SAKO—An Automatic Coding System

LEON ŁUKASZEWICZ

Institute of Mathematical Machines, Polish Academy of Sciences, Warsaw

SUMMARY—Some properties of SAKO, an Automatic Coding System for the medium-scale digital computers XYZ and ZAM II, are described. This system takes into account specific features of these machines (for instance, the fact that they are fixed-point), and provides for logical operations on machine words. In designing SAKO, the author aimed at a system sufficiently effective to eliminate machine language almost entirely in the whole field of numerical and logical problems.

INTRODUCTION

IN VIEW of the growing need for new programs for electronic digital machines, the use of automatic coding systems is particularly helpful. Their purpose is to bring about an essential reduction in the time spent in preparing programs, assuming that the methods to be used to solve the problems are known. Of the systems in use at present, the best known in Poland are FORTRAN, MATH-MATIC and the MANCHESTER AUTOCODING SYSTEM. But their use is still limited. For instance, in problems containing logical operations, when the need arises to use the internal structure of a machine word, these systems are not very suitable. Moreover, in some systems, the efficiency of the object-program is not high, and they are not, therefore, economical, especially in frequently repeated programs.

The SAKO Automatic Coding System was conceived as a basic system which would make the programming of numerical and logical problems in machine language virtually unnecessary. Although instructions written in SAKO may be fairly easily used in conjunction with machine instructions written in SAS (Symbolic Address System),

this is seldom necessary. In fact the purpose of SAS is to act as an intermediary between SAKO and the actual machine language.

The basic principles born in mind when designing SAKO were:

1. The language should take advantage of all machine potentialities and cover the whole field of numerical and logical applications.
2. The resulting object program should be as efficient as possible both from the point of view of time of execution and that of economizing on machine storage.
3. The language should be as easy to use as possible and independent of special machine features.

Clearly, these aims, especially the first and third, are to some extent contradictory. SAKO is a compromise in which, however, the first two points are given priority. In spite of this, as we shall see later, SAKO is almost as general and as easy to use as the other languages we have mentioned.

In general, it seems that a practical realization of a fully universal language, independent of every individual machine feature, will always lead to compromises at the expense of the full economical use of the machine.

Whether the machine is fixed- or floating-point, binary or decimal, whether it can read magnetic tape in one or both directions must of necessity influence the programming system which is based on principles 1 and 2. An additional source of differences is the more or less basic tendency to improve any universal language, thus making it difficult to publish anything as the best of all possible languages now and in the future.

The differences between practical languages designed for different machines should not be thought of as unfortunate, if similarities in general structure and other common features are borne in mind. What is important is that the overall simplicity of autocodes makes it easier to switch from one language to another than is the case when two machine codes are involved.

To obtain similar languages designed for different machines it is helpful to establish some generally accepted pattern, such as ALGOL (Ref. 4). The many advantages of this language were taken as a starting point when developing SAKO. In the present paper, rather than describe SAKO in detail, we shall concern ourselves with those features which distinguish it from other automatic coding systems currently in use. A description of arithmetical formulae and of the use of subroutines in SAKO is given in Ref. (5).

GENERAL STRUCTURE OF 'SAKO': INSTRUCTION AND DECLARATION LIST

Programs written in SAKO are made up of *sentences*. There are several different types of SAKO sentence, each of which has a definite form and meaning; the use of each is governed by a number of rules. SAKO, therefore, is a formalized language.

The information processing done by the machine is in two basic stages:

the introduction of the program into the machine; and
the processing of this program by the machine.

Thus, all SAKO sentences are divided into the following categories.

DECLARATIONS: give information on the structure of the program and determine the meaning of the symbols used. Declarations are operative during the period when the program is being fed into the machine; they are omitted while the program is being executed.

INSTRUCTIONS: are carried out by the machine when the program is running.

COMMENTS: are of an explanatory character only for the use of the programmer; they are ignored during the introduction and execution of the program.

A list of SAKO declarations and instructions is given in Table 1.

Here we recall the need to distinguish between the sequence in which instructions are written and that in which they are executed by the machine. These are, generally, different. The sequence in which SAKO instructions are written is that in which they are input into the machine.

