

AN INTRODUCTION TO THE PROGRAMMING OF A DIGITAL COMPUTER
WITH PARTICULAR REFERENCE TO THE BRUSH AUTOMATIC CODING
SCHEME FOR THE ELLIOTT 803 COMPUTER

by
D. C. Hogg

SUMMARY:- The basic concepts used in working with a digital computer are explained. Enough of the basic facilities of the Brush automatic coding scheme (H-Code) are described to enable programmes to be written. Examples are given. The document serves as an Introduction to the H-Code manual.

August, 1962

BRUSH ELECTRICAL ENGINEERING CO., LTD.
A MEMBER OF THE HAWKER SIDDELEY GROUP
LOUGHBOROUGH, ENGLAND

Contents

1	<u>COMPUTERS</u>	1
2	<u>INTRODUCTION TO PROGRAMMING</u>	1
2.1	PROGRAMME	1
2.2	TACKLING THE PROBLEM	1
3	<u>THE BRUSH AUTOMATIC CODING SCHEME - H-CODE</u>	2
3.1	STORAGE OF NUMBERS	2
3.2	THE NAMING OF PROGRAMMES	2
3.3	THE PROGRAM	2
3.4	ARITHMETIC	2
3.5	REFERENCE NUMBERS AND JUMP INSTRUCTIONS	3
3.6	INPUT AND OUTPUT OF NUMBERS	4
3.7	LOOPS OF INSTRUCTIONS	4
3.8	AUTOMATIC SELECTION OF NUMBERS;MODIFICATION	4
3.9	SUBROUTINES	5
4	<u>EXAMPLES</u>	5

List of Figures

1	Example of flow diagram	2
2	Subroutine flow diagram	5
3	The use of a subroutine	5
4	Copy of teleprinter for example 1.	6
5	The flow diagram for example 1.	7
6	Copy of teleprinter for example 2.	8
7	The flow diagram for example 2.	9

DISTRIBUTION

Dr. L.R.Blake
Dr. D.A.Jones
Research Division Library
Central Reference Library
Chief Engineers

and As Required

AN INTRODUCTION TO THE PROGRAMMING OF A DIGITAL COMPUTER
WITH PARTICULAR REFERENCE TO THE BRUSH AUTOMATIC CODING
SCHEME FOR THE ELLIOTT 803 COMPUTER.

by
D. C. Hogg.

1. COMPUTERS

We are concerned with automatic electronic digital computers. A digital computer handles numbers in the form of a series of digits, such as one does when doing a calculation by hand. For example, the number 53 might be represented by a gearwheel turned through five teeth, and another turned through three teeth or, alternatively, by two trains of pulses containing 5 and 3 pulses respectively. The precision of such machines can be increased indefinitely merely by allowing for a sufficiently large number of digits in the number.

This is in contrast to analogue computers in which numbers are represented by some physical quantity, such as length, angle or electrical potential, and in which arithmetical operations on such numbers are performed by using some law of Physics, e.g. Ohm's Law, and then making a measurement to find the answer. A slide rule is an example of an analogue computer: here the logarithms of numbers are proportional to lengths; these lengths are added and subtracted mechanically to give lengths proportional to the logarithms of products and quotients.

The accuracy of an analogue computer is limited by the precision with which the physical quantity used can be measured; whereas the accuracy of a digital computer depends only on the number of digits allocated to a number in the machine. This facility which allows calculation to a large number of significant figures can be most important. Even if the raw data of a problem is given only to three decimal digits and the results are only required to that accuracy, rounding-off errors (e.g. subtraction of two large nearly equal quantities) may build up invalidating the final result. Most electronic digital computers work with numbers of 11 decimal digits accuracy.

The drawback with a digital computer is that all operations take place serially and, therefore, calculation time may in certain cases build up appreciably in spite of the very fast speed of electronic computation. Furthermore, in order to solve a problem using digital computer, the mathematical solution has, in general, to be known completely whereas analogue computers only require the formulation of the problem.

2. INTRODUCTION TO PROGRAMMING

2.1. PROGRAMME

A digital computer is a general-purpose machine. It is prepared for a particular job by feeding into it a programme, or list of instructions. This programme is stored within the memory or store of the computer. It is important to realise that it is only when a computer has got a programme within it, that it becomes a usable machine.

Because it is the programme which makes a computer do a particular job, it follows that it is only necessary to feed in a different programme to prepare the computer for a different type of work; there is no need to make any changes to the computer itself. This is one reason why it is possible for so many people to do such a variety of work on one machine. No matter what the machine was doing previously, it is immediately set up to do your particular job when you feed in your programme.

