CONFERENCE ON AUTOMATIC PROGRAMMING OF DIGITAL COMPUTERS

MATHEMATICS DEPARTMENT

BRIGHTON TECHNICAL COLLEGE

April 1, 2, 3, 1959

FURTHER DEUCE INTERPRETATIVE PROGRAMMES AND SOME TRANSLATING PROGRAMMES

by

S. J. M. DENISON

(The English Electric Co. Ltd.)

Further DEUCE Interpretative Programmes and some Translating Programmes.

S. J. M. Denison.

1. INTRODUCTION.

The object of automatic programme is, of course, to reduce the effort required to write programmes, and this is achieved by using, instead of the computer's own code of instructions, a pseudo-code which is usually closer to the programmer's habitual way of describing the operations which he wants the computer to perform. (The word 'programmer' is used here rather in the sense of 'anyone wishing to write a programme', than that of 'someone specially trained to write programmes'). A programme having been written in a pseudo-code, there are two ways of making it produce the desired result, viz, by interpretation or by translation (or compiling, as it is often called).

In the first method, each codeword is interpreted as it is reached in the course of the calculation, and many of them will therefore be interpreted many times. Since the interpretations occupy computer time additional to that required for the actual computation, interpretative schemes produce programmes which are slower than those written in the machine-code, although in some applications, e.g. in 'G.I.P.' and 'T.I.P.' (Ref. 1), the amount of computation specified by each codeword is usually so great that the interpretation time is negligible. Another feature of pseudo-codes which tends to produce slow programmes when the interpretative method is used is that they do not fully reflect the storage structure of the computer. It is, of course, desirable that a pseudo-code should be as free as possible from this structure, since it is alien to the conventional description of calculations, etc. On the other hand, the structure is designed mainly to achieve as high a speed of operation as possible for a given overall cost of the computer, and the neglect of it must cause a reduction in speed.

To satisfy both these requirements, which are incompatible in any interpretative scheme, and to avoid the repeated interpretation of codewords, we can use the computer to translate, once for all, a programme written in a pseudo-code free from computer-determined structure into a fast programme of computer instructions, taking the storage structure fully into account. The problems which are presented in the writing of a programme for making such a translation are, however, far from trivial.

In many interpretative schemes, some translation is also done, i.e. the original codewords are first translated into others with properties nearer to those of the machine-code, e.g. symbolic addresses may be converted to absolute computer addresses. (We, at English Electric, usually reserve the term 'compiling' for this process, for convenience, although it is not essentially different from translation, as defined in the previous paragraph). A 'compiler', i.e. a programme for doing this preliminary work, is much easier to write than a 'translater', and may save a good deal of interpretation time.

Descriptions will be given of four interpretative schemes which have been prepared for DEUCE, one of which has been referred to by Mr. Robinson (Ref. 1). In each case, some compiling is done on the codewords before they are interpreted. The pseudo-code used in the first of these schemes, called 'GEORGE', is unusual, being in fact an extension of a notation ('reverse Polish') suggested for mathematics (Ref. 2). The second scheme, 'Alphacode', was inspired by the 'Manchester Autocode' (Ref. 3), but attempts to make every codeword a statement in plain English. The third scheme, 'Steve', employs a special-purpose pseudo-code, similar in form to those used by 'G.I.P.' and 'T.I.P.', but intended solely for calculations on the thermo-physical properties of steam and water. The fourth scheme, 'Easicode', is a general-purpose scheme with a form of compiled instruction giving rapid interpretation.

There follows a description of an existing translating programme whose pseudo-code is known as 'Soda'. Both this and 'Easicode' retain some of the storage structure of DEUCE, which has in some degree facilitated their writing but which makes the writing of programmes in these pseudo-codes rather more difficult than in 'Alphacode', for example.

Finally, some of the difficulties which arise in translating from a pseudo-code which is free from computer-determined structure are discussed, together with some of the techniques which are to be used in a programme for translating from 'Alphacode' to DEUCE machine-code.

2. 'GEORGE', the 'General Order Generator'.