The sequence in which the instructions are executed is determined as follows:

the first instruction obeyed by the machine is always the first instruction written in the program;

having carried out an instruction which is not a control instruction, the machine moves to the next instruction, i.e. the nearest in sequence as written;

transfer to an instruction out of sequence may take place only after a control instruction and the conditions contained therein have been satisfied.

The last sentence written in a program is always the declaration END which signifies that the input of the program has been completed. The

instruction STOP is the last to be performed and signals that the program has been executed in full and that the machine should stop.

When long programs are written in SAKO they must be divided into CHAPTERS which, together with the corresponding data, are stored internally.

The *syntax* of a SAKO sentence is now given:

Sentences in SAKO are built from the following characters:

1. The 25 letters of the Latin alphabet, A, . . ., Z;
2. Digits 0 . . . 9.
3. Characters for operations and relations + − . / = ≡
4. Separating characters . , : ( )
5. Space (space between characters).

These characters are available on Creed Teleprinters, with : and ≡ replacing v and n, respectively.

Certain sequences of characters, with a specified structure, and followed by a space are called *expressions*. Expressions denote, for example:

numbers,
variables and functions,
arithmetical and Boolean expressions.

Other specific sets of characters, again followed by a space, are called *phrases* in SAKO. For example:

READ
NEXT
WHEN.

Some character sequences, constructed freely, but with strictly determined begin and end, are called *texts*. For example,

comments, and
output format specifications.

SAKO sentences are built up of phrases, expressions and texts. For each permissible SAKO sentence-type, rules are laid down giving the exact kind of phrase, type of expression, text restrictions allowed and the order in which such components of a sentence may occur. Each SAKO sentence begins with a new line. The number of lines occupied by a sentence is determined by the sentence structure which also determines the end of the sentence. The majority of SAKO sentences occupy a single line.

TABLE 1

List of SAKO Sentences

1. *Declarations*
   1. CHAPTER: name
   2. BLOCK: list of blocks
   3. STRUCTURE: list of blocks
   4. TABLE: name
   5. INTEGRAL: list of variables and blocks
   6. PARAMETER $\left\{\begin{matrix}\text{DECIMAL}\\\text{BINARY}\end{matrix}\right\}$ SCALE
   7. LANGUAGE SAS
   8. LANGUAGE SAKO
   9. END
2. *Control Instructions*
   1. JUMP TO α
   2. JUMP ACCORDING TO I: list of labels
   3. GO TO CHAPTER: name
   4. IF $A = B$: α, OTHERWISE β
   5. IF $A > B$: α, OTHERWISE β
   6. IF OVERFLOW: α, OTHERWISE β
   7. IF KEY K: α, OTHERWISE β
   8. REPEAT FROM α: $I = \mathcal{J}(K)L$
   9. STOP
3. *Drum Instructions*
   1. READ DRUM FROM I: list of variables and blocks
   2. WRITE DRUM FROM I: list of variables and blocks
4. *Input–Output Instructions*
   1. READ: list of variables and blocks
   2. READ LINE: name of block
   3. PRINT $(I.\mathcal{J}.)$: list of variables
   4. PRINT LINE: name of block
   5. PRINT WORD: list of variables
   6. TEXT:
   7. TEXT LINES:
   8. SPACE
   9. LINE
5. *Arithmetical and Boolean Instructions*
   1. $A := B$
   2. $A \equiv B$
   3. SET $\left\{\begin{matrix}\text{DECIMAL}\\\text{BINARY}\end{matrix}\right\}$ SCALE
   4. INCREASE $\left\{\begin{matrix}\text{DECIMAL}\\\text{BINARY}\end{matrix}\right\}$ SCALE BY I: list of variables and blocks.
6. *Subroutines*
   1. SUBROUTINE: (list of results) = name (list of arguments)
   2. (list of results) = name (list of arguments)
   3. SUBSTITUTE: name (list of arguments)
   4. RETURN
7. *Identifiers*

   $i, j, \ldots$ natural numbers

   $A, B, \ldots$ variables or expressions

   $I, \mathcal{J}, \ldots$ integral variables or numbers

   $\alpha, \beta, \ldots$ labels

Although the rules for forming SAKO sentences allow great heterogeneity in their structure, the majority are formed according to the following system:

phrase—operational part of instruction or declaration
expression—parameters of sentences
phrase—colon
expression—arguments of instructions or declaration in form of a list of variables.

