ELLIOTT 903 ALGOL

OBJECT CODE MANUAL (JUNE 1966)

CONTENTS

1. INTRODUCTION

The purpose of this manual is to describe the object code of the Elliott 903 Algol system. The elements of this code are called "pords" (parameter words) and are output on to paper tape by the Algol translator while it is reading the source program on an input paper tape. These pords are interpreted one at a time at run time.

The object code is based on one described in the book "Algol 60 Implementation" by Randell and Russell, but is by no means identical to it.

A run time stack is used, somewhat like that in the book, in order to evaluate expressions and to deal with the execution of blocks, procedures and for statements. Although a number of features, for example strings, are handled identically in Randell and Russell and in 903 Algol, it has been possible to introduce certain simplifications because of the restrictions imposed by the IFIP subset. Other changes have also been made and the principal differences are:-

1. The absence of recursion in the subset permits scalars to be given fixed addresses in the store; arrays cannot be treated in this way because their size is not known until run time.

2. The treatment of labels and switches is simplified by the absence of designational expressions from switch lists and by the requirement to declare labels in switch lists; this latter is not an IFIP restriction.

3. The type of an expression, i.e. whether it is real, integer or boolean can be decided at translation time because of the rules of the subset. Unfortunately the parameters to formal procedures form an exception to the rules and it was decided to check at every procedure entry that the actual/formal correspondences are legal; this check extends to include the number of parameters or dimensions of a procedure or array parameter respectively.

4. The restricted call-by-name in the subset removes the need for the "implicit subroutines" described in the book.

5. The addressing of parameters is not as elegant as the method described in the book; although the vector called "display" has been removed, a search down the stack at each parameter reference is sometimes necessary.

6. The object program address increases by one word at a time rather than by one syllable at a time.

7. The object program does not accumulate in the store as translation proceeds but is punched out on paper tape (in relocatable binary form). There is therefore no problem in allocating space to hold the object program, but there is the new problem that one cannot alter something which has already been punched. This is overcome by making use of some facilities of the loader program which performs a second pass as it stores the pords (see 5.2 below).

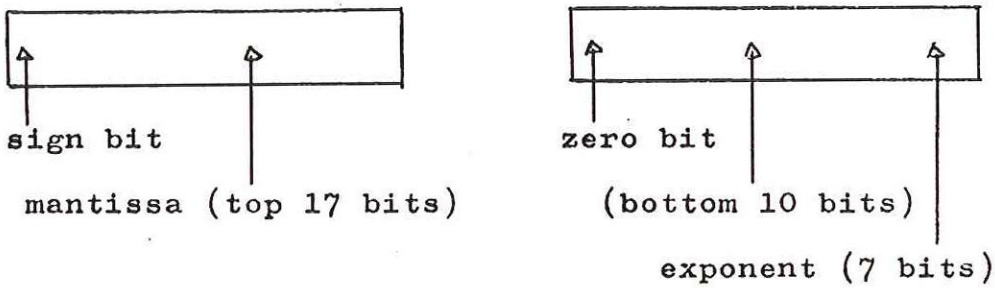
8. Input and output operations which correspond to all the features of read and print lists have direct counterparts in the pords.

9. An object program block is set up for a procedure body, a for statement, or a block which contains an array or switch declaration.) The case where time is saved is where only local variables are declared and this, unfortunately is rare in practice.

2. FORMAT OF REAL, INTEGER AND BOOLEAN VARIABLES

2.1 Real

In the store a real, or an element of a real array occupies two words with the left (more significant) word at an address one lower than the right hand word. There is no restriction for the left hand word to be at an odd or even address.



With mantissa = a and exponent = b a number of value x is represented as

$x = a \times 2^b$ with for $x \neq 0$ $\quad -1 \leq a < -\frac{1}{2}$

$\frac{1}{2} \leq a < 1$

$-64 \leq b < 63$

for $x = 0$, $a=0$, $b=0$

In this form it is said to be packed. It is to be noted that the most significant digit of the exponent acts as a sign bit so that if b= -63 the digits are

1000 001

The largest number which can be held is 9.223×10^{18} approximately and the accuracy is to 8 significant decimal digits (27 bits plus sign bit).

When a real number is brought to the stack it is unpacked into three locations with the exponent occupying the third location and the exponent part of word two cleared to zero. Intermediate results are therefore held to an accuracy of 34 bits plus sign; round off occurs when the result is assigned to a location pair in the store.

2.2 Integer

An integer occupies a single word of 18 bits and lies between -131072 and +131071.

2.3 Boolean

True is represented by the value +1 and false by the value zero; in either case a whole word is used to hold the value.

3. THE STACK

The stack is basic to the operation of the system and is used to hold, among other things, the intermediate results formed during the evaluation of an expression.

3.1 Assignment statement

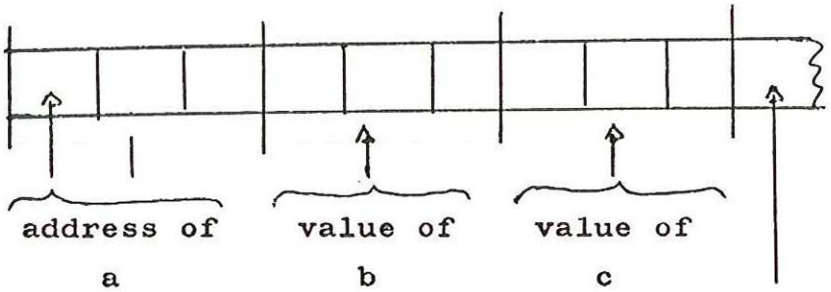
If a, b and c are all real then the Algol text

```
a:= b +c;
```

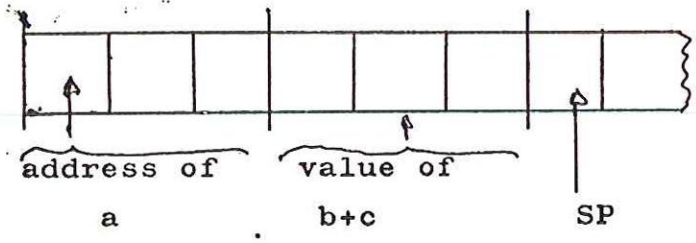
is translated into the pords (for explanation see 6 below):-

```
TRA    "a"
TRR    "b"
TRR    "c"
PRIM   R+R→R
PRIM   ST
```

The interpreter scans down the pords and, in the case of data references, places the address or the value on the stack; three locations are allotted to each entity in the stack. After scanning the first three pords therefore the stack looks like this:-



A "PRIM" pord defines an operation, in this case the addition of two reals; after this pord the stack looks like this:-



Finally the "PRIM ST" pord assigns the value of b+c to a and moves the stack pointer, SP, back to where it was before the statement began. The stack therefore does not grow continuously but returns to a standard place at the end of each statement.

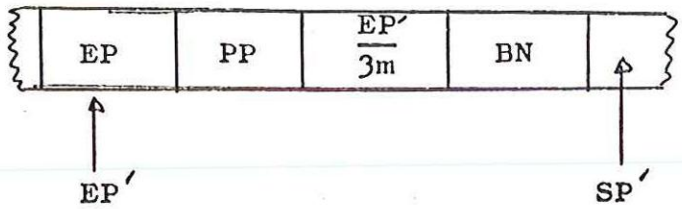
3.2 Procedure calls

When a procedure is entered in Algol it is necessary to store the return address from which the procedure was called. The address within the pord program is called the pord pointer, PP and this is preserved in the stack at the point of call since it indicates the next pord to be taken after execution of the procedure.
In fact it is necessary to preserve four quantities in the stack in order to enter and leave a procedure correctly; these four are:-

Entry pointer	EP	Points to the start of the four locations (called a "stack entry") which hold the status of the calling block.
Pord pointer	PP	Points at next pord for a return address.
EP′-3m		Where there are m parameters, this address is the value to which the stack pointer must return after exit from the procedure.

