ideas in EDSAC programming awaiting rediscovery. (Just to add a little more confusion, note that subroutine M3 used in Reciprocals does not conform to any of the types discussed above. It prints out the text that follows it, and is then overwritten by the program proper, so as not to take up any memory at run time. It was very useful for printing out table headings and the like.)

```
          G      K
          T   47 K   ] Set M parameter
          P   21 θ   ]
          T      Z
        0 S    1 M   ] Set count to -9
19 →    1 T    6 M   ]
        2 A    2 M   ] 1·2^-4 to 0D
        3 T      D   ]
        4 A    7 M   ] n·2^-4 to 0F
        5 T    4 D   ]
        6 A    6 θ   ] Set 0D to 0D/4D(ie. 1/n)
        7 G   56 F   ]  using subroutine D6
        ------------
D6 →    8 O    3 M   ] Output new line
        9 O    4 M   ]
       10 O    5 M     Output decimal point
       11 A   11 θ   ] Print 0D
       12 G   92 F   ]  using subroutine P1
        ------------
       13 P   10 F     Parameter for P1
P1 →   14 A    7 M   ]
       15 A    2 M   ] Increment n
       16 T    7 M   ]
       17 A    6 M   ] Increment
       18 A      M   ]   and test counter
       19 G    1 θ   ]
        ------------
       20 Z      F     Stop

M      O P      D     = 1
       1 P    4 D     = 9
       2 Q      D     = 1·2^-4
       3 θ      F     carriage return
       4 Δ      F     line feed
       5 M      F     decimal point
       6 P      F     count
       7 W      F     = n (=2·2^-4 initially)
```

(a) Master routine

```
space P K

T 56 K

┌──────┐
│  M3  │
└──────┘

θΔ*RECIPROCALSθΔπ    Table heading

space P Z

T 56 K

┌──────┐
│  D6  │
└──────┘

space P Z

┌──────┐
│  P1  │
└──────┘

space P Z

┌──────┐
│Master│
└──────┘

E Z P F
```

(c) Make-up of program tape

Routine	Location of first order	Number of storage locations occupied
D6 (divide)	56	36
P1 (print)	92	21
Master	113	-

(b) Table of routines

```
RECIPROCALS

.4999999999
.3333333333
.2499999999
.2000000000
.1666666666
.1428571428
.1249999999
.1111111111
.0999999999
```

(d) Printout

Fig. 12 Reciprocals program

Scaling and Rounding

The problem of scaling arises because the EDSAC could only store fractions in the range $-1 \leq x < 1$. This was a problem with most early computers, although the advent of hardware or software floating-point in the mid-1950s meant that most users were soon able to forget about it.

In the case of the Reciprocals program, the reciprocals $^1/_2$, $^1/_3$, ... $^1/_{10}$ are all in the range $-1 \leq x < 1$, so there will be no need to scale the results. The denominators 2, 3, ... 10, however, would be out of range for fractions. We therefore scale them by 2^{-4}, so that they are all of modulus less than unity. Now, when we calculate the reciprocal $^1/_n$ using:

$$\frac{1 \times 2^{-4}}{n \times 2^{-4}}$$

the scale factors cancel and the result is correct. Life was not always so easy and often scaling was the hardest part of solving a problem. (For a fuller discussion of scaling see example program "TPK" in the *Program Documentation* pdf file.)

There is a copy of the Reciprocals program in the Reciprocals Folder, in the Tutorial Guide Examples folder. Note that subroutine P1, like most of the EDSAC print routines, did not perform rounding. Thus in the output of Reciprocals shown in Fig. 12c, ½ is printed as 0.4999999999 rather than as 0.5000000000. Here, this is mainly an aesthetic point, but normally a fraction would be rounded to n decimal places by adding the constant $\frac{1}{2} \times 10^{-n}$. A number of useful rounding constants are given in the Appendix, Table 4.

Exercise

1 Modify the Reciprocals program so that it prints the results to a precision of 6 decimal places, unrounded. (*Hint:* Change the parameter for P6 in line 13).