In conventional mathematical notation, symbols denoting operations on two numbers are often placed between them, e.g. a + b, a ÷ b, while many symbols for operations on one number are placed in front of it, e.g. log a, sin θ. The Polish notation unifies this by placing all operational symbols in front of the number or numbers, while the reverse Polish notation places all operational symbols behind it or them. Thus, in the reverse Polish notation one would write, for example, ab+, ab÷ a log, θ sin. It is found that these notations make the use of brackets hardly necessary, since each operational symbol defines the number of numbers either behind it or in front of it which must be associated with it. For instance, in bc+ax, the symbol + obviously operates on b and c, while the symbol x operates on a and the result of bc+, i.e. the corresponding expression in conventional notation would be (b + c) x a. The conventional expression from which this is distinguished by the brackets, viz b + c x a, can be denoted, in reverse Polish, by bcax+. This notation is, however, not unique, since besides interchanging c and a, one can also write this expression as caxb+, just as one could write the conventional expression c x a + b.

Further examples are:-

Conventional Notation.	Reverse Polish Notation.
log {(a+b) ÷ (c+d)}	ab+cd+÷ log
a + b + c + d	ab+c+d+
	or ab+cd++
	or abcd+++
2 sin (p+q) cos (p+q)	pq+ sin pq+ cos x2x
	or pq+ sin pq+ cos 2xx

Now, an advantage of the reverse Polish notation is that it can be regarded as a computer pseudo-code, since expressions can be worked through steadily from left to right, each symbol being interpreted as an instruction, and this is in fact the basis of the 'GEORGE' pseudo-code. Part of the computer is regarded as a 'running accumulator', that is, a succession of cells, each capable of holding one number, through which all internal operations are performed. A symbol representing a number, e.g. a, is interpreted as 'fetch this number into the next vacant cell of the running accumulator'. (It is assumed that the symbol has previously been given a numerical value by reading in data). A monadic operator, i.e. one operating on a single number, is interpreted as 'perform this operation on the last number in the running accumulator and overwrite the number with the result', while a diadic operator, i.e. one operating on two numbers, is interpreted as 'perform this operation on the last two numbers in the running accumulator, taken in order, overwrite the first number with the result and clear the cell containing the second number'. Thus, the result of interpreting any expression is that the numerical value of the expression is to be found in the first vacant cell of the running accumulator, all the following cells having been cleared.

In 'GEORGE', this basic notation is extended to include operations on suffixed variables, repeated operations, discriminations, the reading in and punching out of numbers, the storing of results, etc., the properties of the notation being preserved as far as possible. For example, a doubly-suffixed symbol, a_{ij} is regarded as a diadic operation on the numbers i and j, is written ij//a , (//a being treated as one symbol), and interpreted as 'fetch i into the next vacant cell of the running accumulator, fetch j into the following one, overwrite i by the element (i, j) of the array defined by //a and clear the cell containing j'. This facilitates the specification of arithmetic operations on the suffices, e.g. $a_{i+1,\ j-1}$ is written as i1+j1-//a , which can be interpreted using the specified rules.

As was mentioned in the introduction, some compiling is done on the symbols before they are interpreted; they are in fact translated into 'keywords', which are DEUCE instructions leading to stored routines for carrying out the operations specified. Computation is carried through with floating-point numbers, so that the programmer need not concern himself with their size.

This scheme provides a very compact way of writing a programme and, being addressless, has none of the computer's own structure in it, but has the disadvantage that the notation is unnatural.

---

3. '<u>ALPHACODE</u>'.

The chief aim in designing this pseudo-code was to enable someone not familiar with computers to write programmes, and to this end the instructions take the form, as far as possible, of statements in plain English, free from computer jargon, but aided by mathematical symbols. In the first version of the scheme, all quantities involved in the computation were referred to as X's with suffices, and counters were provided, e.g. for specifying how many times a part of the calculation should be repeated; these were denoted by N's with suffices. A later version, to speed up the interpretation, introduced the option of referring to temporarily used quantities, e.g. intermediate results in evaluating formulae, as suffixed T's, but a programmer can ignore this if he wishes. (The suffices are usually written in line with the letters, for convenience.)

