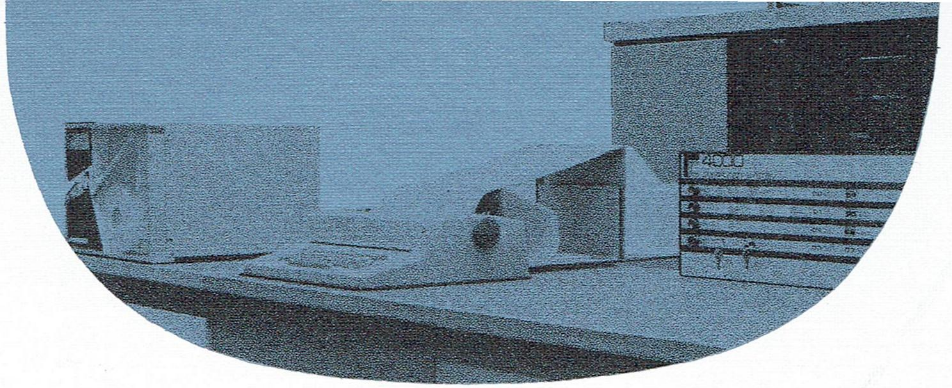


Abs
Add
Arcsin
Arctan
Arg
Blockproc
Blocksread
Case
Changerec
Changerec6
Changevar
Check
Checkvar
Close
Cos
Entier
Exor
Exp
Extend
External
Extract
Field
Getposition
Getshare
Getshare6
Getzone
Getzone6
In
Increase
Inrec
Inrec6
Intable
Invar
Ln
Logand
Logor
Long
Message
Mod
Monitor
Open
Out
Outchar
Outinteger
Outrec
Outrec6
Outtext
Outvar
Overflows
Random
Read
Readall
Readchar
Readstring
Real
Repeatchar
Round
Setposition
Setshare
Setshare6
Setzone
Setzone6
Sgn
Shift
Sign
Sin
Sinh
Sqrt
Stderror
String
Swoprec
Swoprec6
System
Systime
Tableindex
Tofrom
Underflows
Write
Zone

Algol 6

rc 4000 ®
DATAMATICS

RCSL NO: 31-D322

ALGOL 6

USER'S MANUAL

1st edition,

2nd printing

Edited by

Hans Dinsen Hansen

A/S REGNECENTRALEN

ISBN 87 7557 018 1

1. INTRODUCTION — 3 pages
References

2. BASIC SYMBOLS, IDENTIFIERS, NUMERALS, AND STRINGS — 6 pages
2.0.1. Character set and coding
2.0.2. Source text
2.0.3. Source files
2.0.4. Space and New Line
2.1. Letters
2.3. Delimiters
2.4. Identifiers
2.5. Numbers
2.6. Strings
2.7. Quantities, kinds and scope
2.8. Values and types

3. EXPRESSIONS — 9 pages
3.1. Variables and fields
3.2. Function designators
3.3. Arithmetic expressions
3.4. Boolean expressions
3.5. Designational expressions
3.6. String expressions
3.7. Zone expressions

4. STATEMENTS — 3 pages
4.2. Assignment statements
4.6. For statements
4.7. Procedure statements

5. DECLARATIONS — 7 pages
5.1. Type declarations
5.2. Array declarations
5.4. Procedure declarations
5.5. Zone declarations
5.6. Zone array declarations
5.7. Field declarations

6. INPUT/OUTPUT SYSTEM — 19 pages
6.1. Documents
6.1.1. Internal process
6.1.2. Backing storage
6.1.3. Typewriter
6.1.4. Paper tape reader
6.1.5. Paper tape punch
6.1.6. Line printer
6.1.7. Card reader
6.1.8. Magnetic tape
6.1.9. Devices without documents
6.2. High level zone procedures
6.3. Buffering and checking
6.3.1. Multishare input/output
6.3.2. Algorithms for multishare input/output
6.3.3. Standard error reactions
6.3.4. Block procedure
6.4. Primitive level, OS
6.4.1. Communication with documents
6.4.2. Document driver
6.4.3. Operating system

