A MANUAL OF THE DASK ALGOL LANGUAGE

A supplement to the ALGOL 60 report

Second edition

by

Jørn Jensen, Toke Jensen, Per Mondrup, Peter Naur

REGNECENTRALEN, COPENHAGEN

1961

Jens Damgaard Andersen

A MANUAL OF THE DASK ALGOL LANGUAGE

A supplement to the ALGOL 60 report

Second edition

by

Jørn Jensen, Toke Jensen, Per Mondrup, Peter Naur

REGNECENTRALEN, COPENHAGEN

1961

CONTENTS.

INTRODUCTION.

DASK ALGOL is a hardware representation of the ALGOL 60 language, suitable for the machine DASK of Regnecentralen, Copenhagen. Since DASK ALGOL lies very close to the reference language it has been found practical to base the description of DASK ALGOL directly upon the ALGOL 60 report as far as the basic characteristics are concerned. The exact specifications of DASK ALGOL are then defined through the set of corrections and extensions of the ALGOL 60 report given in the present Manual . Because of this intimate relation to the ALGOL 60 report the numbering of sections within the present Manual has been chosen to be a direct continuation of the section numbers of the ALGOL 60 report.

The conventions and methods used in DASK ALGOL were developed over the period February 1959 - August 1960. The great stimulus received at the early stages of the work from the general advisors of the ALCOR group, Professors F. L. Bauer and K. Samelson and Mr. M. Paul, Mainz, is gratefully acknowledged. Discussions with Professor A. von Wijngaarden, Dr. E. W. Dijkstra and Mr. J. A. Zonneveld of the Mathematical Centre, Amsterdam, have also been a great stimulus to the work.

The actual work was done by

Jørn Jensen (JJ)
Toke Jensen (TJ)
Per Mondrup (PM)
Peter Naur (PN)

The general conventions are due to JJ, PM and PN. The actual coding was done by JJ (block and parameter administration, standard input procedures), TJ (input of the ALGOL program), and PM (arithmetics, standard functions, standard output procedures). Checking and debugging was done by PN, who also worked out the present Manual.

The typing of the Manual was done by Kirsten Andersen.

The chapter 12 of the Manual is still in preparation at the present moment.

6. 8-CHANNEL PUNCH TAPE CODE AND FLEXOWRITER KEYBOARD.

6.1. PRINTED SYMBOLS.

Lower case	Upper case	Code	Lower case	Upper case	Code
a	A	oo . o	w	W	o .oo
b	B	oo . o	x	X	oo .ooo
c	C	ooo . oo	y	Y	ooo.
d	D	oo .o	z	Z	o o. o
e	E	ooo .o o	æ	Æ	ooo .
f	F	ooo .oo	ø	Ø	o oo. oo
g	G	oo .ooo	0	^	o .
h	H	oo o.	1	v	. o
i	I	oooo. o	2	×	. o
j	J	o o . o	3	/	o . oo
k	K	o o . o	4	=	.o
l	L	o . oo	5	[	o .o o
m	M	o o .o	6	]	o .oo
n	N	o .o o	7	)	.ooo
o	O	o .oo	8	(	o.
p	P	o o .ooo	9	)	oo. o
q	Q	o oo.	,	10	ooo. oo
r	R	o o. o	.	:	oo o. oo
s	S	oo . o	-	+	o .
t	T	o . oo	<	>	oo . o
u	U	oo .o	_	|	o.oo
v	V	o .o o			

The key for _| does not advance the carriage.

6.2. TYPOGRAPHICAL SYMBOLS.

LOWER CASE	oooo. o	UPPER CASE	oooo.o	SPACE	o .
CAR RET	o .	TAB	ooo.oo		

6.3. CONTROL SYMBOLS.

STOP CODE	o. oo	TAPE FEED	oooo.ooo	PUNCH ADRES	o .
PUNCH OFF	o o.ooo	PUNCH ON	o o.o	AUX CODE	o.o

PUNCH ADRES and AUX CODE insert their respective codes when depressed simultaneously with any other key.

6.4. FLEXOWRITER KEYBOARD.

START READ	STOP READ	PUNCH ADRES									AUX CODE	STOP CODE	TAPE FEED

TAB	PUNCH OFF	× 2	/ 3	= 4	; 5	[6	] 7	(8	) 9	^ 0	v 1	_	|	PUNCH ON

	Q q	W w	E e	R r	T t	Y y	U u	I i	O o	P p	> <	CAR RET

LOWER CASE	A a	S s	D d	F f	G g	H h	J j	K k	L l	Æ æ	Ø ø	LOWER CASE

UPPER CASE	Z z	X x	C c	V v	B b	N n	M m	10 ,	: .	+ -	UPPER CASE

SPACE

6.5. NUMERICAL REPRESENTATIONS.

In the following table the characters have been arranged according to the numerical equivalent of the hole combination (after removal of the parity check hole). The first column gives the decimal value of the character, the second the sedecimal value, and the third and fourth columns give the lower and upper case character, respectively.

		LOWER	UPPER
0	00	SPACE	
1	01	1	∨
2	02	2	×
3	03	3	/
4	04	4	=
5	05	5	;
6	06	6	[
7	07	7	]
8	08	8	(
9	09	9	)
10	0A	(NOT USED)	
11	0B	STOP CODE	
12	0C	END CODE	
13	0D	(NOT USED)	
14	0E	¯	|
15	0F	(NOT USED)	
16	10	0	∧
17	11	<	>
18	12	s	S
19	13	t	T
20	14	u	U
21	15	v	V
22	16	w	W
23	17	x	X
24	18	y	Y
25	19	z	Z
26	1A	(NOT USED)	
27	1B	'	↑0
28	1C	CLEAR CODE	
29	1D	(NOT USED)	
30	1E	TAB	
31	1F	PUNCH OFF	
32	20	-	+
33	21	j	J
34	22	k	K
35	23	l	L
36	24	m	M
37	25	n	N
38	26	o	O
39	27	p	P
40	28	q	Q
41	29	r	R
42	2A	(NOT USED)	
43	2B	ø	Ø
44	2C	PUNCH ON	
45	2D	(NOT USED)	
46	2E	(NOT USED)	
47	2F	(NOT USED)	
48	30	æ	Æ
49	31	a	A
50	32	b	B
51	33	c	C
52	34	d	D
53	35	e	E
54	36	f	F
55	37	g	G
56	38	h	H
57	39	i	I
58	3A	LOWER CASE	
59	3B	.	:
60	3C	UPPER CASE	
61	3D	SUM CODE	
62	3E	(NOT USED)	
63	3F	TAPE FEED	
64	40	CAR RET	

7. THE RELATION BETWEEN DASK ALGOL AND ALGOL 60.

7.1. BASIC SYMBOLS.

7.1.1. Single character symbols.
7.1.1.1. Letters and digits. DASK ALGOL adds the letters
æ Æ ø Ø
to the reference alphabet. The appearance of all letters and digits may be seen from section 6.
7.1.1.2. Delimiters. As apparent from section 6 the following simple reference language symbols are directly available in DASK ALGOL:
+ - × / < = > ∨ ∧ , . $_{10}$: ; ( ) [ ]

7.1.2. Compound symbols.
7.1.2.1. Underlined words. Underlined words are produced in DASK ALGOL by depressing the underline (_) key immediately preceding each letter of the word. The symbols are the following:
true false go to if then else for do step until while comment begin end own boolean integer real array switch procedure string label value
Note: In DASK ALGOL boolean is spelled with small letter.
7.1.2.2. Compound symbols similar to reference language. The following compound symbols, most of which are produced by combining the underline (_) or stroke (|) with other characters, are similar to those of the reference language:
≤ ≥ ≠ ≡ :=
7.1.2.3. Compound symbols differing from reference language. The following compound symbols show a noticable deviation from the reference language:

Reference language	↑	¬	␣	'	'
DASK ALGOL	↑	-,	⊥	{	}

7.1.3. Reference symbols omitted in DASK ALGOL.
The following symbols are not included: ↑ ⊃

7.2. USE OF comment.

7.2.1. Two special forms of ALGOL comments, viz. the forms
comment drum data
and
comment drum program
will be recognized by the DASK ALGOL translator and will influence the storage allocation (cf. section 10). No other comments will be influenced by this convention.

7.2.2. The third form of comment is permitted only in the form:
end <any sequence of digits or letters>

7.3. IDENTIFIERS, NUMBERS.

7.3.1. Identifiers may have any length, but characters following the first 6 will be ignored by the translator.
7.3.2. Variables declared to be integer must lie in the range
$-524288 \leq \text{integer} \leq 524287$
7.3.3. The range of non-zero real variables is
$2.94_{10}-39 \leq \text{abs(real)} \leq 3.40_{10}38$

7.4. RESERVED IDENTIFIERS.

The complete list of reserved identifiers arranged alphabetically is as follows:

Identifier	Reference	Identifiers	Reference
abs	3.2.4	skrvvr	8.6
arctan	3.2.4	sqrt	3.2.4, 7.5
cos	3.2.4	streng	9.6
entier	3.2.5, 7.5	tryk	8.3
exp	3.2.4, 7.5	trykende	8.7
ln	3.2.4, 7.5	trykklar	8.7
læs	9.4	trykkopi	9.7
læsstreng	9.6	trykml	8.5
læst	9.5	trykslut	8.7
sign	3.2.4	trykstop	8.6
sin	3.2.4	tryksum	8.7
skrv	8.3	tryktab	8.6
skrvkopi	9.7	tryktekst	8.4
skrvml	8.5	tryktom	8.5
skrvtab	8.6	trykvr	8.6
skrvtekst	8.4		

7.5. ALARMS OF STANDARD FUNCTIONS.

Misuse of the standard functions will cause alarms during the run of the ALGOL program as follows:

Identifier	Cause of alarm	Typed indication
entier	Argument exceeds the interval for integers (cf. section 7.3.2)	spild entier
exp	The function value exceeds the range for reals (cf. section 7.3.3)	spild exp
ln	The argument is non-positive	spild ln(-)
sqrt	The argument is negative	spild sqrt(-)

7.6. ARITHMETIC EXPRESSIONS.

7.6.1. The operator + will not be accepted in the DASK ALGOL system.

7.6.2. Accuracy.
The accuracy of a real number will correspond to 31 significant binary digits, i.e. the relative accuracy is approximately 10^{-9}.

7.6.3. Alarms.
If the range of real numbers is exceeded or an undefined operation is attempted self explanatory alarm indications will be typed by the machine as follows:
spild +, spild -, spild ×, spild /, spild ↑, spild /0,
spild -↑ ikke hel, spild 0↑-.
Subsequently error output of the values of all variables will be made on the output punch (cf. section 12).

7.7. BOOLEAN EXPRESSIONS.

The operator ⊃ is not included in DASK ALGOL.

7.8. INTEGERS AS LABELS.

Integers cannot be used with the meaning of labels in DASK ALGOL.

7.9. FOR STATEMENTS.

In DASK ALGOL the controlled variable of a for clause must be a simple variable, not a subscripted variable.

7.10. PROCEDURE STATEMENTS.

7.10.1. Recursive procedures.
Generally speaking DASK ALGOL cannot handle procedures which call themselves recursively. This means that the actual parameters must not refer to the procedure itself, neither directly nor indirectly.
An exception is, however, made in case of procedures, which have only one formal parameter called by value. This class includes the standard functions, abs, arctan, cos, entier, exp, ln, sign, sin, sqrt. Because of this exception it is permissible to write, e.g.,

exp(exp(x)).

7.10.2. Handling of types.
The types integer and real will be handled according to the prescriptions of section 4.7.3 except in the case that a formal parameter, which is specified to be real and to which assignments are made, in the call corresponds to an integer declared variable. This special case will be treated incorrectly in DASK ALGOL.

7.11. ORDER OF DECLARATIONS.

DASK ALGOL requires the declarations in each block head to be written in the same order in which they appear in chapter 5, thus:

first: type declarations
second: array declarations
third: switch declarations
fourth: procedure declarations

For further rules concerning the writing of array declarations when it is desired to store arrays on the drum, see section 10.3.

7.12. Own.

In DASK ALGOL own can only be used with type declarations, not with array declarations.

7.13. PROCEDURE DECLARATIONS.

7.13.1. Recursive procedures.

As already stated (cf. section 7.10.1) recursive procedures cannot be handled. Thus within the procedure body no operation which directly or indirectly may cause a call of the procedure itself may occur.

7.13.2. Arrays called by value.

DASK ALGOL cannot handle arrays called by value.

7.13.3. Specifications.

The specifications for formal parameters must be complete, i.e. each parameter must occur just once in the specification part.

8. STANDARD OUTPUT PROCEDURES.

Output of text and results from a program will be controlled by means of output procedures permanently available to the translator (i.e. without explicit declarations). The output will be provided in the form of 8-channel punch tape or printed copy. The symbols and 8-channel code given in section 6. 8-CHANNEL PUNCH TAPE CODE AND FLEXOWRITER KEYBOARD will be used.