2 Print the results of the Reciprocals program rounded to 6 decimal places. (NB. The constant $\frac{1}{2} \times 10^{-6}$ is "W 199 F, P F"; this should be placed in an adjacent even-odd word pair (eg. words 8M and 9M). To add a long constant into the accumulator, use A 8 πM (say); the π forces the length bit in the instruction to be 1.

4 DEBUGGING: GETTING PROGRAMS RIGHT

> By June 1949 ... I was trying to get working my first non-trivial program, which was for the numerical integration of Airy's differential equation. It was on one on my journeys between the EDSAC room and the punching equipment that "hesitating at the angles of stairs" the realization came over me that a good part of the remainder of my life was going to be spent in finding the errors in my own programs.
>
> M.V. Wilkes, *Memoirs*, 1985, p. 145

Like Wilkes, everyone who begins to program soon discovers that the difficulty lies not in writing programs, but in getting them to work. On the EDSAC there were essentially three ways of finding the mistakes in a program: peeping, the post-mortem technique, and checking routines. We will look at each of these in turn.

First, however, let us consider the two common types of bug (or "pitfalls" or "blunders" as they were called in *Wilkes, Wheeler and Gill*, which was published long before the term "debugging" gained currency).

1 *Control errors*. Control or sequence errors occur when the program logic is in someway faulty. Typically a control error causes a program to have unpredictable behaviour and eventually come to a halt in an apparently random location. The most common cause of a control error is a wrong address in a branch order, or faulty subroutine linkage.

2 *Numerical errors*. These are errors in the numerical computation of a program, which do not immediately affect the sequence in which the orders are obeyed. That is to say, the program apparently behaves well, but the answers are wrong. The most common causes of arithmetic errors were due to scaling errors, undetected overflow, and faulty numerical methods.

These two types of errors benefit from different debugging approaches.

4.1 Peeping

In Sections 2 and 3, we single-stepped through a couple of small programs, which demonstrated most of the salient features of peeping. However, real programs with subroutines soon show the limitations of the technique.

First, it is quite difficult to navigate around the monitor tube - which is why it makes sense to locate the origin of the master routine of a program on a tube boundary. Second, interpreting addresses and instructions rapidly and accurately in the Sequence Control Tank and the Order Tank takes a lot of practice; likewise, recognizing binary numbers takes experience. Third, when the program goes into a subroutine, it gets very tedious manually stepping through a hundred or more instructions until you can get back to the main program - one way round this is to plant additional stop orders in the program, so that subroutines can run at full speed. Finally, when you have located the error, the program text has to be corrected before it can be re-tested. On some early computers, it was possible to use hand-switches to "patch" in corrections, but this was deliberately made impossible on the EDSAC because it took up so much machine time. The general philosophy at Cambridge was to use post-mortem and checking routines so that debugging could be done away from the computer, leaving it

free for more productive work. This was further encouraged in 1950 when a computer operator was employed to run users' programs (Fig. 13). Program tapes were placed on a "job queue" with instructions as to what action should be taken in the event of a program failure.

4.2 Post-mortems

A post-mortem - more commonly known in the United States as a terminal dump - was the process of printing out a region of the memory after the execution of a program had been terminated. On the EDSAC a post-mortem routine was loaded by the initial orders in the usual way. (So unless you are pressed for memory space, you should avoid using locations 0-55 for data storage other than for temporary variables.) The post-mortem routines themselves were automatically loaded as high up in the memory as possible, where they were least likely to overwrite the information to be dumped.

When programs were run on the EDSAC by an operator, the programmer would leave instructions as to which post-mortem tape to be used in the event of an abnormal program termination, and the address where the post-mortem was to start - which was dialled in by the operator.

There were six post-mortem routines in the original EDSAC program library, with the following specifications:

PM0 Starting at location *n*, print the order-code letter contained in the top five binary digits of each location; continue until stopped by the operator.

Fig. 13 EDSAC Operator

PM1-4 Starting at location *n*, print the contents of each location as a decimal number in the following form:

PM1: short fractions

PM2: long fractions

PM3: short integers

PM4: long integers

Continue until stopped by the operator.