7. SYSTEM CONTROL, ETC. 1 page

8. THE ALGOL SYSTEM 4 pages
- 8.1. Translation
- 8.2. Assembly, index, spill
- 8.3. Execution

9. ALPHABETIC LIST OF NEW ELEMENTS 86 pages
- 9.1. Abs
- 9.2. Add
- 9.3. Arcsin
- 9.4. Arctan
- 9.5. Arg
- 9.6. Blockproc
- 9.7. Blocksread
- 9.8. Case
- 9.9. Changerec
- 9.10. Changerec6
- 9.11. Changevar
- 9.12. Check
- 9.13. Checkvar
- 9.14. Close
- 9.15. Cos
- 9.16. Entier
- 9.17. Exor
- 9.18. Exp
- 9.19. Extend
- 9.20. External
- 9.21. Extract
- 9.22. Field
- 9.23. Getposition
- 9.24. Getshare
- 9.25. Getshare6
- 9.26. Getzone
- 9.27. Getzone6
- 9.28. In
- 9.29. Increase
- 9.30. Inrec
- 9.31. Inrec6
- 9.32. Intable
- 9.33. Invar
- 9.34. Ln
- 9.35. Logand
- 9.36. Logor
- 9.37. Long
- 9.38. Message
- 9.39. Mod
- 9.40. Monitor
- 9.41. Open
- 9.42. Out
- 9.43. Outchar
- 9.44. Outinteger
- 9.45. Outrec
- 9.46. Outrec6
- 9.47. Outtext
- 9.48. Outvar
- 9.49. Overflows

9.50. Random
9.51. Read
9.52. Readall
9.53. Readchar
9.54. Readstring
9.55. Real
9.56. Repeatchar
9.57. Round
9.58. Setposition
9.59. Setshare
9.60. Setshare6
9.61. Setzone
9.62. Setzone6
9.63. Sgn
9.64. Shift
9.65. Sign
9.66. Sin
9.67. Sinh
9.68. Sqrt
9.69. Stderror
9.70. String
9.71. Swoprec
9.72. Swoprec6
9.73. System
9.74. Systime
9.75. Tableindex
9.76. Tofrom
9.77. Underflows
9.78. Write
9.79. Zone

APPENDIX A, Execution times in microseconds 3 pages

APPENDIX B, File Processor commands 6 pages

APPENDIX C, Error messages 6 pages

Index 4 pages

New algol facilities *)

1) freely placed comments

2) message at end medium

3) improved error messages at error in source

4) file numbers on sources

5) listing of bossline numbers

6) listing of selected parts of the source text by means of list.on list.off

7) listing of source names and dates

8) possibility for dynamic change of list situation as well as selection of a copysource, by means of a new delimiter algol

I. Changes in call, new modificators

bossline.{yes / no}

list.{on / off}

copy.{<copysources>}$_1^\infty$

II. Changes in the source text

new compound, commentstring: <* anything *>

new delimiter: algol ... ;

*) as described in RCSL Nc: 31-D366 by Tove Ann Aris

ad 1 commentstring <* anything *> may be placed whereever space is allowed, except in fat comma. Syntactically it is treated like space. The string must not contain <* . Error messages as for text string.

ad 2 In case the text is not listed, and the algol call does not specify message.no, a message is given for end medium.

ad 3 Error messages for source errors are improved to the following possibilities:

error at source: <name> unknown

error at source: <name> not textfile

error at source: <name>.<integer> not magtape

error at source: <name> illegal kind

error at source: <name> connect error

error at source: <name> connect error

error at source: <name> not text

error at source: <name> hard error
device status <name>
<cause>

ad 4 Files in an entry may be specified by <name>.<integer> so that only one entry must be inserted in the catalog. <name> must be an entry which contains the name of the magnetic tape in question. <integer> must not be 0. The resulting filenumber will be the file number in the catalog entry plus <integer>. File 7, 8, 9 and 10 are chosen as follows:

```
t=set mto mt123456 0 7
p=algol t t.1 t.2 t.3
```

ad 5 the call parameter bossline.yes implies that listing or messages besides the linenumber will state the boss linenumber. Standard is bossline.no.

ad 6 The are three degrees of listing:

1) if list.yes is specified the total source text is listed

2) if list.no is specified nothing whatever is listed.