8.1. CONTROL OF TYPEWRITER AND OUTPUT PUNCH.

Half of the standard output procedures are available in two forms, one controlling the output punch (identifier beginning with tryk), the other controlling the on-line typewriter (identifier beginning with skrv). However, the actual output produced by the machine also depends on the position of a 5 - way selector on the control panel of the machine having positions marked 0, S + P, P, S, +. The following table tells whether output will be produced on the typewriter or the punch or both for all combinations of output procedure identifier and position of selector:

	Typewriter		Punch	
Selector	tryk	skrv	tryk	skrv
0	no	yes	yes	no
S + P	yes	yes	yes	no
P	no	no	yes	yes
S	yes	yes	no	no
+	no	no	no	no

8.2. IDENTIFIERS AND MAIN CHARACTERISTICS.

The identifiers and main characteristics of the standard output procedures are the following:

Identifier	Example, reference	Effect
tryk skrv	tryk({+d$_1$ddd}, q$\uparrow$2) section 8.3.	Outputs the values of an arbitrary number of arithmetic expressions in a specified layout. Other output operations may also be inserted as parameters.
tryktekst skrvtekst	skrv({<Q$_1$=$_\perp$}) section 8.4.	Outputs a specified string of symbols.

trykml skrvml	trykml(8-n) section 8.5.	Outputs a specified number of SPACEs.
tryktom	tryktom (100) section 8.5.	Punches a specified number of TAPE FEED symbols.
trykvr skrvvr	skrvvr section 8.6.	Outputs one CAR RET symbol.
tryktab skrvtab	tryktab section 8.6.	Outputs one TAB symbol.
trykstop	trykstop section 8.6.	Punches one STOP CODE symbol.
trykende	trykende section 8.7.	Punches one END CODE symbol.
trykslut	trykslut section 8.7.	Punches one PUNCH ON symbol.
trykklar	trykklar section 8.7.	Punches one CLEAR CODE symbol and sets internal sum of punched symbols to zero.
tryksum	tryksum section 8.7.	Punches a STOP CODE, a SUM CODE and a code representing the sum of the symbols punched since program read-in, last trykklar or last tryksum.
trykkopi skrvkopi	trykkopi ({</;}}) section 9.7.	Copies a section of the input tape to the output, the section being specified through a parameter.

It holds for all standard output procedures that each output operation will cause an addition to an internal variable of a number which is equivalent to the character. This may be used for checking purposes by means of the mechanisms described in sections 8.7.2 and 9.2. It should be noted, however, that for the checking to work correctly the output tape must not include any character which has been produced by a skrv - operation (cf. section 8.1).

8.3. STANDARD PROCEDURES: tryk, skrv.

8.3.1. Syntax.

```
<sign> ::= <empty>| - | + | ±
<exponent layout> ::= ₁₀<sign>d | <exponent layout>d
<zeroes> ::= 0 |<zeroes>0 | <zeroes>␣0
<positions> ::= d |<positions>d | <positions>␣d
<0-positions> ::= <positions> | <0-positions>0 | <0-positions>␣0
<decimal layout> ::= <0-positions>|<0-positions>.<zeroes>|
                     <positions>.<0-positions>|.<0-positions>
<layout tail> ::= <decimal layout>|<decimal layout><exponent layout>
<layout> ::= <sign><layout tail>|<sign>n<layout tail>|
             <sign>n␣<layout tail>
<formal layout>::= <formal parameter>|
     <if clause><formal parameter> else <formal layout>
<layout expression>::= {<layout>}|<formal layout>
<tryk parameter>::= <arithmetic expression>|<tryk statement>|
     <tryktekst statement>|<trykml statement>|<tryktom statement>|
     <trykvr statement>|<tryktab statement>|<trykstop statement>|
     <trykende statement>|<trykslut statement>|<trykklar statement>|
     <trykcum statement>|<trykkopi statement>
<tryk parameter list>::= <tryk parameter>|
     <tryk parameter list>,<tryk parameter>
<tryk statement>::= tryk(<layout expression>,<tryk parameter list>)|
                    skrv(<layout expression>,<tryk parameter list>)
```

8.3.2. Examples.

```
tryk({ddd.00}, P, trykvr, tryktekst ({<Q=}), w +s)
skrv ({-d₁₀-dd},epsilon/16)
tryk({dd␣ddd},Q,trykml(5),tryk({.ddd},q),W,t-3)
tryk(if s>0 then f1 else f2, Sum)
tryk(1, p-q, s+t)
```

8.3.3. Semantics.

A call of the procedure tryk or skrv causes the following treatment of the parameters specified in the tryk parameter list:

Arithmetic expression: the value will be printed in the layout supplied in the first parameter of the call.

Output statement: the call of the statement will be executed.

8.3.4. The layout.

The layout expression will be evaluated once at the beginning of the execution of the tryk or skrv statement. The evaluation will take place in a way which is completely analogous to that of other expressions (cf. section 3.3.3). The final value must always be of the form {<layout>}.

The symbols of the layout give a symbolic representation of the digits, spaces and symbols as they will appear in the printed number. Indeed, the finally printed number will have exactly the same number of printed characters as is present in the layout (except in case of alarm printing, see section 8.3.6). The various symbols of the layout have the following significance:

8.3.4.1. Sign. The four possible symbols in the sign position signify the following:
8.3.4.1.1. Empty. The number is supposed to be positive. No sign will be printed. If a negative number is encountered, an alarm printing will take place (see section 8.3.6).
8.3.4.1.2. - . The sign will always be printed using SPACE for positive, and - for negative numbers. It will, if possible, move to the right, appearing as the first or second symbol to the left of the first digit (a layout SPACE may appear in between) or immediately in front of the decimal point.
8.3.4.1.3. + . The sign will always be printed using + for positive and - for negative numbers. It will, if possible, move to the right, as in 8.3.4.1.2 above.
8.3.4.1.4. $\pm$. The sign will always be printed, using + for positive and - for negative numbers. It will be printed as the first symbol of the number, before any SPACE or digit.

8.3.4.2. Digits. Letters d and n represent digits. Letter n may only appear as the first symbol following the sign. The total number of letters d and n gives the maximum number of printed significant digits (cf. section 8.3.8).

If n is used in the first digit position, proper decimal fractions will be printed with a 0 in front of the decimal point. If d is used this 0 will be omitted.

8.3.4.3. Zeroes. Zeroes may appear at the end of a decimal layout. They influence the representation of the number in the following manner: If m zeroes are present at the end of the decimal layout the exponent printed will be exactly divisible by m+1. For this to be possible at the same time as the position of the decimal point within the complete layout is kept fixed the significant digits of the number are allowed to move to the right, using the positions of the symbols 0, depending on the magnitude of the number. If no exponent layout is included the exponent 0 is understood and the above rule holds unchanged.

8.3.4.4. Spaces. Spaces will be inserted in all positions where the symbol $\perp$ appears. The symbol $\perp$ may within the layout be replaced by SPACE the effect of SPACE being the same.

8.3.4.5. Decimal point. The decimal point will always be printed in a fixed position within the layout. If decimals are printed it will appear as . otherwise as SPACE.

8.3.4.6. Scale factor. The scale factor will be printed in the same way as in the language. The symbol $_{10}$ will appear immediately in front of the sign of the exponent. If the scale factor is 1 the symbols $_{10}$ and following will appear as SPACEs. Note that it is not possible to print an exponent part without a decimal part.

8.3.5. Round-off.

All numbers will be correctly rounded to the number of significant digits printed.

8.3.6. Limitations.

The total number of symbols n and d in any decimal layout must be ≤ 15.

The total number of symbols n, d, and 0, written to the left of the decimal point must be ≤ 15.

The total number of symbols d and 0 written to the right of the decimal point in a decimal layout must be ≤ 15.

The number of symbols d in any exponent layout must be ≤ 7.

The total number of the symbols $_{\perp}$ and SPACE and . in any layout must be ≤ 7.

8.3.7. Alarm printing.

By alarm printing is meant that the printing will consume more positions on the paper than are present in the layout. Alarm printing will occur as follows:

8.3.7.1. Negative number printed with layout having empty sign position. The correct - will be inserted, consuming one extra position.

8.3.7.2. Number too large for layout. Whenever the number to be printed is too large for the layout given, an actual layout is used which will accomodate the number by inserting an exponent layout, or by increasing the number of exponent digits.

8.3.8. Small numbers.

Printing of small numbers will never give rise to alarm printing. Instead the number of printed significant digits will be smaller than the maximum (section 8.3.4.2).

8.3.9. Examples of printed numbers.

In order to indicate the exact number of characters printed, commas are inserted immediately preceding and following each number.

Layout $n_{\perp}dd_{\perp}dd.d0_{\perp}0$	$+d_{\perp}ddd.ddd_{\perp}d$	$-ddd.d00_{10}+d$	$\pm dd.0_{10}-dd$
Normal printing			
0.00 1,	+.001 2,	$1.235_{10}-3$,	+12 $_{10}-4$,
0.01 2,	+.012 3,	12.35 $_{10}-3$,	+ 1.2 $_{10}-2$,
0.12 3,	+.123 5,	123.5 $_{10}-3$,	+12 $_{10}-2$,
1.23 5,	+1.234 6,	1.235 ,	+ 1.2 ,
12.34 6,	+12.345 7,	12.35 ,	+12 ,
1 23.45 7,	+ 123.456 8,	123.5 ,	+ 1.2 $_{10}$ 2,
12 34.57 ,	+1 234.567 9,	$1.235_{10}+3$,	+12 $_{10}$ 2,
1 23 45.7 ,		12.35 $_{10}+3$,	+ 1.2 $_{10}$ 4,
	-.001 2,	$-1.235_{10}-3$,	-12 $_{10}-4$,
	-1 234.567 9,	$-1.235_{10}+3$,	-12 $_{10}$ 2,
Alarm printing			
-0.00 1,			
-1 23 45.7 $_{10}$3,	-1 234.567 9_{10}4,		
1 23.45 7_{10}15,	+1 234.567 9_{10}14,	123.5 $_{10}+15$,	

8.4. STANDARD PROCEDURES: tryktekst, skrvtekst.

8.4.1. Syntax.

```
<tryktekst parameter>::= {<<proper string>}|<formal parameter>
<tryktekst parameter list>::= <tryktekst parameter>|
        <tryktekst parameter list>,<tryktekst parameter>
<tryktekst statement>::= tryktekst(<tryktekst parameter list>)|
                         skrvtekst(<skrvtekst parameter list>)
```

8.4.2. Examples.

```
tryktekst ({<Result is}, a, {<than expected})
skrvtekst({<Q⊥=⊥})
```

8.4.3. Semantics.

The execution of a tryktekst statement causes the strings of characters referred to in the parameters to be outputed, taking the parameters in order from left to right. The characters outputed are given directly in the form of a proper string if the parameter has the form

{< <proper string> }

Otherwise the formal parameter must refer to an actual parameter having this form, and the formal prameter must call by name (cf. section 4.7.3.2).

8.4.3.1. The string quote.

Note the difference between the string quotes used here

{< }

and those used in layout expressions (cf. section 8.3.1).

8.4.3.2. Treatement of SPACE and CAR RET.

All characters of the proper string, including SPACEs and CAR RETs will be outputed. The symbol for space ⊥ will however be equivalent to SPACE, i.e. it will be printed, not as it stands, but as a SPACE.

8.5. STANDARD PROCEDURES: trykml, skrvml, tryktom.

8.5.1. Syntax.

```
<trykml statement>::= trykml (<arithmetic expression>)|
                      skrvml (<arithmetic expression>)
<tryktom statement>::= tryktom (<arithmetic expression>)
```

8.5.2. Examples.

```
trykml(n + m - 7)
tryktom (75)
skrvml (if p > 0 then 3 else 4)
```

8.5.3. Semantics.

The execution of a trykml statement causes the number of SPACE symbols (mellemrum) specified as actual parameter to be outputed.

A call of the procedure tryktom causes the number of TAPE FEED symbols specified as actual parameter to be outputed.

The value of the arithmetic expression will, if necessary, be rounded to the nearest integer. If it assumes a non - positive value no symbols will be outputed.

8.6. STANDARD PROCEDURES: trykvr, skrvvr, tryktab, skrvtab, trykstop.

8.6.1. Syntax.

<trykvr statement>::= trykvr|skrvvr
<tryktab statement>::= tryktab|skrvtab
<trykstop statement>::= trykstop

8.6.2. Semantics.

A trykvr statement causes a CAR RET symbol (vogn retur) to be outputed. Note that this will cause the combined operation of return of carriage and line feed to take place.

A tryktab statement causes output of a TAB symbol.

A trykstop statement causes the STOP CODE to be punched.

