Rosemary Candlin

INTRODUCTION TO PROGRAMMING

FOR EDSAC 2

SEPTEMBER 1957

C O N T E N T S .

AN INTRODUCTION TO PROGRAMMING FOR EDSAC 2

PART I

1. Introduction.

The purpose of this booklet is to provide an introduction to programming for newcomers to the subject. By the 'program' is meant the schedule of operating instructions by means of which the machine carries out the calculation. Each of these instructions (or 'orders') specifies a single operation (such as addition or multiplication), and the program must be drawn up in such a way that the machine not only carries out the operations required, but does so in the right sequence.

The ideas involved in programming for different machines are essentially similar. However, in order to fix one's ideas and to provide examples for illustration and practice, it is necessary to work in terms of the programming system for a particular machine. Since this booklet is intended to be used in connection with the Summer School in programming held at the University Mathematical Laboratory, Cambridge, it is written in terms of the system used for EDSAC 2. It is meant to be only a first introduction to the subject: in particular it leaves out many of the advanced facilities available on EDSAC 2.

The main things that a programmer needs to know about a machine are:

(i) Its 'order code', that is, the different elementary operations that the machine can carry out, and how they are specified.

(ii) How the machine, having carried out the operation specified by one instruction, determines the next one.

(iii) The input and output media: that is, the physical forms in which the program and data are supplied to the machine and in which results are supplied by it; and the auxiliary equipment with which the input can be prepared and the output processed.

(iv) The coded form in which program and data must be specified to the machine.

(v) The range of numbers that can be stored inside the machine, and the precision with which they can be stored.

In the following sections the order code is introduced gradually, so that the student becomes accustomed by practice to some of the more commonly used orders without having to master the whole order code at

once. Orders are of the 'one-address type; that is, each order which refers to the store (some do not) refers to one number in the store only. Orders are normally placed in the store in the sequence in which they are to be obeyed. Input and output are both by means of five-hole punched paper tapes. The input tape is prepared on a keyboard perforator, similar to a typewriter.

In EDSAC 2, numbers are stored in floating binary form. However, in the early stages of learning how to program, the programmer has no need to know the precise form in which numbers or orders are stored, or the details of the processes by which the machine operates on them. He should, however, know the range of numbers which can be stored and used, and their precision. In EDSAC 2, numbers up to 10^{40} can be stored and used, with a precision of about 10 significant decimal figures.

2. The Store.

The store consists of 1024 registers, each capable of holding one number or a pair of orders. A single order occupies only half a register; a half register is frequently called a storage location, or simply a location. In order to identify the storage locations, each is labelled by a number, called its address; these addresses run from 0 to 2047, and the addresses of the two halves of a register are always an even number and the odd number following it - that is, 2q and 2q+1. The whole storage register is referred to by the even address 2q. (This leads to no confusion, since it is always clear whether we are talking about a register or a location.) It is sometimes convenient to use the term 'word' to describe the contents of a register without specifying whether it is a number or a pair of orders. The notation C(q) will be used for the content of location q, and F(q) - where q is even - for the content of register q.

3. The Arithmetic Unit.

The central part of the arithmetic unit is a register which plays the part of the result register in a desk calculating machine; it accumulates the sum of numbers added into it. This register is known as the accumulator and its content will be written F(Acc). Ordinarily, the effect of any arithmetic operation is simply to change the value of F(Acc).

At any stage in a calculation, the number in the accumulator can be copied into any specified storage register q by means of an appropriate

instruction. When this is done, the previous content of the register is obliterated, and is replaced by the content of the accumulator. The number in the accumulator itself is not altered.

The accumulator, or a storage register, is said to be 'clear' if its content is zero; and to 'clear' a register means to make its content zero.

4. Storage of orders.

Orders are normally placed in storage locations numbered in the sequence in which they are to be obeyed. The control unit is so designed that after the machine has carried out the order in location q it automatically takes next the order in location q+1, unless the order in location q has specified otherwise (see Section 7).

5. Written form of orders.

When orders are written, the function and the address are written as two integers, normally separated by the letter f. The written forms of some of the more important arithmetical operations are as follows:

10 f q place in the accumulator the number in storage register q.

11 f q place in the accumulator minus the number in storage register q.

12 f q add to the number in the accumulator the number in storage register q.

13 f q subtract from the number in the accumulator the number in storage register q.

14 f q multiply the number in the accumulator by the number in storage register q.

15 f q divide the number in the accumulator by the number in storage register q.

19 f q copy the number in the accumulator into storage register q.

In all these, q stands for the address of a storage register, that is, an even integer in the range 0 to 2046 inclusive: it is called the address part of the order. For orders with function numbers 10 to 15, the result of the operation is placed in the accumulator.

The contents of the store are not affected by any of these orders except the order with function number 19, and this order does not affect the content of the accumulator.

An order with function number x is usually called an 'x order'.

The 10 order is said to set F(q), the number in storage register q, in the accumulator; similarly the 11 order is said to set -F(q) in the accumulator. The results of these operations are independent of the previous content of the accumulator, which is deleted and is lost unless it has previously been placed in some register in the store.

6. Some simple examples.

We are now able to write down the sequences of orders required to perform some simple calculations. These are to be thought of as forming part of a larger program: the numbers to be operated on have been calculated and placed where they are at an earlier stage of the work. For the reader's convenience, we note with each order the operand (that is, the number in the store to which the order refers), and also the number in the accumulator after the order has been obeyed. These are purely explanatory: they do not form part of the program as it would be read into the machine.

In writing our sequences of orders, we shall draft them in such a form that the required calculation is carried out regardless of the content of the accumulator before the first order of the sequence is obeyed. This usually requires that the sequence starts with a 10 or 11 order.

Example 1. Given x = F(200), y = F(202) - that is, the numbers x and y are in registers 200 and 202 respectively; to form x+y and place it in 204.

Order	Operand	F(Acc)
10 f 200	x	x
12 f 202	y	x+y
19 f 204	x+y	x+y .

Since the orders are to be obeyed in this sequence, and the control unit takes orders in sequence from successive storage locations unless explicitly instructed to do otherwise, these orders must be placed in successive storage locations, say 100, 101, 102:

Storage location	Order
100	10 f 200
101	12 f 202
102	19 f 204

Example 2. Given a = F(300), b = F(302), x = F(350), y = F(352); to place ax in 420 and y(ax+by) in 430.

Location	Order	Operand	F(Acc)
100	10 f 350	x	x
101	14 f 300	a	ax
102	19 f 420	ax	ax
103	10 f 302	b	b
104	14 f 352	y	by
105	12 f 420	ax	ax+by
106	14 f 352	y	y(ax+by)
107	19 f 430	y(ax+by)	y(ax+by).

7. Jump orders.

Unless it is explicitly instructed otherwise, the machine obeys orders in the sequence in which they are stored. Any order which makes it do otherwise is known as a jump order. The effect of a jump order, with address part q, is to make the machine obey a new sequence of orders starting at storage location q (a 'jump to q'). A jump order may be unconditional, in which case the machine always jumps to q, or conditional, in which case the machine jumps to q if a certain condition is satisfied, and otherwise proceeds serially; that is, if the conditional jump order is in location r, it takes the order in (r+1) as the next to be obeyed. The form of an unconditional jump order is

50 f q

Two of the most common conditional jump orders, in which the criterion for a jump depends on the sign of the number in the accumulator, are as follows:

54 f q jump to q if F(Acc) $\geqslant 0$; otherwise proceed serially.