NOTE: If as well list.no as list.yes is specified in the call, only the last specified is valid.

3) if list.yes and list.no are not specified in the call, it is possible to list selected parts of the source text.

This is governed as described below by:

fps mode listing.yes
list.on
list.off

Change of list situation.

If list.on is specified in the parameter list, algol will list the source text of all the sources following this parameter, until a possible parameter list.off is specified.

If list.off is specified in the parameter list, algol will omit listing of the following sources, until a possible parameter list.on is specified.

If the call specifies a source before a list parameter, the source will be listed only in case fps mode listing is yes.

ad 8 Source names are listed unless the call specifies message.no. Date and clock is listed only when the source is selected, i.e. not at unstack, see example.

ad 9 $\text{algol}\left\{\text{<modifier>}\right\}_0^1 \left\{\text{copy.}\begin{Bmatrix}\text{<copysource>}\\ \text{<integer>}\end{Bmatrix}\right\}_0^1$

$\text{<modifier>} ::= \text{list.}\begin{Bmatrix}\text{on}\\ \text{off}\end{Bmatrix}$

If a list parameter is not followed by a copysource, it means that the listmode of the actual source is changed. If a listparameter is followed by a copysource, the list parameter relates only to the copysource.

If no listparameter is specified for the copysource, the copysource will be listed in case the actual source is listed.

An integer parameter is matched with the call in which the parameters are numbered 1,2,... A listparameter in front of copy.<integer> will be blind since the list mode specified in the call will be valid.

Further the delimiter algol is treated as message, i.e. it is listed unless the parameter message.no is specified, and the delimiter must follow either begin or semicolon, further it must be terminated by semicolon.

E X A M P L E.

```
prog=algol list.on copy.t1.t2 list.off copy.t3 t0 bossline.yes

source 1 =      t0
copysource 1 = t1
copycource 2 = t2
copysource 3 = t3


t0:    begin                          listed
       comment 0;                     not listed
       algol list.on;                 not listed, but message
       comment 1;                     listed
       algol list.off;                listed
       comment 2;                     not listed
       algol copy.1<*t1*>;            not listed, but message
       comment 3;                     not listed
       algol list.on copy.t4;         not listed, but message
       comment 4;                     not listed
       algol copy.2<*t2*>;            not listed, but message
       algol copy.3<*t3*>;            nct listed, but message
       comment 5;                     not listed
       end                            not listed

t1:    comment copysource no.1;       listed

t2:    comment copysource no.2;       listed

t3:    comment copysource no.3;       not listed

t4:    comment copysource t4;         listed
```

OUTPUT:

```
prog=algol list.on copy.t1.t2 list.off copy.t3 t0 bossline.yes

t0 2401.75    14.36
    10     1 begin
 1. line      30     3  algol list.on;
    40     4 comment 1;
    50     5 algol list.off;
    line      70     7  algol copy.1<*t1*>;
t1 2401.75    14.33
    10     7 comment copysource no.1;
    20     8

t0
    line      90     9  algol list.on copy.t4;
t4 2401.75    14.33
    10     9 comment copysource t4;
    20    10

t0
    line     110    11  algol copy.2<*t2*>;
t2 2401.75    14.33
    10     11 comment copysource no.2;
    20    12

t0
    line     120    12  algol copy.3<*t3*>;
t3 2401.75    14.33
    line      20    13  end medium
t0
algol end 9
```

1. Introduction

This is a revision of the Algol 5 User's Manual, transforming it into a manual of Algol 6.

The present editor of the Algol manual wishes to express his admiration of the high standard set by his predecessor, Søren Lauesen, therefore large parts and, as far as possible, the style of description were taken over directly from Søren Lauesen's manual. The present editor was not always able to follow the style set in the Algol 5 manual. In some rare cases he could not restrain his urge to make things in his own way.

1.1. Format of the manual

The manual consists of 3 rather different parts:

Chapters 2 to 5 follow section by section the Algol 60 report (ref. 3) and give changes in syntax and semantics relative to the reference. Certain of these sections are new in that they have no counterpart in Algol 60, others only contain changes to Algol 60. It should be obvious from the context which is which.