Block number BN This is a unique 9 bit integer for each run time block. Its presence in the stack enables a parameter to be referenced by searching down the stack if necessary. It also allows a "go to" to leave the procedure body.

The stack entry comprises all of these as shown below, primes denote quantities of the current block and unprimed quantities refer to the calling block.



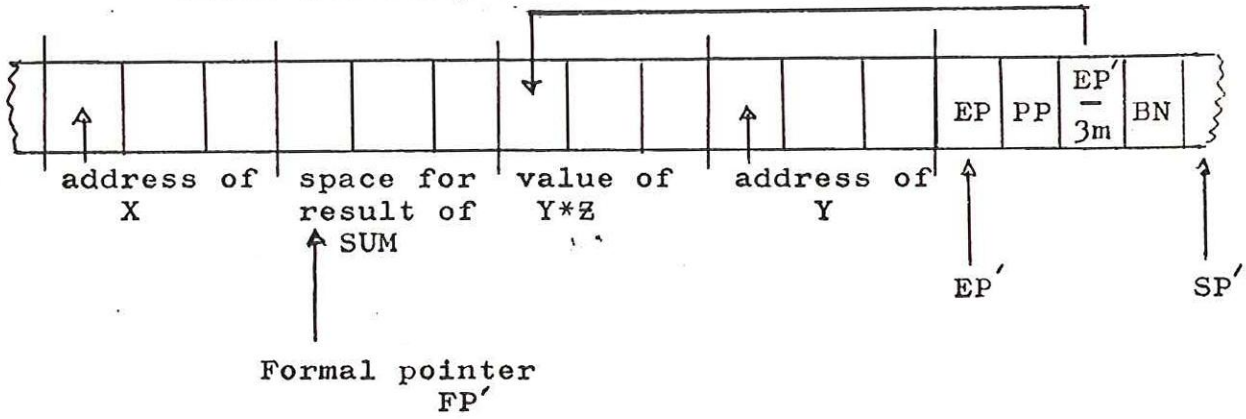
The first two locations are filled in at the point of call and the last two at the procedure's entry point.

The parameters to a procedure and the space for its result, if any, are also held in the stack. Parameters called by value are developed in the stack as values at the point of call and parameters called by name have their addresses in the stack.

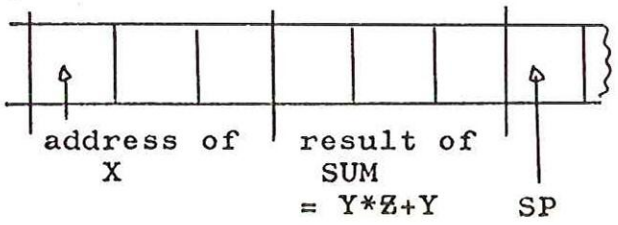
Example

```
begin integer procedure SUM (A,B); value A;
                                   integer A,B;
                      SUM:= A+B;
      integer X, Y, Z;
            X:= SUM (Y*Z, Y);
```

After the entry to SUM the stack contains



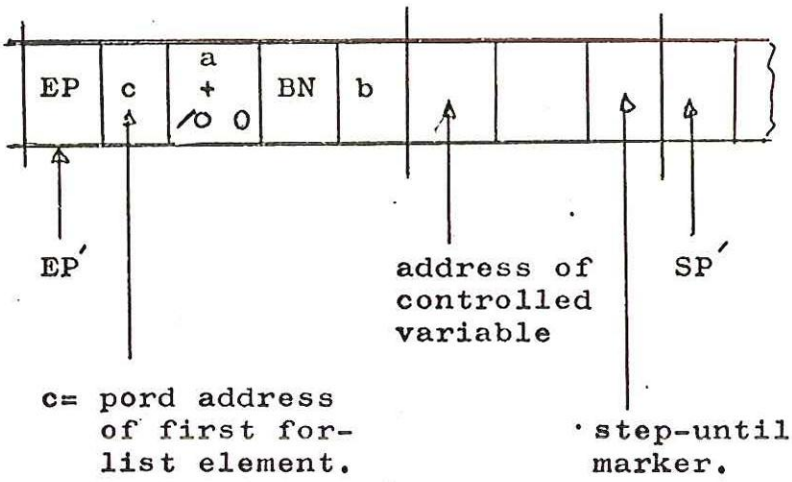
During the body of SUM reference is made to another pointer called the formal pointer FP. This points to the result space if a type procedure is involved and a parameter numbered n is always referenced by the item stored at FP + 3n. FP is set up by the pord at the entry to the procedure body. After exit from the body the stack contains



ready for the assignment to X to be made.

3.3 For-statements

A for-statement is made into a run time block with the help of a stack entry involving five quantities and this is always followed by the address of the controlled variable:-



The address 'a' is the address of the controlled statement and the address b is that of the next statement. The sign bit of the word containing 'a' is set equal to one to mark the entry as a for-block in case a go to causes an exit from the block.

A while or step-until element continually executes the statement beginning at 'a' which eventually returns control to 'c' the for-list element in question where evaluation of the condition begins again; during this evaluation the stack is used as workspace in the normal way with the help of SP. When the element is exhausted the address c gets overwritten with that of the next for-list element and so the process goes on.

A simple for-list element merely overwrites c before execution of the controlled statement.

The last for-list element is followed by a special pord which causes exit from the entire for-block and execution resumes at pord address b.

The location at EP+7 is the step-until marker used to prevent the incrementing of the controlled variable at the first iteration. It is important that BN is stored in the same relative position to EP for a procedure and a for-block stack entry because of parameter references and go to.

3.4 Arrays

An array has space set aside for it in the stack just after a block entry in which it has been declared. For the details of this action see MAMPS in 6.4 below. An array parameter called by value causes all the elements of an array to be copied from one part of the stack to another (see 8 below).

4. STORAGE ALLOCATION

For a basic machine there are alternative ways of dividing up the store depending on whether the library of Algol procedures is held after the end of the interpreter or is held as a relocatable binary paper tape.

With a multi module machine it would appear reasonable to confine the interpreter, library and special machine code to module 1 and to place the object code etc in module 2 onwards. The only restriction is that the length of the object code, constants and scalars must not individually exceed 8191 words.

Figure 1 shows the case of a basic machine containing the whole of the library in the store. Dealing with the areas in turn:-

4.1 Interpreter

This part is a machine code program which contains the subroutines for input/output, the simpler Algol standard procedures, floating point arithmetic, interpretation of the pord code and management of the run time stack. The interpreter is aware of the starting address of the pord program even if the library changes size because the pord program is loaded by entry at a special address.

4.2 The library

This consists of the machine coded Algol procedures:-

arctan, cos, sin	under library name "qatrig"
sqrt	
instring, outstring	under library name "qastri"
lowbound	
range	

The linking of these procedures to their calls from within the object code is done by the loader program which acts in a similar manner to that of the SIR loader. It follows therefore that all the above names including qatrig and qastri are held in the loader's dictionary before the relocatable Algol object code tape is fed in.