55 f q " " " " F(Acc) < 0; " " " .

It will be seen that these two conditional jump orders are complementary. Logically it would be sufficient for the programmer to have only one of them; but it is much more convenient to have both.

Example 3. If x = F(200) is negative, replace it by 0; otherwise leave it unchanged.

Location	Order
100	10 f 200
101	54 f 103
102	13 f 200
103	19 f 200

The sequence of orders actually carried out by the machine is different according as $x \geqslant 0$ or $x < 0$. If $x \geqslant 0$ the orders carried out are:-

Location	Order	Operand	F(Acc)	Notes
100	10 f 200	x	x	
101	54 f 103		x	
103	19 f 200	x	x	F(200) = x.

In this case the number in the accumulator when the 54 order is encountered is positive or zero. For a 54 order this is the condition for a jump, and the machine therefore jumps to the address 103 specified in the 54 order.

If $x < 0$, however, the orders carried out are:-

100	10 f 200	x	x	
101	54 f 103		x	
102	13 f 200	x	0	
103	19 f 200	0	0	F(200) = 0.

In this case, the number in the accumulator when the 54 order is encountered is negative. This is the condition for proceeding serially, and the machine takes its next order from 102. This causes F(200) = x to be subtracted from F(Acc) = x, leaving in the accumulator the number zero, which is the result required to be put in 200 when x is negative.

Note. The fact that the order in 103 may be reached by a jump from 101 is often indicated by the notation '101→' written to the left of the 'Location' column on the line referring to location 103, thus:-

101→103 19 f 200

This notation is purely for the guidance of the programmer, and plays no part in the orders in the machine.

8. Cycles of orders.

Many long calculations involve repeated application of the same group of orders to different sets of numbers. Such a group is called a cycle. The number of times the cycle has to be repeated may be known in advance or may depend on the numbers produced in the course of the calculation. In either case, the cycle must include at least one jump order, in order to return from the end of the cycle to the beginning: moreover, it must include at least one conditional jump, since otherwise it will be impossible ever to leave the cycle.

A short example in which the number of repetitions is not known in advance, but is controlled by the results of the calculation, is the following.

Example 4. F(4) is negative. Add F(2) (which is positive) to it repeatedly until the result becomes zero or positive, and place the result in 4.

Let the initial value of F(4) be -x (x is positive) and F(2) = y.

	Location	Order	Operand	F(Acc)
	100	10 f 4	-x	-x
102 ⟶	101	12 f 2	y	-x+y, -x+2y, -x+3y, ..., successively.
	102	55 f 101		do.
	103	19 f 4		final value of -x+my.

The order in 102 results in a jump back to 101 if -x+my is still negative; the first time -x+my becomes zero or positive, the machine proceeds serially from the order in 102 and, by obeying the order in 103, it places the final value of -x+my in 4.

Notes. 1. A procedure of this kind could be used for reducing a negative angle to the range 0 to 2π, by successive addition of 2π.

2. Here the conditional jump for leaving the cycle is also the jump to repeat the cycle.

The following is a less trivial example of a process in which the number of repetitions of a cycle depends on the results obtained as the calculation proceeds.

Example 5. Given x = F(200), a = F(300), where $0 < a < x < 1$; form the sum of the series $x+x^2+x^3+\ldots$, up to (but not including) the first term which is less than a, and place the result in 202.

We shall build up the sum term by term in 202, so that when the calculation is complete the result will already be in that register, as required. We shall also need to keep a record of x^n, the term last added to the sum, in order to form x^{n+1}, the next term to be added (if it is not less than a); let this be kept in 204. If we start with x in 202 and 204, we want the cycle to be such that, at the end of n repetitions of it, x^{n+1} is in 204 and $x+x^2+\ldots+x^{n+1}$ is in 202; for brevity we shall write S_{n+1} for this sum of n+1 terms, so that

$$S_{n+1} = S_n + x^{n+1}.$$

In programming a set of orders including a cycle, it is often most convenient to program the cycle first, and then to add the preliminary orders, such as those, in this example, which set the initial contents of 202 and 204. There are often several points in a cycle at which one might start the programming of the cyclic group of orders. We shall start the programming at the point where x^n has just been added to the sum and the result placed in 202. We then want to form x^{n+1}, test whether it is greater than a and, if it is not, jump out of the cycle. Let the first of these orders be in 103; then they are

Location	Order	Operand	F(Acc)	Notes
103	10 f 204	x^n	x^n	set x^n in the accumulator from 204
104	14 f 200	x	x^{n+1}	multiply x^n by x to form x^{n+1}
105	19 f 204	x^{n+1}	x^{n+1}	place x^{n+1} in 204
106	13 f 300	a	$x^{n+1}-a$	
107	55 f		$x^{n+1}-a$	

A 55 order is used to test whether the condition for leaving the cycle is fulfilled. This condition is $x^{n+1}-a < 0$, so a jump occurs if it is satisfied. The address to which the jump occurs cannot yet be specified. It is left blank for the moment, but will have to be inserted later.

If the condition $x^{n+1}-a <$ is not satisfied, the machine will take its next order from 108, and we want it then to add the value of x^{n+1}, just calculated, to the sum up to the term x^n which is in 202, and to place the result in 202. Since at this stage the accumulator contains $x^{n+1}-a$, which we do not want to be added into the sum, we must start the addition process with a 10 order:

108	10 f 204	x^{n+1}	x^{n+1}	
109	12 f 202	S_n	$S_n+x^{n+1} = S_{n+1}$	
110	19 f 202		S_{n+1}	place S_{n+1} in 202.

We are now ready to return to the beginning of the cycle, which is the order in 103, and we can do this by an unconditional jump order, since if these orders in 108 - 111 are reached at all, the condition for leaving the cycle is not yet satisfied. This jump order is therefore

111 50 f 103

Since the order in 103 is a 10 order, the content of the accumulator when this jump is made is irrelevant to the further calculation.

Alternatively we could use the group of orders:-

Location	Order	
100	10 f 200	
101	19 f 202	
102	19 f 204	
→103	10 f 204	orders 100 - 106 of (1)
104	14 f 200	
105	19 f 204	
106	13 f 300	
←107	54 f 110	
108	10 f 202	set sum in accumulator
←109	50 f 114	
→110	10 f 202	
111	12 f 204	orders 108 - 111 of (1)
112	19 f 202	
←113	50 f 103	
→114	next order.	

Here the cycle consists of the orders in locations 103 - 107 and 110 - 113; 100 - 102 are prologue and 108 - 109 epilogue.

The orders in locations 100 - 106 are the same as before. Now, however, we want the conditional jump order in 107 to be succeeded by the epilogue in 108 - 109 if the condition $x^{n+1}-a < 0$ <u>is</u> satisfied, and by a jump to 110 to continue the cycle if this condition is <u>not</u> satisfied: so we now require a 54 order at this point.

9. <u>Cycles of orders with a count</u>.

In the last example, the number of terms of the series we had to take could not be predicted in advance. We might instead have wished to take a definite number, say 50 terms of the series. One way to do this is to keep somewhere a count of the number of cycles still to be performed, to subtract 1 from it every time we go round the cycle and to leave the cycle when the count becomes 0. (This of course means that in the 'prologue' we must set the initial value of the count - in this case 49, since the first term of the series was dealt with in a special way, outside the cycle, by the orders in 100 - 102.) It would be perfectly possible to do the counting in the ordinary registers; however, this state of affairs is so frequent that the machine is provided with two special registers (called <u>modifier registers</u>,

for reasons which will appear in the next section) which can be used for this purpose. These are called the s register and t register. Their contents are integers which are denoted by s and t. The operations involving these registers will be explained in terms of the s register: similar statements, with t substituted for s, apply to the t register.