When a programme is fed into a computer, the whole programme goes in and is stored before the computer begins to

obey the instructions of the programme. The instructions are NOT, repeat NOT, obeyed as they are read in. This is a very important idea, even though it is quite simple. Each programme is designed to do a particular type of work. When the programme has been stored in the computer, the instructions are then obeyed one by one, and one of the first instructions will be to read and store the data for the particular job being done. Thus the same programme may be used several times with different sets of data, without reading in the programme each time.

The computer, having stored the data it requires, then continues to work through the instructions of the programme, taking items of stored information, producing and storing intermediate results and finally producing the answers. These are then punched on to paper tape for subsequent printing on a teleprinter page, attention being paid to the layout in columns and the insertion of headings where desirable.

2.2. TACKLING THE PROBLEM

It is important to realise that a computer is a pure automaton. It has no intelligence, no initiative and no capacity for dealing with unforeseen circumstances. This means that the programme must be written in such a way that every possibility is catered for and the programmer must foresee any exceptional cases that could arise.

When preparing a problem (programming) for a computer the procedure is as follows:—

- Stage (i) The problem and the solution of the problem suitable for computing, must first be precisely defined. In mathematical problems this is probably done in the mathematical statement of the problem, but in other problems, particularly those involving logical decisions, this first stage may, in fact, be a considerable part of the work.
- (ii) We must assign a letter of the alphabet and numerical suffix to each variable which occurs in the calculation. (see Section 3.1)
- (iii) The steps of the problem are written down in the form of a flow diagram (see below)
- (iv) The flow diagram has to be converted to a form (or code) that the computer can understand. Using the Brush Automatic Coding Scheme, this is done quite simply, using a code which resembles normal mathematical and logical statements.
- (v) When the programme has been written, it has to be put into a form which can be "read" by the computer. This is done with a teleprinter which produces a punched paper tape and a typed sheet with a copy of the programme.

Each time a number, letter or symbol is printed, a corresponding combination of up to 5 holes is punched across the tape, and these holes can be sensed by the photoelectric tape-readers on the computer. The information on this programme tape is read in by the computer, automatically converted to a form suitable to it, and stored.

The numbers involved in the calculation (data) are similarly punched on a paper tape, referred to as the data tape. This is usually a separate tape, but it may be attached to the end of the programme tape.

EXAMPLE OF FLOW DIAGRAM.

The computer is to read a list of numbers (A), one by one, form and print $1/\sqrt{A}$ and stop when a negative number is encountered. If $A = 0$ then print "No solution" and read the next number.

A possible flow diagram for this very trivial problem would be:

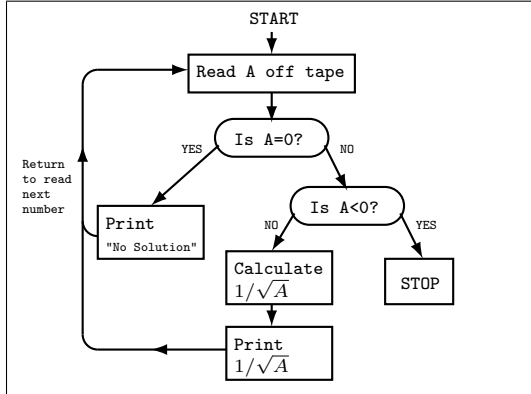


Figure 1: Example of flow diagram

3. THE BRUSH AUTOMATIC CODING SCHEME - H-CODE

3.1. STORAGE OF NUMBERS

The operation of the computer can be visualised by thinking of it as a huge collection of pigeon-holes called storage-locations or simply locations. Problems are solved on the computer by simple basic arithmetical and logical operations on data which has been stored in convenient locations.

In the Brush automatic coding scheme (H-Code) the stored numbers are identified symbolically. While the programme is being written it is inappropriate to consider the values of the numbers, they will in fact be different each time the programme is used. A symbolic form is therefore required and it is, of course, similar to simple algebraic notation.

A number in a location is referred to by any letter of the alphabet together with a numerical suffix (if needed) e.g.

A_{13}, B_{16}, C, Z_2 .

C is understood to be equivalent to C_0 .

$A, A_1, A_2, \dots, A_{13}$ etc., etc., will be consecutive locations in the machine. It is not up to the programmer to allocate particular locations in the machine to a particular letter. This is done automatically. It is sufficient to tell the machine that $A_0 - A_{45}, B_0 - B_{50}$ etc., are required and the computer will do the actual allocating.