An example of an instruction formed in this way is:

PRINT (6.5): *A,B,X*

The fact that the majority of SAKO sentences exhibit a similiarity of structure is of considerable help to the programmer and to the machine when analysing them.

DECIMAL REPRESENTATION OF NUMBERS IN 'SAKO'

1. *Introductory remarks*

Both XYZ and ZAM II are binary, fixed-point machines in which, however, the scale of computation may be varied by instructions shifting the contents of the double length accumulator after multiplication and before division (Ref. 1). The way in which numbers are represented and the manner in which arithmetical operations are carried out depend upon these features.

We recognize that it is generally more convenient, from the programmer's angle, to use floating-point arithmetic, which may be simulated in fixed-point machines by means of suitable subroutines. However, as this leads to a serious reduction in the speed of the machine, this would be contrary to the first principle laid down in the introduction, and has not, therefore, been incorporated in SAKO.

2. *Fractional numbers*

Though XYZ and ZAM II normally work in binary, a decimal interpretation of all numbers either occurring in a program or processed by a program is possible. One storage cell is then required for one fractional number, the cell being imagined to consist of 12 positions. The first position is occupied by the sign character. The remaining positions are occupied in sequence by decimal digits and the point separating the integral and fractional part of the number, thus, for instance:

—	0	0	2	7		5	3	4	8	0	3
0	1	2	3	4	5	6	7	8	9	10	

The number of the position in which the point is written determines the *decimal scale* of the number. In the above example, the number is written in scale 4.

The following are the basic rules for operating with fractional numbers:

1. Arithmetical operations on such numbers may only be performed when both operands are stored in the machine in the same scale.
2. The scale of the resultant is the same as the scale of the two operands.
3. The particular scale used is determined by a SET SCALE instruction, which must be compatible with the scale in which all operands have been written. Clearly if the scales are different, the result obtained is wrong.
4. The scale of all non-integral numbers occurring in the program is given by an immediately preceding PARAMETER SCALE declaration.
5. If as a result of the particular operation carried out, more significant digits are obtained before the point than is permissible in the scale adopted, overflow occurs and results in a wrong number being transferred to store.

(It follows from what we have said that we can always write, in the scale N, a number having N significant decimal digits before the point. However, some extension of this limit is possible, as will be shown below.)

6. Overflow is indicated by the machine by the appearance of a 1 in the special register called *overflow indicator*. The state of this register, which may be 0 or 1, is tested by an IF OVERFLOW instruction, which automatically zeros the overflow indicator.

The scale currently in use may be changed by a CHANGE SCALE instruction.

3. *Integers*

An integer occupied half a machine store cell. We imagine that this consists of 6 positions, the first of which is occupied by the sign character, the remainder by the digits of the number, the least significant of which occupies the last position. A SET SCALE instruction does not affect the representation of an integer.

The rules for operating with integers in SAKO are:

1. The result of adding, subtracting and multiplying two integers and of raising one integer to an integral power is always an integer.
2. The result of dividing one integral number by another or of adding, subtracting, multiplying or exponentation with mixed numbers is always

a non-integral number in the scale determined by the last SET SCALE instruction.

3. If overflow occurs, the overflow register automatically records the fact as in the case of non-integral numbers.

4. *Example*

The use of the SET SCALE instruction and of a PARAMETER SCALE declaration is illustrated below:

Assume that the cubic $y = a_0 + a_1x + a_2x^2 + a_3x^3$ is to be evaluated for the 101 values $x = 0(0.01)1$, the given a_i's having absolute values less than 1. The appropriate SAKO program may be written:

```
C) VALUES OF POLYNOMIAL
   PARAMETER SCALE 1
   TABLE(4): A
     0.38461        −0.86461
   −0.09830          0.77667
   INTEGER: K
   SET SCALE 1
*2) Y = 0
*1) Y = Y . X + A(K)
   REPEAT FROM 1: K = 3(−1)0
   LINE
   PRINT (6.2): X
   PRINT (6.4): Y
   REPEAT FROM 2: X = 0.00 (0.01) 1.00
   STOP
   END
```

Then first sentence is comment (C) which has no significance when the program is run. The second sentence is a declaration establishing the scale according to which the fractional numbers occurring later in the program are to be written in the machine. Declaration TABLE specifies the *four* numbers following as components of a vector A; these will be stored in the machine in scale 1 in accordance with the previous PARAMETER SCALE declaration. The declaration INTEGER declares K to be a variable taking integer values only.