Chapters 6, 7, and 8 serve as introduction to the input/output system, to the facilities for programming of operating systems, and to the coupling of the Algol system to the surroundings.

Chapter 9 and appendices A, B, and C are the parts used in daily programming. Chapter 9 is therefore an alphabetic list of all standard identifiers and operators, provided with realistic examples.

1.2. Changes relative to Algol 60

1. The representation of the language is changed to enable a good use of the ISO alphabet.
2. A new quantity 'zone' is introduced. Zones are the basis for introduction of a general input/output system, where the user can work on a high level with automatic buffering and error recovery, but where he also may interfere with the administration or work on the most basic level. He may even program operating systems (batch processing, real time, time sharing, etc.) in algol.
3. Procedures may be translated alone, in this way new, Algol-coded, standard procedures may be produced.
4. Field variables, beeing pointers to 'fields' in arrays or zones are introduced. Fields are subsets of arrays and zones, and they do not have to be of the same type as the type of the array.
5. Case expressions and case statements, first suggested by C.A.R. Hoare, are admitted.
6. Operators for working on parts of operands are introduced (pattern operators).
7. A new type, long, is introduced. Longs possess integral values, but their range is extended relative to the range of integers.
8. Some details, left undefined by the Algol 60 report, are defined: owns are initially 0 or false, the controlled value at exit from a for-statement is defined.
9. Errorful programs may be executed until the bad spots are touched.

These changes aim at converting algol from a sophisticated plaything to a more realistic tool for software production.

1.3. Use of the manual

This is not a text, but a user's manual. It is therefore expected that the reader has been introduced to Algol in some other way.

If you are familiar with Algol, but not with Algol 6, you may start reading until section 2.5 to learn the external representation of Algol 6 program. Next, read section 3.1.6 with subsections to become familiar with the internal representation of numbers. Third, read the introduction to chapter 6. Fourth skim chapter 9 and try to get an impression of the procedures and operators you can utilize. Appendix B explains how you call the compiler and execute the translated program.

Keep away from chapters 3 to 5 unless you are familiar with the Algol 60 report (ref. 3), and keep away from 6.3 to 8 unless you want to explore the multiprogramming system and the peripheral devices.

1.4. Acknowledgements

The system is based on the Algol 5 compiler. It was designed by Jørn Jensen, Bo Tveden Jørgensen, Søren Lauesen, and Jørgen Zachariassen. The re-programming of the compiler was made by Jørn Jensen, Bo Tveden Jørgensen, and Jørgen Zachariassen. Hans Rischel and the editor participated in performing certain of the necessary changes in the standard procedures.

The editor wishes to thank Nils Andersen, The Institute of Datalogy, for provoking this revision of the manual, and at the same time pointing out some important spots needing re-consideration. He also wants to thank Søren Lauesen, E. Johansson, and Kirsten Andersen for their great help - each one in his line - during the work with this new edition.

A/S Regnecentralen, August 1974

Hans Dinsen Hansen.

References:

Ref. 1: P. Brinch Hansen: Multiprogramming System.
RCSL 55-D140, A/S Regnecentralen, Copenhagen.

Ref. 2: Søren Lauesen: File Processor, Users Manual.
RCSL 55-D21, A/S Regnecentralen, Copenhagen.

Ref. 3: J.W. Backus, et.al., Revised Report on the Algorithmic Language Algol 60 (ed. Peter Naur), Regnecentralen, Copenhagen (1962); Comm. ACM 6 no. 1 (1963), pp 1-17.

Ref. 4: P. Brinch Hansen: RC 4000 Reference Manual.
RCSL 55-D1, A/S Regnecentralen, Copenhagen.

Ref. 5: P. Lindblad Andersen: Monitor 3.
RCSL 31-D300, A/S Regnecentralen, Copenhagen.

Ref. 6: Hans Rischel: Utility programs, Part 1:3.
RCSL 31-D106, 31-D233, 31-D320, A/S Regnecentralen, Copenhagen.

Ref. 7: Søren Lauesen: Boss 2, Users Manual.
RCSL 31-D310, A/S Regnecentralen, Copenhagen.

Ref. 8: Kirsten Mossin: External processes.
RCSL 31-D37, A/S Regnecentralen, Copenhagen.