There are two types of numbers used by the computer:-

(a) Floating-point variables or simply variables

These usually refer to physical quantities etc., and are of any form, integral, fractional, or mixed numbers within the approximate range $\pm 10^{77} - \pm 10^{-77}$

(b) Fixed-point variables, suffices, integer-variables, or simply integers

These are used mainly for counting and organisation of the programme. They are whole numbers i.e. integers and exist within the approximate range $0 - \pm 10^{12}$

Except under special circumstances, variables and integers cannot be mixed in arithmetic statements.

¹N.B. (i) * is used for the multiplication sign. As well as distinguishing it from a letter x it is the teleprinter symbol for X (presumably for the same reason).

It is now apparent that not only do we have to tell the machine how many locations we have to use for a particular letter but also whether the letter is to represent a variable or an integer.

The actual instructions in H-Code which appear at the beginning of the programme are, for example

```
:SETV (A50, B3, C, X1)
:SETS (N13, L2, T)
```

On reading these the computer will allocate 51 locations to be identified as $A, A_1, A_2, \dots, A_{50}$ and to be understood to be variables. Similarly for B, C and X .

Under SETS the letters will similarly identify locations which will hold integers.

If a letter is set to be a variable it cannot also be an integer and vice-versa.

However, as explained above, it is of no concern to the programmer which locations are actually allocated to the variables or integers. His only concern is that a number placed in location A_{10} (say) in one part of the programme will have the same value when A_{10} is referred to later, unless changed by a step in the programme itself.

3.2. THE NAMING OF PROGRAMMES

In H-Code a number of independently written programmes (or procedures) can be used together, with the restriction that all use the same variables and integers.

In order that we can refer to and use one programme inside another, we give to each procedure a name (consisting of letters only).

So that the computer is prepared to understand reference to a procedure and to receive a procedure when put into the computer, we have to set the procedure names in the same way as we have set the variables and integers. Example:

```
:SETR (LTV, DESIGN 6, ENDW 3)
```

The number appearing after the name is a reference number. This is explained in 3.5

3.3. THE PROGRAM

The set statements (SETV, SETS, SETR) are merely directives to the machine to reserve locations for programmes or procedures of given names written with specified variables and integers.

After the sets, the Computer is in a position to receive the programme. This programme it translates and stores in a form which it can subsequently be obeyed.

Each procedure is headed by a statement indicating to the machine the name of the procedure. It is written e.g.

```
:DEFINE (DESIGN)
```

The actual orders of the procedure DESIGN then follow.

The name in the DEFINE statement will have been set in the SETR instruction, as will all the subsequent procedures.

3.4. ARITHMETIC

In the computer arithmetical operations (+, -, *, /, See note(i) below¹) take place only between numbers in the storage locations and the number in a special location called an accumulator. For example, if we want to add two numbers together, one of the numbers must first be transferred from the store to the accumulator, and the other added to it. The result of any operation is always in the accumulator.

In order that the computer be efficiently and quickly programmed, the form of the arithmetic in H-Code follows this principle of accumulator working, so that if we write the following statement in our programme:

$$A + B * C \rightarrow Y$$

the computer translates it operation by operation viz.

A Bring $c(A)$ into ACC (see note (ii) below)
 +B Add $c(B)$ to ACC giving $c(A) + c(B)$ in ACC
 *C Multiply $c(ACC)$ by $c(C)$ giving $[c(A) + c(B)] * c(C)$ in ACC
 $\rightarrow Y$ Send the $c(ACC)$ to Y giving $c(Y) = [c(A) + c(B)] * c(C)$

i.e. in mathematical terms we have written a programme to calculate

$$Y = (A + B)C$$

As we can see, this does not correspond exactly to standard mathematical conventions, but if desired we can put in suitable brackets i.e.

$$(A + B) * C \rightarrow Y$$

These brackets are ignored by the computer but they must occur in pairs. This principle of accumulator working is extended to functions so that if we wanted an order in a programme to calculate

$$Y = Ae^{\sqrt{BX+C}}$$

we would write

$$(B * X + C) : \text{SQRT} : \text{EXP} * A \rightarrow Y$$

which, step by step, is:-

(Ignored
 B $c(B) \rightarrow ACC$
 *X $c(ACC) * c(X) \rightarrow ACC$
 +C $c(ACC) + c(C) \rightarrow ACC$
) Ignored
 :SQRT Square root of $c(ACC) \rightarrow ACC$
 :EXP Exponential of $c(ACC) \rightarrow ACC$
 *A $c(ACC) * c(A) \rightarrow ACC$
 $\rightarrow Y$ $c(ACC) \rightarrow Y$

