ALGOL BULLETIN SUPPLEMENT no. 5.
Supported by The Rask-
Ørsted Fond, Copenhagen

A MANUAL OF THE DASK ALGOL LANGUAGE.

A supplement to the ALGOL 60 report.

First edition, November 1960.

Regnecentralen, Copenhagen.

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 chapters 11 and 12 of the Manual are still in preparation at the present moment.

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	x
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		
27	1B	'	$_{10}$
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.

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.

7.3. 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 operationwhich 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.

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,ddd}, q\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,=,}) 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>|
     <tryksum 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 ⊥ 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⊥dd⊥dd.d0⊥0	+d⊥ddd.ddd⊥d	-ddd.d00$_{10}$+d	±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({<$Q_{\perp}=_{\perp}$})

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 $\perp$ 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 ne cessary, 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 immidiately 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 symbolswith 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 assigament 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. Assigament 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 goto 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. Examples 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 transferringinformation 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 bolean, 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 being 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.