PM5 Starting at location *n*, interpret each word of store as an order and print the appropriate order; continue until stopped by the operator.

All the post-mortem routines except PM0 have been written afresh for the Edsac simulator; unfortunately all the original tapes have long since vanished and there are no extant listings. They are slightly more user-friendly than the original routines, but otherwise conform closely to the original specifications. These are the only items in the Edsac library that are not original artefacts. (Incidentally, the PM0 routine, which *is* an original artefact, is a very clever piece of coding that shows what was possible with the initial orders. It occupies exactly four words of memory.)

To use PM5 (say) proceed as follows. First, run Recprocals so there is a user program in the store. Next, click the "Postmortem Routines" tab in the toolbar, and select PM5. Confirm that the simulator title bar reads "Output from: PM5". Press Start - without pressing Clear, otherwise you will lose everything. When the program stops, dial the 3-digit location where you want the post-mortem to start (eg. 113). The store will now be printed out from word 113 onwards. Press Stop when you have enough output. If you wish, the output can be saved and printed for study away from the machine.

```
113  S  135 F
114  T  140 F
115  A  136 F
116  T      D
117  A  141 F
118  T    4 D
119  A  119 F
120  G   56 F
121  O  137 F
122  O  138 F
123  O  139 F
124  A  124 F
125  G   92 F
126  P   10 F
127  A  141 F
128  A  136 F
129  T  141 F
     .
     .
     etc.
```

Fig. 14 Post-mortem using PM5

Fig. 14 shows part of the P5 post-mortem printout of Reciprocals. Notice how all the addresses are in absolute form - this quite often makes errors in code-letter usage immediately obvious. Note also that zero words are not printed, and that order codes which correspond to a "stunt" character (shift, line feed, return, etc.) temporarily upset the alignment of the printout. You will find that PM5 is by far the most helpful debugging aid you are likely to use. This was not the case on the original EDSAC because printing on a $6^2/_3$ character per second teleprinter meant it took several minutes to print a substantial region of store; PM0 was much faster, though not so useful.

Exercise

1 Try using PM0. The version of PM0 provided prints the function letter of each instruction from location 56 upward. It is easy to modify the program for another starting point (see the program text). Modify the program to print from 113 upward - and compare the output (STATATAGO...) with Fig. 14.

4.3 Checking Routines

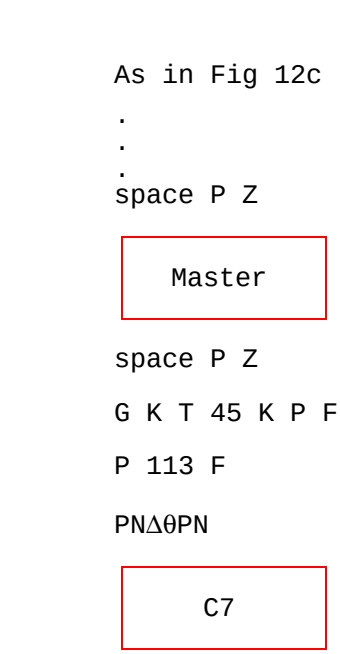
The EDSAC pioneered the technique of interpretive trace routines - although the term "trace" was not then in use, and they were called "checking" routines. Checking routines were invented by Stanley Gill - the third author of *Wilkes, Wheeler and Gill*; he was then a research student and was later Professor of Computing Science at Imperial College, University of London, and one-time president of the British Computer Society.

The idea of a trace routine is that, instead of obeying the orders of a program directly by the control circuits of the computer, they are obeyed by an interpretive program or simulator. It is then possible to print out diagnostic information - ie. a trace - while the program is being executed. There were several checking subroutines in the EDSAC library, although just the two provided with the simulator will suit most purposes. These are subroutines C7 and C10. C7 is useful for checking control errors, while C10 is most useful for checking numerical errors. Using checking routines needs a little planning and forethought, but they are very powerful, and once the technique has been mastered it can be more effective than peeping.