8.7. STANDARD PROCEDURES: trykende, trykslut, trykklar, tryksum.

8.7.1. Syntax.

<trykende statement>::= trykende
<trykslut statement>::= trykslut
<trykklar statement>::= trykklar
<tryksum statement>::= tryksum

8.7.2. Semantics.

The four output procedures described here all serve to insert characters on the output tape with a view to a later use of this output tape as input tape to an ALGOL program.

The trykende statement punches the END CODE. When later the tape is read into the machine this will cause a stop of the machine (cf. section 9.2.6).

The trykslut statement punches the PUNCH ON symbol. This is intended to be used as a non - printing terminator for læs and læst (cf. sections 9.4 and 9.5).

The trykklar statement punches the CLEAR CODE and sets the internal sum of the punched characters to zero. This prepares for the use of the checksum mechanism (cf. section 9.2.5).

The tryksum statement punches a STOP CODE, a SUM CODE and a character representing the value of the internal sum of all punched characters and sets this sum to zero. During input this combination will cause an automatic sum check to take place (cf. section 9.2.5).

9. STANDARD INPUT PROCEDURES.

Input of information from 8-channel punch tape may be carried out at any stage of an ALGOL program through calls of standard input procedures permanently available to the translator.

In order to provide flexibility several different kinds of standard input procedures are available. These differ both with respect to the interpretation of the single symbols supplied on the input tape and the internal effect of the input operation.

9.1. IDENTIFIERS AND MAIN CHARACTERISTICS.

The identifiers and main characteristics of the standard input procedures and the associated procedure streng are the following:

Identifier	Example, reference	Effect
læs	læs(a, b, c) section 9.4.	Reads numbers and assigns to variables or arrays.
læst	p × læst section 9.5.	real procedure læst has the next number on the input tape as its value.
læsstreng	læsstreng section 9.6.	Reads a string of symbols to an internal variable for later comparison by means of the
streng	streng({<P}) section 9.6.	boolean procedure streng. The value of streng is true if the string supplied as parameter agrees with the string read by the last call of læsstreng.
trykkopi skrvkopi	trykkopi({</;}) section 9.7.	Cause a copying of the characters on the input tape to be output punch (trykkopi) or the typewriter (skrvkopi).

9.2. UNIVERSAL INPUT MECHANISMS.

Certain characters on the input tape will be handled in the same way no matter which of the standard input procedures is controlling the input operation. The universal mechanisms are the following:

9.2.1. Skipping between PUNCH OFF and PUNCH ON.

All characters between PUNCH OFF and the first following PUNCH ON, these two characters included, will be completely ignored during input.

9.2.2. Ignoring of BLANK TAPE, TAPE FEED, and ALL HOLES.

The characters

.	BLANK TAPE
oooo.ooo	TAPE FEED
ooooo.ooo	ALL HOLES

will be ignored during input.

9.2.3. Standard error reaction.

Various kinds of errors may be detected during input (cf. sections 9.2.4, 9.4.3.6, 9.5.3). In each of these cases an error type indication will immediately be typed on the output typewriter and then the machine will execute the following standard error reaction: The following characters on the input tape will be copied to the output punch. When two lines have been copied the machine control is returned to the translator system (cf. section 12).

9.2.4. Error combinations.

Outside the sections of the tape between PUNCH OFF and PUNCH ON (cf. section 9.2.1) the reading of a hole combination with wrong parity, or of any NOT USED code (including 63 symbols with decimal values from 65 to 127, not listed in section 6.5) will cause typing of

læs fejl

and the machine will do the standard error reaction (cf. section 9.2.3).

9.2.5. The checksum mechanism.

When the standard input procedures read tapes which have been prepared by the standard output procedures the checksums included on this tape in consequence of calls of the tryksum procedure will automatically be verified. If the check symbol does not check with the corresponding symbol as formed during previous read-in the machine will print

læs fejl sum

and the machine will stop. If the START key is pressed the reading will continue. The internal variable which holds the current sum of the symbols which have been read in may be reset to zero by the inclusion of the CLEAR CODE on the tape. This is the symbol produced by the trykklar procedure (cf. section 8.7.2). On the flexowriter use:

AUX CODE with 0

9.2.6. Stop produced by END CODE.

The reading will stop whenever the END CODE symbol appears. If the START key is pressed the reading will continue. The END CODE may be produced by an ALGOL program by a call of the trykende procedure (cf. section 8.7.2). On the flexowriter it is produced by depressing

AUX CODE with SPACE.

9.2.7. The effect of UPPER CASE and LOWER CASE.

For printed symbols (cf. section 6.1) the meaning and effect of a given hole combination depends on the most recent CASE symbol on the tape (UPPER CASE or LOWER CASE).

For typographical and control symbols (cf. sections 6.2 and 6.3) the effect is usually independent of the case.

9.3. TERMINATORS, INFORMATION SYMBOLS, AND BLIND SYMBOLS.

The effect of the input characters which do not give rise to an action of a universal input mechanism (cf. section 9.2) depends on the particular standard input procedure. In describing this effect it is convenient to make use of the following concepts:

9.3.1. Terminators. A terminator is a symbol on the input tape which indicates to the input procedure that the reading of a piece of information (e.g. a number) has been completed.

9.3.2. Information symbols. An information symbol is a symbol on the input tape supplying positive information which is transferred to the running ALGOL program by the input procedure.

9.3.3. Blind symbols. A blind symbol is a symbol on the input tape which is ignored by the input procedure.

As explained more concisely in the following sections we have for the procedures læs and læst:

Terminators: <letter> all signs except $+-.{}_{10}$ TAB PUNCH ON CAR RET
Information symbols: <digit> $+-.{}_{10}$
Blind symbols: SPACE _ STOP CODE
and for læsstreng:
Terminators: all signs TAB PUNCH ON CAR RET
Information symbols: <digit> <letter>
Blind symbol: SPACE _ STOP CODE

Each input operation will in general read three sections of the input tape:

1. Any mixture of terminators and blind symbols.
2. A legal sequence of information symbols mixed with blind symbols.
3. One terminator.

9.4. STANDARD PROCEDURE: læs.

9.4.1. Syntax.

```
<læs parameter>::= <variable>|<array identifier>
<læs parameter list>::= <læs parameter>|
                        <læs parameter list>,<læs parameter>
<læs statement>::= læs(<læs parameter list>)
```

9.4.2. Examples.

```
læs(P)
læs(A[i,j], V, MATA)
læs(k, B[1,k])
```

9.4.3. Semantics.

A call of the procedure læs will cause the values of numbers supplied on the input tape to be assigned to the variables and/or arrays of subscripted variables specified as parameters. The assignments will in detail be executed as follows:

9.4.3.1. Order of assignment. The parameters will be taken in order from left to right and the assignment will be completely finished for each parameter before the next is treated. Thus the statement læs(k, B[1,k]) will first assign a value from the input tape to k and this value of k will then define the particular component of B to which the next number on the tape will be assigned.

9.4.3.2. Assignment to array. If an array identifier is supplied as parameter an assignment to all the components of the array will take place. The order of assignment may be described as follows: Denoting the lower and upper subscript bounds of the array declaration by l1, l2, ... ln, u1 u2, ... un, the input operation is equivalent to

```
for i1:= l1 step 1 until u1 do
for i2:= l2 step 1 until u2 do
      .......

for in:= ln step 1 until un do
  A[i1, i2, ... , in]:= input number
```

where i1, i2, ... in are internal variables.

9.4.3.3. Input tape syntax. The characters appearing on the input tape during the execution of læs must conform to the following syntactic rules:

```
<læs terminator>::= ∨|×|/|=|;|[|]|(|)|||∧|<|>|,|TAB|PUNCH ON|:|CAR RET|
                    <letter>
<læs information>::= <digit>|.|10|+|-
<læs blind>::= SPACE|_|STOP CODE
<læs prelude>::= <empty>|<læs blind>|<læs terminator>|
      <læs prelude><læs blind>|<læs prelude><læs terminator>
<digit sequence>::= <digit>|<digit sequence><digit>|
      <digit sequence><læs blind>|<læs blind><digit sequence>
<input integer>::= <digit sequence>|+<digit sequence>|-<digit sequence>
<input fraction>::= .<digit sequence>
<input exponent>::= 10<input integer>
<input decimal>::= <digit sequence>|<input fraction>|
                   <digit sequence><input fraction>
<unsigned real>::= <input decimal>|<input exponent>|
                   <input decimal><input exponent>
<input real>::= <unsigned real>|+<unsigned real>|-<unsigned real>
<input ditto>::= -|<input ditto>-|<input ditto><læs blind>
<tape integer>::= <læs prelude><input integer><læs terminator>|
                  <læs prelude><input ditto><læs terminator>
<tape real>::= <læs prelude><input real><læs terminator>|
               <læs prelude><input ditto><læs terminator>
```

9.4.3.4. Examples of input tape for læs.

Tape integers:

```
17 283;
i = +138,
S[25]
funktion(-12)
p: -/
```

Tape reals:

```
w:= 3.857_392 <
eps:= -10-14,
pi:= 3.141592 65;
Sæt x = 4,
q: 1.38410-11,
```

9.4.3.5. Semantics of input tape. Depending on the type of the variable each læs assignment will cause the reading of one tape real or tape integer. If these contain digits they will be interpreted according to the usual ALGOL prescriptions (cf. sections 2.5.3 and 2.5.4), ignoring all læs blinds and læs terminators. An input ditto, on the other hand, will cause the læs assignment to be skipped for the particular variable, thus leaving its value unchanged.

9.4.3.6. Errors. The standard procedure læs checks that the syntactic rules of section 9.4.3.3 are satisfied. In particular, while assigning to a variable of type integer the symbols . and $_{10}$ must not occur. Errors of this kind will cause typing of the error indication

læs fejl

and the standard error reaction will take place (cf. section 9.2.3). In addition a protest is made against numbers whose absolute value is greater than $3.4_{10}38$. In this case the error indication

læs spild

is typed before the standard error reaction is made.

9.5. STANDARD PROCEDURE: læst.

9.5.1. Syntax.
<læs function designator>::= læst

9.5.2. Examples.
w:= (læst + y)/q
B[læst, læst]:= læst

9.5.3. Semantics.
læst is a real procedure having an empty formal parameter part. Every time it is called it will read the next tape real appearing on the input tape (cf. section 9.4.3.3). This information on the input tape will define its value according to the rules of section 9.4.3.5, except that læst will treat an input ditto as a syntactic error (cf. section 9.4.3.6).

9.5.3.1. Example of input tape for læst. A reasonable input tape for the second example of section 9.5.2 would be the following:
B[3,7]:= 3.847,
Note that the correct execution of this input operation is directly dependent on the strict adherence to the rules of sections 4.2.3.1 - 4.2.3.3 for assignment statements.

9.6. STANDARD PROCEDURES: læsstreng, streng.

9.6.1. Syntax.

<læs streng statement>::= læsstreng
<formal string>::= <formal parameter>|
    <if clause><formal parameter> else <formal string>
<string expression>::= {<<proper string>}|<formal string>
<streng function designator>::= streng(<string expression>)

9.6.2. Examples.

læs streng
if streng({<A}) then go to T

9.6.3. Semantics.

The standard procedures læsstreng and streng serve to read identifying information from the input tape and to compare this information with information supplied by the program. The detailed operation is defined below.

9.6.3.1. Input tape syntax. During execution of læsstreng the characters on the input tape are treated according to the following syntax:

<læsstreng terminator>::= ∨|×|/|=|;|[|]|(|)|||∧|<|>|,|$_{10}$|TAB|-|+|PUNCH ON|.|:|CAR RET
<læsstreng information>::= <digit>|<letter>
<læsstreng blind>::= SPACE|_|STOP CODE
<læsstreng prelude>::= <empty>|<læs streng blind>|
    <læsstreng terminator>|<læsstreng prelude><læsstreng blind>|
    <læsstreng prelude><læsstreng terminator>
<input string>::= <læsstreng information>|<input string><læsstreng blind>|
    <input string><læsstreng information>
<tape string>::= <læsstreng prelude><input string><læsstreng terminator>

9.6.3.2. The internal string. Each call of læsstreng will read the first following tape string from the input tape and assign the five first information symbols of the input string, which is a part of it, to a unique internal variable. If the input string has less than five information symbols it will be extended with the appropriate number of unique dummy characters.

9.6.3.3. F amples of tape strings and internal strings.

Symbols on tape	Internal string
b7.	b7
(Matrix A)	Matri
[×]:A and B;	AandB
true,	true

9.6.3.4. Standard procedure streng. This is a boolean procedure, requiring a string expression as parameter. It has the value true if all the characters of the value of the string expression agree with the same number of characters of the internal string, assigned by the previous læsstreng, both strings taken in order from left to right, otherwise the value false. Note that the agreement of the two strings puts the following restrictions on the string supplied as parameter to streng:
9.6.3.4.1. It cannot contain more characters than the number of information symbols in the internal string (never more than 5).
9.6.3.4.2. It can only contain digits and letters.