An arithmetic operation is written, as, (for example):

```
         X3  =  X1  PLUS  X2
         X5  =  X3 MINUS  X4
     or  X6  =       ROOT X5
```

Trigonometrical and other operations are included, and are written in the form:

```
         X2  =       SIN   X1
         X6  =       LOG   X4
         X4  =      TANH   X7
```

Repetitions of a group of instructions are specified by writing after the group an instruction like:

```
         Count   N1   UP TO   10,   jumping to R2.
```

and putting the reference number (2 in this case) to the left of the first instruction of the group. The effect of this particular instruction would be to repeat the instructions between the reference 2 and itself 10 times and then go on to the next instruction. (The instructions are normally obeyed in the order in which they are written down). The reason for naming a counter, N1, to be used for this operation is that this counter can then be used to modify instructions in the loop. For instance, we may write:

```
         N1  and  N2   MODIFY next instruction.   N3 also.

         X201  =  X1   PLUS                       X101
```

which will cause the suffices of the X's in the second instruction to be modified by the current values of the corresponding counters in the modifying instruction. (The same counter may be used to modify the suffices of two or all three X's).

It is admitted that instruction modification is a concept peculiar to the use of computers and it has been found that it does cause some difficulty to non-specialist programmers learning 'Alphacode'; it is a limitation of this sort of pseudo-code.

It will be noticed that some instructions contain several words and that, in the example just given, one or two of these are in capitals. The words in capitals are those essential for defning the operations, and are in fact used by DEUCE for this, while the others complete the sense. The instructions are punched on Hollerith cards for reading into DEUCE, one per card, to simplify the changing of instructions or the insertion of others, and a 'pulling-file' is used from which a prepunched card for any instruction can be drawn, it being then necessary only to punch the X's, N's, R's etc. Thus the programmer can specify the operation he requires in an instruction in any (unambiguous) way he chooses, e.g. he can write PLUS or +, he can write the extra words or leave them out; as long as the person punching the cards knows which to draw from the pulling-file, the operation will be specified to DEUCE in standard form. The words which complete the sense are included in the prepunched cards, so that a punched programme can be printed in an intelligible form. Appendix 1 shows such a printed programme, produced by a card-controlled typewriter. The constant shown on the second line of the programme is 100 in floating decimal ($1{\cdot}0 \times 10^2$), and the function 'AS BIG' represents $\geqslant$. The aims of this pseudo-code have obviously not been fully realised. (Where the word defining an operation has more than six letters, only the first six are used by DEUCE. That is why only these are printed as capitals).

This appendix also shows some of the more powerful instructions included in the code, together with the method of specifying a subroutine of codewords such as that referred to in the instruction for solving a set of differential equations. It is assumed that the equations are of the first order and degree, with only the derivative on the left-hand side; the subroutine specified in the instruction must calculate the values of the right-hand sides, given the current values of the variables. Instructions are also included for carrying out arithmetic and other operations on complex numbers. They are designed in such a way that each pair of real numbers forming a complex number is specified as a single entity.

A feature of the scheme is that the programmer has the option of punching out intermediate results to help him test his programme. The result of obeying an arithmetic or trigonometric function, etc., will be punched out if a 'P' is written to the right of it, provided that a key on the DEUCE control panel is in the depressed position while the calculation is being performed. When testing is complete, the programme is run with the key in its normal position and, with no alteration to the programme, only the final results are produced.

---

4. 'STEVE'.

This interpretative scheme is an example of one made specially for a particular sort of calculation, viz. that involving the thermo-physical properties of steam and water, which occurs frequently in problems associated with steam turbines and heat exchangers. The pseudo-code used is very much like that used by 'T.I.P.' i.e. each instruction consists basically of four parts, a, b, c and r; a and b in general specify data upon which the function specified by r operates, and c indicates where the result is to be stored. As in 'T.I.P.', the values of r correspond to a fixed set of functions, but a, b and c refer to single numbers. Many of the functions are, of course, for calculating properties of steam or water under given conditions, but ordinary arithmetic functions are also provided, together with counting, discrimination, input, output and other facilities. As in 'Alphacode', instructions can be given references to avoid absolute addressing purpose for the purpose of jumping out of sequence, and an optional stopping or punching parameter can be included to aid programme testing.

Special attention was paid in designing 'STEVE' to its behaviour after a failure during either testing or use, the latter often occurring through faulty data preparation or through trying to stretch the scheme beyond its limits. For example, division by zero may be called for, or an attempt made to calculate some property of steam under conditions where steam would not exist. After a failure, cards are punched out showing the codeword at which it oocurred, its number and the values of the quantities a and b, so that the programmer can fairly easily discover the error even though he may not have been present during the use of the programme. An indication is then given to the operator that a failure has occurred, and if this happens too frequently the programme may be taken off the computer. Finally, a jump is made to a special reference which the programmer can use as he thinks best, e.g. to punch out partial results and go on to the next case.