The orders which set the value of s are as follows:

70 s q put the integer q in the s register.

71 s q put the integer -q in the s register.

Note that q here stands for a number (not the content of storage register q) whereas s is a letter identifying the modifier register involved; also that these orders do not contain a letter f.

It is usually convenient (for reasons which will appear in the next section) to count in multiples of 2; thus for counting we need a way of increasing or decreasing s by 2, and we need a conditional jump order which will test whether s is 0. These are both, in fact, done by the same order:

74 s q increase s by 2; if the new value of s is 0 proceed serially, if not jump to q.

75 s q decrease s by 2; if the new value of s is 0 proceed serially, if not jump to q.

Example 6. Given $x = F(200)$, form the sum $x+x^2+\ldots+x^{50}$. Place the sum in 202.

The arithmetic of this calculation is simpler than that of Example 5, since we do not have to form and test $x^{n+1}-a$: the arithmetical orders of the cycle which we still need are those in locations 103 - 105 and 109 - 110 of the schedule (1) on p.9 Further, since we do not have to form and test $x^{n+1}-a$ before adding x^{n+1} to the sum, the conditional jump order for leaving the cycle can be put at the end of the cycle instead of in the middle; that is, we do the arithmetic before the counting. The main new point is the organisation of the count. In this case the cycle has to be performed 49 times. We can either count upwards from -98 to 0 or downwards from 98 to 0 (remembering that we count in multiples of 2); both versions are given here.

Counting downwards		Counting upwards		
Location	Order	Location	Order	
100	10 f 200	100	10 f 200	prologue
101	19 f 202	101	19 f 202	
102	19 f 204	102	19 f 204	
103	70 s 98	103	71 s 98	

Counting downwards		Counting upwards		
Location	Order	Location	Order	
→104	10 f 204	→104	10 f 204	
105	14 f 200	105	14 f 200	
106	19 f 204	106	19 f 204	arithmetical operations
107	12 f 202	107	12 f 202	
108	19 f 202	108	19 f 202	
109	75 s 104	109	74 s 104	count and test.

Notes. (1) The counting does not affect the accumulator. (This is one reason for using a modifier register.)

(2) If a different number of repetitions has to be made, only the address part of the order in 103 has to be changed. Moreover, the group of orders includes the setting of the count; thus it can be used any number of times in the course of the calculation.

(3) After k repetitions of the cycle (counting downwards), the initial value of s has been diminished by 2k. Another cycle will then be started unless s has been diminished to 0. Thus if the original value of s was 2j (an even number), the cycle will be performed just j times.

In preparing for a cycle, it is often convenient to place 0 in some storage register. For instance, in both ~~the~~ examples 5 & 6, ~~of this section~~ it would have been convenient to place 0 in 202 as the initial value of the sum ('the sum over 0 terms'). We got around this by treating the first term of the sum in a special way by the orders in 100, 101; but this is frequently very inconvenient. There is therefore a special order which has the effect of putting 0 in storage register q:

9 f q clear q; that is to say, set F(q) = 0.

This does not disturb the accumulator. To show its use, we give the following alternative version of Example 6:

Location	Order	Notes
100	71 s 100	set count (counting upwards)
101	9 f 202	clear 202 (initial value of sum)
102	10 f 200	put x in accumulator (as initial value of x^n)
103	50 f 106	jump into cycle
→104	10 f 204	
105	14 f 200	
→106	19 f 204	
107	12 f 202	
108	19 f 202	
109	74 s 104	

Here it is inconvenient to put 1, the initial value of x^n, into 204, since no storage location contains the number 1; so instead we enter the cycle at the point where x^{n+1} has been formed and is about to be stored, and we do this with x, the initial value of x^{n+1}, in the accumulator. The 74 order is obeyed at the end of this first part-cycle, which therefore is counted as if it were a complete cycle. Hence the number of cycles is now 50 and not 49, consequently the initial value of s has to be set to -100.

10. Automatic modification of orders. It often happens that a group of orders has to be repeated a number of times, with slight and systematic changes being made before each repetition. These changes will normally be in the address parts of some of the orders. For instance, if $x_1,\ldots,x_{40}$ are stored in 302,...,380 and $y_1,\ldots,y_{40}$ in 402,...,480, and we wish to form $x_1y_1+\ldots+x_{40}y_{40}$ in 200, then we have to go through a cycle 40 times; and when we go through the cycle for the n^{th} time we want its arithmetic orders to have the effect of those shown below (we leave out the count, for the moment, and write S_n for the sum up to and including x_ny_n):

Location	Order	Operand	F(Acc)	
102	10 f 300+2n	x_n	x_n	
103	14 f 400+2n	y_n	x_ny_n	
104	12 f 200	S_{n-1}	$x_ny_n+S_{n-1} = S_n$	(2)
105	19 f 200	S_n	S_n	

Suppose that we have chosen to count upwards to 0; then the n^{th} time we go through the cycle we have s = 2n-82, since the first time we have s = -80. In effect, therefore, the value of 2n is available in the modifier register. (This is why we chose to count up, and not down, in this case.)

To cope with this situation, there is an arrangement by which an order may be modified (before being obeyed) by having the content of the s (or t) register added to the address in it. This is indicated by using the letter s (or t) instead of f to separate the function and address parts of the order. It must be understood that the order is not altered in the store every time it is obeyed: what happens is that when the order is put from the store into the control unit s (or t) is added to its address part before it is obeyed.

Example 7. Given $x_1,\ldots,x_{40}$ stored in 302,...,380, and $y_1,\ldots,y_{40}$ stored in 402,...,480; form $x_1y_1+\ldots+x_{40}y_{40}$ in 200.

We already know what the arithmetical orders in the cycle should look like in the control unit: they are given in (2) above. The cycle can now be built up round them:

Location	Order	Operand	Notes
100	9 f 200	0	clear 200 ready to form sum
101	71 s 80		set count initially at -80
102	10 s 382		
103	14 s 482		
104	12 f 200		
105	19 f 200		
106	74 s 102		

The simplest way to determine the address parts of the two modified orders (in 102 and 103) is to note that the first time we go through the cycle, when s = -80, they are to reach the control unit in the form 10 f 302 and 14 f 402.

An order can only be modified in this way if it does not in its specification refer to a modifier register: the s in a 74 order, for instance, means that the order refers to the s register, and does not mean that the order is to be modified by s before being obeyed. Thus there are two kinds of orders in which s (or t) may appear. In orders of one kind, of which orders with function numbers 9 - 15 and 19 are examples, the letter s in the order indicates that the number s is to be added to the address part of the order before it is obeyed. In those of the other kind, of which orders with function numbers 70, 71, 74 and 75 are examples, the letter s denotes the modifier register which is involved in the operation specified by the order, but this operation does not involve adding the number s to the address part of the order. The orders of this second kind are those which have function numbers from 66 to 90 inclusive; they are called modifier orders.

When s is added to the address part of a modified order, the sum is taken modulo 2048; thus a meaningful order always results, and there is no overflow from the address part of the order into anywhere else.