If the library gets deliberately overwritten by input of an object program it can be re-established by a start at 12; this also resets the name into the loader's dictionary.

4.3 The pord code

This is described in detail in the rest of this manual. Each word has, like machine code, a function part of 5 bits and an address part of 13 bits and these words are examined in turn by the interpreter. All the operations required by the Algol source text are done by interpreter subroutines which are reached from the pords.

4.4 Constants

Each constant mentioned in the Algol text is stored here (once only) as a positive integer or a positive floating point number. Information relating to switches and labels is also held here and the contents of the whole area are punched out at the end of a translation.
This part of the output tape is preceded by the left hand global name "qacodl" (Algol Constants Object Data Load) and this is noted by the loader. All references to constants, labels or switches at run time require the interpreter to refer to the address of qacodl.

4.5 Scalars

Each integer, boolean or real variable declared in the source text has space reserved for it here. The area is labelled globally by the name "qavnda" (Algol Variables Notional Data Area) and this is also noted by the loader. All references to these variables at run time require the interpreter to refer to the address of qavnda.

The space is reserved by using a SIR skip code and hence the area is not cleared to zero during input, nor is it cleared to zero at the start.

This scheme is not so efficient as one which places all scalars on the stack, but the loss of space is never serious in practice and there is a slight gain in speed.

4.6 Machine code

A User may wish to have special procedures written in machine code and these can be input by a start at 11 after input of the Algol pord program.

4.7 Stack

As described in section 3.

4.8 Loader names

This is a list of names and addresses which is used by the loader to link procedures to their calls.

4.9 Loader

This loads relocatable binary paper tape.

4.10 Systems area

No use is made of this area at present. It could be used for transferring the Algol translator or interpreter from magnetic tape or disc into the store.

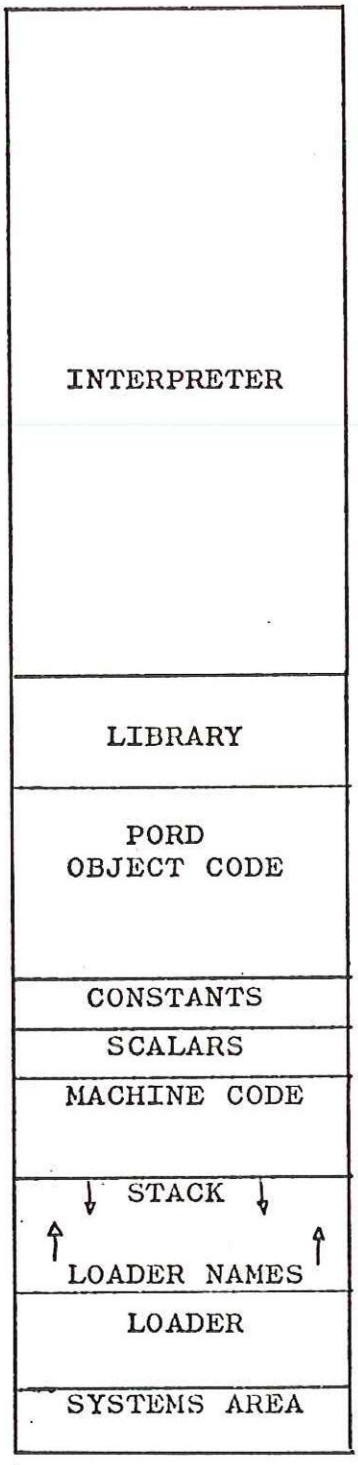
4.11 Summary

There is better use of available storage if the Algol is translated in "library mode" (start at 12). Figure 2 shows the layout. In the first place it is likely that the program does not need all the library and only the part which is needed is present; in the second place the space needed for scalars can overlap the loader but to take advantage of this the scalar space must come last. This is what happens when a translation is done in library mode. Figure 3 shows a suggested use of store in a multi module machine.

FIGURE 1.

Basic machine; library entirely stored

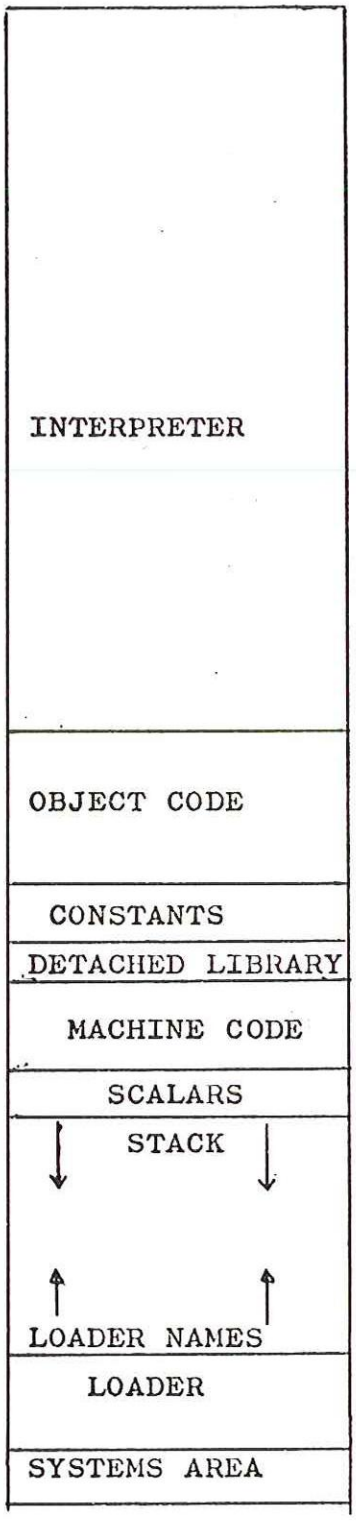
upside down



Address	Remarks
8	
4200	
4900	
6000	Labelled QACODL
6200	Labelled QAVNDA
6400	If any is present
6800	
7500	
8000	

FIGURE 2.