This scheme, in common with those already described, uses floating-point arithmetic.

---

5. 'EASICODE'.

Like 'G.I.P.', 'T.I.P.', 'Alphacode' and 'Steve', this scheme employs a basic 3-address pseudo-code, but whereas the others assume the 'pseudo-computer' to have virtually a single level of storage capacity, 'EASICODE' takes account of the storage structure of DEUCE (Ref. 1, section 2). 6 blocks of stores, named A to F, in the rapid-access store, are made available to the programmer, each comprising 32 one-word stores. (The blocks are, in fact, 6 delay-lines). 192 blocks of backing stores (tracks on the magnetic drum) are also available. The subroutines required for the most commonly used arithmetic and organisational functions are permanently available in the rapid-access store, but those required for additional functions must be chosen by the programmer from the ordinary DEUCE subroutine library, read in with the programme and stored on the drum; they are then brought into the rapid-access store as required, by instructions in the programme. All computation is in fixed-point arithmetic, all numbers other than those used for counting being assumed to have their modulus less than 1 and provision has been made for scaling numbers up or down by 2 or 10 during the computation. The decimal codewords are read in, one per card as for 'Alphacode', but are compiled into DEUCE instructions leading to appropriate subroutines, like the 'keywords' associated with 'GEORGE'. These arrangements have produced a scheme which is much faster in operation than 'Alphacode', but which requires more skill and care in writing programmes.

Codewords are basically of the form f(x, y) z, though the function may be organisational, e.g. for fetching a subroutine from the drum, instead of mathematical. The function is represented by a number between 1 and 96, numbers 1 to 32 being reserved for those permanently available. x, y and z normally represent addresses in the rapid-access store, by the block letter followed by a number between 1 and 2, but in codewords causing transfers on the drum or between the drum and the rapid-acoess store they may represent a block on the drum, by its number, or a whole block in the rapid-access store, by the block letter followed by 00. A check for the compatibility of f with x, y, and z is incorporated. An example of a simple arithmetic codeword is:

```
f      x      y      z
1      A01    F29    B07
```

which would cause the number in store 1 of block A to be multiplied by that in store 29 of block F and the result put in store 7 of block B. A transfer from the drum to the rapid-access store would have the form:

```
f      x      y      z
19     101    -      A00
```

The codewords are normally obeyed in the order in which they are read in, conditional and unconditional jumps being provided as usual and specified by a symbolic next address (S.N.A.) and corresponding symbolic location (S.L.) on the codeword to which the jump is made. These symbols are numbers between 1 and 96.

Loop-counting and modification of instructions are catered for, modification is specified in the same codeword as the instruction modified; and can take two different forms. An example of the first is:

f	x	y	z	M_1	M_2	M_3
4	A02	C20	A15	010	010	010

the effect of which is to subtract each of the 10 numbers starting with C20 from the corresponding numbers in the string of 10 starting at A02 and to put the results in A15 onwards. If any M is left blank, the corresponding address is unmodified, and the same number is used in each calculation of the string. An example of the second form of modification is:

f	x	y	z	M_1	M_2	M_3
3	F32	A01	E32	B05	-	B05

When this codeword is obeyed, the number (assumed to be an integer) in B05 is added, modulo 32, to the addresses corresponding to M_1 and M_3, viz. x and z, e.g. if B05 contains 6, the codeword will cause F6 to be added to A01 and the result put in E6. (All three addresses can be modified independently if necessary). Where the first form of modification can be used, it produces the required results much more quickly than the repeated use of the second form.

Provision is made for reading in and punching out strings of consecutive numbers to or from a block of the rapid-access store, and to or from one or more complete blocks on the drum. From 1 to 6 numbers may be punched on the same card.

The programme-testing facilities provided include stopping the programme anywhere and, in particular, at a codeword whose number has been set on the control panel; jumping to an instruction not next in sequence; and punching out intermediate results. All these, and also the failure indications, have been arranged so that no knowledge of the binary notation is required.

Appendix 2 shows the functions which have been allocated fixed numbers in the present version of 'Easicode'.