(We put a colon (:) before words in the programme to distinguish English from letters which refer to locations)

This simple method of indicating to the computer the mathematical formulae it has to calculate can be applied to innumerable equations. However, if we have an equation such as:-

$$Y = ab + cd$$

it cannot be programmed in one expression with the facilities mentioned so far. This is because after multiplying a and b together, the result is in the accumulator and this has to be added to the product of c and d. Clearly, this second multiplication has to be performed using the accumulator, therefore the first product has to be stored away temporarily. In our programme, we can write this as:-

$$A * B + (C * D) \rightarrow Y$$

In this case the brackets are most important and indicate to the computer that the quantity in brackets must be computed separately.

step by step this is:-

A $c(A) \rightarrow ACC$
 *B $c(ACC) * c(B) \rightarrow ACC$
 +($c(ACC) \rightarrow$ Special storage location X_1
 C $c(C) \rightarrow ACC$
 *D $c(ACC) * c(D) \rightarrow ACC$
) $c(ACC) + c(X_1) \rightarrow ACC$
 $\rightarrow Y$ $c(ACC) \rightarrow Y$

We can extend this facility to any depth effectively e.g. the instruction to calculate:-

$$Y = A/\sqrt{AB + CD}$$

is written

$$A / ((A * B + (C * D)) : \text{SQRT}) \rightarrow Y$$

The outer pair of brackets are necessary to indicate that the contents are to be calculated separately.

The inner pair of brackets are necessary to indicate that $C * D$ is to be computed separately.

But the brackets around $A * B + (C * D)$ are not necessary as the result of this computation is in the accumulator and we can immediately take the square root of it. They are merely put in to "clarify" the statement.

If a pair of brackets is immediately preceded by + - * or / then the expression between the brackets is computed separately. If not preceded by one of these operators then the pair of brackets are ignored.

N.B. Note that in the preceding examples, and those that follow wherever we have used A, B, C etc, we could as well have used:

$$A_{13}, B_{56}, C_2, \dots \text{etc.}$$

Also variables can be replaced by mixed number constants e.g. 167.63, 123.0 etc.

Integers can be replaced by integer constants e.g. 3, 176 etc. Numbers which are variables should always have a decimal point. Numbers which are integers should never have a decimal point.

3.5. REFERENCE NUMBERS AND JUMP INSTRUCTIONS

In a programme, the instructions are usually obeyed in the sequence in which they are written. The jump instructions, however, enable this sequence to be broken and can specify which instruction is to be obeyed next at a particular point in the programme.

To do this, it is necessary to be able to identify the instruction to which the jump has to go. This is done by labelling. A label or reference number is a positive integer constant written with a bracket, before the instruction e.g.

$$3) A + B \rightarrow Y$$

The simplest jump instruction is written:e.g.

$$: \text{GOTO } 7$$

This means that the next instruction to be obeyed is that with reference number 7.

Jump instructions may be made conditional on some equality or inequality e.g. we can write:

$$: \text{IF } (A + B > X : \text{SIN}) \text{ GOTO } 9$$

This means "obey the instruction with reference number 9 if a + b is greater than sin x and continue the sequence from there, otherwise go on to the next instruction as usual".

Any arithmetical quantities can be compared and > can be replaced by:-

$$<, =, \neq, \leq, \geq.$$

²(ii) $c(A)$ means "the contents of location A". ACC stands for "the accumulator" and $c(ACC)$ "the contents of the accumulator".

In the SETR instruction which appears at the head of the programme, we have to suffix each routine name with the highest reference number or label which is going to occur in that particular routine. This is so that the computer can reserve locations for the organisation of the reference numbers.

Each individual routine has its own reference numbers, i.e. if reference 3 occurs in a particular routine it has no connection with reference 3 in another routine.

3.6. INPUT AND OUTPUT OF NUMBERS

The Instruction

```
:READ (A,B,C,D,E)
```

is used to read numbers punched on paper tape. When it is obeyed the computer will read the first number on the data tape and store it in location A, the second in B etc.

When the computer has finished the calculation it has to communicate its results to the outside world. It punches the requisite numbers on to paper tape which is then fed into a teleprinter which produces a typewritten copy of the numbers.

The order in the programme which instructs the computer to do this is typified by:

```
C : 2.4 or even (B + A) : 5.3
```

in the second example (B + A) is formed in the accumulator and :5.3 signifies that the number in the accumulator is to be punched out with up to 5 digits before the decimal point and 3 after it.