9.6.3.5. Example. The following table shows the value of streng for various input strings and parameters:

	Parameter:		
Input string	A	Alg	ALGOL
ALGOL 60	true	false	true
A	true	false	false
Blg	false	false	false
Algol	true	true	false
Algorithm	true	true	false

9.7. STANDARD PROCEDURES: trykkopi, skrvkopi.

9.7.1. Syntax.

<trykkopi statement>::= trykkopi(<string expression>)|
skrvkopi(<string expression>)

9.7.2. Examples.

```
trykkopi(‹<+/›)
skrvkopi(if s>0 then w else y)
trykkopi(fs)
```

9.7.3. Semantics.

A call of a trykkopi statement causes a copying of characters from the input tape to the output. The section of the input tape to be copied is defined by the value of the string expression supplied as parameter. This value must have the form

‹< <proper string> ›

where the proper string consists of one or two characters. If one character is supplied the copying will take place from the actual position of the input tape until the first occurrence of the character specified as parameter. If two characters are supplied the copying will start from the first character on the tape which is the same as the first of the two characters supplied as parameters and will continue until the first occurrence of the second of these symbols on the tape. The characters indicating the begin and end of the section of the input tape to be copied will not themselves be copied.

The copying will include all legal characters except those associated with the universal input mechanisms (cf. section 9.2) and superfluous case shifts.

9.7.3.1. Example of call, input tape, and output.

The call

```
trykkopi(‹<[]›)
```

operating on the following input tape:

```
Heading: [
Problem number:]
```

will produce as output:

```
Problem number:
```

10. STORING BLOCKS AND VARIABLES ON THE MAGNETIC DRUM.

10.1. INTRODUCTION.

ALGOL programs involving only a few hundred arithmetic operations and/or variables may be handled direcly by the DASK ALGOL translator and system. However, if problems exceeding a certain size are presented, the translator or the system will refuse to accept the problem (cf. sections 10.4.3 and 12). What has happened is that the capacity of the directly available internal store of the machine, the so-called core store, has been exceeded.

This does not mean that the larger problems cannot be handled, since there is available in the machine a storage capacity on the so-called magnetic drums of 8 times that of the core store. What it does mean, however, is firstly that the user must supply extra information to the ALGOL translator system telling the system to place certain blocks or variables on the drum and secondly that time will be spent transferring information between the drum and the cores making the program less efficient. Now the loss of efficiency of an ALGOL program which uses the drum may depend very greatly on the manner in which the different parts of it are distributed between the core and the drum stores. Since this distribution is performed on the basis of information given by the user, it means that in order to make efficient use of the facilities for storing blocks and variables the user must know something about how an ALGOL program will be stored and how the machine will handle it.

Preoccupation with this kind of consideration admittedly lies very far from the spirit of the universal language ALGOL, and it is clear that the ideal ALGOL translator system would handle the storage problem automatically. However, it is believed that the extra burden placed on the user, who must use the DASK ALGOL drum mechanisms will prove in practise to be rather modest.

10.2. MACHINE CHARACTERISTCS AND SPACE REQUIREMENTS.

With a view to the descriptions of the following sections the present section will give some information of the characteristics of DASK and of the length of the translated program.

10.2.1. The core store.

For the machine to be able to execute an ALGOL statement directly all the individual instructions and all the variables must be present in the core store. The capacity of the core store is divided into 2048 half cells, of which 64 are permanently reserved by the ALGOL system. The remaining 1984 half cells may be used for instructions or variables.

10.2.2. The drum store.

The drum store forms a reservoir of information for the core store. The total capacity is 8 times that of the core store, i. e. 16384 half cells. While the instructions and the variables of the core store are directly accessible, the information in the drum store must be transferred to the core store before it can be used. This transfer can only be done in lumps of 64 half cells at a time, so-called tracks, and is a comparatively slow process since the machine may perform about 12 arithmetic operations on real numbers in the time taken to transfer one track.

10.2.3. Storage requirements of ALGOL programs.

The exact storage requirements of an ALGOL program are given in detail in section 11. For a rough estimate the following rules may be used:

Declared variables, whether simple or subscripted, occupy one half cell each if they are of type integer or boolean, and two half cells each if of type real. Each formal parameter of a procedure occupies two half cells.

To make an estimate of the storage occupied by the instructions make a count of the symbols of the program (omitting type declarations and procedure headings) as follows:

Each occurrence of a number or an identifier counts as one.

Each occurrence of a delimiter, except for comma (,), semicolon (;) and parentheses () counts as one.

To obtain the number of half cells occupied by instructions multiply the count by a factor of from 1.2 to 1.5.

The total storage requirement will be that of the variables plus that of the instructions.

10.3. STORING VARIABLES ON DRUM.

10.3.1. Syntax.

<drum array declaration>::=
comment drum data [<bound pair list>];<array declaration>|
<drum array declaration>;<array declaration>

10.3.2. Examples.

```
comment drum data [1:p, 7:s-1];
array P, Q[1:p, 7:s-1, 3:8, v:w];
integer array I, K[1:p, 7:s-1, 2:7]

comment drum data [2:7];
boolean array Boo 1, Boo 2[2:7, 1:n, 1:q]
```

10.3.3. Semantics.

A drum array declaration declares one or several identifiers to represent arrays, in exactly the manner explained in section 5.2, but, in addition, specifies one or more of the subscripts of these arrays to refer to the drum. This means the following:

10.3.3.1. The arrays declared in a drum array declaration will in the normal way define the meaning of corresponding subscripted variables. Thus the fact that the arrays are stored on drum is visible only in the declaration.

10.3.3.2. The number of subscripts referring to the drum and the bounds for these subscripts are given in the initial drum data comment. All following array declarations must have identicaly the same bound pairs in their first subscript positions. The bound pairs in any remaining subscript positions will refer to the core store.

10.3.3.3. All arrays declared in one drum array declaration are referred to as a drum array group. The array declarations belonging to one drum array groups are all those which follow the initial drum data comment until the first following drum data comment, switch declaration, procedure declaration or statement. Thus the rules for the writing of declarations (cf. section 7.11) must be extended to read:

Array declarations must be written in the following order: First all declarations for not-drum arrays (in any order), then drum array declarations.

10.3.3.4. As far as the storing is concerned a drum array group is treated as one large (generally not rectangular) array. The complete set of components of this array is stored on the drum while in the core store only a sub array corresponding to one set of values of the subscripts referring to the drum is stored. Since this sub array occupies only a fraction of the total storage of the drum arrays a considerable saving of core store may be achieved.

10.3.4. Illustration.

The scheme for storing drum arrays may be further explained by a simple example: let the declaration read

```
comment drum data[1:3];
integer array I, K[1:3, 4:5];
array R[1:3, 7:8, 9:10]
```

The components of these arrays will be stored on the drum in the following order:

```
I[1,4], I[1,5], K[1,4], K[1,5], R[1,7,9], R[1,7,10], R[1,8,9], R[1,8,10]
I[2,4], I[2,5], K[2,4], K[2,5], R[2,7,9], R[2,7,10], R[2,8,9], R[2,8,10]
I[3,4], I[3,5], K[3,4], K[3,5], R[3,7,9], R[3,7,10], R[3,8,9], R[3,8,10]
```

In the core store only one of these sections will be available at any one time, for instance the one having the first subscript equal to 3:

```
I[3,4], I[3,5], K[3,4], K[3,5], R[3,7,9], R[3,7,10], R[3,8,9], R[3,8,10]
```

As already stated (section 10.3.3.1 above) the components of drum arrays will be used exactly like any other subscripted variables. However, it is clear that since any reference to a component having its drum subscripts different from those last referred to will cause several transfers of tracks to and from the drum, the process may become very time consuming. Thus the following statement:

```
R[2,7,10]:= I[1,5] + K[3,4]
```

will cause at least four, and perhaps six, track transfers to take place. The rule for the ALGOL programmer to follow in order to avoid this is:

Frequent references to altered values of the drum subscripts of arrays within one drum array group should be avoided.

10.4. STORING BLOCKS ON DRUM.

10.4.1. The drum program comment.

Any block, whether a statement in the program or the body of a procedure declaration, may be specified to be stored on the drum. This is achieved by adding the symbols:

comment drum program;

immediately following the begin of the block. In order to enable the ALGOL programmer to exploit this facility some information of the manner in which an ALGOL program is stored in the core store will be given below.

10.4.2. Storage of instructions and variables.

At any one stage of the run of a program one end of the core store, the low end, will be occupied by instructions, constants, simple variables, and formal variables while the high end of the store, the so-called stack will contain the components of all those arrays, which are defined at this stage, and any intermediate results of expressions in the process of beeng evaluated.

10.4.2.1. Low end storage. Within the low end the order of storing of the various blocks is as follows:

First segment: all such parts of the program which do not belong to any drum block, including constants, but not including the storage for simple variables of ordinary blocks of the program.

Second segment: instructions and simple variables of all drum procedure bodies, which will all share a certain section of the core store. Thus this segment may at any one time store only one drum procedure body.

Third segment: the simple variables of ordinary blocks of the program, sharing space with the instructions and simple variables of all drum blocks of the program. This segment will thus at any one time contain either the simple variables of an ordinary block or one complete drum block.

If drum blocks are written within drum blocks the outer drum block will be stored in three segments in exactly the same manner as the complete program, forming, so to speak, a microcosmos of its own.

Note particularly that since all drum procedures which are declared in the same block head share the same place in the core store NO TWO DRUM PROCEDURES WHICH ARE DECLARED IN THE SAME BLOCK HEAD MUST EVER CALL EACH OTHER, NEITHER DIRECTLY OR INDIRECTLY.

10.4.2.2. Stack storage. The storing in the stack (the high end of the core store) is arranged strictly according to the dynamic order in which the various blocks have been entered. Thus each time an entry into a block is made the components of arrays declared in the block head will be placed in order downwards from the last reserved cell of the stack. Again, every time an exit from a block is made the part of the stack used by this block is released and may thus be used by any other block.

10.4.3. Dynamic storage control.

Since arrays of arbitrary size may be declared in any block head it is necessary to keep a continuous check on the extent of the parts of the store used by the low and high end sections, if disastrous overlaps between instructions and variables are to be avoided. This is performed as follows: When a block is entered the amount of low end storage which must be reserved while this block is working is calculated and compared with the current limit of the stack. Again, every time a new item is added to the stack a similar control is made. If the control detects that the two parts of the store are about to overlap the error indication

ferritlager sprængt

will be typed and the machine will immediately proceed to output information about the block causing the error.

Similarly, since any declaration of a drum array may use an arbitrary amount of the drum storage, a check on the occupation of the drum will be made every time a drum array is declared. An overlap will cause typing of the error indication

tromlelager sprængt

to be followed by output as above.

10.4.4. Preservation of ordinary drum blocks.

The first time a drum block is entered it will automatically be transferred to the core store. At the same time the administrative system makes a note that this particular block is now available.

When a new entry is made into the same block the system will omit the transfer from drum if it is certain that the part of the store occupied by the block has not been disturbed since the last exit. The conditions for this are 1) that no entry into (or exits from) any ordinary block has been made, and 2) that the stack has not made use of the relevant part of the store in the meantime. Note, however, that calls of procedures may well be executed, without disturbing an available drum block.

10.4.5. Preservation of procedure drum blocks.

The system for avoiding unneccessary transfers from the drum store in case of procedure drum blocks is similar to that of ordinary drum blocks, but rather less economical. The rules are: That procedure drum block into which an entry has last been made will be available without a repeated transfer provided 1) the stack has not in the meantime made use of the relevant part of the store and 2) no exit from the block, from which the drum procedure was first called, has been made.

For more details of the storage allocation of DASK ALGOL see the article: J. Jensen, P. Mondrup, P. Naur: A Storage Allocation Scheme for ALGOL 60, BIT vol. 1 no. 2 (1961).

11. MACHINE REPRESENTATION OF PROGRAMS AND PARAMETERS.

In the present chapter the representation of all the possible constituents of an ALGOL program within the machine will be given. This information is given mainly for the benefit of those programmers who wish to mix ALGOL language with machine language and those who wish to make use of the programs of the wired store (cf. section 11.2).

11.1. NOTATION.

The notation of the present chapter necessarily deviates strongly from that of all the previous chapters. Indeed we are concerned exclusively with specifications of machine addresses and the contents of storage locations.