6. AN EXISTING TRANSLATING PROGRAMME.

The pseudo-code, 'SODA', associated with this programme (Ref. 4) assumes a pseudo-computer in some respects like DEUCE itself. It has a 'short accumulator' a 'lower accumulator' and an 'upper accumulator', the two latter being combined in some instructions to form a 'double accumulator'. (c.f. Ref. 1, Section 2). Arrays of data are stored on the drum, but access to them is through two 'working stores' which can usually be regarded as independent storage units of indefinite size, though for some purposes it must be recognized that each consists of only 32 positions and that data are automatically transferred to or from these positions at appropriate times. Constants are also stored on the drum, and there is similar access to them. Arithmetic and logical operations take place between a number in one of the accumulators and either a number in an array brought into one of the working stores or a constant, and the result remains in the accumulator.

That is, the code is 'one-address'; each instruction consists essentially of an address, which is usually symbolic, and a function specification which includes an indication of which accumulator is to be used. Arrays of numbers, and constants, can be arbitrarily named by the programmer, provided that not more than 5 alpha-numeric characters are used and at least 1 is alphabetic.

For example, arrays may be named A1, A2, and so on, or, if there is only one of them, simply ARRAY, while the constant π may be given the obvious name PI. The address used in an instruction referring to such an array or constant is then its name. Provision is made for referring to any element of an array by means of 4 index registers, any one of which can be specified in the instruction. This method is used where the same or a similar calculation is to be done with each element of the array; if, however, the elements are disconnected and are to be used individually in the computation, the programmer can name the array by a single letter and then refer to any element by this letter followed by the position of the element in the array, e.g. if the array is called A, then A0002 would be the address of its second element. One of the index registers is also required when this sort of reference is made. Numerical addresses refer to numbers currently in the working stores or the constants buffer store, to the accumulators or to special constants.

Functions are specified by groups of 3 letters which have a mnemonic connection with the description of the operation, e.g. CAS represents 'Clear and Add to the Short accumulator', while SBL represents 'SuBtract from the Lower accumulator'. A list of the functions built into the scheme is given in Appendix 3. Almost any standard DEUCE subroutine can, however, be used to introduce a new function. Both fixed - and floating-point working are catered for.

To enable the normal sequence to be broken, any instruction can be given a 'next instruction location' which is an arbitrary symbol of up to 5 alphanumeric characters. The instruction to which the jump is made must, of course, have the same symbol, as its 'location'. Normally a next instruction location refers either to an instruction further back in the programme or, at the entry to a loop for example, to the next instruction in sequence; in other cases a special forward jump indicator must be included in the instruction or the translation programme will assume that an error has been made.

The only other part of an instruction is the 'decrement', which is used only with the counting functions (JIX, JXH, JXE). The decrement may be symbolic, in which case its numeric value is found in the corresponding constant address, or it may be numeric, and it may be positive or negative. The function JIX causes the decrement to be subtracted from the specified index register, the result remaining there, and a jump to be made to the specified next instruction if this result is positive. With both JXH and JXE, the specified index register is compared with the decrement; in the first case a jump is made if the number in the register is algebraically greater than or equal to the value of the decrement and in the second case a jump is made only if they are equal.

Control cards, with characteristic punchings, are inserted at various places in the programme to define constants, extra DEUCE subroutines, arrays, loops (optional), and the end of the programme.

Appendix 4 shows a 'SODA' programme for forming a vector product. A loop control card could be inserted between cards 9 and 10; its effect may be to make the resultant DEUCE programme faster by causing a fresh block of instructions to be brought from the drum at the beginning of the loop.

The translation programme reads in a set of 'SODA' codewords together with the control cards, etc., but not, of course, the data on which the calculation is to be done. It then produces a set of DEUCE instructions for carrying out the calculation specified, as a complete DEUCE programme punched out on cards and ready for use in the normal way after adding a standard pack of subroutine cards and the data.

To assist programme testing, the translation can, at the discretion of the programmer, include a 'trace' facility. If the facility is included the DEUCE programme can be made to punch out, after sequences corresponding to most 'SODA' codewords, a codeword identification number and the contents of the accumulators and index registers, it can be made simply to stop and show the identification number after each such sequence, or it can be made to run normally. A correct programme should, of course, be finally translated without the trace facility to produce an efficient DEUCE programme.