Ref. 9: Jens Hald and Allan Wessel: RC 4000 Fortran.
RCSL 31-D103, A/S Regnecentralen, Copenhagen.

Ref. 10: Tom Sandvang: Code procedures and the run time organisation of algol programs.
RCSL 31-D199, A/S Regnecentralen, Copenhagen.

2. BASIC SYMBOLS, IDENTIFIERS, NUMBERS, AND STRINGS

2.0.1. Character set and coding

The source text to the algol compiler must be represented in the ISO 7-bit character code. At run time, the program may choose any alphabet, but the ISO 7-bit code is offered as a standard. It is possible in a simple way to use paper tapes in flexowriter code as source and data, because the monitor may convert the code to ISO 7-bit code (see ref. 1 and 2).

The table below shows for each character of the ISO 7-bit alphabet: the internal value (V), the graphic representation or the name of the character (G), the character class as source to the translator (S), and the character class as data read with the standard alphabet (D).

V	G	S	D	V	G	S	D	V	G	S	D	V	G	S	D
0	NUL	blind	0	32	SP	basic	7	64	@	graphic	7	96	`	graphic	7
1	SOH	illegal	7	33	!	basic	7	65	A	basic	6	97	a	basic	6
2	STX	illegal	7	34	"	graphic	7	66	B	basic	6	98	b	basic	6
3	ETX	illegal	7	35	£	graphic	7	67	C	basic	6	99	c	basic	6
4	EOT	illegal	7	36	$	graphic	7	68	D	basic	6	100	d	basic	6
5	ENQ	illegal	7	37	%	graphic	7	69	E	basic	6	101	e	basic	6
6	ACK	illegal	7	38	&	basic	7	70	F	basic	6	102	f	basic	6
7	BEL	illegal	7	39	'	basic	5	71	G	basic	6	103	g	basic	6
8	BS	illegal	7	40	(	basic	7	72	H	basic	6	104	h	basic	6
9	HT	illegal	7	41	)	basic	7	73	I	basic	6	105	i	basic	6
10	NL	basic	8	42	*	basic	7	74	J	basic	6	106	j	basic	6
11	VT	illegal	7	43	+	basic	3	75	K	basic	6	107	k	basic	6
12	FF	basic	8	44	,	basic	7	76	L	basic	6	108	l	basic	6
13	CR	blind	0	45	-	basic	3	77	M	basic	6	109	m	basic	6
14	SO	illegal	7	46	.	basic	4	78	N	basic	6	110	n	basic	6
15	SI	illegal	7	47	/	basic	7	79	O	basic	6	111	o	basic	6
16	DLE	illegal	7	48	0	basic	2	80	P	basic	6	112	p	basic	6
17	DC1	illegal	7	49	1	basic	2	81	Q	basic	6	113	q	basic	6
18	DC2	illegal	7	50	2	basic	2	82	R	basic	6	114	r	basic	6
19	DC3	illegal	7	51	3	basic	2	83	S	basic	6	115	s	basic	6
20	DC4	illegal	7	52	4	basic	2	84	T	basic	6	116	t	basic	6
21	NAK	illegal	7	53	5	basic	2	85	U	basic	6	117	u	basic	6
22	SYN	illegal	7	54	6	basic	2	86	V	basic	6	118	v	basic	6
23	ETB	illegal	7	55	7	basic	2	87	W	basic	6	119	w	basic	6
24	CAN	illegal	7	56	8	basic	2	88	X	basic	6	120	x	basic	6
25	EM	basic	8	57	9	basic	2	89	Y	basic	6	121	y	basic	6
26	SUB	illegal	7	58	:	basic	7	90	Z	basic	6	122	z	basic	6
27	ESC	illegal	7	59	;	basic	7	91	Æ	basic	6	123	æ	basic	6
28	FS	illegal	7	60	<	basic	7	92	Ø	basic	6	124	ø	basic	6
29	GS	illegal	7	61	=	basic	7	93	Å	basic	6	125	å	basic	6
30	RS	illegal	7	62	>	basic	7	94	^	graphic	7	126	‾	graphic	7
31	US	illegal	7	63	?	graphic	7	95	_	in text	7	127	DEL	blind	0