11.1.1. Machine addresses.

Machine addresses will be written either
a) as absolute addresses (integers in the range from 0 - 2047)
b) ALGOL identifiers. Addresses associated with definite classes of ALGOL quantities will be distinguished through the first letter(s) of the identifier used, thus:

First letter(s)	Associated ALGOL quantity
ni	Integer
nr	Number
lv	Logical value
la	Label
layout	Layout
st	String
r	real variable -
i	integer variable |Simple or formal by value
b	boolean variable -
a	Array
p	Procedure
s	Switch
f	Formal parameter, called by name
fi	Formal integer parameter, called by name
fr	Formal real parameter, called by name

Symbols for machine addresses will be written in front of the corresponding contents followed by colon, like ALGOL labels.

11.1.2. Contents of storage locations.

For convenience both an adapted NL 1 representation and the NL 4 representation will be given. The definition of NL 1 will be found in Lærebog i Kodning for DASK, Regnecentralen 1958. The official description of NL 4 is not yet published.

11.2. THE WIRED STORE.

All the standard administrative and arithmetic functions needed when an ALGOL program is running have been permanently wired into the machine in the form of a wired store. This has the same capacity as the core store and relieves the core store of all standard mechanisms except for the first 64 half cells which contain working locations for the wired store and some universal parameters of the running ALGOL program (cf. section 11.8). Entry into the wired store is made through 17-jumps, which are otherwise similar to 16-jumps.

11.3. USE OF MACHINE REGISTERS AND WORKING LOCATIONS.

All the working registers of the machine are used by the standard routines of the wired store, and their contents must therefore always be expected to be changed by any jumps into the wired store.

Input parameters to the wired store are always supplied in the index registers IRB and IRD, and occasionally in IRC. By far the most important parameter is the contents of IRB which defines the last used full word in the stack (cf. section 10.4.2.2.). Since the stack is used as working space by all wired routines the user must always put such an address into IRB that the locations 2046B, 2044B, and 2042B are at the disposal of the wired routine.

11.4. NUMBERS, LOGICAL VALUES, LABELS, AND STRINGS.

11.4.1. Real number in stack.

The given number y is rewritten as

$$y = y1 \times 2\uparrow y2$$

where y2 is an integer and y1 is a fraction rounded to have 31 binary places with $-1 \leq y1 \leq -2\uparrow(-19)$
or $y1 = 0$
or $2\uparrow(-19) \leq y1 \leq 1 - 2\uparrow(-19)$.
Also $-127 \leq y2 \leq 128$
If $y1 = 0$ then $y2 = 19$
The number is packed into a full word in the stack as follows:

$$y \text{ in stack} = y1 + (128 - y2) \times 2\uparrow(-39).$$

The greatest number $= (1 - 2\uparrow(-19)) \times 2\uparrow 128 = 3.4028 \times 10\uparrow 38$
The smallest normalized number $= 2\uparrow(-128) = 2.9387 \times 10\uparrow(-39)$

11.4.2. Real number in store.

The representation of y is

$$y \text{ in store} = y \text{ in stack} - 109 \times 2\uparrow(-39)$$

Thus if $y = 0$ the number zero is stored.

Note that if $y2 \geq 20$ a carry from the exponent part will change the stored modulus.

11.4.3. Integer in stack.

The integer z is always represented as

$$z = z1 \times 2\uparrow 19.$$

It is stacked exactly like a real number, i.e.

z in stack = z1 + 109 × 2↑(-39) = z × 2↑(-19) + 109 × 2↑(-39)

The greatest positive integer = 2↑19 - 1 = 524287
The greatest negative integer = -2↑19 = -524288

11.4.4. Integer in store.

The integer z is stored as

z × 2↑(-19)

in a half cell. This representation is analogous to that of a real.

11.4.5. Logical values.

true is represented as any negative integer. false is represented as any non-negative integer.

11.4.6. Labels.

The label pointing to the machine instruction la in block number bn in the program is stored in a half cell as

la × 2↑(-11) + bn × 2↑(-19)

11.4.7. Layouts (cf. section 8.3.4.).

A layout is stored in a full cell as ten sedecimal characters:

B bhdfs s bhdfs
B pqrst s pqrst

b = number of letters n and d in number part.
h = number of characters n, d, and 0 before decimal point in number part.
d = number of characters d and 0 following the decimal point in number part.
f = 4 × fn + fe, where fn and fe represent the signs of the number and the exponent according to the following table:
<empty> = 0, - = 1, + = 2, ± = 3.
s = number of letters d in exponent + 8 × number of letter n.
p,q,r,s,t are the numbers of characters n, d, 0 in the consecutive digit groups, as separated by SPACE and decimal point.

Examples:

Layout	Sedecimal characters
$n_1dd_1dd.d0_10$	65308 12221
$+d_1ddd.ddd_1d$	84480 13310
$-ddd.d00_{10}+d$	43361 33000
$\pm dd.0_{10}-dd$	221D2 21000

11.4.8. Strings.

Each character (following the left string bracket ‹<), together with its case, is represented as an integer:

1 + numerical representation according to section 6.5 +
(if LOWER then 128 else 0)

Each integer is represented as 8 binary digits, or equivalently 2 sedecimal digits.

A string is represented as the string of binary digits followed by the end mark, eight zeros. It is stored in consecutive full cells, the first of which carries the address of the string, st.

Example:	NL 1 representation
ALGOL 60.	B 32243 B 82724 B 01879 B 1BC00

11.5. EXPRESSIONS.

11.5.1. General conventions.

During the evaluation of expressions the ALGOL system will make use of the stack for storing any intermediate values and counting in IRB will take place. However, when the evaluation is completed IRB will have returned to its original value and the result will be present in the AR register. The form of the value in AR is that given in section 10.4, where for numbers the store form (section 11.4.2) is used.

These same rules hold for the evaluation of the more complicated primaries, subscripted variables, function designators, relations, and switch designators which also make use of the stack, but deliver their result in AR.

Expressions within parentheses, on the other hand, are not treated as closed subexpressions. Instead the parentheses are taken into account implicitely through the order of execution of operations.

11.5.2. Values, simple variables, function designators without parameters, formal parameters.

The values of primaries, which are either themselves values, or in the ALGOL program are represented by single identifiers, are brought to the AR by means of the following instructions:

Integer	ni A 60	0 + H	ni
Number	nr A 40	0 + F	nr
Logical value	lv A 60	0 + H	lv
Label	la A 60	0 + H	la
Simple variable, integer	i A 60	0 + H	i
Simple variable, real	r A 40	0 + F	r
Simple variable, boolean	b A 60	0 + H	b
Function designator without parameters, declared	p A 16	SEK	p
Function designator without parameters, standard	p A 17	SKL	p
Formal real parameter	fr A 37	UDF	fr
Formal integer parameter	fi A 37	UDF	fi

A formal integer must always be rounded before it is used.

The list of addresses of standard procedures without parameters is as follows:

læsstreng	1337
læst	1232
skrvtab	1736
skrvvr	1733
trykende	1727
trykklar	1723
trykslut	1731
trykstop	1729
tryksum	1712
tryktab	1737
trykvr	1734

11.5.3. Subscripted variables.

The value of a subscripted variable is brought to AR through the following steps: 1) The values of all subscript expressions are stacked. 2) The internal array identifier is brought to AR. 3) A wired subroutine is called in. These three steps will be treated separately:

11.5.3.1. Stacking a subscript. There are three cases, according to how the value of the subscript expression has been evaluated:

Result of expression	Bring to stack through		
Integer in AR	595 A 17	SKL	595
Real or formal in AR	569 A 17	SKL	569
Real in stack	566 A 17	SKL	566

The subscripts must be stacked one by one in this way.

11.5.3.2. Array identifier to AR. There are two cases:

Array identifier is			
Not formal	a A 40	0 + F	a
Formal	f A 37	UDF	f

11.5.3.3. Subscripted variable to AR. This is done by

	162 A 17	SKL	162

This also counts the stack back.

11.5.3.4. Example.

a[i, r1, r2×r3, fi] will appear as			
	i A 60	0 + H	i
	595 A 17	SKL	595
	r1 A 40	0 + F	r1
	569 A 17	SKL	569
	r2 A 40	0 + v	r2
	594 A 17		
	r3 A 40	a × v	r3
	330 A 17		
	566 A 17	SKL	566
	fi A 37	UDF	fi
	569 A 17	SKL	569
	a A 40	0 + F	a
	162 A 17	SKL	162

11.5.4. Function designators with parameters.

The value of a function designator is brought into AR by a call instruction followed by instructions representing the parameters. The instruction (or last instruction) of the last parameter is always marked with the C - index mark.

11.5.4.1. The call instruction may be one of three things:

Declared procedure	p A 16	SEK	p
Formal procedure identifier	f A 37	UDF	f
Standard procedure	p A 17	SKL	p

The addresses of the available standard procedures, to be used in p A 17 instructions, are the following:

Identifier	Address	Identifier	Address
abs	1763	skrvvr	1733
arctan	1906	sqrt	1791
cos	1888	streng	1356
entier	1774	tryk	1401
exp	2020	trykende	1727
ln	1992	trykklar	1723
læs	1197	trykkopi	1373
læsstreng	1337	trykml	1745
læst	1232	trykslut	1731
sign	1778	trykstop	1729
sin	1830	tryksum	1712
skrv	1402	tryktab	1737
skrvkopi	1372	tryktekst	1692
skrvml	1746	tryktom	1747
skrvtab	1736	trykvr	1734
skrvtekst	1691		

Only some of these standard procedures define the values of function designators. The complete list is given here for ease of reference.

11.5.4.2. Single identifier parameters. All parameters which in the ALGOL program are values or single identifiers are represented by single instructions. For those of them which define values the representation is exactly the instruction for bringing the value into AR as given in section 11.5.2. The remaining cases are represented as follows:

Array identifier	a A 44	0 + F	a
Switch identifier	s A 16	SEK	s
Procedure identifier (declared)	p A 16	SEK	p
Procedure identifier (standard)	p A 17	SKL	p

Remember in all cases the C - index mark for the last parameter.

11.5.4.3. Subscripted variable as parameter. A subscripted variable appearing as parameter is represented by a series of instruction having four parts as follows:

11.5.4.3.1. The instruction w A 74 D>P w

where w refers to the last instruction of the fourth part (see section 11.5.4.3.4).

11.5.4.3.2. Instructions for stacking the values of the subscripts, in the same way as described in section 11.5.3.1.

11.5.4.3.3. Array identifier to AR. As described in section 11.5.3.2.

11.5.4.3.4. Two fixed instructions: 150 A 17 SKL 150

w: 0A/C75 N > D (,c)

The last of these is the one referred to in section 11.5.4.3.1. above. It must be C - index marked if it is the last instruction of the complete function designator.

11.5.4.4. Other expressions. All other expressions, i.e. all such expressions which are not values, single identifier expressions, or subscripted variables, are represented by a series of instructions having three parts as follows:

11.5.4.4.1. The instruction

```
        w A 74    D > P  w
```

similar to the one of section 11.5.4.3.1. w refers to the last instruction of the third part.

11.5.4.4.2. Any set of instructions bringing the value of the expression into AR (store packed form). These instructions may make use of the stack in the usual manner, but must of course return the value of IRB to its original value. This will happen automatically if the instructions represent a proper ALGOL code.

11.5.4.4.3. Two fixed instructions:

```
        147 A 17    SKL   147
w:      0A/C75      N > D (,c)
```

cf. section 11.5.4.4.1 above. The C - index mark must be used if the expression is the last parameter of the function designator.

11.5.4.5. Example.

p(r1, a[i1], i2 - r2) will appear as

```
        p A 16      SEK   p
       r1 A 40      0 + F r1
       w1 A 74      D > P w1
       i1 A 60      0 + H i1
      595 A 17      SKL   595
        a A 40      0 + F a
      150 A 17      SKL   150
w1:     0 A 75      N > D 0
       w2 A 74      D > P w2
       i2 A 60      0 + H i2
      594 A 17      0 + v
       r2 A 40      a - v r2
      264 A 17
      361 A 17      a > v
      147 A 17      SKL   147
w2:     0 C 75      N > D ,c
```

11.5.5. Relations.

The value of a relation is brought to AR through a series of instructions having two parts:

11.5.5.1. Stacking the difference of the two arithmetic expressions. The difference of the values of the first and second arithmetic expression is brought to the stack, using real mode arithmetics (cf. section 11.5.7.3). If the second expression consists of the constant 0 the subtraction will be omitted.