Example 8. Given $a_0,\ldots,a_{40}$ stored in 300,...,380 and x = F(200); form $a_0+a_1x+\ldots+a_{40}x^{40}$ in the accumulator.

Here, for the first time, we meet an important point of tactics: it is not enough to know what one wants to calculate, one must also decide the particular way in which it is to be calculated. Here one's first idea might be to build up the series term by term, as we did earlier in Example 5. There is, however, a much more economical way, which is expressed by the formula

$$((\ldots((a_{40}x+a_{39})x+a_{38})x+\ldots)x+a_0).$$

This can be evaluated by a cyclic process, of which the arithmetical steps in a cycle consist simply of a multiplication by x and the addition of a coefficient a_j to the result.

When we go through the cycle the n^{th} time, the arithmetical orders must be obeyed in the form

14 f 200	multiply by x
12 f 380-2n	add a_{40-n}

and we have to go through the cycle 40 times. It follows that we must count <u>down</u>, so that at the n^{th} repetition we have s = 82-2n. (If we counted <u>up</u>, we would have s = 2n-82, which is no good to us.) Thus the group of orders is

<u>Location</u>	<u>Order</u>	<u>Notes</u>
100	70 s 80	set count initially at 80
101	10 f 380	put a_{40} in the accumulator
→102	14 f 200	cycle to multiply by x
103	12 s 298	and add a_{40-n}.
104	75 s 102	

11. <u>Special orders</u>.

In addition to the normal orders, there are in EDSAC 2 certain <u>special orders</u> (also called <u>permanent subroutines</u>). From the point of view of the programmer these are to be regarded as single orders, but, for engineering reasons, they all have the function number 59 and are distinguished by their address parts. Most of the operations they perform are more complicated than those of normal orders. Five of them replace the number in the accumulator, x, by some function of it:

59 f 11	place $x^{\frac{1}{2}}$ in the accumulator
59 f 12	" e^x " " "
59 f 13	" $\log_e x$ " " "
59 f 14	" sin x " " "
59 f 15	" cos x " " "

One is concerned with the input of numbers:

59 f 10 read a number from the input tape and place it in the accumulator.

(The form in which numbers have to be punched on the input tape is discussed in Section 14.)

Several are concerned with the output of numbers, in various forms and with various precisions. A typical one is

59 f 27 punch on the output tape the number now standing in the accumulator (in floating decimal form, to 7 significant figures). This does not affect the content of the accumulator.

These special instructions widen considerably the range of calculations which can be programmed quite simply.

Example 9. Read a number x from the input tape, calculate $y = e^{(x-e^x)}$ and punch x and y on the output tape.

In this example no repeated groups of orders occur, and it is easiest to draft the orders of the program in the sequence in which they will be carried out by the machine.

The machine must first read x from the input tape, which is done by the order

Order	Operand	F(Acc)
59 f 10	x	x.

Anticipating that we shall need x again at a later stage of the calculation we must plant it somewhere in the store. This we do by the order

19 f 2	x	x,

where we are using 2 as working space. Since x is still in the accumulator, we can punch it on the output tape by the order

59 f 27	x	x,

and, since this still leaves x in the accumulator, we can form e^x by

59 f 12	x	e^x,

and then e^x-x by

13 f 2	x	e^x-x.

To obtain y, we need the exponential of minus this last quantity, so the first thing to do is to change its sign. Since we no longer need the value of x, we can use 2 as working space, and change the sign of the content of the accumulator by the orders

19 f 2	e^x-x	e^x-x
11 f 2	e^x-x	$x-e^x$.

The value of y is now formed in the accumulator by the order

Order	Operand	F(Acc)
59 f 12	$x-e^x$	$y = e^{(x-e^x)}$

and punched on the output tape by the order

59 f 27	y	y.

What comes next in the program depends on what we want to do next. It may, for example, happen that we want to perform the same calculation with a new value of x read from the tape, in which case the next order will be a jump back to the beginning of the sequence. It can, however, happen that this one calculation is all we want to do. In this case we now want an order which stops the machine and signals to the operator that the calculation is ended. Such an order is the stop order:

101 f 0 stop the machine and light the stop warning lamp.

This order must be used to end any complete program, since otherwise the machine might go on running for ever.

12. Punched form of orders.

We have so far been concerned solely with the design of a program, that is to say with writing down the orders on paper. EDSAC 2, however, does not respond to written symbols, and in order to get the program into the machine it must first be punched out on paper tape, which is the actual input medium of EDSAC 2. The tape is punched out on a keyboard rather like that of a typewriter, and is the same sort of tape as is used to control teleprinters in telegraphy. The manner in which the tape must be punched is so designed that if the tape is passed through a tape reader connected to a teleprinter the result will be a printed copy of the program as written out in the manner laid down in the previous sections of this booklet.

Each symbol on the keyboard corresponds to a single row of holes on the tape. In this row there are five positions in which holes can be punched; in addition there is a small hole, known as the sprocket hole, which helps to guide the tape through the tape reader but has no programming significance. There are thus 32 possible symbols, including blank tape, which can be represented. These, together with their representations on the tape, are given in Appendix 1.

In addition to the decimal digits, those lower case (small) letters which are used in writing a program, and certain symbols

such as brackets and a decimal point, the code includes 'carriage return', 'line feed', 'space', 'letter shift', and 'erase'. The first three of these are the 'layout symbols': they do not themselves correspond to any mark on the paper (if the tape is printed out), but they control the way in which the other symbols are laid out on the printed page. They must be used in punching a program, as will be explained below. The use of the letter shift symbol on the input tape is restricted to the printing of titles (see Section 20). The erase symbol consists of all five holes and is ignored by the program input routine (except in a title - see Section 20). It may therefore be used to cancel any symbol that has been incorrectly punched on the input tape.

In punching a program, each order must be punched as it is written: that is, the function number, followed by the letter f, s or t, followed by the address (which may be in parametric form, as explained in Section 19). Single spaces may be included, and will improve the appearance of the printed page, but two spaces must never be punched together. Moreover, each order must be followed by the two symbols 'carriage return' and 'line feed'. This is not merely to produce an elegant layout on the page when the program is printed out: it is necessary to have some terminating symbol at the end of each order, and the carriage return symbol has been adopted for this purpose.

13. Directives.

The program input routine reads the orders and numbers of a program into the store. It must therefore be given, for example, some indication of where in the store the program is to be put, and when it is to stop reading in the program and begin obeying it. Such indications are called directives.

There are two types of directives which concern us at the moment. The first of them is written

p1 = q

where q is an integer. This causes the orders following on the tape to be read into successive storage locations, starting with location q. The effect of this continues until another directive is read. The second kind of directive is written

s q

This causes the machine to stop reading tape and start obeying the orders of the program, beginning with that in location q and

Items terminated either with C-r

or with 2 spaces.

Can have 1 space anywhere between symbols

" " erase " (any no.)

Don't have blank tape or line feed within

items, but as many as you like between items

proceeding serially in the usual way.

A directive must be terminated by a carriage return and a line feed, just like an order, so that it would appear in the printed copy as a single line of print.

As an example, the written form of Example 9, including the directives needed to put it in locations 100 - 109 and then begin obeying it, is as follows:

```
p1 = 100
59 f 10
19 f 2
59 f 27
59 f 12
13 f 2
19 f 2
11 f 2
59 f 12
59 f 27
101 f 0
s 100
```

The punched form would consist of the same symbols (with single spaces if desired), each line, including the last, being terminated by a carriage return and a line feed.