C7: Sequence Checking

C7 prints out the order-code letter of each instruction as it is obeyed. This enables the flow of the program to be checked against the program manuscript: where control departs from the expected sequence, then that is where the error lies. A particularly attractive feature of C7 is that it only interprets code from sections of the memory designated by the programmer. This enables the tracing of subroutines - which can be assumed to be correct - to be suppressed, so that the amount of print-out produced is minimized.

To use C7 on an existing program, we simply replace the final control combination (typically E Z P F) with C7 preceded by its control combinations. The control combinations are a bit messy, but the scheme shown in Fig. 15a works for simple cases.

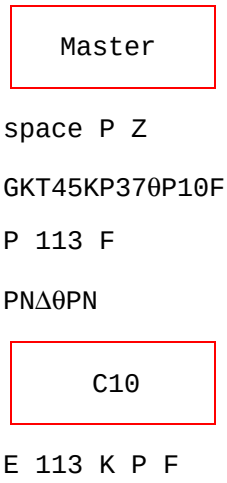


(a) Make-up of program tape

```
RECIPROCALS

STATATAG

OO
OMAG
4999999999
AATAAG
TATATAG

OO
OMAG
3333333333
AATAAG
TATATAG

OO
OMAG
2499999999
AATAAG
TATATAG
.
.
etc.
```

(b) Printout

Fig. 15 Use of checking subroutine C7



(a) Make-up of program tape

```
RECIPROCALS

+7541198730-0001373291+0625000000
+1250000000

.
4999999999+1875000000
-0001220703+0625000000+1875000000

.
3333333333+2500000000
-0001068115+0625000000+2500000000

.
2499999999+3125000000
-0000915527+0625000000+3125000000

.
2000000000+3750000000
-0000762939+0625000000+3750000000
.
.
.
etc.
```

(b) Printout

Fig. 16 Use of Checking Subroutine C10

There is a copy of Reciprocals with C7 appended in the Reciprocals Folder (in the file Reciprocals+C7). Fig. 15b shows the print out produced by Reciprocals+C7. The full specification for C7 is given in the *Program Documentation*.

Note how effectively this trace enables you to navigate around the master routine and to follow its control logic. Note that the subroutine prints a new line after a branch order, and that a clear line is left whenever instructions are obeyed "silently" (unless the silent instructions themselves cause printing to occur).

C10: Numerical Checking

The C10 subroutine helps to trace numerical errors by printing the contents of the accumulator (as a long fraction) every time the user program executes a T-order. Thus the printout will contain all the intermediate results computed in the program.

The C10 subroutine, like C7, is appended to the end of the program, replacing the final control combination (usually E Z P F). Again, the control combinations are rather messy. Fig. 16 shows the make-up of the program tape and printout produced by Reciprocals+C10. Note that the first word is always junk, and that the next three numbers printed represent the integer -9 (which looks strange when printed as a long fraction.), 1 x 2^{-4}, and 2 x 2^{-4} respectively. A new line is printed after every branch statement, and a line feed is output whenever a subroutine is obeyed silently.

Frankly, the C10 subroutine is quite painful to use, and it was used very much as a last resort when a numerical calculation would not give exactly the right results. Numerical errors on the EDSAC could be extraordinarily stubborn. Probably you will never have occasion to use this subroutine in anger, but it is there should you need it, and full documentation is given in the *Program Documentation* pdf file.

5 PROBLEMS FROM THE SUMMER SCHOOL AND ELSEWHERE

If you understood all or most of the material in the *Tutorial Guide*, you might now like to try developing an EDSAC program yourself.

Beginning in 1950, the Mathematical Laboratory at Cambridge organized Summer Schools in programming for people inside the University and for other universities and industry. The course was of a fortnight's duration and during that period students were expected to write and get running some simple programs on the EDSAC. Programs 1 to 5 below were all Summer School problems. They were all small, though not trivial, problems; for example they generally need to make use of the subroutine library, and sometimes scaling is required. The remaining problems are more challenging.