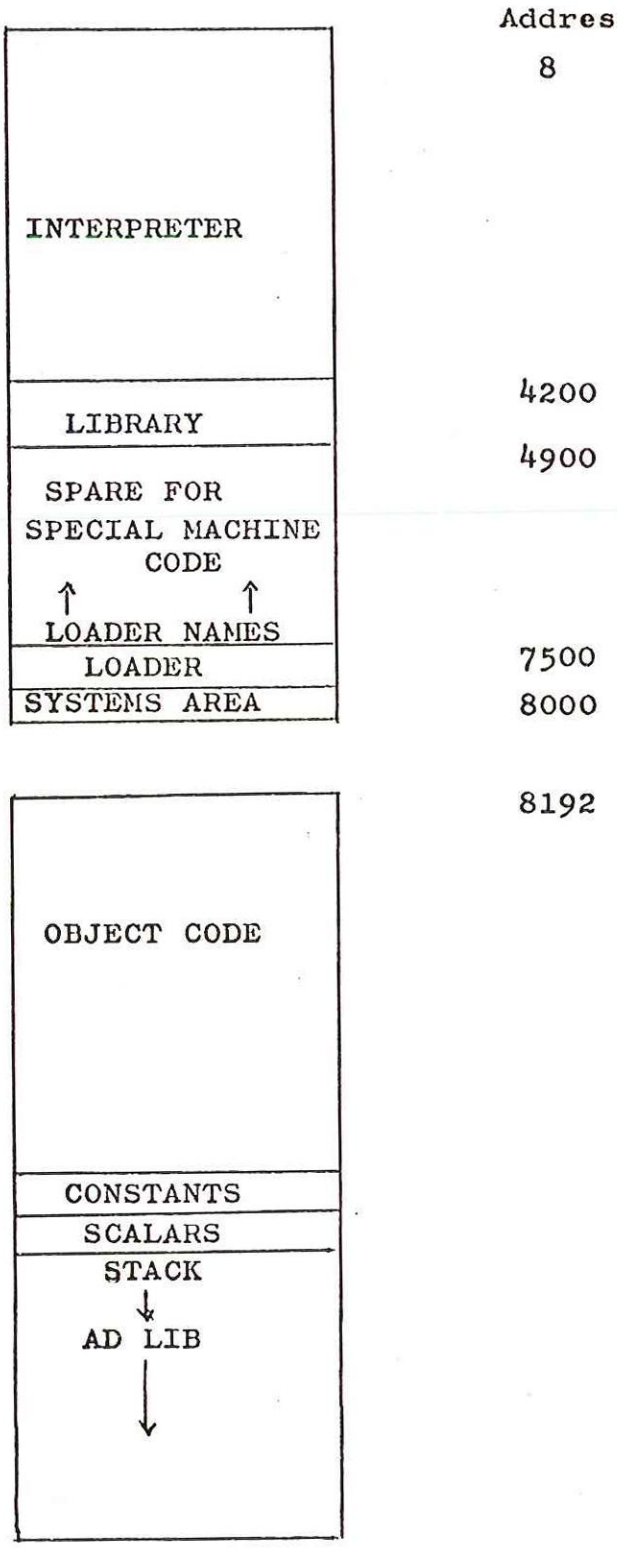
Basic machine; library detached



Address	Remarks
8	
4200	
5300	Labelled QACODL
5500	
5700	If any is present
6100	Labelled QAVNDA
6300	
7500	
8000	

FIGURE 3.

Multi module machine; suggested scheme



5. OBJECT CODE

5.1 As held in the store

The object code consists of 18 bit words which are interpreted at run time. Like machine code each word has a 5 bit (mnemonic) function part and a 13 bit address which is often a data reference:-

TIR N "Take Integer Result from location N in the Notional Data Area and store it in the stack; advance the stack pointer by three".

When the address is not a data reference then it is either a small integer:-

INDR 3n "Store in the stack an element from an n dimensional array".

why 3n

or it is the address of part of the object program being executed:-

IFJ N "If False Jump. The top-most stack location is regarded as a boolean result; reduce the stack pointer by three and, if false, jump to address N from the beginning of the object code".

or it is some miscellaneous quantity, sometimes a packed pair of items.

Of the 32 possible functions there are about 30 which are described below; one of these, called PRIM, is used to reach one of about 60 further subroutines and the entire collection of 90-odd subroutines forms the bulk of the interpreter.

Example, if a and b are integers, the Algol text:-

```
b: = a+5;
```

is translated into the following object code:-

```
TIA    "b"      Take Integer Address of b
TIR    "a"      Take Integer Result     a
TIC    "5"      Take Integer Constant   5
PRIM   I+I      Add Integers
PRIM   ST       Store result in b.
```

The references to b, a, 5, I+I and ST are all small integers in the actual code.

5.2 As punched on paper tape

There are two formats on the paper tape corresponding to the two input modes for the 920 and 903.

920 mode (7 bits in)		903 mode (8 bits in)
A B C D E . F G H	↑	A B C D E . F G H
I J K L M . N O P		I J K L M . N O P
Q R S T U . V W X		Q R S T U . V W X

direction of motion

A B C ←——— Loader code →B C D

D L T ←——— Parity bits →A I Q and are all zero.

EFGHIJKMNOPQRSUVWX ← Word itself → EFGHJKLMNOPRSTUVWX

It will be noted that in 903 mode the parity bits are all zero; in 920 mode the parity bits are correct on tape, but ignored by the hardware. However there is a checksum accumulated for every row of tape for protection against faulty punching.

The following loader facilities are used:

code 1 — Load as it stands

code 2 — Load after adding base address

code 3 — Update an implicit jump such as is needed around a procedure body or along a conditional expression.
Code 3 accompanies the word containing the address of the word to be updated. For such an update the following word is zero and is accompanied by loader code zero and this is followed by three blanks.
Code 3 followed by a non zero word is used to update an array or procedure checking word (see 8 below).

code 4 — With subcode 1 for the left hand global labels "qacod1 and "qavnda"; with subcode 2 for punching the call of library procedure names. These names are never accompanied by an increment.

code 5 — Skip n locations. This is used to reserve data space at the end of the program.

code 6 — This is a checksum.

code 7 — Stop loading and print "FIRST NEXT" message.

Generally speaking the chaining facility of the loader is only used for multiple calls on a library procedure where this latter is attached to the end of the pord tape.

ELLIOTT 903 ALGOL

Addition to Pord Manual December 1966

Section 6

The code number 5 has been allotted to a new function, INDFS, Index Formal Switch. It is punched with loader code 1 and has BN, n as its address part.

Action

"Find FP using BN; address := contents of (3n + FP); subtract 3 from SP; get integer from stack;

If integer is zero or negative or greater than contents of address then FAIL;

store 2 * integer - 1 + address at SP; add 3 to SP; "

Purpose

INDFS is needed at the call of P in the program below.

```
TEST;

begin
        switch S := L1, L2, L3;
        procedure P (A); value A; label A; goto A;
        procedure Q (B); switch B; P (B [2]) ;
        comment start of program;
        Q (S);
L1 : L3 : print "L` BAD` ; stop;
L2 :      print" L` GOOD`;

end;
```

D. Hunter

6. FUNCTION CODES

6.1 Glossary

PP	Pord Pointer value	13 bits
SP	Stack Pointer "	16 "
EP	Entry Pointer "	16 "
FP	Formal " "	16 "
BN	Block Number	9 " stored in bits 6-14 inc.
N	A relative address	13 bits
n	A parameter number	4 " stored in bits 15-18 inc.
m	The number of parameters	"
d	The number of dimensions	6 bits stored in bits 6-12 inc.
a	The number of arrays	6 bits stored in bits 13-18 inc.
PBA	Primitive Base Address (= address of beginning of a table of subroutine entry addresses	13 bits
QACODL	Address of start of Object Data Load	13 bits
QAVNDA	Address of start of Notional Data Area	13 bits
P	input output parameter	a small integer
BA	Base Address of programs	

6.2 Names of codes

Alphabetically listed names for 5 bit pord functions.