14. Input of numbers.

We have so far been concerned with the input of numbers during the operation of the program, and for this we have used the order 59 f 10. It is sometimes convenient to read numbers into the store at the same time as the program, by means of the program input routine. For example, if values of a function are to be evaluated by means of a polynomial approximation to it, the values of the coefficients have to be read in at some point, and it is easiest and most convenient to read them in with the rest of the program.

The program input routine can read numbers as well as orders. The numbers must be punched in a suitable style; this is described below, and is the same as that required to read a number by the order 59 f 10. The storage registers into which numbers are read are controlled by a directive p1 = q, just as with orders; a succesion of numbers will be read into successive registers. It is indeed possible to mingle numbers and orders. The program

Location	Content	Operand	F(Acc)
100	10 f 200	a	a
101	14 f 202	b	ab
102	12 f 108	12 6	ab+12 6
103	14 f 106	π	π(ab+12) 6
104	19 f 252	π (ab+12) 6	π(ab+12)6
105	50 f 156		π(ab+12)6
106	3·14159265		
107			
108	12 6		
109			

The tape entries would consist of the items in the column headed 'Content', each followed by the symbols for carriage return and line feed; thus if the input tape were used to give a printed copy on a teleprinter, these items would appear on successive lines, as written in this column.

One advantage of this way of reading numbers which are known before the calculation is started, as part of the program, is that no orders are needed for this in the program; another advantage will be mentioned in Section 19. A further way of reading numbers in the course of the input of a program will be described in Section 20 (a).

15. Output of numbers.

It has been pointed out in Section 11 that a number may be punched on the output tape by the special order

59 f 27

The number thus punched is in floating decimal form, with seven significant figures; that is, when the output tape is used to give a printed copy on a teleprinter, the number will appear as: minus sign or space (according as the number is negative or positive), first digit, decimal point, remaining digits, suffix $_{10}$ symbol, exponent. The exponent is an integer of one or two* digits preceded, in the case of a negative exponent, by a minus sign.

The results of a calculation are printed by passing the output tape through a tape reader connected to a teleprinter. If the results consist of more than two or three numbers, the

---

* If the exponent is zero it is omitted altogether, the number ending with the suffix $_{10}$.

layout of these numbers on the printed page must be programmed in order to provide them in a readable form. For this purpose we must insert, in the part of the program concerned with the output of results, orders for punching on the output tape the symbols for space, carriage return and line feed. These orders are as follows:

107 f 2 punch on the output tape the symbol for carriage return
107 f 8 " " " " " " " " line feed
107 f 30 " " " " " " " " space.

Note that when carriage return and line feed are both required the carriage return must precede the line feed, and not vice versa.

PART 2

16. Further orders in the order code.

This section introduces some further orders in the order code of EDSAC 2.

(a) An order which is often useful is:

8 f q exchange the content of the accumulator with that of storage register q.

(b) The following are orders, similar to those with function numbers 10 to 13, but involving the modulus of the content of a specified storage register:-

20 f q set |F(q)| in the accumulator
21 f q set -|F(q)| in the accumulator
22 f q add |F(q)| to the content of the accumulator
23 f q subtract |F(q)| from the content of the accumulator

(c) The following are additional jump orders:-

51 f q jump to q and clear the accumulator
52 f q jump to q if the content of the accumulator is zero, otherwise proceed serially
53 f q jump to q if the content of the accumulator is not zero, otherwise proceed serially.

Since the orders with function numbers 10, 11, 20 and 21 provide means of setting the content of any storage register in the accumulator, it is seldom necessary to clear the accumulator; if this has to be done, a 51 order provides one way of doing it.

(d) The following are additional modifier orders:

72 s q add the number q to s
73 s q subtract the number q from s
79 s q set the address in the order in location q equal to s
80 s q set s equal to the address in the order in location q.

Similar orders, containing t instead of s, concern the t register.

Orders with function numbers 79 and 80 are useful for temporary storage and replacement of the content of a modifier register if this register is required for another purpose in the course of a calculation.

(e) It is sometimes necessary to halt the machine temporarily, for instance to give time for a new tape to be put in the tape reader. For this purpose we have the order:

102 f 0 wait until the machine is manually restarted.

17. Subroutines.

Since a calculation has to be broken down into very simple individual operations, it will normally need a great many of them. Instead of considering the program as a whole, it is therefore convenient to break it up into groups of orders, each doing some self-contained job such as a quadrature or the multiplication of two matrices. Such groups of orders are called subroutines. Thus the group of orders required to evaluate a power series (Example 7) could be regarded as a subroutine, though it is really too short for this to be worth while.

The main reason for thinking in terms of subroutines is that this enables a program to be broken up into pieces small enough to be grasped but, at the same time, self-contained enough to be checked individually. Quite a number of jobs of the sort one would do in a subroutine turn up time and again. It would obviously be a waste of time if each programmer had to devise a set of orders for doing them: suitable sets of orders are therefore put in a library, so that one can merely copy them and can rely on their not containing any mistakes. Such copying can be carried out and verified mechanically; all that the user has to do is to take the original tape from the library and place it in the copying and verifying equipment.

It may happen that we wish to use a subroutine at several different points in a program. It is then convenient to place it in the store in some position separate from the main program and, on each occasion on which the subroutine is used, to enter it by a jump order (sometimes called the cue to the subroutine) and to return from it by another jump order (called the link). If the cue is in location r, it is most convenient if the link produces a jump to (r+1); then, from the point of view of the programmer, the cue which results in the operation of the subroutine is a

single order which is placed in the sequence of orders just like an order for any other operation.

To simplify and standardise the process of entering and leaving subroutines, there are in the order code of EDSAC 2 two special orders; their function numbers are 58 and 60. Subroutines using these orders for cue and link are called closed subroutines. For most purposes the programmer need not know the exact specifications of the 58 and 60 orders. Provided that (i) within a closed subroutine he does not use register 0 nor call in another closed subroutine, and (ii) if the subroutine changes the value of t he does not use after the subroutine the value of t set before it, then all he needs is the partial description:

58 f q enter the closed subroutine of which the first order is in location q;

60 f 0 return from a closed subroutine to the location immediately following the cue order 58 for entry to the subroutine, restoring s to the value it had when the 58 order was obeyed.

(Note that, by contrast to everything earlier, s and t are NOT treated symmetrically. It follows that the s register is freely available to the programmer within the subroutine, but the t register only if he can afford to destroy its previous content.)
However, in some contexts in which these orders may be used, it may be advisable to know rather more about them, and for this reason they are considered further in the next section.

All library subroutines are closed in the sense just defined; and even if a subroutine is only going to be used once, the programmer will usually find it most convenient to write his program as a number of closed subroutines called in by a brief master routine. It is then easy to check the subroutines, and the master routine is short enough to be error-free; the waste of time and orders involved is negligible.

18. Entering and leaving a closed subroutine.

The orders which are used to enter and leave a closed subroutine are as follows:

58 f q if r is the location of this order, make C(0) = 0 f s and C(1) = 0 f (r+1), and jump to q;

60 f 0 set the value of s equal to the address part of C(0), and jump to the address specified by the address part of C(1).