1 Print the value of the function $\sqrt{\frac{n}{n+1}}$ for $n = 1, 2, \ldots 10$.

2 Read a sequence of 20 long fractions from the input tape and print the sum of their squares. (*Note:* Use subroutine R1.)

3 Print the inverse factorials $\frac{1}{n!}$, and their sum, of the numbers $n = 2, 3, \ldots 10$.

4 Print the sum of $\frac{1}{n}$ and the sum of $\frac{1}{n^2}$ for $n = 1, 2, \ldots 100$. (*Note*. The results will need to be scaled.)

5 A traffic census is to be taken using a tape punched as follows. Whenever a bicycle passes a B is punched, and whenever a motor vehicle passes a V is punched. Every minute an M is punched, except at every tenth minute when a T is punched. At the end of the tape an E is punched. Rows of blank tape, and erase, carriage return and line feed symbols may appear anywhere. Prepare a program to process this tape as follows:

   (a) Check that, apart from rows of blank tape and erase, carriage return and line feed symbols, only the symbols B, V, M, T and E appear on the tape;

   (b) Check that exactly nine M's intervene between consecutive T's;

   (c) Print the greatest number of bicycles that pass within any consecutive 15 minutes, and the greatest number of motor vehicles that pass within any 15 consecutive minutes.

[Author: D. J. Wheeler, c.1950]

Note The above problems are roughly in ascending order of difficulty It is possible to solve all of them using only the subroutines D6, P1, P6, R1, and S2, whose specifications are given in Table 3 of the Appendix.

Here are some more substantial problems, not from the Summer School.

6 *Library Square-Root Subroutine* Write a subroutine S99 for the EDSAC library

which calculates the square root, x, of the argument, a, stored in 0D. Use the Newton-Raphson iterative formula:

$$x_{n+1} = \frac{1}{2}\left(x_n + \frac{a}{x_n}\right)$$

Incorporate the subroutine in a driver program which tests it for a variety of arguments. Note that to keep subroutines in the EDSAC library short, arguments were not normally validated - bad arguments simply produced bad results.

7 *Programmed Multiplication Test* Write a program which will test if the Edsac multiplier is functioning correctly (assuming all other machine functions are OK). On the original EDSAC this test was used to ensure the hardware was serviceable, but on the simulator it would verify the correct implementation of the multiplier.

8 *Big Letters* Write a program that prints the contents of a paper tape in big letters:

```
BBBB   I   GGGG       L      EEEEE  TTTTT TTTTT EEEEE  RRRR    SSS
B   B  I  G          L      E        T     T   E      R   R  S
BBBB   I  G   GG     L      EEE      T     T   EEE    RRRR    SSS
B   B  I  G    G     L      E        T     T   E      R  R       S
BBBB   I   GGGG      LLLLL  EEEEE    T     T   EEEEE  R   R   SSS
```

9 *Pretty Printing* A program (now lost) was developed in 1953 that would take an EDSAC tape, and list it in formatted form, one instruction per line. Since the Greek letters phi, theta, delta and pi could not be printed, they were substituted with /, $, +, and . (period).

10 *Highland Dancer* A demonstration program, now lost, displayed an animation of a highland dancer on the main-memory monitor tube. Write a program to produce an entertaining animation.

11 *Sieve of Eratosthenes* The sieve of Eratosthenes is used to determine the prime numbers between 2 and n as follows:

> Write down the numbers 2 to n; starting at 4 (ie. 2 squared) cross out all the multiples of 2; from 9 (ie. 3 squared) cross out all the multiples of 3; ignore 4 because it has already been crossed out; from 5^2 cross out all the multiples of 5; and so on. At the finish only the primes remain.

In about 1950 Wheeler used this idea to calculate primes on the EDSAC at high speed by avoiding the operation of division:

> [On the monitor tube] it was possible to show 16 35-bit words at a time, that is 560 bits altogether. In Wheeler's program the ... numbers were represented in order by binary digits. To begin with all these digits were present. As the sieve operated and numbers were eliminated the ones were replaced by zeroes. The speed of the machine was such that it was possible to watch this happening on the screen." (Wilkes, *Nature*, October 1975, p. 544).