Mnemonic name	Explanation	Notes
CF	Call Function	
CFF	Call Formal Function	
GT	Go To	Refers to labels area
GTF	Go To Formal	
GTFS	Go To Formal Switch	
GTS	Go To Switch	
IFJ	If False Jump	Conditionals
IFUN	Integer Function	
INDA	Index Address	Subscript references
INDR	Index Result	
INDS	Index Switch	
INOUT	Input/Output	
MAMPS	Make Array Maps	At array declaration

Mnemonic name	Explanation	Notes
PE	Procedure Entry	
PEM	Procedure Entry machine code	
PRIM	Primitive Entry	
RFUN	Real Function	
TA	Take Address	In program
TF	Take Formal	Parameter references
TIA	Take Integer Address	
TIC	Take Integer Constant	
TICA	Take Integer Constant Address	
TIR	Take Integer Result	
TLA	Take Label Address	
TRA	Take Real Address	
TRC	Take Real Constant	
TRCA	Take Real Constant Address	
TRCN	Take Result Call by name	
TRR	Take Real Result	
UJ	Unconditional Jump	
(TRAP	Take Real Address in Program)	Hand pording only

6.3 Address parts

Mnemonic	Loader code	Function code	Punched Address	Referenced Address
CF	2 or 4	21	N or NAME	N+BA or library NAME entry
CFF	1	22	BN,n	3n+FP
GT	1	10	N	N+QACODL
GTF	1	11	BN,n	3n+FP
GTFS	1	14	BN,n	"
GTS	1	9	N	N+QACODL
IFJ	1 or 2	7	8191 or N	N+BA
IFUN	1	28	BN,n	3n+FP
INDA	1	12	3n	SP-3n etc see below
INDR	1	13	"	"
INDS	1	27	N	N+QACODL
INOUT	1	15	P	see below
MAMPS	1	6	d,a	see below
PE	1	23	BN,n	
PEM	1	30	n	
PRIM	1	31	N	N+PBA
RFUN	1	29	BN,n	3n+FP

Mnemonic	Loader code	Function code	Punched Address	Referenced Address
TA	2	0	N	N+BA
TF	1	24	BN,n	3n+FP
TIA	1	1	N	N+QAVNDA
TIC	1	18	N	N+QACODL
TICA	1	17	N	N+QACODL see also TLA
TIR	1	2	N	N+QAVNDA
TLA	1	17	N	N+QACODL see also TICA
TRA	1	3	N	N+QAVNDA
TRC	1	20	N	N+QACODL
TRCA	1	19	N	"
TRCN	1	26	BN,n	3n+FP
TRR	1	4	N	N+QAVNDA
UJ	1 or 2	8	8191 or N	N+BA
(TRAP	2	16	N	N+BA cf TA)

6.4 Actions specified

CF The intention of this is to make a partial stack entry before jumping to a procedure body.

```
"address:= N+BA;
LI: store EP at SP; store SP in Entry Pointer
    register; add 1 to SP; store PP at SP;
    add 1 to SP;
    PP:= address;"
```

CFF This is for a call of a formal function.

```
"Find FP using BN; address:= contents of
 3n+FP;  go to L1 in CF above;"
```

GT An ordinary GO TO is executed by reference to two words in QACODL, of which the first contains the label address and the second the Block Number. The Block Number is needed in case a jump to an outer block is involved.

```
"address:= N+QACODL;
L2: if BN equals contents of (address + 1)
    then go to contents of address;

L3: if contents of (EP+3) equals contents
    of (address + 1) then to to L4;
    EP:= contents of EP; go to L3;

L4: if contents of (EP+2) <0 then go to L5;
    SP:= contents of (EP+2);
    FP:= SP-3; go to L6;

L5: SP:= EP;

L6: EP:= contents of EP;
    BN:= contents of (EP+3);
    go to contents of address;"
```

GTF "Find FP using BN; address: = contents of (3n+FP); go to L2 in GT above;"

GTFS "Find FP using BN; address: = contents of (3n+FP); go to L7 in GTS below;"

GTS "address: = N+QACODL;
L7: subtract 3 from SP; get integer from stack; if integer is zero or greater than contents of address then FAIL;
address: = 2 x integer -1 + address; go to L2 in GT above;"

IFJ "subtract 3 from SP; get boolean (integer) from stack; if zero then go to address N+BA;"

IFUN This stacks the address of a function name if n = o or of a parameter if n ≠ o.

"Find FP using BN; store(3n+FP) at SP; add 1 to SP; store the constant +1 at SP; add 2 to SP;"

The constant +1 indicates that the address is that of an integer or boolean variable; this is checked at a procedure entry against the parameter checking word (see 8 below).

INDA The purpose of this is to place the absolute address of an array element in the stack having been given the address of the map entry and n index values. If the array has real elements then the most significant digit of the result is a one, picked up from the array map. Next to this is placed +1 for an integer or +2 for a real array element. See 8 below.

INDR As for INDA except that the value of the array element is brought to the stack and unpacked into 3 locations if real.

INDS "address: = N+QACODL; subtract 3 from SP; get integer from stack;

if integer is zero or greater than contents of address then FAIL;

store 2 x integer -1 +address at SP; add 3 to SP;"

INOUT See section 9; the address part of the pord, p, determines the action in an identical way to that of the address part of a PRIM pord.

MAMPS This operation is executed immediately after entry to a block in which local arrays are declared. The number of dimensions, d, and arrays, a, are

packed with 6 bits each at the right hand end of the word containing MAMPS.

begin real array A,B,C,D [P:Q, R:S, T:U];

is translated to

```
        TIR     P
        TIR     Q
        TIR     R
        TIR     S
        TIR     T
        TIR     U
        MAMPS   3,4
A:      /0      0       / indicates real
         3      7       3 dimensions; 7 locations
                        onwards to map address

B:      /0      0
         3      5
C:      /0      0
         3      3
D:      /0      0
         3      1
        +0              Map address filled in here
                        by action of MAMPS (equal to
                        SP).
```

The indications "A:" etc do not mean that a global label is punched, merely that reference to array A is by specifying the address of the "array pair" marked "A:".

Using the top six entries in the stack an array map is calculated and placed in the stack, advancing SP appropriately. The array map for a d, dimensional array contains 2d + 1 words in all. These are arrayed as follows (for d =3):-

```
array map :     TOTAL SIZE
                OFFSET
                l1
                c1
                l2
                c2
                l3
```

Note that there is one more lowbound value, l, than map coefficient, c, TOTAL SIZE is the number of locations occupied by the array itself. OFFSET is the number of locations between the leading element of the array and the element all of whose index values are zero; it is positive if the element with zero indices lies at a higher address than the leading element.

The coefficients c1, c2 etc are calculated as follows:-

c1 is the range of the first subscript multiplied by two if the array is real.

c2 is the range of the second subscript multiplied by c1.

The lowbound values $l1$, $l2$ etc are not necessary for the evaluation of the address of an element of the array, but are necessary for the standard procedure lowbound.

The address of the leading element of each array is filled in at A:, B: etc., the first being equal to SP, the next to SP + TOTAL SIZE etc. OFFSET is then calculated

OFFSET = - 2 x P - R x c1 - T x c2

Finally there is a jump to execute the next pord beyond the map address.

The address of element B [i,j,k] is -
the contents of location B: with its "/" to indicate real plus
OFFSET
plus
2i + j x c1 + k x c2

PE This operation is at the head of a block bearing the number BN′ and expecting m parameters. The action is to complete the stack entry set up by CF or by CFF, and do parameter checking (see 8 below).

BN occupies 9 bits, m 4 bits.

"store EP-3m at SP; add 1 to SP; store (old) BN at SP; add 1 to SP; store BN′ in Block Number register; store EP-3m-3 in Formal Pointer register;"

PEM This is placed at the head of a piece of machine code to make it look like pords when called from within the User's Algol program.

Example.

```
[ARCTAN]
 ARCTAN   PEM    1
          +0                for return address
          ...
          ...
          ...
          0  ARCTAN+1       normal exit in machine
          /8 1              code
```

The stack entry is completed as for a PE operation but using the current value of BN in place of BN'. There is then a call of the machine code from within the interpreter. On returning to the interpreter in the normal way for a machine code subroutine the action for RETURN (see 7.2 below) are executed.

PRIM Enter a subroutine at address N+PBA.