The 58 order provides the cue for entry to a closed subroutine, and the 60 order provides the link for return to the main program. Provided that the contents of locations 0 and 1 (that is, of register 0) have not been disturbed during the subroutine, the effect of the 60 order is to reset s to the value which it had on entry to the subroutine, and to return to the main program at the order immediately after the cue to the subroutine. Since the address (r+1) for the jump back is set by the cue order, the link order is the same wherever in the store the subroutine is placed. It should be noted that, in contrast to what happens with other orders, the s and t registers are treated unsymmetrically. The content of the s register is stored at the beginning, and recovered at the end of a closed subroutine; but nothing is done about t. Thus, if the t register is used by the subroutine, the original value of t is lost.

Since register 0 is used by the 58 and 60 orders, it should not normally be used for any other purpose. Care is also needed if in the course of one closed subroutine it is necessary to call in another: in this case, the contents of register 0 after entering the first subroutine (i.e. its link with the main program) must be put somewhere else in the store before the second subroutine is entered, and put back in 0 again afterwards in order to return to the main program. A typical sequence of orders for this would be

```
                        10 f 0 ⎤   transfer C(0) and C(1) to accumulator
                        19 f 2 ⎦   and store them in locations 2 and 3.
                        58 f q     jump to second subroutine.
Second subroutine ——→   10 f 2 ⎤   recover original contents of locations
                        19 f 0 ⎦   0, 1, and put them back in 0, 1.
```

19. Parametric addresses.

In many sequences of orders, the addresses in one or more orders depend on the locations of others. For instance, in the set of orders (1) in Example 5 (Section 8), the address 103 in the order in location 111 is the address specifying the location of another order in the sequence, and depends on the position in the store in which these orders are placed. Other instances of this can be seen in Examples 4, 6, 7 and 8; it has so far only occurred with jump orders, but there are other contexts in which it may occur. Further, it has so far been supposed that the addresses of all registers containing numbers involved in the calculation have been determined beforehand.

If +ve write p followed by no.

−ve " m " " "

e.g. 17 p70 m30 m30

$= 17 + p_{70} - 2p_{30}$ modulo 2048

The result is, in effect, that a programmer cannot write a sequence of orders until he has decided where in the store these orders, and the numbers on which they operate, are to be placed; and this can be very inconvenient. For a subroutine intended for the library it can be disastrous, since a feature of a library subroutine should be that it can be put in any position in the store without alteration.

We require, therefore, a means of referring to locations of orders, and to registers holding numbers, which does not require that the programmer should know the addresses of these locations or registers at the time he programs the sequence of orders referring to them. At some later stage in the process of programming these addresses must be determined, but the process of programming a long or involved calculation is very much simplified if this step can be postponed. Further, such a facility enables a group of orders to be tested before it has been decided where in the store these orders, the numbers on which they operate, or any other groups of orders referred to, shall be placed in the complete program of which these orders will eventually form a part.

On EDSAC 2 this facility is provided by the use of symbols termed <u>parameters</u> in the addresses in the written (and punched) form of orders; each such parameter stands for an integer which is the address, or a contribution to the address, in an order. A parameter is written (and punched) as the letter p followed by a number which identifies it, for example

p12

Note that the number 12 here <u>identifies</u> the parameter and is NOT its <u>value</u>; The combination p12 must be regarded as a single symbol, denoting a single number which is the address, or a contribution to the address, in an order. It is often convenient to use several independent parameters in a single program. There are in fact 97 parameters available to the programmer, written as p3 to p99 inclusive. There are also parameters p1 and p2, but these should not be used by the programmer. It is possible to form an address by the addition of a known number and a parameter; this is expressed in the written (and punched) form of an order by writing the parameter symbol immediately after the number. An address containing a parameter is called a <u>parametric address</u>.

<u>Example 11</u>. As an example of the use of parameters, suppose that when drawing up the orders (1) of Example 5 (Section 8), it had

not been decided in which register of the store the number x should be stored, or where the records of the current values of x^n and S_n in the course of the calculation should be kept, but it had been decided that the latter two records should be kept in successive registers. Then if we denote by q the address of the register which is to contain x, and by b the address of the register which is to contain x^n, we should want the first three orders of this group to be

```
10 f q
19 f b
19 f b+2
```

If we use the parameter symbol p20 to denote the value of q, and p21 to denote the value of b, these three orders, in the form appropriate to the machine, would be

```
10 f p20
19 f p21      (a)
19 f 2p21
```

the address 2p21 in the third of these orders being the form used for writing the address

2 + (value of parameter p21).

The conversion from parametric addresses to absolute addresses is carried out by the program input routine. On different occasions on which a group of orders using parametric addresses is used, different values may be given to the parameters; but on any one occasion the parameters will have definite values, which must be specified during the input of the program so that they can be used for this conversion. There are two ways of specifying the value of a parameter, one explicit and the other implicit.

The value of a parameter can be set explicitly by a directive. The directive to set the parameter p3 to the value q is

```
p3 = q
```

For instance, if in Example 11 the values of the parameters p20 and p21 are set by the directives

```
p20 = 200
p21 = 202
```

the orders (a) of this example would be placed in the store as

```
10 f 200
19 f 202
19 f 204
```

which are the first three orders of the sequence (1) of Example 5.

Forward reference successful if parameter is +ve and whole address of order - coped with by input routine Can only use in the address part of order, not as directive.

Directive = 10 followed by c.r. unsets parameters set implicitly from p 10 --- p 99. Explicitly set parameters can be overridden by directive p ?=? e.g.

s 100 Checks that there are no unfilled forward references

s+ 100 just starts at 100

Alternatively, a parameter may be set implicitly, by attaching a label to an order, the label being the number which identifies the parameter to be used in referring to the location of this order. The label is written by following the relevant order with an opening bracket and the identifying number of the parameter to be set, thus:

14 f 200 (3

The value of the parameter p3 will then be set to the address of the location in the store into which this order is placed by the program input routine.

Example 12. The orders of Example 8 (Section 10) in a form independent of position in the store.

The sequence of orders in its original form, designed to be put into locations 100 to 104, is

Location	Order
100	70 s 80
101	10 f 380
102	14 f 200
103	12 f 298
104	75 s 102

The last of these is the only one in which the address depends on the position of this group of orders in the store. If we label the order 14 f 200 by (say) the label (3, the location of this order, to which the 75 order jumps back, can be denoted by the parameter p3, so that the orders take the form

(p3)-2	70 s 80
(p3)-1	10 f 380
(p3)	14 f 200 (3
(p3)+1	12 f 298
(p3)+2	75 s p3

and in this form this group of orders is independent of the position in the store which it may occupy. It is still necessary that these orders should occupy successive locations, but the use of parametric addresses frees the programmer from having to consider what particular sequence of locations shall be used. It has the further advantage that if, at a later stage of programming, it is found that some orders have to be inserted in the middle of a program already drafted, this can be done without requiring any renumbering of addresses in orders already written. It also enables different parts of a long or complicated program to be largely 'uncoupled' from one another during programming, while retaining full freedom

of cross-reference between them; their detailed placing in the store can be considered separately, and later.

The replacement of a parameter symbol by its numerical value in a particular case is done, as already stated, by the program input routine. It might seem that the value of a parameter must be set, explicitly or implicitly, before this parameter is used. However, if a parameter is the whole of the address in an order (as often happens when a parameter is used), then it will be correctly interpreted even though the value of that parameter is only set at a later stage of the input process.

In Example 12 the item labelled (3 is an order. It is also possible to set a value of a parameter by labelling, in the same way, a number which is read in the course of the input of a program (but not one read in by means of a 59 f 10 order).