Write a Sieve of Eratosthenes program to determine the primes between 2 and about 500.

12 *EDSAC's First Achievement* The first program run on the EDSAC on 6 May 1949 printed a table of squares of the integers 0 to 99. Unfortunately, the original program is lost, although several copies of the printout have survived (Fig. 17) Recreate the program, using Initial Orders 1.

If you are in London with time on your hands, you might like to visit the Science Museum, where this historic printout is on permanent display.

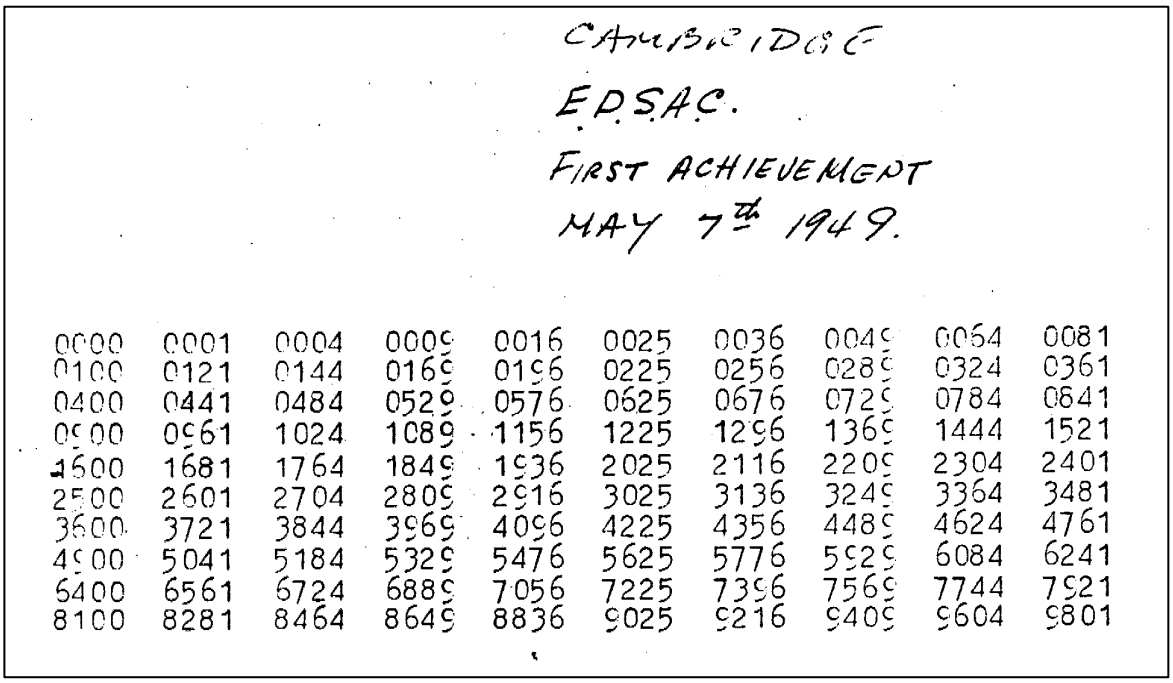


Fig. 17 EDSAC's First Achievement

Acknowledgements

The EDSAC simulator draws on the work of my former students Martin Smedley, Ken Fowler, Paul Waldron, and John Martin. My thanks to John Cowan, Ross Hamilton, Lee Wittenberg, Bill Purvis and numerous beta testers. Finally, I acknowledge the help of Maurice Wilkes, David Wheeler, and Don Hunter who supplied me with numerous unrecorded historical details about the EDSAC and its programming techniques.

Martin Campbell-Kelly

Bibliography

If you enjoyed using the simulator and would like to explore EDSAC programming in more depth, you will need to get hold of a copy of the first edition (1951) of *Wilkes, Wheeler and Gill*. This was reprinted by MIT Press in 1982, but is now unfortunately out of print again; you should, however, be able to find a copy in any major university library or through inter-library loan. My article "Programming the EDSAC" (1980) discusses the later development of EDSAC programming and includes a full bibliography. Another article "The Airy Tape" (1992) describes Wilkes's attempt to get his first real program working - and the discovery of debugging. Something of Wheeler's programming philosophy can be discovered from his article "The EDSAC Programming Systems" (1992). Finally, Maurice Wilkes's very readable *Memoirs* are invaluable for understanding the EDSAC milieu.