The instruction SET SCALE 1 establishes the scale in which subsequent calculations will be carried out.

The next instruction, assigning Y the initial value zero, is labelled number 2.

An asterisk on the left of a label means the beginning of a REPEAT cycle. Generally, before each label we must put exactly as many asterisks as there are instructions REPEAT referring to this label. Each label must begin with a digit.

The two next instructions cause the machine to evaluate the polynomial using Horner's method. The computing will be done in scale 1, in accordance with the previous instruction SET SCALE.

The three next instructions cause the values X and Y to be printed out in sequence. In printing X, 6 positions before the decimal point, and 2 after the point are reserved. This gives, together with the point, 9 positions. In printing Y, 11 positions are reserved, the result being rounded to four digits after the point. Instruction LINE causes the transfer to the next line.

The following REPEAT instruction causes the section of the program from label 2 to this instruction to be repeated for values of X varying from 0.0 by 0.01 to 1.0.

Instruction STOP stops the machine.

Declaration END indicates the end of the program.

BINARY REPRESENTATION. BOOLEAN WORDS

1. *Binary word*

A decimal interpretation of the storage places was given above. However, the binary interpretation and its use is nearer to the real machine construction, and may give additional advantages in many cases.

In binary interpretation each storage cell is treated as a set of 36 positions containing one of two digits 0 or 1. Such a set we call a *binary word* or, briefly, a *word.*

Depending upon interpretation we can treat the word as a fractional number, an integral number or a Boolean word.

2. *Fractional numbers*

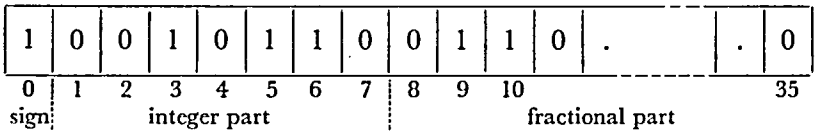
We write fractional and integral numbers in a binary word using the binary system to represent them.

The digit in position zero of the word determines the sign of the number:

0—positive
1—negative

In a word the integral and fractional part of a number occupy two sections immediately following each other. The number of the position in which the last digit of the integral part is placed indicates the *binary scale* used.

Thus:



represents the number −22.375, in binary scale 7.

If the fractional part begins with position 1, we say that the number is written in scale 0. If the integral part ends in position N, we say that the number is written in binary scale N.

The binary scale used is not explicitly indicated in the word when writing fractional numbers, but it is taken into account in the object program by an appropriate shifting of the contents of the double length accumulator. These shifts are brought about by the machine instruction EXECUTE INSTRUCTION LOCATED IN THE CELL n directly after every instruction to multiply, or EXECUTE INSTRUCTION LOCATED IN THE CELL m placed directly before every instruction to divide. The cell with the address n contains the instruction LONG SHIFT LEFT s, and the cell with the address m LONG SHIFT RIGHT s. By changing the number s in the storage cells with addresses n and m, we change the scale of all subsequent calculations.

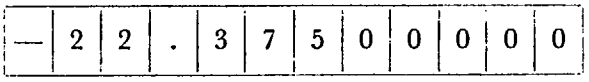
In the SAKO language the binary scale to be used may be indicated to the machine by the instruction SET BINARY SCALE or by the declaration PARAMETER BINARY SCALE, for instance:

```
SET BINARY SCALE 12B
PARAMETER BINARY SCALE 8B
```

To each decimal scale previously described corresponds a binary scale as shown in the following table:

Decimal scale	0	1	2	3	4	5	6	7	8	9	10
Binary scale	0	4	7	10	14	17	20	24	27	30	35

Thus numbers written in decimal scale 2:



in the machine will be represented in binary scale 7 in the way shown above.

3. *Integral numbers*

To write an integral number in the machine we use only half of the word, i.e. 18 binary positions. Therefore, in one machine storage cell we can write two integral numbers simultaneously.

The first position of the half word is used to indicate the sign of the integer. The least significant binary digit is written in the last of the 18 positions.

While performing operations on integers, the machine treats each number as if it occupies all the 36 positions, with half of the word occupied by the number (positions from 0 to 17) and the remaining half (positions 18 to 35) occupied by zeros.