7. SOME TRANSLATION DIFFICULTIES.

The main source of difficulty in writing a translation programme for a computer such as DEUCE is the multiplicity of its storage levels; there are a few rapid-access stores, a few hundred medium-access 'main' stores and a few thousand slow-access stores. This multiplicity is introduced for economy's sake, of course. In DEUCE, the actual computation takes place in the 'fast' stores and instructions are obeyed from some of the main stores, the others being used to contain numbers not wanted immediately; the 'slow' stores usually contain instructions or numbers, or both, to be used later in the programme. To make efficient use of the computer care must be taken in deciding which part of the store shall hold any particular number or instruction at any time during the running of the programme. It has already been remarked, however, that a good pseudo-code should not reflect this structure, i.e. the codewords should all appear to be equally accessible and the numbers should either be all on the same level or, if there are differences, they should be introduced only for the convenience of the pseudo-code user, e.g. the X's, N's and T's in 'Alphacode'. Therefore, a translation programme, if it is to produce an efficient machine-code programme from a good pseudo-code, must allocate positions at the various storage levels to the (translated) instructions and to the numbers, according to the span of programme over which they are required and the frequency with which they are used, and must arrange for transfers between the levels to take place at appropriate times. For example, to quote extreme cases, an intermediate result which is used in the next codeword and never again should be left in a fast store, while a group of numbers which are required at the beginning and also at the end of a long programme should be allocated positions in the slow store, though, of course, they will have to be transferred, through the main store, to the fast store when they are used. Also, a loop of instructions which is obeyed many times should, if possible, remain entirely in the main store while it is being obeyed, though it may have to be put originally in the slow store to make room for another part of the programme.

The problem of storage-level allocation is complicated by the topology of the programme. Loops may be broken into from other parts of the programme, there may be alternative paths and there may be complicated combinations of these, so that the requirements of the various paths must be weighed against each other.

An 'Alphacode' DEUCE translation programme is being developed, in which many of these difficulties are overcome. The techniques used are described in a paper (Ref. 5) to be presented at the UNESCO International Conference on Information Processing in Paris in June. Briefly, the programme is first divided into sections which have no branches or junctions except at their ends, and the references to numbers in each section in turn are examined for span and frequency, those with lowest span and highest frequency being given highest priority, in general, for the fast stores. Any vacancies are then filled by using topological considerations. A similar process is used to allocate positions in the main store, though here subroutines of DEUCE instructions also compete for space. Finally, the instructions linking the DEUCE subroutines are divided into appropriate groups and allocated main storage space, the instructions necessary to bring them from the slow store having been inserted.

8. ACKNOWLEDGMENTS.

The author is indebted to the originators of the schemes described for the use made of their own descriptions, and also to the Director of Research, Nelson Research Laboratories, English Electric Co. Ltd., Stafford, for permission to present this paper.

9. REFERENCES.

1. Robinson, C., 'Automatic Programming for DEUCE', to be presented at this conference. (English Electric Report No. NS y 133).

2. Hamblin, C.L. 'Computer Languages', Australian J. Of Science, Dec. 1957, p. 135.

3. Brooker, R.A., 'An Attempt to Simplify Coding for the Manchester Electronic Computer', Brit. J. of App. Phys. (1955) 6 p. 307.

4. Bell, C.G, and Brigham, R.C., 'SODA Manual of Operation', Report from Dept. of Electrical Eng., N.S.W. University of Technology, Sydney, Australia (1958).

5. Duncan, F.G., and Hawkins, E.N., 'Pseudo-code Translation on Multi-level Storage Machines', UNESCO Conference, June 1959 (English Electric Report No. NS y 125).

APPENDIX 1.

(i) An automatically typed 'Alphacode' programme.

```
R         A           B        FUNCTION            C                     D

        X0007 =                CONSTAnt
                                                        +1.00000000   +002
7             Read   0006      DATA         into X0001  onward.
        N0001 =     X0004      MOVED
        N0002 =     X0005      MOVED
        N0003 =     X0006      MOVED
1       N0004  and  N0004      MODIFY next inst.  0000  also.
        X0001 =     X0001      DIVIDEd      by  X0007
              Count N0004      UP TO              0003 ,   jumping   to  R 1
        X0004 =     X0001      MOVED
6       X0005 =      0001      PLUS              X0004
        N0004 =     N0001      MOVED
        X0006 =      0000      PLUS               0001
2       X0006 =     X0006      MULTIPlied   by  X0005
              Count N0005      UP TO             N0004 ,   jumping   to  R 2
5       X0008 =     X0006      MULTIPlied   by  X0004
        X0009 =     X0006      MINUS              0001
        N0006  and   0000      MODIFY next inst.  0000  also.
        X0016 =     X0008      DIVIDEd      by  X0009
        N0006 =     N0006      PLUS               0001
              If    N0004  is  AS BIG        as N0003         jump   to  R 3
        N0004 =     N0004      PLUS              N0002
4       X0006 =     X0006      MULTIPlied   by  X0005
              Count N0005      UP TO             N0002      jumping   to  R 4
                               JUMP                                  to  R 5
3 Batch  0000 Print N0006      RESULTs     from X0016  onward.     Type  0
        N0006 =      0000      PLUS               0000
              If    X0004  is  AS BIG        as X0003         jump   to  R 7
        X0004 =     X0004      PLUS              X0002
                               JUMP                                  to  R 6
                               FINISH
```