Any characters which appear in the body of the programme between inverted commas e.g.

```
'LBS/FT.3'
```

will be punched on the output tape when the order is obeyed. This facility is normally used in conjunction with the above number print in order to give explanation and layout to the numerical answers.

3.7. LOOPS OF INSTRUCTIONS

An essential feature of using a digital computer is that repetitive processes are used. In this way, a group of instructions which are only written once may be used repeatedly during the calculation, probably operating on different numbers each time.

In H-Code the instructions (or rather pairs of instructions) which facilitate the use of repetitive processes are, for example:—

```
(i)
:LOOP
    } programme to be obeyed
    } repetitively
:REPEAT
```

The programme between the :LOOP and :REPEAT is obeyed repetitively. That is after the :REPEAT control goes back automatically to the instruction immediately following the :LOOP instruction.

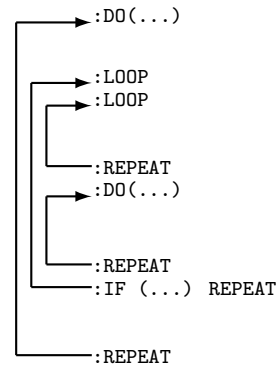
```
(ii)
:LOOP
    } programme to be obeyed
    } repetitively
:IF (A<C) REPEAT
```

This is similar to (i) but the :REPEAT is conditional. The range of conditions is the same as a conditional :GOTO instruction.

```
(iii)
:DO (5)
    } Programme to be obeyed
    } 5 times
:REPEAT
```

The programme between the :DO and :REPEAT is obeyed 5 times or more generally, obeyed the number of times specified by the :DO instruction. The :REPEAT must be unconditional.

Loops can occur with each other and are paired as shown in the following example:—



A :REPEAT is paired with the last unpaired :LOOP or :DO. Other programme can occur anywhere between the :DOs, :LOOPS, and :REPEATs.

3.8. AUTOMATIC SELECTION OF NUMBERS;MODIFICATION

Often during a computation, we require to work systematically through sets of numbers stored as variables in the computer. To do this, we require the methods of programming mentioned in the previous section but we also need to be able to select in turn a succession of variables, referring to a different variable each time we go round the loop.

This technique is made possible by being able to write:

A_N (punched as AN or A (N))

instead of e.g. A_3 , when referring to a variable. When the programme is obeyed the computer uses the current value of N to determine the particular A which is to be used.

So if we write:

$A_N \rightarrow X$

and on obeying this order $N = 3$ then the actual order obeyed is

$A_3 \rightarrow X$

A can be a variable or an integer, but N must be an integer.

An integer used in this way is called a modifier.

To obtain a succession of values the N (say) is modified inside the loop so that A_N will refer to another variable next time around.

This technique of modification is one of the most valuable features available in a computer.

To illustrate this, we give an example:

Form $C_i = A_i + B_i$, for $i = 0(1)99$ i.e. i takes all the values from 0 to 99 in steps of 1.

The orders which will do this are:

```
0 → I
:DO (100)
AI + BI → CI
I + 1 → I
:REPEAT
```

Initially $I = 0$ so that the order:—

```
AI + BI → CI
```

will be:-

$A0 + B0 \rightarrow C0$

Before the loop is repeated I is increased by 1 so that the second time round the loop $I = 1$ and the order is:-

$A1 + B1 \rightarrow C1$

This is repeated 100 times. When the loop is obeyed for the last time $I = 99$ so that the order is:-

$A99 + B99 \rightarrow C99$

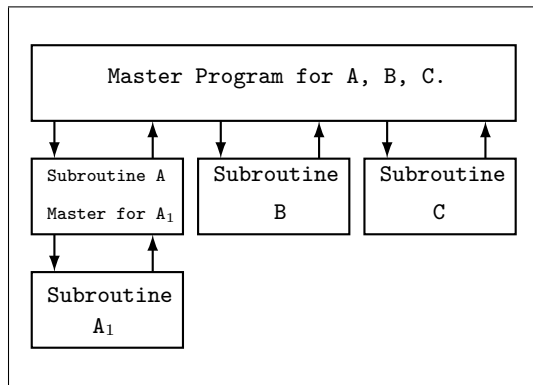


Figure 2: Subroutine flow diagram

3.9. SUBROUTINES

When writing a programme it is often convenient to split up the program into a number of distinct parts, each of which carries out a process which is more or less an independent operation. We might also want to incorporate into our program a procedure which has been written by someone else, for example, a procedure to solve a set of linear equations whose coefficients have arisen in the main programme. Procedures or programmes which are used in this way are called subroutines. Subroutines can of course use other subroutines themselves etc. A programme which uses the subroutine is called its master. A flow diagram incorporating subroutines might look like fig. 2.