11.5.5.2. Forming the logical value. The logical value is brought to AR through the following instructions (the constant at minus: 0 C 00 A+F, c. minus = 66):

11.5.5.2.1. Operator < (and ≥).

```
          2 B 35    N > B  2,b
(minus A 20)        (A + H  minus)
```

11.5.5.2.2. Operator > (and ≤).

```
        252 A 17     0 - a
          2 B 35     N > B  2,b
(minus A 20)         (A + H  minus)
```

11.5.5.2.3. Operator ≠ (and =).

```
          w A 51     HP-  k + 2
        252 A 17     0 - a
w:        2 B 35     N > B  2,b
(minus A 20)         (A + H  minus)
```

11.5.5.3. Example.
i1 ≥ i2 + i3 is represented:

```
       i1 A 60     0 + H  i1
      594 A 17     0 + v
       i2 A 60     0 + H  i2
      594 A 17     0 + v
       i3 A 60     0 + H  i3
      257 A 17     a + v
      261 A 17     s - a
        2 B 35     N > B  2,b
minus A 20         A + H  minus
```

11.5.6. Switch designators.

The value of a switch designator is brought to AR through a series of instructions having two parts:

11.5.6.1. Value of subscript as integer to AR. Form of instructions as described in section 11.5.7.

11.5.6.2. Call switch. There are two cases:

```
Declared switch identifier     s A 16     SEK    s
Formal switch identifier       f A 37     UDF    f
```

11.5.6.3. Example.
s[1 + i] is represented

```
one A 60     0 + H  one
  i A 20     A + H  i
  s A 16     SEK    s
```

11.5.7. Simple arithmetic expressions.

11.5.7.1. Method of evaluation.

The evaluation of simple arithmetic expressions proceeds as follows: The values of the primaries are found, taking them in order from left to right. However, as soon as the value of a primary has been found as many of the operations are performed as is permitted by the rules of precedence. Thus, in between the evaluation of two primaries any number of operations (zero included) may take place. This method makes it possible to store all intermediate results in the stack and to operate only on the results stored in the top of the stack.

Complications arise, however, from the differences between real and integer mode arithmetics. Thus while real mode arithmetics must always have one operand in the stack, in integer mode it is sometimes possible to operate between AR and the store. This will be further explained below.

11.5.7.2. The use of real and integer mode arithmetics. The choice of mode is made primarily from the type of the variable (declared or internal) to which the value of the expression will be assigned, and secondarily by the types of the constituents.

11.5.7.3. Real mode arithmetics. Expressions to be assigned to a real variable, those of relations (cf. section 11.5.5.1) and actual parameters of function designators and procedure statements, will be evaluated in the real mode throughout, irrespective of the types of the constituents. The instructions for bringing the values of primaries into AR have already been described (sections 11.5.2, 11.5.3, 11.5.4, and 11.5.5). Note that since integers are represented like reals (cf. section 11.4.4) they may enter into real mode operations without modification. The arithmetic operators are represented by six kinds of instructions, which all require the contents of AR to be represented in the store packed form (cf. sections 11.4.2 and 11.4.4):

11.5.7.3.1. Unpack AR to stack. There are three cases:

IRB:= IRB - 2; stack[IRB]:= AR	594 A 17	0 + v
IRB:= IRB - 2; stack[IRB]:= -AR	242 A 17	0 - v
IRB:= IRB - 2; stack[IRB]:= round (AR)	552 A 17	SKL 552

The variants including round-off are used in case of formal integer parameters (cf. section 11.5.2).

11.5.7.3.2. Rounding in AR. Two cases.

AR:= entier(AR + 0.5)	557 A 17	SKL 557
AR:= -entier(AR + 0.5)	559 A 17	SKL 559

11.5.7.3.3. Change sign in stack.

stack[IRB]:= -stack[IRB]	252 A 17	0 - a

11.5.7.3.4. Operate stack with AR. Five cases.

stack[IRB]:= stack[IRB] + AR	257 A 17	a + v
stack[IRB]:= stack[IRB] - AR	264 A 17	a - v
stack[IRB]:= stack[IRB] × AR	330 A 17	a × v
stack[IRB]:= stack[IRB] / AR	340 A 17	a / v
stack[IRB]:= stack[IRB] ↑ AR	368 A 17	a ↑ v

11.5.7.3.5. Operate in stack. Five cases.

IRB:= IRB + 2; stack[IRB]:= stack[IRB] + stack[IRB - 2]	254 A 17	s + a
IRB:= IRB + 2; stack[IRB]:= stack[IRB] - stack[IRB - 2]	261 A 17	s - a
IRB:= IRB + 2; stack[IRB]:= stack[IRB] × stack[IRB - 2]	327 A 17	s × a
IRB:= IRB + 2; stack[IRB]:= stack[IRB] / stack[IRB - 2]	337 A 17	s / a
IRB:= IRB + 2; stack[IRB]:= stack[IRB] ↑ stack[IRB - 2]	365 A 17	s ↑ a

11.5.7.3.6. Pack stack to AR.

AR:= stack[IRB]; IRB:= IRB + 2	361 A 17	a > v

11.5.7.4. Examples of expressions in real mode arithmetics:

i + r - fr is represented:

```
  i A 60     0 + H  i
594 A 17     0 + v
  r A 40     a + v  r
257 A 17
 fr A 37     UDF    fr
264 A 17     a - v
361 A 17     a > v
```

i1 ↑ r1 × r2 / r3 - i2 is represented:

```
 i1 A 60     0 + H  i1
594 A 17     0 + v
 r1 A 40     a ↑ v  r1
368 A 17
 r2 A 40     a × v  r2
330 A 17
 r3 A 40     a / v  r3
340 A 17
 i2 A 60     0 + H  i2
264 A 17     a - v
361 A 17     a > v
```

- r1 + fr / r2 / fi ↑ i is represented:

```
 r1 A 40     0 - v  r1
242 A 17
 fr A 37     UDF    fr
594 A 17     0 + v
 r2 A 40     a / v  r2
340 A 17
 fi A 37     UDF    fi
552 A 17     SKL    552
  i A 60     0 + H  i
368 A 17     a ↑ v
337 A 17     s / a
254 A 17     s + a
361 A 17     a > v
```

- (fr + i / r) × fi is represented:

```
 fr A 37     UDF    fr
594 A 17     0 + v
  i A 60     0 + H  i
594 A 17     0 + v
  r A 40     a / v  r
340 A 17
254 A 17     s + a
252 A 17     0 - a
 fi A 37     UDF    fi
557 A 17     SKL    557
330 A 17     a × v
361 A 17     a > v
```

11.5.7.4. Integer mode expressions. Integer mode arithmetics will be used for the evaluation of subscripts and the expressions of assignment statements having left part variables of type integer. In integer mode expression the evaluation (proceeding from left to right) will be carried out using the fixed-point arithmetic of the machine as long as possible, i.e. as long as only variables and function designators of type integer and the operators +, -, and ×, are involved. If an operand of type real or one of the operators / and ↑ is encountered the rest of the expression will be evaluated in real mode and the final result rounded.

In the following the notation ARi and stacki.will be used to denote contents of AR and the stack known to be on integer form.

11.5.7.4.1. Round-off. Round-off to the nearest integer, and a conversion to the integer representation (section 11.4.3), will be performed in two kinds of cases: (1) when a formal name parameter specified to be an integer has been brought to AR (cf. section 11.5.2) and (2) before a result of real arithmetics is used as a subscript or assigned to an integer. The following operations are available:

ARi:= round (AR)	557	A 17	SKL	557
ARi:= -round (AR)	559	A 17	SKL	559
ARi:= round (stack [IRB]); IRB:= IRB + 2	554	A 17	hel	
ARi:= stack i [IRB] + round (AR); IRB:= IRB + 2	546	A 17	SKL	546
ARi:= stack i [IRB] - round (AR); IRB:= IRB + 2	548	A 17	SKL	548
ARi:= stack i [IRB] × round (AR); IRB:= IRB + 2	550	A 17	SKL	550
stack i [IRB]:= round (stack [IRB])	566	A 17	SKL	566
IRB:= IRB - 2; stack i [IRB]:= round (AR)	552	A 17	SKL	552

11.5.7.4.2. Integer mode operations. The result of an integer mode operation will always be placed in AR. However, the operands may have to be taken from the store and AR or from AR and the stack, depending on the circumstances. The operations are:

ARi:= - ARi	561	A 17	SKL	561
IRB:= IRB - 2; stack i [IRB]:= AR i	594	A 17	0 + v	
One operand in store				
ARi := i	i	A 60	0 + H	i
ARi:= - i	i	A 61	0 - H	i
ARi:= ARi + i	i	A 20	A + H	i
ARi:= ARi - i	i	A 21	A - H	i
ARi:= ARi × i	zero	A 24	A + H	zero
	i	A 2A	M × H	i
	19	A 0C	VSK	19
One operand in stack				
ARi:= stack i [IRB] + ARi; IRB:= IRB + 2	602	A 17	SKL	602
ARi:= stack i [IRB] - ARi; IRB:= IRB + 2	600	A 17	SKL	600
ARi:= stack i [IRB] × ARi; IRB:= IRB + 2	605	A 17	SKL	605

11.5.7.4.3. Examples of expressions in integer mode arithmetics (cf. section 11.5.3.1.)

b[fi1 + i1, i2 - fi2, i3 ×(i4 - i5)]	fi1 A 37	UDF	fi1
	557 A 17	SKL	557
	i1 A 20	A + H	i1
	595 A 17	SKL	595
	i2 A 60	0 + H	i2
	594 A 17	0 + v	
	fi2 A 17	SKL	fi2
	548 A 17	SKL	548
	595 A 17	SKL	595
	i3 A 60	0 + H	i3
	594 A 17	0 + v	
	i4 A 60	0 + H	i4
	i5 A 21	A - H	i5
	605 A 17	SKL	605
	595 A 17	SKL	595
c[-i1 - i2 × i3]	i1 A 61	0 - H	i1
	594 A 17	0 + v	
	i2 A 60	0 + H	i2
	zero A 24	A + H	zero
	i3 A 2A	M × H	i3
	19 A 0C	VSK	19
	600 A 17	SKL	600
	595 A 17	SKL	595

11.5.8. Simple Boolean expressions.

11.5.8.1. Method of evaluation. The evaluation of Boolean expressions is based on the following system: Before the operation the first operand must be placed in the stack while the second must be placed in the AR. The result of the operation, on the other hand, may be placed either in AR or in the stack, depending on the need.

11.5.8.2. Operations. The notation ARb and stack b will be used to denote boolean values placed in the AR and the stack.

```
ARb:= b                                          b A 60     0 + H  b

IRB:= IRB - 2; stack b [IRB]:= ARb             595 A 17     SKL    595

ARb:= stack b [IRB]; IRB:= IRB + 2             362 A 57     OSK    362
ARb:= -, ARb                                minus A 20     A + H minus

ARb:= stack b [IRB] ∧ ARb; IRB:= IRB + 2         w A 11     HP+      w
                                                 0 B 60     0 + H  0,b
                                           w:    2 B 35     N > B  2,b

ARb:= stack b [IRB] ∨ ARb; IRB:= IRB + 2         w A 51     HP-      w
                                                 0 B 60     0 + H  0,b
                                           w:    2 B 35     N > B  2,b

ARb:= stack [IRB] ≡ ARb; IRB:= IRB + 2          w1 A 11     HP+     w1
                                                w2 A 50     OHP     w2
                                          w1: minus A 60     0 + H minus
                                          w2:   362 A 17     SKL     362

stack b [IRB] := stack [IRB] ∧ ARb                w A 51     HP-      w
                                                  0 B 28     A > H  0,b
                                           w:

stack b [IRB]:= stack b [IRB] ∨ ARb               w A 11     HP+      w
                                                  0 B 28     A > H  0,b
                                           w:

stack b [IRB]:= stack b [IRB] ≡ ARb               w A 51     HP-      w
                                              minus A 60     0 + H minus
                                                  0 B 26     H + A  0,b
                                           w:
```

11.5.8.3. Examples of Boolean expressions.

```
b1 ∨ (b2 ≡ b3)                                  b1 A 60     0 + H  b1
                                               595 A 17     SKL    595
                                                b2 A 60     0 + H  b2
                                               595 A 17     SKL    595
                                                b3 A 60     0 + H  b3
                                                w1 A 11     HP+     w1
                                                w2 A 50     OHP     w2
                                          w1: minus A 60     0 + H minus
                                          w2:   362 A 17     SKL     362
                                                w3 A 51     HP-     w3
                                                 0 B 60     0 + H  0,b
                                          w3:    2 B 35     N > B  2,b
```

```
r > 0 ∧ r < 1                          r A 40     0 + v  r
(cf. section 11.5.5.)                594 A 17
                                     252 A 17     0 - a
                                       2 B 35     N > B  2,b
                                     595 A 17     SKL    595
                                       r A 40     0 + v  r
                                     594 A 17
                                     one A 60     0 + H  one
                                     264 A 17     a - v
                                       2 B 35     N > B  2,b
                                       w A 11     HP+    w
                                       0 B 60     0 + H  0,b
                                  w:   2 B 35     N > B  2,b
```

11.5.9. If clauses and else.

If clauses, whether they occur in expressions or in statements, are always represented alike. They have two parts.

11.5.9.1. Instructions for bringing the value of the Boolean expression to AR, according to section 11.5.2, 11.5.3, 11.5.4, 11.5.5, or 11.5.8.