(ii) Some other 'Alphacode' instructions.

```
R          A          B        FUNCTION            C                     D

         X0001 = sum 0006 term SERIES, argument  X0002
               Solve 0003      DIFF.Equations at X0101 interval, using S 1
         X0001 =      0050 step INTEGRal,interval X0021
 Read    X0010  from 0100  pt. GRAPH         at  X0300

1                              SUBROUtine
                               ......
                               ......
                               ......
                               END OF                   subroutine     S 1
```

APPENDIX 2.

The permanent 'Easicode' functions, with their numbers:

1. Multiply.
2. Divide.
3. Add.
4. Subtract.
5. Modulus.
6. Count Integers.
7. Count up to.
8. Equal)
9. Big As) (Conditional Jumps)
10. Exceeds)
11. Unequal)
12.
13.
14. Buzz and Halt.
15. Enter Subroutine (of codewords)
16. End of Subroutine (of codewords)
17. Transfer Fast to Fast Store.
18. Transfer Fast to Slow Store.
19. Transfer Slow to Fast Store.
20. Transfer Slow to Slow Store.
21. Fetch Subroutine.
22.
23. Programme Constant.
24. Read One Parameter (Integer)
25. Read One Number.
26. Punch One Number.
27. General Read to Fast Store.
28. General Punch from Fast Store.
29. General Read to Slow Store.
30. General Punch from Slow Store.
31. Finish.

APPENDIX 3.

List of 'SODA' Instructions.

A. Instructions Pertaining to the Short Accumulator.

CAS	-	Clear and Add to the Short.
CSS	-	Clear and Subtract from the Short.
ADS	-	ADd to Short.
SBS	-	SuBtract from the Short.
STS	-	STore the Short.
SSL	-	Shift the Short to the Left.
SSR	-	Shift the Short to the Right.
LAS	-	Logical And with the Short.
LOS	-	Logical Or with the Short.
LNS	-	Logical Non-equivalence with the Short.

B. Instructions Pertaining to the Lower Accumulator.

CAL	-	Clear and Add to the Lower.
CSL	-	Clear and Subtract from the Lower.
ADL	-	ADd to the Lower.
SBL	-	SuBtract from the Lower.
STL	-	STore the Lower.
SLL	-	Shift the Lower to the Left.
SLR	-	Shift the Lower to the Right.
LAL	-	Logical And with the Lower.
LOL	-	Logical Or with the Lower.
LNL	-	Logical Non-equivalence with the Lower.

C. Instructions Pertaining to the Upper Accumulator.

CAU	-	Clear and Add to the Upper.
CSU	-	Clear and Subtract from the Upper.
ADU	-	ADd to the Upper.
SBU	-	SuBtract from the Upper.
STU	-	STore the Upper.
SUL	-	Shift the Upper to the Left.
SUR	-	Shift the Upper to the Right.
LAU	-	Logical And with Upper.
LOU	-	Logical Or with the Upper.
LNU	-	Logical Non-equivalence with the Upper.
MPY	-	MultiPlY.
DIV	-	DIVide

D. Instructions Pertaining to the Double Accumulator.

CAD	-	Clear and Add to the Double.
CSD	-	Clear and Subtract from the Double.
ADD	-	ADd to the Double.
SBD	-	SuBtract from the Double.

ASD	-	Add a Single length word to the Double.
SSD	-	Subtract a Single length word from the Double.
STD	-	STore the Double.
SDL	-	Shift the Double to the Left.
SDR	-	Shift the Double to the Right.