M. Campbell-Kelly, "Programming the EDSAC: Early Programming Activity at the University of Cambridge", *Annals of the History of Computing 2* (1980) pp. 7-36.

M. Campbell-Kelly, "The Airy Tape: An Early Chapter on the History of Debugging", *Annals of the History of Computing 14* (1992), pp. 18-28.

M. V. Wilkes, *Memoirs of a Computer Pioneer*, MIT Press, 1985.

M. V. Wilkes, D. J. Wheeler and S. Gill, *The Preparation of Programs for an Electronic Digital Computer*, 1951, Addison-Wesley. Reprinted as Vol. 1 of the Charles Babbage Institute Reprint Series for the History of Computing, MIT Press, 1982.

M. V. Wilkes, "The EDSAC", and B.H. Worsley, "The EDSAC Demonstration", in *Report of a Conference on High-Speed Automatic Calculating Machines, 22-25 June 1949,* Cambridge University Mathematical Laboratory, 1950. Reprinted as pp. 415-429 of B. Randell, *Origins of Digital Computers*, Springer-Verlag, 1983.

D. J. Wheeler, "The EDSAC Programming Systems", *Annals of the History of Computing 14* (1992), pp. 34-40.

Appendix of Tables

Table 1 The EDSAC Instruction Set (1949)

A	n	Add the number in storage location n into the accumulator
S	n	Subtract the number in storage location n from the accumulator
H	n	Copy the number in storage location n into the multiplier register
V	n	Multiply the number in storage location n by the number in the multiplier register and add the product into the accumulator
N	n	Multiply the number in storage location n by the number in the multiplier register and subtract the product from the accumulator
T	n	Transfer the contents of the accumulator to storage location n and clear the accumulator
U	n	Transfer the contents of the accumulator to storage location n and do not clear the accumulator
C	n	Collate [logical *and*] the number in storage location n with the number in the multiplier register and add the result into the accumulator
R	2^{n-2}	Shift the number in the accumulator n places to the right
L	2^{n-2}	Shift the number in the accumulator n places to the left
E	n	If the sign of the accumulator is positive, jump to location n; otherwise proceed serially
G	n	If the sign of the accumulator is negative, jump to location n; otherwise proceed serially
I	n	Read the next character from paper tape, and store it as the least significant 5 bits of location n
O	n	Print the character represented by the most significant 5 bits of storage location n
F	n	Read the last character output for verification
X		No operation
Y		Round the number in the accumulator to 34 bits
Z		Stop the machine and ring the warning bell

Table 2 Edsac Character Codes

Perforator		Teleprinter		Binary	Decimal
Letter shift	Figure shift	Letter shift	Figure shift		
P	0	P	0	00000	0
Q	1	Q	1	00001	1
W	2	W	2	00010	2
E	3	E	3	00011	3
R	4	R	4	00100	4
T	5	T	5	00101	5
Y	6	Y	6	00110	6
U	7	U	7	00111	7
I	8	I	8	01000	8
O	9	O	9	01001	9
J		J		01010	10
π		Figure Shift		01011	11
S		S	"	01100	12
Z		Z	+	01101	13
K		K	(	01110	14
Erase[1]		Letter Shift		01111	15
Blank tape[2]		(no effect)		10000	16
F		F	$	10001	17
θ		Carriage Return		10010	18
D		D	;	10011	19
ϕ		Space		10100	20
H	+	H	£	10101	21
N	-	N	,	10110	22
M		M	.	10111	23
Δ		Line Feed		11000	24
L		L	)	11001	25
X		X	/	11010	26
G		G	#	11011	27
A		A	-	11100	28
B		B	?	11101	29
C		C	:	11110	30
V		V	=	11111	31

Notes

1 Erase is represented by an asterisk ("*") in the simulator. When this character is *output*, it sets the teleprinter into letter shift.