11.5.9.2. The conditional instruction `w A 11 HP+ w` where w is the point to be reached if the expression is false.

11.5.9.3. The delimiter else is represented by the unconditional instruction `w A 10 HOP w`

11.5.9.4. Example of conditional expression.

```
if r1 < 0 then r2 else r3 + r4         r1 A 40     0 + v  r1
                                     594 A 17
                                       2 B 35     N > B  2,b
                                      w1 A 11     HP+    w1
                                      r2 A 40     0 + F  r2
                                      w2 A 10     HOP    w2
                                  w1: r3 A 40     0 + v  r3
                                     594 A 17
                                      r4 A 40     0 + F  r4
                                     257 A 17     SKL    257
                                     361 A 17     a > v
                                  w2:
```

11.6. STATEMENTS.

11.6.1. Block begin.

There are four kinds of blocks: 1) Core block, 2) Core procedure body, 3) Drum block, 4) Drum procedure body. The representation of any block begin involves the following parameters:

store limit	This is the address of the highest point of the store used by the block.
block number	An identifying integer obtained by counting the block begin s from the begin of the program.

In addition, for drum blocks the following parameters must be supplied:

drum track	Drum track address of first track.
first order	The address of the first order in the core store (even).
last order	The address of the last order (odd).

Using this notation the representation is as follows:

Core block begin

```
        722 A 17  SKL 722
store limit A block no.
```

Core procedure body begin

```
        726 A 17  SKL 726
store limit A block no.
```

Drum block begin

```
        755 A 17  SKL 755
store limit A block no.
 drum track A 1C  SEL drum track
first order A 55  N>C first order
 last order A 00  A+F last order
```

Drum procedure body begin

```
        762 A 17  SKL 762
store limit A block no.
 drum track A 1C  SEL drum track
first order A 55  N>C first order
 last order A 00  A+F last order
```

It holds generally that the different parts of an ALGOL program are stored in exactly the order in which they appear in the ALGOL program itself. This order is, however, broken by each appearance of a drum block. Drum blocks are in their entirety removed from their original place in the program. Only the drum block begin as shown above will appear in the original place while the remaining parts are stored in the section defined by the parameters: first order, and last order. The drum block begin administration will both perform a transfer from drum (if necessary) and jump to first order. Again, drum block end will cause a return to the instruction following the parameter: last order, while drum procedure body end will return to the place from where the procedure was called.

11.6.1.1. Block parameters in stack. Upon entry into a block various parameters belonging to the previous level are placed in the stack and new values are placed in certain fixed locations (cf. section 11.8.1). These latter include in location 48 the address of the latest block entry in the stack, the stack reference SR. The block entry consists of two full words as follows:

```
SR:                                          ss A ba
SR+1: For current block = core block          0 D 55     N > C  0,d
      For current block ≠ core block       exit A 55     N > C  exit
SR+2:                                        sr A tb
SR+3:                                         g A fk
```

The meaning of the parameters is the following:

ss The normal value of IRB in current block. If there are arrays declared in the current block it will be the address of the last full word used for these. Otherwise SR = ss.

ba The block number of the current block.

exit exit + 1 is the address to which control must be transferred after end of the current block.

sr The stack reference (SR) belonging to the previous block.

tb The block number of an available drum block in the previous level (cf. section 11.8.1 location 54).

g Store limit of previous block (cf. location 49).

fk Last drum track used by previous block (cf. location 55).

The meaning of some of the parameters at a given moment may be illustrated by the following picture of the central part of the core store:

```
                  g:          Program

Current store limit:
                  - - - - - - - - -

              arrow:          Free store
                  - - - - - - - - - -
                IRB:          Parts of expressions, subscripts, etc. in
                              current block.

                  - - - - - - - - -
                 ss:          Subscripted variables in current block.

                  - - - - - - - - -

                 SR:  ss A ba Block entry
                       0 D 55
                      sr A tb
                       g A fk
                  - - - - - - - - -
                              Parts of expr., previous block.
                  - - - - - - - - -
                              Subscr. var. previous block
                  - - - - - - - - -
                 sr:          Block entry.
```

11.6.2. Block end. There are two cases:

11.6.2.1. End of procedure body defining the value of a function designator:

```
                                        p1 A 40     0 + F  p1
                                       797 A 17     SKL    797
```

11.6.2.2. All other block end s:

```
                                       797 A 17     SKL    797
```

11.6.3. Assignment statements. The representation has three parts:

11.6.3.1. The left part list. Taking the left part variables in order from left to right instructions are compiled as following:
11.6.3.1.1. Simple variables and formal variables called by value: Nothing is compiled.
11.6.3.1.2. Formal variables called by name: f A 77 OUD f
11.6.3.1.3. Subscripted variables. There are three sections:
11.6.3.1.3.1. The values of subscripts are stacked. See section 11.5.3.1.
11.6.3.1.3.2. Array identifier to AR. See section 11.5.3.2.
11.6.3.1.3.3. Array identifier to stack. One instruction:

```
                                       595 A 17     SKL    595
```

11.6.3.2. The value of the expression to AR. For arithmetic expressions the type of the left part variables will determine whether this evaluation will be done in real or integer mode arithmetics (cf. the complete section 11.5).

11.6.3.3. Assignment. Taking the left part variables in the reverse order (from right to left) the assignment will be represented by:
11.6.3.3.1. Simple, and formal value, variables,

```
type real:                            r/fr A 08     A > F  r/fr
type integer or boolean               i/fi A 28     A > H  i/fi
```

11.6.3.3.2. Formal variables called by name: f+1 A 37 UDF f+1
11.6.3.3.3. Subscripted variables: 158 A 17 SKL 158

11.6.3.4. Example of assignment statement.

```
fr:= f1[i]:= r1:= r2 - r3              fr A 77     OUD    fr
                                        i A 60     0 + H  i
                                       595 A 17    SKL    595
                                       f1 A 37     UDF    f1
                                       595 A 17    SKL    595
                                       r2 A 40     0 + v  r2
                                       594 A 17
                                       r3 A 40     a - v  r3
                                       264 A 17
                                       361 A 17    a > v  r1
                                       r1 A 08
                                       158 A 17    SKL    158
                                      fr+1 A 37    UDF    fr+1
```

11.6.4. Go to statements. There are two cases:

11.6.4.1. Go to local label. Representation:

```
la A 10     HOP   la
```

11.6.4.2. Go to non-local or computed label. There are two sections:
11.6.4.2.1. Label to AR. See sections 11.4.6, 11.5.6, 11.5.9.
11.6.4.2.2. Go to administration:

```
811 A 17    SKL   811
```

11.6.4.3. Example of go to statement.
go to if b1 then s[i1 - i2] else la

```
      b1 A 60     0 + H  b1
      w1 A 11     HP+    w1
      i1 A 60     0 + H  i1
      i2 A 21     A - H  i2
       s A 16     SEK    s
      w2 A 10     HOP    w2
w1: la1 A 60      0 + H  la
w2: 811 A 17      SKL    811
```

la1 must contain la A block no.

11.6.5. For statement.
The three kinds of for list elements and the controlled statement are represented as follows (where w refers to section 11.6.5.4.):

11.6.5.1. Arithmetic expression. Two sections (1) Assign the value to the controlled variable, as in sections 11.5.3.2. and 11.6.3.3. (2) The instruction

```
w A 16      SEK    w
```

leading to the first instruction representing the controlled statement (section 11.6.5.4.).

11.6.5.2. Step-until-element. The representation depends on the type of the controlled variable.
Integer: for i/fi:= a step b until c

```
    Instructions for
    i/fi:= a
    1009 A 75      N > D 1009
      w1 A 74      D > P w1
w2: Instructions for
    ARi:= b
    594 A 17       0 + v
    Instructions for
    ARi:= c
      1 B 28       A > H  1,b
w1: (0) A 17       SEK
   i/fi A 60/37    0+H/UDF i/fi
      w A 16       SEK    w
     w2 A 10       HOP    w2
```

Real: for r/fr:= a step b until c

```
          Instructions for
          r/fr:= a
        1034 A 75      N > D 1034
          w1 A 74      D > P w1
w2: Instructions for
IRB:= IRB - 2 ; stack [IRB]:= b
          Instructions for
IRB:= IRB - 2 ; stack [IRB]:= c
w1:   (0) A 17      SKL   (0)
      r/fr A 40/37  0+F/UDF r/fr
         w A 16     SEK   w
        w2 A 10     HOP   w2
```

11.6.5.3. While-element. The element E while F is represented:

```
 w1: Instructions for assigning
 E to the controlled variable;
 Instructions for ARb:= F;
        w2 A 11     HP+   w2
         w A 16     SEK   w
        w1 A 10     HOP   w1
w2:
```

11.6.5.4. The controlled statement:

```
        w3 A 10     HOP   w3
w:      w4 A 74     D > P w4
  Instructions representing
  the controlled statement action
w4:   (0) A 75      N > D (0)
        1 D 10      HOP   1,d
w3:
```

11.6.5.5. Example of for statement.
for fi:= i1 step i2 until i3 do

```
          i1 A 60     0 + H i1
       fi+1 A 37      UDF   fi+1
       1009 A 75      N > D 1009
         w1 A 74      D > P w1
 w2:     i2 A 60      0 + H i2
        594 A 17      0 + v
         i3 A 60      0 + H i3
          1 B 28      A > H 1,b
 w1:   (0) A 17      SKL   (0)
         fi A 37      UDF   fi
          w A 16      SEK   w
         w2 A 10      HOP   w2
         w3 A 10      HOP   w3
 w:      w4 A 74      D > P w4
          - - - -
 w4:   (0) A 75      N > D (0)
        1 D 10       HOP   1,d
 w3:
```

11.6.6. Procedure statement.

The representation of procedure statements follows exactly the same rules as those of function designators, see sections 11.5.2. and 11.5.4.

11.7. DECLARATIONS.

11.7.1. Type declarations.

These only influence the storage allocation, but do not give rise to instructions in the running program.

11.7.2. Array declarations.

In the running program an array declaration disposes of three different parts of the store:

(1) Instructions representing essentially the subscript bound expressions. These must be executed every time an entry into the block is made.

(2) The coefficients in the storage mapping function. These are calculated once at each block entry by the instructions of part (1). Subsequently they are used every time a reference to a subscripted variable is made. They occupy a fixed amount of storage space depending only on the number and dimensions of the arrays, but not on the subscript bounds.

(3) The values of the components. These use a variable amount of storage space, depending on the values of the subscript bounds which have been calculated last. Therefore they are placed in the stack.

11.7.2.1. Structure and subscript bounds. All array declarations of a block head are represented by one call of an administrative subroutine, the subscript bounds and structure of the declarations being represented by parameters of this call. This subroutine call is built up from the following constituents and parameters:

Call and first parameter. a is the even address of the first array identifier (cf. section 11.7.2.3).	847 A 17 SKL 847 a A 08 A > F a
The next parameter must be 2×n A (type) or 0 A 06.	
Begin subscript bounds for n core arrays, or the core bounds for n drum arrays. (type) = 01 for integer or boolean, = 02 for real.	2×n A (type)
The next parameters must represent bound expression.	
Begin subscript bounds for drum.	0 A 06 F + A 0
Next parameters: bound expressions.	
Lower and upper bound expressions	Same representation as that of expressions as parameters of function designators (cf. sections 11.5.4.2, 11.5.4.3, 11.5.4.4.).
Final termination of declarations not involving drum.	0 A 0A M × F 0
Termination of one complete set of drum array parameters, start a new group. Next parameters: bound expressions.	0 A 07 M > A 0
Final end of declaration involving drum array.	0 A 0B A / F 0

11.7.2.2. Example of structure and subscript bound representation.

```
integer array A, B [i1 : i2 + i3, i4 : i5]
comment drum data [i6 : i7, i8 : i9];
real array C [i6 : i7, i8 : i9, i10 : i11];
integer array D [i6 : i7, i8 : i9];
integer array E, F [i6 : i7, i8 : i9, i12 : i13]
```

is represented:

```
                    847 A 17     SKL      847
                      a A 08     A > F    a
                      4 A 01     A - F    4
                     i1 A 60     0 + H    i1
                      w A 74     D > P    w
                     i2 A 60     0 + H    i2
                     i3 A 20     A + H    i3
                    147 A 17     SKL      147
w:                  (0) A 75     N > D    (0)
                     i4 A 60     0 + H    i4
                     i5 A 60     0 + H    i4
                      0 A 06     F + A    0
                     i6 A 60     0 + H    i6
                     i7 A 60     0 + H    i7
                     i8 A 60     0 + H    i8
                     i9 A 60     0 + H    i9
                      2 A 02     A + F    2
                    i10 A 60     0 + H    i10
                    i11 A 60     0 + H    i11
                      2 A 01     A - F    2
                      4 A 01     A - F    4
                    i12 A 60     0 + H    i12
                    i13 A 60     0 + H    i13
                      0 A 0B     A / F    0
```

11.7.2.3. Representation of array identifiers. The address associated with an array identifier (cf. section 11.1.1.) gives access to the coefficients of the storage function. Note that all arrays having the same subscript bounds will refer to the same set of coefficients, and also that all arrays having the same drum subscript bounds will use the same drum coefficients even when the core bounds differ.