RFUN This stacks the address of a real type procedure or a real parameter so that an assignment can be made to it.

```
"find FP using BN;
store FP+3n + /0 0 at SP;  add 1 to SP;
store /0 2 at SP;  add 2 to SP;
```

The sign bit on the address word indicates a real address, and the sign bit on the +2 indicates an unpacked quantity. The +2 indicates a real quantity for parameter checking (see 8 below).

TA "store N+BA at SP; add 3 to SP;"

```
TF   "find FP using BN;
     store contents of (3n+FP) at SP;
     store contents of (3n+FP+1) at SP+1;
     store contents of (3n+FP+2) at SP+2;
     add 3 to SP;
```

TIA "store N+QAVNDA at SP; add 1 to SP; store +1 at SP; add 2 to SP;"

TIC "store contents of (N+QACODL) at SP; add 3 to SP;"

TICA, TLA "store N+QACODL + the constant 8 0 at SP; add 1 to SP; store +1 at SP; add 2 to SP;"

The +1 written by TLA gets overwritten by +9 if a label is an actual parameter.

TIR "store contents of (N+QAVNDA) at SP; add 3 to SP;"

TRA "store N+QAVNDA + the constant /0 0 at SP; add 1 to SP; store +2 at SP; add 2 to SP;

TRC "store contents of (N+QACODL) at SP; unpack the contents of (N+1+QACODL) and store at SP+1, SP+2; add 3 to SP;"

TRCA "store /8 0 plus N+QACODL at SP; store +2 at SP+1; add 3 to SP;

TRCN "Find FP using BN; address: = contents of 3n+FP; store contents of address at SP; if address has sign bit =1 then
begin if contents of 3n+FP+1 has sign bit =1 then transfer contents of (address +1) to SP+1 and (address +2) to SP+2 else unpack contents of (address +1) and transfer to SP+1, SP+2 end;
add 3 to SP;

TRR "store contents of (N+QAVNDA) at SP; unpack the contents of (N+1+QAVNDA) and store at SP+1, SP+2;
add 3 to SP;

UJ "PP: = N+BA;"

TRAP "store /8 0 plus N+BA at SP;
store +2 at SP+1; add 3 to SP;"

This is only for referring to real constants in hand ported programs. This function is not produced by the translator.

7. PRIMITIVES

7.1 Listed in numerical order

Each primitive is a subroutine and is numbered from 1 to 70; those which are numbered 1-29 and 63-70 are organisational and the remainder are arithmetic, relational, logical or standard procedure routines such as log and exp.

The numberical value of the primitive is used to enter the routine via a table of addresses, and each pord has the function part 31 or /15.

Primitive number	Mnemonic	Explanation
1	CBL	Call Block
2	CHECKB	Check boolean
3	CHECKI	Check integer
4	CHECKR	Check real
5	CHECKS	Check string
6	DO	
7	STW	Store While
8	FINISH	End of program
9	FOR	Start of FOR statement
10	FR	For Return
11	FSE	For statement end
12	DIV	Integer divide
13	ITOR1	Integer to Real 1

Primitive number	Mnemonic	Explanation
14	ITOR2	Integer to Real 2
15	NEGI	Negate integer
16	NEGR	Negate real
17	RETURN	At end of block
18	RTOI1	Real to integer 1
19		
20	ST	Store
21	STA	Store Also
22	STEP	
23		
24	WAIT	Wait
25		
26	UNTIL	
27	UP	
28	R↑I→R	Arithmetic
29	WHILE	
30	I+I→I	
31	R+R→R	
32	I-I→I	
33	R-R→R	
34	I*I→I	
35	R*R→R	Arithmetic
36	I/I→R	
37	R/R→R	
38	I↑I→I	
39	I↑I→R	
40	R↑R→R	
41	I<I→B	
42	R<R→B	
43	I≤I→B	Relational
44	R≤R→B	
45	I=I→B	
46	R=R→B	
47	I≠I→B	
48	R≠R→B	
49	I>I→B	Relational
50	R>R→B	
51	I≥I→B	
52	R≥R→B	
53	B∧B→B	
54	B∨B→B	
55	B≡B→B	Logical
56	B⊃B→B	
57	¬B→B	
58	ABS	
59	ENTIER	
60	EXP	Standard procedures

Primitive number	Mnemonic	Explanation
61	LN	Standard procedures
62	SIGN	
63		These place integer x in stack at SP-2 where x runs from 3 to 10 for primitives 63-70. The purpose is for parameter checking.
64		
65		
66		
67		
68		
69		
70		

7.2 Actions specified

1 CBL — This has the same effect as a function CF to a point two pords on and therefore makes a partial stack entry.

2-5 CHECK — These punch "newline, asterisk" followed by a boolean, integer or real whose value is at the top of the stack or a string whose address is at the top of the stack. SP is unchanged except for CHECKS which reduces it by 3.

6 DO — This makes use of the common procedure ASSIGN (see below).

"ASSIGN; store PP at EP+1;
store contents of (EP+2) in Pord Pointer register;"

7 STW — "ASSIGN;"

8 FINISH — Punch 100 rows of blank tape, the word FINISH and a halt code. Then stop.

9 FOR — A stack entry is made. In the object program FOR is followed by three pords.

PRIM FOR
a — address of controlled statement
BN'
b — address of next statement

"store EP at SP;
store (PP+4) at SP+1;*
store a + /0 0 at SP+2;
store old BN at SP+3;
store b at SP+4;
EP: = SP; add 5 to SP;
store BN' in Block Number register;"

10 FR — This ends a controlled statement.
"PP: = contents of (EP+1);"

* PP+4 points to 5 locations after PRIM FOR

11	FSE	This undoes the effect of FOR and is the last for list element. "BN: = contents of (EP+3); PP: = contents of (EP +4); SP: = EP; EP: = contents of EP;"
12	DIV	See arithmetic primitives.
13 14	ITOR1 ITOR2	Convert contents of SP-3 and SP-6 from integer to unpacked real respectively.
15 16	NEGI NEGR	Replace the contents of SP-3 by minus the contents.
17	RETURN	Used at the end of a block or procedure. "BN: = contents of (EP+3); SP: = contents of (EP+2); PP: = contents of (EP+1); EP: = contents of EP; FP: = contents of (EP+2) minus 3;"
18	RTOI1	Convert contents of SP-3 from unpacked real to integer with FAIL if overflow.
20	ST	This makes use of the common procedure ASSIGN (see below) "ASSIGN; subtract 3 from SP;
21	STA	"ASSIGN; copy the contents of SP to SP-3 (SP+1) to SP-2 (SP+2) to SP-1;"
22	STEP	"ASSIGN; store PP at EP+1; clear location EP+7;"
24	WAIT	Halt but prepare to resume after a start at 9.
26	UNTIL	"If the contents of (EP+7) are zero then make them non zero else add the contents of (EP+8) to the variable whose address is at EP+5; If the sign of (the increment held at EP+8 times (the controlled variable whose address is at EP+5 minus the final value given in (EP+11)) is greater than zero then store PP at EP+1; this corresponds to element exhausted; subtract 6 from SP; If the above sign is $\leq$ 0 then store the contents of (EP+2) in the pord pointer register; this causes the statement to be executed;"

The type of arithmetic is determined* by the address of the controlled variable held at EP+5.