We have to make sure before the master enters a subroutine that the variables which the subroutine requires for its calculation have been set. The subroutine must not ruin any information which the master might want retained.

When a subroutine is being written, the writer, in general, does not know in what context the subroutine is going to be used, so that he cannot put an instruction to pass control-back to a particular master programme. The return must therefore be general. In H-code we write the instruction

`:EXIT`

In the subroutine. This has the effect of passing control back to its master, whatever programme that master might be.

If we wanted to use the procedure XYZ as a subroutine and start obeying it at reference 3 (say) we would write in the master the instruction

`:XYZ3`

When the instruction `:EXIT` is obeyed in the subroutine control passes back to the master and the computer obeys next the instruction immediately after the instruction `:XYZ3`. This is illustrated in fig.3

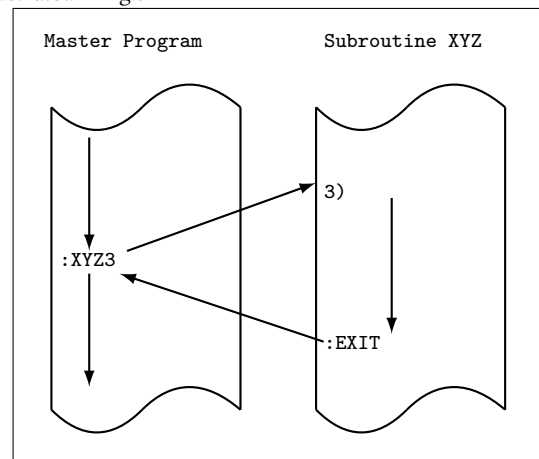


Figure 3: The use of a subroutine

4. EXAMPLES

Enough of H-Code has been introduced to enable us to give an example of the programming of two simple problems.

(It should be emphasised that only a minimum of the available facilities have been introduced. The more powerful facilities, and a list of all facilities, are described in the H-Code Manual).

A. The programme sheet.

```
:SETV(A,B,C,D1)
:SETS(S)
:SETR(SOLUTION OF QUADRATIC (5))

:DEFINE(SOLUTION)
0 →S
1):READ(A,B,C)
'
'S+1 →S:5
:IF(A=0.0)GOTO4
B*B-(A*C*4) →D
:IF(D<0.0)GOTO3
:IF(D=0.0)GOTO2

D:SQRT →D1
' REAL      '(-B+D1)/(A*2):4.5'      '(-B-D1)/(A*2):4.5,:GOTO1
2)
' EQUAL REAL  '-B/(A*2):4.5,:GOTO1
3)
' COMPLEX     '-B/(A*2):4.5'      '-D:SQRT/(A*2):4.5,:GOTO1
4)
:IF(B=0.0)GOTO5
' SINGLE ROOT  '-C/B:4.5,:GOTO1
5)
' NO SOLUTION':GOTO1

:CLOSE
:START (0,SOLUTION)
::
```

B. The data sheet with 5 sets of coefficients.

```
0  4.321  3.8506
2  4  2
1  -2.145  -3.564
1.0  2.0  2.0
0  0  3.6
)
```

C. The result sheet.

```
1  SINGLE ROOT  -0.89114
2  EQUAL REAL   -1.00000
3  REAL         3.24373      -1.09873
4  COMPLEX      -1.00000      1.00000
5  NO SOLUTION
```

Figure 4: Copy of teleprinter for example 1.

EXAMPLE 1.

The object of this is to solve a series of quadratic equations whose coefficients are punched on tape. The roots, real and imaginary, are printed in rows preceded by the serial number of the equation.

As the problem must be precisely defined, the following information is given:-

- The coefficients of the general quadratic equation $ax^2 + bx + c = 0$ will be given on tape, in the order a,b,c.
- if the equation has two different real roots we are to print "REAL" followed by the roots:-

$$\frac{-b + \sqrt{b^2 - 4ac}}{2a}, \frac{-b - \sqrt{b^2 - 4ac}}{2a}$$

If equal roots print "EQUAL REAL" followed by the double roots: $\frac{-b}{2a}$

If $a = 0$ print "SINGLE ROOT" followed by solution of $bx + c = 0$ i.e. $-c/b$ unless in addition $b = 0$ when print "NO SOLUTION".