4. *Boolean word.*

By a *Boolean word* we understand a word whose digits 0 and 1 are treated as elements of the two-element Boolean algebra. Every fractional and integral number written in the machine may be treated as a Boolean word. We write Boolean words in octal notation, expressing each sequence of three binary digits as digits from 0 to 7. For instance:

776.000.000.377

represents the Boolean word 111 111 110 000 000 011 111 111.

In accordance with operations in Boolean algebra, we use the idea of *identity* and the operations *sum, product and negation.* Moreover, we make use of what is called *cyclic shifting* of a word.

Two Boolean words A and B are identical:

$$A \equiv B$$

if and only if all digits of one word are identical with the corresponding digits of the second word.

The *sum* of the Boolean words B and C, $A \equiv B + C$, is the Boolean word A, in which at any position the digit is 1 if and only if there is a 1 in this position in at least one of the two words B or C.

Boolean *products* and *negations* $A \equiv B \times C$, $A \equiv -B$ are similarly defined. The result of a *cyclic shift* of one position to the right of the Boolean word B, written

$$A \equiv B*1$$

is the Boolean word A, which is such that, for all $\alpha \leqslant 34$, the digit in position α in the word B is in position $\alpha + 1$ in word A. The digit in word B in position 35 become in word A the digit in position zero.

The result of a cyclic shift of the word B of N positions:

$$A \equiv B*N$$

is the word A, resulting from N cyclic shifts of one position to the right of the word B.

A cyclic shift of the word B by N positions to the left is written

$$A \equiv B*(-N)$$

A *Boolean expression* is formed of Boolean operations.

Of two operations in a Boolean expression that which is enclosed in the greater number of parentheses is executed first. In case of an equal number of parentheses the operations are executed with the following priority:

1. Shifts
2. Products
3. Sums and negations

Operations enclosed in the same number of parentheses and of the same priority are executed in sequence from left to right.

To illustrate these rules we take the following example. We require to form the Boolean product of positions 0–9 of the word B, and positions 10–19 of the word C, and add to it the negation of positions 20–29 of the word D. The result is to be placed in positions 26–35 of the word A, the remaining positions being filled with zeroes.

This operation may be expressed by the formula:

$$A \equiv (B*20 \times C*10 + (-D))*6 \times 000.000.001.777$$

It should be emphasized once more, that since a number is represented in the machine by 36 0's or 1's, any number may be treated as a Boolean word, and any Boolean word as a number. The value of A in this example is the set of 36 bits, which may be treated equally well as a Boolean word, or as a number.

Boolean and arithmetical operations in the formula may be distinguished by the use of the characters $\equiv$ or $=$.

BLOCK STRUCTURE

The declaration BLOCK, which always contains constant parameters, determines the structure of all blocks mentioned in the list, and at the same time, reserves an appropriate amount of storage for these blocks.

The declaration STRUCTURE, in which variable parameters may occur, changes the block structure introduced by the previous BLOCK declaration,

but does not reserve storage space. A new block structure must not require more storage space than has been previously allocated.

Instructions involving the structure of a block always assume the structure determined in the last occurring BLOCK or STRUCTURE declaration.

Example

Consider the following section of a program:

```
BLOCK (100) : A
INTEGER : N, M
READ : N, M
STRUCTURE (N, M) : A
READ : *A
.....
```

The first BLOCK declaration determines the block A as a vector, at the same time it also reserves 100 storage cells for the location of the vector components.

The declaration INTEGER and the instruction READ cause the integer numbers N and M to be read into the machine.

The STRUCTURE declaration specifies the new structure of the block A as a matrix of size N × M. This product must be less than 100, because, otherwise, it would not be possible to store this matrix in the locations previously reserved.

The next READ instruction causes all the elements of the matrix A to be stored. The asterisk before A in the READ instruction indicates that A is a block.

SOME INPUT AND OUTPUT INSTRUCTIONS

1. *Read*

The input data introduced into the machine by this instruction must satisfy the following conditions:

1. Integer numbers are written as a sequence of no more than five digits.
2. Fractional numbers are written as a sequence of no more than ten digits separated by decimal point.
3. Negative numbers are preceded by the character —. Positive numbers can, but need not, be preceded by the character +.
4. Boolean words are written in octal and preceded by the character ≡.