27 UP — This makes space in the stack for the result of a type procedure.

"Add 3 to SP;"

29 WHILE — If the topmost value is false then the next pord is taken, but if true then a pord jump occurs to the address given in EP+2.

```
"subtract 3 from SP;
if false store PP in EP+1;
if true store contents of (EP+2) in
Pord Pointer register;"
```

```
procedure ASSIGN; "subtract 3 from SP;
address: = contents of (SP-3);
type: = contents of (SP-2);

if address has an "8 0" bit then
FAIL because assignment to a
constant is being attempted;

store contents of SP at address;

if address < 0 then begin,if type
< 0 then

    transfer contents of (SP+1) to
    address+1 and contents of (SP+2)
    to address +2

                else
    pack together the contents of (SP+1)
    and (SP+2) with round off and store
    at address +1

end;"
```

The primitives numbered 30-56 have two inputs and one output. The left hand input is at SP-6, the right hand input is at SP-3. The output occupies SP-3 or, if real, the three words starting at SP-3. The nett effect of each primitive is to reduce SP by 3. Failure actions are not indicated.

Arithmetic	30-40	(and div no.12)
Relational	41-52	
Logical	53-57	(note 57 is like NEGI)

* Packing is determined by the contents of (EP+6).

The primitive numbered 58-62 are the ones which execute the Algol procedures which are always in the store. A call of one of these procedures does not produce a PRIM UP as an ordinary procedure call would. The actions of all these primitives are:-

"replace contents of (SP-3) by the function of the contents".

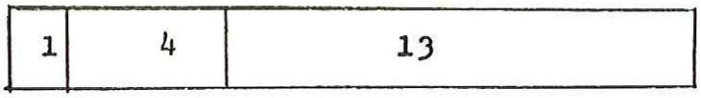
The primitives numbered 63-70 serve the purpose of filling the stack with type information alongside the address of any call-by name actual parameter. This is for parameter checking.

8. PARAMETER CHECKING

At each procedure entry having m parameters there are m parameter checking words following the "PE BN,m" pord. During the execution of the PE pord, a check is made between the relevant word which describes the formal parameter, and the corresponding actual parameter given in the stack; the check includes number of dimensions or parameters.

No check is made for a formal parameter called by value.

If an array parameter is called by value then the copying of the array is done within the PE pord.



a checking word

v x dim

v=1 if called by value

x has values as given below.

dim contains the number of dimensions plus the base address of the pord program; this latter is a nuisance but is an inevitable consequence of the way the checking word is updated by the loader program.

value of x	parameter type	value of dim
1	integer or boolean	0
2	real	0
3	integer or boolean array	n+BA
4	real array	n+BA
5	integer or boolean procedure	n+BA
6	real procedure	n+BA
7	procedure	n+BA
8	switch	0
9	label	0
10	string	0

It will be noticed that all pords which stack a real or integer address also fill the adjacent location with +2 or +1 as in the table above. Pords corresponding to actual parameters of types 3-10 are each followed by the appropriate primitives.

9. INPUT OUTPUT

These operations all have a function part of 15 , INOUT and an address part p.

	value of p	action	
*	1	read an integer	
*	2	read a real	
	3	print an integer	
	4	print a real	
	5	aligned	global
	6	punch	
	7	digits	
	8	freepoint	
	9		
	10		
*	11	prefix	global
	12	sameline	
	13	scaled	
	14	reader	
*	15	print a string within a print list	
	16	aligned	local
	17	punch	
	18	digits	
	19	freepoint	
	20	reset the local settings from the global settings.	
	21		
*	22	prefix	local
	23	sameline	
	24	scaled	
	25	reader	

The parameters to all the above lie in the stack - if there is one at SP-3 and if there are two at SP-6 and SP-3. The operations with an asterisk * have addresses in the stack and all the others, with the exception of sameline, have values.

All parameters are cleared from the stack by decreasing SP appropriately.

The global and local presumed settings are affected by the operations as indicated.

A string is stored with its opening and closing string quotes packed three characters to a word left justified and space filled; inner strings in the 503 Algol sense are permitted. The inner string is interpreted on output.

10. LIBRARY AND MACHINE CODE PROCEDURES

The library is a set of machine code procedures each introduced by a PEM pord but there is no reason why hand porded procedures should not be added. A hand porded procedure must start with a PE pord containing a Block Number less than 50 and be followed by the correct parameter checking words.

Example

10.1 As declared near the front of the Algol text.

```
code

    integer  procedure   SUM (a, b);
    value a;  integer a, b;

algol;
```

This is not needed for library procedures such as arctan which behave exactly as if such a declaration had already been made before the User's program is read.

10.2 As called from further on in the text

"x : = SUM (y+y , z);" is translated to

```
TIA    "x"
PRIM    UP        make room for result
TIR    "y"
TIR    "y"
PRIM   I+I → I
TIA    "z"       call by name
CF     SUM       using loader global name
PRIM   ST
```

10.3 As offered for assembly by SIR for "SUM: = a+b"

```
[SUM]
 SUM      /14     2
          +0
          0       38
          /0      6
```

```
        /4    0
         0    38
        /1    3
        /5    0
         0    SUM+1
        /8    1
%
```

10.4 As hand porded

```
[SUM]
 SUM    /7    642        Block Number 40
        /1    0          Checking words
         1    0
        /12   640
        /8    641
        /10   642
        /15   30
        /15   20
        /15   17
%
```

11. EXAMPLE TRANSLATIONS

In what follows the pords have been written symbolically as far as possible and in a notation which is close to SIR in the belief that this helps the explanation.

11.1 Notation

The function parts of pords are given mnemonically e.g. TIA, TRR etc.

A variable name in small case replaces its address in QAVNDA e.g. TIA x replaces TIA 5 if the address of x is 5.

A parameter name replaces BN,n.

A primitive name replaces its number e.g. PRIM ST for PRIM 20.

The address part of a pord referring to a constant is given correctly in its absolute form and the value of the constant mentioned in a comment.

The names of switches, labels, arrays and procedures are freely used as local SIR labels for explanatory purposes only.

Implicit forward jumps use a series of local SIR labels beginning R1, R2 for historical reasons.

Global labels are declared for QACODL, QAVNDA, the name of the program itself, any library calls and any calls of machine coded procedures.

The semicolon convention e.g. ;+2 and 84; is used

11.2 Program without a run time block

```
TEST1;
      begin  integer  a, b, c;
             b: = 6;
             a: = c: = b+5
      end;
*+0

[TEST1   QACODL   QAVNDA]
TEST1    INPUT    20
         TIC      2        (+3)
         INOUT    17
         UJ       9;
         £''L
         £3`.
         £TES
         £T1
         £`
         TA       4;
         INOUT    15       (PUNCHES PROG NAME)

         TIA      b
         TIC      3        (+6)
         PRIM     ST

         TIA      a
         TIA      c
         TIR      b
         TIC      4        (+5)
         PRIM     I+I→I
         PRIM     STA
         PRIM     ST

         PRIM     FINISH

QACODL    +0
          +1
          +3
          +6
          +5

QAVNDA
>+4
%
```

11.3 Program with run time block.

```
TEST2;
      begin  switch  s: = loop, ret;
             integer a, b;
             ret   .: b: = 4;
             loop: a: = b; go to ret
      end;
```

As for TEST1 for the first 1ϕ pords then

```
        PRIM    CBL
        UJ      R1
        PE      816         (BN = 51)

RET     TIA     b
        TIC     3           ( +4)
        PRIM    ST

LOOP    TIA     a
        TIR .   b
        PRIM    ST

        GT      6           (RET)

        PRIM    RETURN

R1      PRIM    FINISH

QACODL   +0
         +1
         +3
S        +2
          0     LOOP
         +816
          0     RET
         +816
         +4
QAVNDA
>+3
%
```

The only difference between a program translated from Algol and the example of 11.2 or 11.3 as assembled by SIR in relocatable binary form, is that the Algol tape has a checksum inserted just before QAVNDA; if this is not done then the first library tape which gets copied after QACODL will cause a checksum failure at load time.