If complex roots print "COMPLEX" followed by real and imaginary parts.

$$\frac{-b}{2a}, \frac{\sqrt{4ac - b^2}}{2a}$$
- Each equation is to be preceded by the serial number of the equation.

Parameters:

As only capital letters are available on the teleprinter, we have to allocate a letter (and numerical suffix if necessary) to each variable and integer to be used in the computation. We shall use:-

Variable	A	for	a
"	B	"	b
"	C	"	c
"	D	"	$b^2 - 4ac = D$
"	D1	"	\sqrt{D}

Integer S for s the serial number of the current set of coefficients.

The flow diagram:

Shown in fig. 5

Notes on the print-out

The following points have not been explained in the text. Referring to fig. 4:-

- Although we have referred to the programme as both SOLUTION OF QUADRATIC and SOLUTION, in fact the computer only reads up to the first 4 letters of a name, any further letters being ignored.

Instead of using a separate line for each order we can separate them with a comma. In the case of a title instruction the comma is implied.

The instruction :CLOSE indicates that the procedure it follows is terminated.

The :START Instruction indicates that the first order to be obeyed is that following reference 0 in procedure

SOLUTION. The first order of a procedure is addressed as reference 0.

The :: at the end of the programme sheet is a device to stop the computer reading more tape.

- The) at the end of the data sheet tells the computer, in reading data, to stop,

Each item of data should be terminated by at least 2 spaces or a new line.

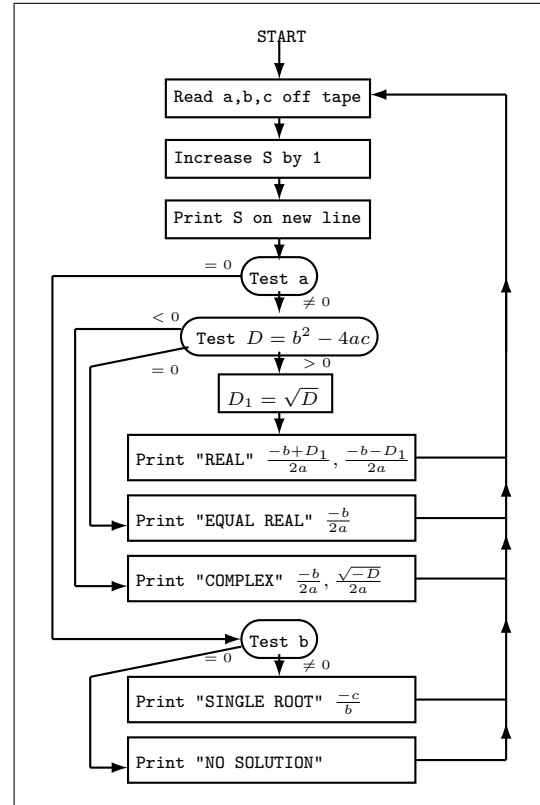


Figure 5: The flow diagram for example 1.

A.B. In 3.4 it was stated that numbers which are variables should always have a decimal point. Although this is a rule which is always correct, there are situations when it can be violated. For example, when a number follows a variable the computer knows that the number represents a floating point quantity therefore we can write $A*2$ but we can not write $2*A$.

In data the variable in the :READ instruction determines whether the data is to be fixed or floating point, therefore no decimal point is necessary.

But if in doubt, use the decimal point.

EXAMPLE 2.

This example illustrates the use of a standard procedure as a subroutine.

The object of the problem is to find the value of the integral

$$z = \int_a^b \frac{1}{d^2 - x^2} dx$$

for given values of a, b, d.

A. The programme sheet.

```

:SETR(MASTER,SIMPSON,AUXILIARY)
:SETV(Y2,A,B,T,H1,S,E,X,Z1,D)
:SETS(C)

:DEFINE(AUXILIARY)
1.0/(D*D-(X*X)) →Y,:EXIT
:CLOSE

:DEFINE(MASTER)
:READ(D,A,B)
'
      ERROR      NO. OF INTERVALS      VALUE OF INTEGRAL
'
0.1 →E,:DO(8)
:SIMPSON
'
'E:3.8'      'C:4 '      'Z:2.8
E/10 →E,:REPEAT
((D+B)*(D-A)/((D-B)*(D+A)))LOG/(D*2) →Z
'

      VALUE OF INTEGRAL BY ANALYTICAL SOLUTION

' Z:2.8
:STOP
:CLOSE

:DEFINE(SIMPSON)
A →X,:AUXILIARY,Y →Y1, B →X,:AUXILIARY,Y →Y2, (A+B)/2 →X,:AUXILIARY,Y →T
(B-A)/2 →H,(T*4+Y1+Y2)*H/3 →Z1,0.0 →S,1 →C
:LOOP
S+T →S,H/2 →H →H1,Z1 →Z,0.0 →T
:DO(C+C →C)
A+H1 →X,:AUXILIARY,Y:ON(T)
(H+H)ON(H1)
:REPEAT
:IF(((T*4+S+S+Y1+Y2)*H/3 →Z1-Z)MOD>E)REPEAT
:EXIT
:CLOSE

:START(0,MASTER)
::