5. A space must not occur between any of the characters composing a number or a Boolean word.
6. Several numbers or Boolean words may be written in one line, but at least one space must follow each number or word.
7. Numbers or lines may be provided with comments which are omitted by input program. Every sequence of characters of which the first is a letter, and ending with the character : or = will be treated as comment. Comment may occupy a whole line.
8. Every block written on an input data form must begin with a new line.

Suppose we have a program involving square matrices, and that we must input:

an integer number showing the order of the matrix;
a block of fractional numbers, the matrix elements;
a Boolean word, to be used in some logical operation or operations.

The data will be written in the form, for example:

```
MATRIX RANGE = 2
MATRIX ELEMENTS :
  0.43        7.21
 -0.28       -3.89
MASK : ≡ 774.000.000.000
```

and will be read in accordance with the instruction

READ : N, *A, M

which will cause the single number, the matrix and the Boolean word to be read in that order.

2. *Read line*

This instruction causes the machine to read a line of input data characters, beginning at the left of the next line. Each character in the line, represented by six bits, is written in the last six positions of successive words of the block referred to in the instruction, the remaining word positions being zeroes. Reading stops either at the end of the line or if there are no more words in the block.

3. *Print line.*

This instruction causes the machine to print out on a new line, beginning from the left, a sequence of characters determined by the

content of the six least significant positions of successive words of the given block. Printing stops either at the end of the block or at a CARRIAGE RETURN character.

These two instructions allow data to be input or results output in accordance with an accepted code. One application may be the reading of texts by a machine, their analysis and processing with the use of Boolean formulae, and the final editing of such processed texts.

REMARKS ON TRANSLATION

No mention has been made above of the translator used. But economy and speed were aimed at when designing SAKO. At first it was feared that, for such a simple machine as XYZ, for which SAKO was originally designed, the writing of the translator might be very difficult. It transpired, however, that the very simplicity of XYZ determined the simplicity of the translator. This became quite clear in the machine ZAM II, which was designed with SAKO in mind. The SAKO translator for this machine, including subroutines and dictionaries, contains less than 3,000 machine words (one word contains two one-address instructions). Object programs produced by this translator are almost optimal.

The SAKO translator was written with the help of the SAKO language. The various parts of this program were written in SAKO and manually translated into SAS language. This translation was done quite mechanically on the basis of SAKO–SAS translation rules. The resulting program was then optimized. This considerably reduced the time spent in writing the translator.

That the SAKO language is also suitable for use in some non-numerical data processing was confirmed at the same time.

CONCLUSIONS

By taking into account the specific features of a machine when designing an automatic programming system, one can ensure that, with such a system, the machine in question is used efficiently over the full range of numerical and logical problems for the solution of which it is employed. In particular, the inclusion of Boolean formulae, exploiting machine word-structure, makes possible the programming of many types of logical operation. In practice, SAKO has eliminated almost entirely the need to use machine-code when programming for XYZ and ZAM II, the machines for which it was designed.

ACKNOWLEDGEMENT

The development of SAKO was possible thanks to the efficient work of the whole staff of the Automatic Coding Group of the Department of Programming of the Institute of Mathematical Machines of the Polish Academy of Sciences. Outstanding contributions were those made by A. Mazurkiewicz, J. Swianiewicz, J. Borowiec, M. Łacka, P. Szorc and many others.

The author is also extremely grateful to the Editor who was troubled with the proper formal adjustments.

REFERENCES

1. Buchholz, Werner, 'The System Design of the I.B.M. 701 Computer', *Proc. Inst. Radio Engrs.*, pp. 1262–1275, October, 1953.
2. Backus, J. W. and Beeber, R. J. *et al.*, 'The FORTRAN Automatic Coding System', Proc. of the Western Joint Computer Conference, 1957.
3. Brooker, R. A., Richards, B., Berg, E. and Kerr, R. H., The Manchester Mercury Autocode System, Computing Machine Laboratory, The University, Manchester, May, 1959.
4. Perlis, A. J. and Samelson, K., 'Report on the Algorithmic Language ALGOL', *Numerische Mathematik*, Bd. 1, s. 41–60, 1959; *Annual Review in Automatic Programming*, Vol. 1, Appendix Three.
5. Mazurkiewicz, A., 'Arithmetical Formulae and the Use of Subroutines in SAKO'. (These Proceedings, p. 177.)