Example 13. Example 10 with parametric addresses for the numbers π and 12.6

The required sequence of tape entries is:

```
10 f 200
14 f 202
12 f p5
14 f p4
19 f 252
50 f 156
3.14159265(4
12(5  6(5
```

Notes. (1) In this form, the orders are independent of their position in the store.

(2) The address in the third order of this group must not be written 2p4 since this is a forward reference and, as such must consist of a parameter alone (see above).

20. Additional facilities of the program input routine.

(a) Listing of constants.

When a program is being drafted, it happens quite often that we wish to use some specific number as the operand of an instruction. For example we may wish to divide the number in the accumulator by 360 to convert degrees into revolutions. The program input routine makes it possible to do this in a simple manner. The programmer merely writes the number itself (instead of the address

where h is the interval of integration in the independent variable x; and c+4, c+10,... are used as working space during the operation. The contents of c+4, c+10,... must be cleared (this may be done most simply by repeated use of the 9 order) before the integration is started, and must not subsequently be altered by the programmer. The auxiliary subroutine must be so written that, given any numbers y_0, y_1,... in registers c,c+6,... it evaluates hf_0, hf_1,... and places them in registers c+2, c+8,...; in addition, on every exit from it t must be 2n. The subroutine must therefore contain, at the end for preference, the order 70 t 2n. If this is done, the t register as well as the s register is available for the programmer to use in the auxiliary.

It is also necessary that on reaching the order 58 f 17 we should have s = c; the simplest way to ensure this (since a closed subroutine restores the value of s) is to have the order 70 s c immediately before the order 58 f q. Thus the natural entry to the integration subroutine will be

```
70 s c    set s = c
58 f q    jump to auxiliary subroutine
59 f 17   advance integration one step.
```

(Note that the auxiliary will have been obeyed once before the order 59 f 17 is reached: this order must not be reached by a jump from elsewhere in the program.)

Example 14. Auxiliary subroutine for the set of equations

$$dy_0/dx = y_1y_2,\ dy_1/dx = -y_0y_2,\ dy_2/dx = -k^2y_0y_1.$$

Suppose F(4) = h, F(8) = k^2, F(10) = y_0 (i.e. c = 10); then F(16) = y_1, F(22) = y_2. We evaluate and plant the values of hdy_0/dx, hdy_1/dx and hdy_2/dx in succession.

Order	Operand	F(Acc)	
10 f 16	y_1	y_1	
14 f 22	y_2	$y_1y_2 = dy_0/dx$	
14 f 4	h	hdy_0/dx	
19 f 12	hdy_0/dx	hdy_0/dx	plant hdy_0/dx in (c+2) = 12
11 f 10	y_0	$-y_0$	
14 f 22	y_2	$-y_0y_2 = dy_1/dx$	
14 f 4	h	hdy_1/dx	
19 f 18	hdy_1/dx	hdy_1/dx	plant hdy_1/dx in (c+8) = 18
11 f 10	y_0	$-y_0$	
14 f 16	y_1	$-y_0y_1$	
14 f 8	k^2	$-k^2y_0y_1 = dy_2/dx$	
14 f 4	h	hdy_2/dx	
19 f 24	hdy_2/dx	hdy_2/dx	plant hdy_2/dx in (c+14) = 24
70 t 6			3 equations, set t = 2n = 6
60 f 0			link.

An integration will usually require some preliminary operations for setting initial values of the y_i's, clearing the set of registers c+4, c+10,..., setting the value of h, and perhaps reading in numerical data such as, in this example, the value of k^2 for which the integration is carried out. It is often convenient to combine these into a separate subroutine which is called in before the integration process is started. The complete program will also have to contain some orders controlling the number of steps of integration carried out, and the stages at which results are punched.

On completion of the order 59 f 17, the contents of registers c, c+6, c+12,... are the values of y_0, y_1, y_2,... at the current stage of the integration; but the contents of registers c+2, c+8, c+14,... are <u>not</u> in general the corresponding values of hdy_0/dx, hdy_1/dx,... though they will be approximations to them. If values of the derivatives are required (for example for punching on the output tape) they must be calculated by the auxiliary, which is entered and left in the ordinary way for a closed subroutine (see Section 17).

If the value of any of the derivatives dy_i/dx depends explicitly on x, the most convenient procedure in many cases is to include among the dependent variables one (say y_0) which is x itself, or a constant multiple of it, and which therefore satisfies the equation

$$dy_0/dx = C,$$

so that $hdy_0/dx = hC$. If h is going to be kept constant throughout an integration, the value of hC can be placed in register (c+2) as part of the preliminaries to the integration itself; it will not be disturbed by the integration process, and does not need to be reset by the auxiliary. It may be convenient to take $C = 1/h$; then $hC = 1$ and $hy_o = x$.

If h is changed in the course of an integration, but all values which are used are multiples of some number h_o (which may, for example, be 0·1 or 0·01 or 0·005) it may be convenient to take $C = 1/h_o$.

Similarly it may be convenient to evaluate other functions, occurring in the differential equation to be solved, by integration of one or more auxiliary differential equations; for example Bessel functions by integration of Bessel's equation

$$d^2y/dx^2 + (1/x)dy/dx + (1 - n^2/x^2)y = 0 \qquad (3)$$

APPENDIX 1 EDSAC 2 teleprinter code

Code	Figure Shift	Letter Shift
00.000	No effect	
00.001	f	F
00.010	Carriage return	
00.011	0	O
00.100	r	R
00.101	7	K
00.110	2	U
00.111	s	S
01.000	Line feed	
01.001	8	L
01.010	5	H
01.011	/	A
01.100	9	M
01.101	(	Z
01.110	=	E
01.111	10	D
10.000	p	P
10.001	4	G
10.010	3	Y
10.011	2	W
10.100	6	J
10.101	t	T
10.110	n	N
10.111	-	B
11.000	1	I
11.001	*	C
11.010	)	X
11.011	Letter shift	
11.100	.	V
11.101	+	Q
11.110	Space	
11.111	Figure shift	

Note: Tapes are punched in accordance with the left-hand column, a hole being punched for each 1. The full stop in each row represents the sprocket hole, which is always punched.

APPENDIX 2 Summary of that part of the EDSAC 2 Order Code used in this booklet.

8 f q exchange the content of the accumulator with that of storage register q.

9 f q clear storage register q; that is to say, set F(q) = 0.

10 f q place in the accumulator the number in storage register q.

11 f q place in the accumulator minus the number in storage register q.

12 f q add to the number in the accumulator the number in storage register q.

13 f q subtract from the number in the accumulator the number in storage register q.

14 f q multiply the number in the accumulator by the number in storage register q.

15 f q divide the number in the accumulator by the number in storage register q.

19 f q copy the number in the accumulator into storage register q.

20 f q set $|F(q)|$ in the accumulator.

21 f q set $-|F(q)|$ " " " .

22 f q add $|F(q)|$ to the content of the accumulator.

23 f q subtract $|F(q)|$ from the content of the accumulator.

50 f q jump to q.

51 f q jump to q and clear the accumulator.

52 f q " " q if the content of the accumulator is zero; otherwise proceed serially.

53 f q jump to q if the content of the accumulator is <u>not</u> zero; otherwise proceed serially.

54 f q jump to q if F(Acc) $\geqslant 0$; otherwise proceed serially.

55 f q " " q " F(Acc) < 0; " " " " .

58 f q enter the closed subroutine of which the first order is in location q. (If r is the location of this order, make C(0) = 0 f s and C(1) = 0 f (r+1) and jump to q.)