```

B. The data sheet.

1	0	.5
---	---	----

C. The result sheet.

ERROR	NO. OF INTERVALS	VALUE OF INTEGRAL
0.10000000	2	0.55000000
0.01000000	2	0.55000000
0.00100000	2	0.55000000
0.00010000	4	0.54936267
0.00001000	8	0.54931002
0.00000100	16	0.54930639
0.00000010	32	0.54930616
0.00000001	64	0.54930614
VALUE OF INTEGRAL BY ANALYTICAL SOLUTION		
		0.54930614

Figure 6: Copy of teleprinter for example 2.

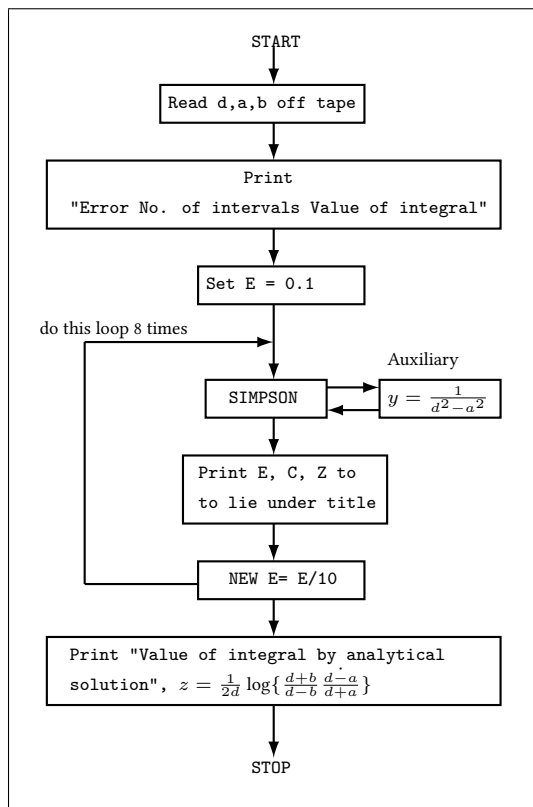


Figure 7: The flow diagram for example 2.

Although this integral has an analytical solution, namely

$$z = \frac{1}{2d} \log_e \left\{ \frac{d+b}{d-b} \frac{d-a}{d+a} \right\}$$

we shall for the sake of interest calculate it by using this result and also by using a standard procedure which calculates an integral by a method based on Simpson's rule. To use this procedure as a subroutine we do not have to know how it works, it is sufficient to be given the following information:-

The name of the procedure is SIMPSON.

It has no references.

It uses the variables A,B,Y,Y1,Y2,T,H,H1,S,E,X,Z,Z1, and the integer C.

It is entered at reference 0 with the following variables set:-

A,B as the limits of integration.

E as the accuracy to which the integration is to be performed.

It is obvious that SIMPSON must also be able to calculate the dependent from the independent variable, therefore we must define a subroutine which calculates y given x where

$$z = \int_a^b y dx$$

SIMPSON becomes the master of this subroutine. A subroutine used in this way is called an auxiliary.

SIMPSON does not need to know the interval of integration. It calculates its own interval in order to obtain the required accuracy.

On leaving SIMPSON Z is the value of the integral and C is the number of intervals into which SIMPSON divides the range of integration to obtain the required accuracy.

To illustrate how the number of intervals increases with the increase in accuracy required we are asked to print out the number of intervals and the value of the integral for the error E = .1, .01, .001,00000001. Finally we shall print out the value of the integral calculated analytically.

Parameters:

We shall use variables and suffices corresponding to those in the mathematical statement of the problem.

The flow diagram

Shown in fig. 7

Notes on the print-out:

Referring to fig. 6:-

A. The function LOG is $\log_e(x)$. Other functions available include SIN, COS, TAN, ARCTAN, EXP, MOD (the modulus), FRAC (The fractional part).

The instruction :STOP is self-explanatory.

Note that the procedures can be 'written' in any order as the order of appearance in the programme has no relation to the order in which they are obeyed.