The remaining examples are usually excerpts from complete programs.

11.4 Subscript variables

```
TEST3;
      begin  array  a,b,c [1:10, -5, 6];
             real  x,y;
             integer i,j;
             i: = j: = 4;
             c [i,j]: = 0;
             x: = a [2, 5];
      end;
```

As for TEST1 and TEST2 then

```
          PRIM    CBL
          UJ      R1
          PE      816

          TIC     1          (+1)
          TIC     3          (+10)
          TIC     4          (+5)
          PRIM    NEGI
          TIC     5          (+6)
          MAMPS   131        (d=2, a=3)

A         /0      0
          2       5
B         /0      0
          2       3
C         /0      0
          2       1
          +0

          TIA     i
          TIA     j
          TIC     6          (+4)
          PRIM    STA
          PRIM    ST

          TA      C
          TIR     i
          TIR     j
          INDA    6          (3n)
          TIC     0          (+0)
          PRIM    ITOR1
          PRIM    ST

          TRA     x
          TA      A
          TIC     7          (+2)
          TIC     4          (+5)
          INDR    6
          PRIM    ST
```

```
        PRIM    RETURN

R1      PRIM    FINISH

QACODL  +0
        +1
        +3
        +10
        +5
        +6
        +4
        +2

QAVNDA
>+7
%
```

11.5 Conditional statement

if ∝ < y then go to CYCLE;

```
        TIR     ∝
        TIR     y
        PRIM    R<R→B
        IFJ     R17
        GT      cycle
R17     ...
        ...
```

11.6 Conditional expression

This illustrates the IFIP rule that the type of an expression can be taken to be real if this depends on something at run time -

i: = if k < 0 then j else j + 2.0;

```
        TIA     i
        TIR     k
        TIC     0           (+0)
        PRIM    I<I→B
        IFJ     R10
        TIR     j
        UJ      R11

R10     TIR     j
        TRC                 (+2.0)
        PRIM    ITOR2
        PRIM    R+R→R
        UJ      ;+2

R11     PRIM    ITOR1
        PRIM    RTOI1
        PRIM    ST
```

11.7 Switch

```
        go to S[k];

        TIR   k
        GTS   S
```

11.8 Input output

11.8.1 print x*y, sameline, digits (3), z;

```
        INOUT   20          (reset settings)
        TIR     x
        TIR     y
        PRIM    IxI→I
        INOUT   3           (print integer)

        INOUT   23          (sameline local)

        TIC     2           (+3)
        INOUT   18          (digits local)

        TRR     z
        INOUT   4           (print real)
```

11.8.2 read a[i,j], k;

```
        INOUT   20          (reset settings)
        TA      A
        TIR     i
        TIR     j
        INDA    6
        INOUT   2           (read a real)

        TIA     k
        INOUT   1           (read an integer)
```

11.8.3 digits (4);

```
        TIC                 (+4)
        INOUT   7           (digits global)
```

11.8.4 print ''L2`TABLE`;

```
        INOUT   20
        UJ      R23
        £''L
        £2`T
        £ABL
        £E`
   R23  TA      84;        (assumes £''L at location
                            84)
        INOUT   15
```

11.9 for statement

for x := 5,6,x + 1 while x < 10,12 step 1 until 18 do

```
        PRIM    FOR
        0       R14
        +864                        (BN = 54)
        0       R15
        TIA     x

        TIC                         (+5) <---
        PRIM    DO
        TIC                         (+6) <---
        PRIM    DO
        TIR     x
        TIC     1                   (+1) <---   for list
        PRIM    I+I                             elements
        PRIM    STW
        TIR     x
        TIC                     (+10)
        PRIM    I<I→B
        PRIM    WHILE
        TIC                     (+12)
        PRIM    STEP
        TIC     1                (+1) <---
        TIC                     (+18)
        PRIM    UNTIL

        PRIM    FSE             <------------ end
                                              dummy
R14     ...
        controlled statement
        ...
        PRIM    FR
R15
        ...
        next statement
        ...
```

11.10 Library Procedure call

11.10.1 x: = abs (y);

```
        TRA     x
        TRR     y
        PRIM    ABS    (always in interpreter)
        PRIM    ST
```

11.10.2.x: = sin (y);

```
        TRA     x
        PRIM    UP
        TRR     y
        CF      SIN    (punched as right hand
                                global name)
        PRIM    ST
```

August 1966

In this example it is as if SIN had been added to the SIR declarations at the start of the program. The loader does not care whether SIN occurs before or after this point and establishes a chain for a series of calls.

11.11 Procedure body and call

```
begin integer procedure  SUM (a,b);  value a,
                                     integer a,b;

                SUM: = a+b;

      Integer x,y;

       x: = SUM (4,5);

end;
```

```
            PRIM    CBL
            UJ      R1
            PE      816         (BN = 51)
            UJ      R2
SUM         PE      834         (BN = 52, m=2)
            /1      0
            1       0
            IFUN    832
            TF      a
            TRCN    b
            PRIM    I+I → I
            PRIM    ST
            PRIM    RETURN      (END OF PROC BODY)
R2          TIA     x
            PRIM    UP
            TIC                 (+4)
            TICA                (+5)
            CF      SUM
            PRIM    ST
            PRIM    RETURN

R1          PRIM    FINISH
```

11.12 Permissible actual/formal correspondence

The general idea is that at the point of call, a value is left in the stack for every formal parameter called by value and an address is left for those which are called by name. Type conversion is explicitly done. The pords at the point of call are shown below:-.

11.12.1 Formal real called by value

```
Declared  real      TRR
    "     integer   TIR  and PRIM  I TO R1
```

```
array element      ... INDR and possibly
                                  convert.
type procedure     PRIM UP ... CF and
                          possibly convert.
constant           TRC or TIC and PRIMITOR1

Formal real on value list     TF
Formal real called by name    TRCN
Formal integer      etc       as real with
                                 conversion.
```

11.12.2 Formal real called by name

```
Declared real          TRA
Constant               TRCA
Formal real on value list  RFUN
Formal real called by name TF
```

Inside the body the parameter may appear on the left or the right of a: = sign and it may be called by name or value. These four cases produce pords as follows:-

	On the left	On the right
By name	TF	TRCN
By value	RFUN	TF

For arrays, procedures and strings which all have address among the program pords, a TA pord is used at the original point of call; if the parameter is to be handed on to an inside procedure then at the inner point of call a TF pord is produced.

A label may or may not be on the value list; if so then the actual parameter can be either a label TLA, or the element of a switch TIR, INDS ... It is regretted that it cannot be the element of a formal switch.

Array and procedure parameters must always correspond in type and number of dimensions or parameters. At the point of call the pord which refers to the following actual parameters is followed by the indicated primitive which places an integer in the stack for checking purposes.

boolean or integer	array	PRIM	63	+3
real	array		64	+4
" "	integer procedure		65	+5
	real procedure		66	+6
	procedure		67	+7
	switch		68	+8
	label		69	+9
	string		70	+10

Inside the procedure body, at the first reference to an array or procedure parameter the checking word which was punched out at the head of the body is altered to indicate the number of dimensions or parameters. This is done by using loader code 3. If there is no reference to the parameter in question, then the address part of the checking word contains +8191 at run time.

DGNH.