59 f 10 read a number from the input tape and place it in the accumulator.

59 f 11 place $x^{\frac{1}{2}}$ in the accumulator.

59 f 12 " e^{x} " " " .

59 f 13 " $\log_e x$ in the accumulator.

59 f 14 " $\sin x$ " " " .

59 f 15 " $\cos x$ " " " .

x = original F(Acc).

59 f 17 advance integration of differential equations by one step.

59 f 27 punch on the output tape the number now standing in the accumulator (in floating decimal form, to 7 significant figures). This does not affect the content of the accumulator.

60 f 0 return from a closed subroutine to the location immediately following the cue order 58 for entry to the subroutine, restoring s to the value it had when the 58 order was obeyed.

(set the value of s equal to the address part of C(0), and jump to the address specified by the address part of C(1).

70 s q put the integer q in the s register.

71 s q put the integer -q in the s register.

72 s q add the integer q to s.

73 s q subtract the integer q from s.

74 s q increase s by 2; if the new value of s is 0 proceed serially, if not jump to q.

75 s q decrease s by 2; if the new value of s is 0 proceed serially, if not jump to q.

79 s q set the address in the order in location q equal to s.

80 s q set s equal to the address in the order in location q.

101 f 0 stop the machine and light the stop warning lamp.

102 f 0 wait until the machine is manually restarted.

107 f 2 punch on the output tape the symbol for carriage return.

107 f 8 " " " " " " " " line feed.

107 f 30 " " " " " " " " space.

APPENDIX 3

Programming Exercises I

Construct sequences of orders to carry out the following operations on EDSAC 2. The orders may be placed anywhere in the store, but their locations should be indicated. Registers 2, 4, 6,... may be used for temporary storage of constants if required.

Sequences of orders should preferably be capable of repeated application.

For sets A and B,

x, y, & z are in registers 100, 102, & 104 respectively;

a, b, & c are in registers 110, 112, & 114 respectively.

Set A

1. Place x+y in register 120.
2. Place 2x-y in register 122.
3. Place x^2 in register 124.
4. Place y^3 in register 126.
5. Place x^2+y^3 in register 128 and x^2-y^3 in register 130.
6. Given that π^2 is in register 118 place $(x^2+y^2)/\pi^2$ in register 120.
7. Place xy+yz+zx in register 122.
8. Place $ax^2+2bxy+cz^2$ in register 124.
9. Place ax^2+bx+c in register 126.
10. Place cx^2 in 120 and bx^2 in register 122.

Set B

1. Place $-|x|$ in register 120.
2. Place $|x-y|$ in register 122.
3. Place the greater of x and y in register 106.
4. Place the greater of F(106) and z in register 106.
5. Replace x by y if $y > z$.
6. The number in register 118 is either 1/2 or 1/3. Whichever is there replace it by the other.
7. Replace the number in 118 by the number in 118 subtracted from (1/2+1/3).
8. Write a sequence of orders which cause a jump to location 300 the first time they are encountered, to location 400 the second time, and so on alternately.
9. If $F(100) < 0$ place F(104) in register 106; if $F(100) > 0$ put F(108) in register 106. In either case reverse the sign of F(100). What does your program do if F(100) = 0? Modify

it, if necessary, so that it stops in this case.

10. Given a, b, c, d, e, f in registers 110 to 120, place u and v in registers 130 and 132 where

$$au+bv+c = 0$$

$$du+ev+f = 0.$$

Set C

1. Clear registers 100 to 198 inclusive.
2. Place in register 4 the sum of the squares of the numbers stored in registers 100 to 198 inclusive.
3. Replace the numbers stored in registers 100 to 198 by those stored in registers 1000 to 1098 respectively.
4. Place the number which is stored in register 4 in the first register which is now clear after register 100. Then clear the following register.
5. Test the numbers in registers 200, 202, 204, etc. until one is found which is greater than 1/2 but less than 1. Place this number in register 6.
6. Given that register 10 contains x, place x^{13} in register 4 using the fewest possible orders. Repeat the programming trying to minimise the time taken. Assume that multiplication orders take ten times as long as any other order.
7. Place in register 100 the largest of the numbers in registers 100 to 198 and place the number from 100 in the register which previously held the largest number.
8. Evaluate and place in register 6 the number
   $a_o+a_1x+a_2x^2+a_3x^3+....+a_{10}x^{10}$
   where x is stored in register 8 and the coefficients a_n are stored in registers 120-2n.
9. Evaluate $1+4x+9x^2+16x^3+...+n^2x^{n-1}+...+100x^9$ without holding all the coefficients in the store.

Programming Exercises II

Write complete programs, including any necessary directives, for the following examples. It may be assumed that a separate data tape will be placed in the tape reader after the program has been read and placed in the store. The program should contain a wait order to allow time for this tape handling. The program should be terminated by a stop order.

Set A

1. Print $(3.142)^2+72.85$.
2. Print $(\sqrt{2}-1)^2$.
3. Print exp(3·14159).
4. Print cos33° and sin33°.
5. Read x from the data tape and print $\log_e(x-\sqrt{x})$.
6. Read x and y from the data tape. Print
   $x\cos(\pi y)$ and $y\sin(\pi x)$.
7. Print $\exp[(1/5)\log x]$ where x is read from the data tape.
8. The data tape contains a group of positive numbers, terminated by a negative number. Print the largest of the positive numbers.
9. Read x and y from the data tape. Let $u_0 = 1$, $u_1 = 1$, and
   $u_{n+1} = xu_n+yu_{n-1}$.
   Print u_{100}/u_{99}.
10. Read x and y from the data tape. $u_0 = x$, $v_0 = y$; and
    $u_{n+1} = (u_n+v_n)/2$,
    $v_{n+1} = \sqrt{u_n v_n}$.
    Print u_{10} and v_{10}.
11. Using the same relations as in Example 10, print u_N where $N \geq 1$ is the smallest value for which $v_N \geq u_N$.

Set B

1. Read 16 numbers x_n from the data tape.
   Print $x_0+x_1\cos12°+x_2\cos24°+\ldots+x_{15}\cos180°$.
2. Read 16 numbers x_n from the input tape.
   Print y_m, m = 1(1)5, where $y_m = \sum_0^{15} x_n\cos(nm12)°$.
3. Print the solutions of $ax^2+bx+c = 0$, where a, b, c are read in sequence from the input tape.
4. Print the sum $1 + 1/2 + 1/3 + 1/4 + \ldots+1/1000-\log_e 1000$
5. Evaluate and print $(1 + 1/4)(1 - 1/9)(1 + 1/16)(1 - 1/25)\ldots(1+1/10{,}000)$.
6. Print the sums
   $1 + x + x^2/2 + x^3/1.2.3 + x^4/1.2.3.4. +\ldots+ x^n/1.2\ldots n +\ldots$
   for x = 1(·2)2. Terms which contribute less than 10^{-10} to the sum should be neglected.
7. The times of departure and arrival of trains from Cambridge to King's Cross and from Cambridge to Liverpool Street, expressed in minutes after midnight, are given on a data tape. The data are expressed in pairs of numbers, the first giving time of departure and the second time of arrival. The trains are in order of departure time, with all the King's Cross trains first. The end of the tape is marked by a negative number. Using the order 59 f 24 for printing, print a timetable expressed in minutes giving the times of departure of <u>useful</u> trains, i.e. those which take less than 2 hours and are not passed by another train during the journey.

INDEX