E. Instructions Pertaining to Floating Point Arithmetic.

PRF	-	PRepare a Floating point number.
CAF	-	Clear and Add a Floating point number
CSF	-	Clear and Subtract a Floating point number.
STF	-	STore a Floating point number.
FAD	-	Floating point ADd.
FSB	-	Floating point SuBtract.
FMP	-	Floating point MultiPly.
FDV	-	Floating point DiVide.
FSR	-	Floating point Square Root.
FLG	-	Floating point LoGarithm.
FEX	-	Floating point EXponential.
FSN	-	Floating point SiNe
FCS	-	Floating point CoSine
FAT	-	Floating point Arc-Tangent

F. Instructions Pertaining to the Index Registers.

LXP	-	Load an indeX register Positive.
LXL	-	Load an indeX register positive Less one.
LXN	-	Load an indeX register Negative.
ADX	-	ADd to an indeX register.
SBX	-	SuBtract from an indeX register.
STX	-	STore an indeX register.

G. Decision or Jump Instructions.

JSZ	-	Jump if the Short is Zero.
JSP	-	Jump if the Short is Positive.
JLZ	-	Jump if the Lower is Zero.
JLP	-	Jump if the Lower is Positive.
JUZ	-	Jump if the Upper is Zero.
JUP	-	Jump if the Upper is Positive.
JDZ	-	Jump if the Double is Zero.
JDP	-	Jump if the Double is Positive.
JIX	-	Jump on IndeX.
JXH	-	Jump on indeX High or equal.
JXE	-	Jump on indeX Equal.

H. Instructions Permitting the Block Transfer of Data.

RWO	-	Read into Working storage One.
RWT	-	Read into Working storage Two.
WWO	-	Write from Working storage One.
WWT	-	Write from Working storage Two.

I. Instructions Pertaining to Input and output.

RDO	-	ReaD into One from the reader.
RDT	-	ReaD into Two from the reader.
RDC	-	ReaD a Card.
PHO	-	PuncH from working storage One.
PHT	-	PuncH from working storage Two.
PHC	-	PuncH a Card.
RDA	-	ReaD Array.
PHA	-	PuncH Array.

J. Miscellaneous Instructions.

HPR	-	Halt and PRoceed
STP	-	SToP
STZ	-	STore a Zero.
STO	-	STore a One.
STA	-	STore an Address unit.
STH	-	STore a High position bit.
STM	-	STore a Minus one.
ACA	-	ACtivate Alarm.
SPA	-	StoP Alarm.
RIL	-	Read from the Input Lights.
WOL	-	Write into the Output Lights.
COL	-	Clear the Output Lights.
CWO	-	Clear Working storage One.
CWT	-	Clear Working storage Two.
ENS	-	ENter Subroutine.
LVS	-	LeaVe Subroutine.

APPENDIX 4.

'SODA' Programme for Forming Vector Product (Order n $\leq$ 1024)

	Location.	Function.	Address.	Next Inst.	Index Reg.	Decrement	Comments.
1.	Type 3			3, A ----, 255, 1024			Define array A
2.	Type 3			3, B ----, 223, 1024			Define array B
3.	BEGIN	RDC	00000				n in address 0
4.		LXP	00000		1		Set n in index 1.
5.		RDA	A----				Read vector A
6.		RDA	B----				Read vector B
7.		RWO	A----				A in working store 1
8.		RWT	B----				B in working store 2
9.		STZ	00100	LOOP-			Clear short acc.
10.	LOOP-	SBX	00104		1		Subtract 1 from index
11.		CAU	A----		1		A_j in upper acc.
12.		MPY	B----		1		A_j x B_j formed
13.		ADS	00102				Added to sum.
14.		JXE		LOOP-	1	00106	Count.
15.		CWO					Clear working store.
16.		STS	00000				Store result in 0.
17.		PHC	00000				Punch result.
18.		STP					Stop
	Final Control Card.			Punching.			
19.	Type 1			1, BEGIN			Define entry.

NOTES:

Address 0 is in working store 1 (Hence the need for card 15)

Address 100 is the short accumulator.

Address 102 is the upper accumulator.

Address 104 produces a 1 in the form required for index registers.

Address 106 produces zero.

The numbers 255 and 223 in the initial control cards define the last tracks of the drum occupied by arrays A and B respectively.