2 Blank tape is represented by a period (".") This character has no effect on output.

3 The symbols θ, ϕ, Δ or π are typed as @, !, & and #, respectively.

Table 3 Specifications of Basic Library Subroutines

Subroutine	Length	Description and Notes
Input-output		
P1	21	Print positive long fraction in 0D to n decimals. Program parameter is P n F. See Reciprocals program for example of use.
P6	32	Print short positive integer in 0F. See Cubes program for example of use.
P7	35	Print long positive integer in 0D. Must start in an even location.
P14	46	Print signed decimal fraction in preset layout. See specification in the Program Documentation pdf file, and example of use in TPK program.
R1	55	Input a sequence of signed, long decimal fractions. See documentation in the Edsac Texts folder, and example of use in TPK program.
R3	41	Input a signed long-decimal fraction. Reads a fraction punched in decimal form followed by sign into 0D.
R4	22	Input a signed integer. Reads an integer punched in decimal form followed by sign. Short integers placed in 0F; long integers placed in OD.
Mathematical		
D6	36	Division. Divides 0D by 4D; result in 0D.
E2	19	Exponential. See specification in Program Documentation pdf file.
S2	22	Square root. Forms square root of 4D; result in 4D.
S3	25	Cube root. Forms cube root of 6D; result in 0D.
T1	44	Cosine. See specification in Program Documentation pdf file.
Miscellaneous		
M3	-	Print a heading. Copies information from the tape to the teleprinter. Occupies 10 locations (temporarily). See Reciprocals program for example of use.
M20	-	Read in a three-digit decimal number from the dial. See specification in Program Documentation pdf file.

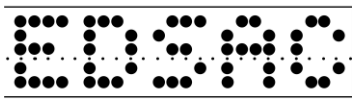
Continued ....

Table 3 continued

Checking		
C7	61	Checking routine - trace of function code letters.
C10	88	Checking routine - trace of accumulator contents.
Post-mortem routines		
PM0		Starting at location n, print the order-code letter contained in the top five binary digits of each location; continue until stopped by the operator. Occupies locations 41-44.
PM1-PM4		Starting at location n, print the contents of each location as a decimal number in the following form: PM1: short fractions PM2: long fractions PM3: short integers PM4: long integer Continue until stopped by the operator. All routines occupy locations 955-1023.
PM5		Starting at location n, interpret each word of store as an order and print the appropriate order; continue until stopped by the operator. Occupies locations 940-1023.

Table 4 Some Useful Constants

-0.1	L 1229 F C 819 F	$+1/3$	H 682 D T 682 D	$1/2 \times 10^{-4}$	F 1464 D P 1 D
+0.2	S 1638 D E 409 D	$+1/7$	θ 585 F W 585 F	$1/2 \times 10^{-6}$	W 199 F P F
-0.3	S 1638 D G 409 D	$1/9$	K 455 F C 455 F	$1/2 \times 10^{-10}$	P D P F
+0.4	L 1229 F Y 819 F	$-1/11$	π 1303 D C 1117 F	$+10^{-2} \times 2^{6}$	W1147 F J 419 D
-0.6	L 1229 F N 819 F	$-\pi/4$	O 699 D D 888 F	$+10^{-3} \times 2^{9}$	G 1327 F I 393 F
+0.7	S 1638 D π 409 D	$-\pi/6$	G 1149 F M1274 D	$+10^{-4} \times 2^{13}$	T 1714 F Z 219 D
-0.8	S 1638 D D 409 D	$-2/\pi$	V 291 D H 1667 F	$+10^{2} \times 2^{-7}$	S 1024 F
+0.9	L 1229 F K 819 F	$+e/4$	H 177 D J 1788 F	$+10^{3} \times 2^{-10}$ $+10^{4} \times 2^{-14}$	K 3328 F O 1568 F



The Edsac Replica Project • The National Museum of Computing
Bletchley Park • Milton Keynes MK3 6EB • United Kingdom