Consider the array a, of core dimension p and drum dimension q (p or q may be zero), with the core subscript bounds Lc[i] : Uc[i] (i = 1, ... p) and the drum subscript bounds Ld[i] : Ud[i] (i = 1, ... q). Further form the following coefficients:

```
cc[p]:= if type is real then 2 else 1;
sc:= 0
for i:= p step - 1 until 1 do
begin cc[i - 1]:= (Uc[i] - Lc[i] + 1) × cc[i];
sc:= cc[i] × Lc[i] + sc end;

dc[q]:= 2 × (number of drum tracks per core section of the complete drum
            array group);

for i:= q step - 1 until 2 do
        dc[i - 1]:= (Ud[i] - Ld[i] + 1) × dc[i];
```

Then the array identifier and coefficients are represented as follows:

```
a0: address of first element of array.   a:    a0 C 40/60   0+F/0+H   a0,c
40 for real 60 for integer and boolean.        a1 A 55      N > C     a1

Core coefficients:                       a1:   sc×2↑(-11)
                                               cc[p]×2↑(-12)
                                                - - - - -
                                               cc[i]×2↑(-12)
                                               cc[1]×2↑(-12)
                                 cc[0]×2↑(-12)+(if q≠0 then -1/2 else -1)
If q ≠ 0:                                      k1 A 55      N > C     k1

Drum coefficients:                       k1:   k00 A 1C     SEL       k00
                                               -dc[q]×2↑(-12)
                                                 - - - - -
                                               -dc[2]×2↑(-12)
                                               -dc[1]×2↑(-12)
                                              (k) A 1C     SEL       (k)
                                          First A n1
                                           Last A n2
```

k00 is the track address corresponding to all drum indices = 0.
(k) is the variable track address of the section of the drum array which is currently in the core store.
2×n1 is the track address of the first track in the array.
n2 is the total number of drum tracks in the array.
First is the even first address reserved in the core store for the drum array group.
Last is the odd last address reserved in the core store.

11.7.2.4. Example of array identifiers. For the sake of clarity it is assumed that when the following arrays were declared IRB had the value 1536 and the location 55 the value 384 × 2↑(-20) indicating 384 to be the last drum track used (cf. section 11.8.1).

```
integer array A, B [2 : 4, -1 : 4];
comment drum data [-4 : -3, -1 : 0];
real array C [-4 : -3, -1 : 0, -12 : 9, 0 : 9];
integer array D [-4 : -3, -1 : 0];
integer array E, F [-4 : -3, -1 : 0, 1 : 3];
```

is represented:=

```
A:      1524 C 60     0 + H  1524,c
          A1 A 55     N > C  A1
= B:    1512 C 60     0 + H  1512,c
          B1 A 55     N > C  B1
A1: B1:    9 A 00     A + F  9
           0 B 00     A + F  0,b
           2 A 00     A + F  2
           6 C 00     A + F  6,c
k1:      404 A 1C     SEL    404
        2046 C 00     A + F  2046,c
        2044 C 00     A + F  2044,c
         368 A 1C     SEL    368
        1424 B 38     38     1424,b
        1511 A 08     A > F  1511
= C:    1432 C 40     0 + F  1432,c
          C1 A 55     N > C  C1
= C1:   1808 C 00     A + F  1808,c
           1 A 00     A + F  1
          10 A 00     A + F  10
        1064 C 00     A + F  1064,c
          k1 A 55     N > C  k1
             A
= D:    1431 C 60     0 + H  1431,c
          k1 A 55     N > C  k1
= E:    1428 C 60     0 + H  1428,c
          E1 A 55     N > C  E1
= F:    1425 C 60     0 + H  1425,c
          F1 A 55     N > C  F1
E1: F1:    1 A 00     A + F  1
           0 B 00     A + F  0,b
        1025 D 00     A + F  1025,d
          k1 A 55     N > C  k1
```

11.7.3. Switch declarations.

A switch declaration is generally represented by a set of instructions having three parts, as follows:

11.7.3.1. Pass-by jump. The first switch declaration of a block head will start with an unconditional jump instruction

```
                    w A 10      HOP    w
```

leading past all other switch and procedure declarations of the block head.

11.7.3.2. Call switch administration. Two fixed instructions:

```
          s:        0 D 55      N > C  0,d
                  824 A 17      SKL    824
```

11.7.3.3. The switch list. The designational expressions of the switch list are represented in exactly the same manner as the parameters of a function designator (cf. sections 11.5.4.2, and 11.5.4.4). The very last instruction of those representing the list must be C-index marked.

11.7.3.4. Example of switch declaration.

```
switch s1:= la1, if b then la2 else la3, s2[i], f
                   (w1 A 10)   (HOP    w1)
          s1:        0 D 55     N > C  0,d
                   824 A 17     SKL    824
                  la10 A 60     0 + H  la10
                    w2 A 74     D > P  w2
                     b A 60     0 + H  b
                    w3 A 11     HP+    w3
                  la20 A 60     0 + H  la20
                    w4 A 10     HOP    w4
          w3:     la30 A 60     0 + H  la30
          w4:      147 A 17     SKL    147
          w2:      (0) A 75     N > D  (0)
                    w5 A 74     D > P  w5
                     i A 60     0 + H  i
                    s2 A 16     SEK    s2
                   147 A 17     SKL    147
          w5:      (0) A 75     N > D  (0)
                     f C 37     UDF    f,c
```

where the labels must be available in the store as follows:

```
          la10:   la1 A (block no.)
          la20:   la2 A (block no.)
          la30:   la3 A (block no.)
```

11.7.4. Procedure declarations.

The first procedure declaration of a block head will begin with a pass-by jump, unless a switch declaration has come before (cf. section 11.7.3.1).

11.7.4.1. Procedure heading. If the procedure has no parameters the heading is represented by two fixed instructions:

```
  0 D 55     N > C  0,d
660 A 17     SKL    660
```

If there are parameters the heading will appear as:

```
  0 D 55     N > C  0,d
f00 A 08     A > F  f00
660 A 17     SKL    660
             Value and type code
```

Here f00 is the even address of the first formal location. The value and type code will consist of one or more half-words constructed from the specifications as follows:

1) The first bit of each word is zero.
2) The following 16 bits of each word contain codes for the formal parameters, each parameter using 2 bits as follows:
   Parameter is called by name 10
   Parameter is called by value, not integer 01
   Parameter is called by value, integer 11
3) The last half word is filled up with zeroes to the right, at least 3 of them. All other half words must end with the bits 001.

It is seen that one half-word will accomodate up to 8 parameters, two half-words up to 16, etc.

11.7.4.2. Procedure body. The procedure body will always be translated as a block, i.e. begin and end will if necessary be inserted by the translator. The representation of procedure block begin and end is given in section 11.6.1. and 11.6.2. All the rest of the procedure body is represented according to the general rules for declarations and statements. If the procedure defines a value this value will be stored in the full word immediately preceding the first formal location (i.e. in address f00 - 2).

11.7.4.2. Example of procedure declaration.

```
procedure Innerproduct (a, b) Order: (k, p) Result:(y) ;
value k; integer k, p ; real y, a, b ;
begin real r ;
      r:= 0 ;
      for p:= 1 step 1 until k do r:= r + a × b ;
      y:= r
end
```

is represented (formal locations from 0 AF = F):

```
                          0 D  55     N > C  0,d
                          0 AF 08     A > F  F
                        660 A  17     SKL    660
                         B 57400      s 57400
                        726 A  17     SKL    726
                store limit A (block no.)
                          r A  48     0 > F  r
                        one A  60     0 + H  one
                          7 AF 37     UDF    F + 7
                       1009 A  75     N > D  1009
                         w1 A  74     D > P  w1
          w2:         one A  60     0 + H  one
                        594 A  17     0 + v
                          4 AF 60     0 + H  F + 4
                          1 B  28     A > H  1,b
          w1:         (0) A  17     SKL    (0)
                          6 AF 37     UDF    F + 6
                         w3 A  16     SEK    w3
                         w2 A  10     HOP    w2
                         w4 A  10     HOP    w4
          w3:         w5 A  74     D > P  w5
                          r A  40     0 + v  r
                        594 A  17
                          0 AF 37     UDF    F
                        594 A  17     0 + v
                          2 AF 37     UDF    F + 2
                        330 A  17     a × v
                        254 A  17     s + a
                        361 A  17
                          r A  08     a > v  r
          w5:         (0) A  75     N > D  (0)
                          1 D  10     HOP    1,d
          w4:           8 AF 77     OUD    F + 8
                          r A  40     0 + F  r
                          9 AF 37     UDF    F + 9
                        797 A  17     SKL    797
```

11.8. PARAMETERS WITH FIXED LOCATIONS IN THE CORE STORE.

As already mentioned (cf. section 11.2) some of the core locations 0 - 63 reserved for the use of the wired store contain the current values of certain universal parameters. Grouped according to the program with which they are associated they are the following:

11.8.1. Universal block parameters (cf. section 11.6.1.1).

Location 29. Arrow (stack warning limit). This is an address used for checking whether the stack runs into the program. It will always take a value somewhere in the interval between the current store limit (the highest address used by the program, cf. location 49) and the top of the stack (defined by IRB). The value of arrow will only be changed when necessary to keep it in the above mentioned interval. The need for changing it is a warning that the stack and the program may be about to overlap. The contents of 29:

(2048 - arrow) B 55

Location 44. First free drum track. The contents of 44 is

2↑(-20) × address of first free drum track.

This is the last drum track which may be used by drum arrays (these are stored from the high address end of the drum). Cf. section 10.4.3.

Location 48. Stack reference, SR. SR is the even address of the current block information in the stack (cf. section 11.6.1.1). Contents of 48:

SR A 35 N > B SR

Location 49. Current store limit. The highest address reserved by the current block. This defines the absolute lower limit to the part of the store available to the stack. Contents of 49:

Current store limit A 00

Location 52. Drum procedure depth. This is used to determine whether upon exit from a block it is necessary to cancel the drum procedure quoted in location 53 (if any). Using the letter q, the value is given as follows:

Upon entry into drum procedure q:= -3
Upon entry into other block q:= q - 2
Upon exit from any block

```
q:= q + 2 ; if q > 0 then begin q:= 0 ;
                cancel drum procedure end
```

Contents of location 52:

2↑(-11) × drum procedure depth.

Location 53. Available drum procedure. This gives the block number of the last drum procedure which has been called, unless an exit from the block from where this call took place has been made in the meantime (cf. location 52). Contents of location 53:

2↑(-19) × drum procedure block number

Location 54. Available drum block. This gives the block number of a drum block accessible from the current block without having to be transferred from the drum. Contents of location 54:

$2\uparrow(-19)$ × drum block number.

Location 55. Last drum track used. This gives the smallest drum track address used for drum arrays. This cannot be smaller than first free drum track (cf. location 44). Contents of location 55:

$2\uparrow(-20)$ × address of last drum track used.

11.8.2. Input procedure parameters.

Location 50,51. Input string. These locations hold the string last inputed by means of læs streng (cf. section 9.6.3.2). The form is the normal string form (cf. section 11.4.8).

Location 57. Input sum. This holds the sum of the symbols read from the input tape since the last CLEAR CODE (cf. section 9.2.5).

Location 58. Input case. The case last read from the input tape. Form:

Upper case:	0 A 00	A + F 0
Lower case:	0 A 40	0 + F 0

11.8.3. Output procedure parameters.

Location 18. Last case symbol for the medium (typewriter or punch) not in use (cf. location 62). Form:

No case:	0 A 13	TOM	0
Upper case:	20 A 11	HP+	20
Lower case:	20 A 51	HP-	20

Location 19. Sum of output symbols for medium not in use (cf. location 57).

Location 56. Medium changer. This is used while determining the output medium. Except for very brief periods while changing to typewriter its value should be in the interval $2\uparrow(-11)$ to 1. The medium may however be forced to be either punch or typewriter by inserting an irregular value into 56 as follows:

Type writer	0 C 01	A - F 0,c
Punch	0 A 01	A - F 0

Location 60,61. Medium and output instruction. Normal values:

Punch (tryk)	60	20 A 51	HP-	20
	61	0 A 7A	PA8	0
Typewriter (skrv)	60	20 A 11	HP+	20
	61	0 A 5A	TA8	0

Location 62,63. Case and sum for medium in use. Form as locations 18,19.