D,Data classes

0,blind:	The character is skipped by all read procedures.
1,shift character:	Not used in the standard alphabet (see 9.32).
2,digits:	May be used as digits in a number or in a text string.
3,signs:	May be used as the sign of a number or in a text string.
4,decimal point:	May be used as the decimal point of a number or in a text string.
5,exponent mark:	May be used as the exponent mark of a number or in a text string.
6,letters:	May be used as part of a text string. Will terminate a number.
7,delimiters:	Will terminate a number or a text string.
8,terminator:	Works as class 7, but terminates a call of readall (9.52). EM (25) will immediately terminate a call of read (9.51) or readstring (9.54).

S,Source text classes

Basic:	Significant in all contexts.
Blind:	Skipped in all contexts.
Graphic:	Significant inside text strings, causes a warning outside.
Illegal:	Produces a warning during the translation, but does not harm.
In text:	Works as a space inside text strings,blind outside.

Control characters

The control characters which are used in algol are the following:

10,NL:	New Line. The change-to-new-line character.
12,FF:	Form Feed. Causes a change of page on the printer, but works syntactically as New Line outside text strings.
25,EM:	End Medium. See 2.0.3.
32,SP:	Space.
127,DEL:	Delete. Used for overpunching of wrong characters.

2.0.2. Source text

The program consists either of one block, of one compound statement, or of one procedure declaration surrounded by 'external' and 'end' (see 9.20).

All characters up to the first 'begin' or 'external' are skipped, but appear in a possible listing.

After the last 'end', the compiler reads as many characters as are necessary to distinguish the 'end' (usually a space or a new line).

2.0.3. Source files

The source text to the compiler consists of one or more files of text as specified in the File Processor command that started the translation (see app. B). The compiler may read source files from paper tape, cards, typewriter, magnetic tape, and backing storage.

A file terminates either when an EM-character is read from the file or when the file physically is exhausted. A file on a roll of paper tape is exhausted when the tape end is met. A file on the backing storage is exhausted when the end of the backing storage area is met. A file on magnetic tape is exhausted when a tape mark is met.

When the compiler meets the file termination before the source text is complete, it looks for the next file specified in the File Processor command and continues reading from that file. If the list of files is exhausted, the compiler prints an error message, generates the necessary number of string terminations and 'end's, and compiles the program completed in this way.

The compiler handles the peripheral devices in accordance with the rules of the File Processor (ref. 2 or 6).

2.0.4. Space and New Line

Space and New Line may be used freely in numbers and between identifiers, compound symbols, and other delimiters. They are not, however, allowed inside identifiers, compound symbols, or delimiters.

Space and New Line are significant characters in a text string and will be printed out at run time when the string is printed.

The character '_' represents a space inside strings, but is completely blind outside. The latter property may be used to divide identifiers and compound symbols (cf. 2.3).

2.1. Letters

The set of small and capital letters of the reference language is extended with the Danish letters

æ ø å Æ Ø Å

2.2.1. Logical values

Logical values are written as compound symbols without underlining: true false (cf. 2.3).

2.3. Delimiters

The underlined delimiters (compound symbols) of the reference language are written without underlining. A Space or a New Line is required to separate a compound symbol from a preceding identifier or a succeeding letter or digit. Thus the delimiter space is forbidden inside a delimiter, but the symbol '_' may be used instead. The delimiters 'goto' and 'boolean' may not be written as 'go to' and 'Boolean'. Algol 6 adds the following delimiters to the reference:

abs	entier	field	of	zone
add	extend	long	or	
and	external	message	round	
case	extract	mod	shift	

Other delimiters differing from the reference are shown in the following table:

Algol 60	Algol 6	Algol 60	Algol 6	Algol 60	Algol 6
×	*	≡	==	$_{10}$	'
÷	//	$\supset$	=>	␣	_
↑	**	∨	! or *)	[ ]	( )
≤	<=	∧	& and *)	‘ ’	<::> +)
≥	>=	¬	-,		<<> +)
≠	<>				

*) 2 alternative representations are allowed.
+) The first is used for text strings, the second for layout strings.

The delimiter 'message' is syntactically equivalent with 'comment', but it may cause a listing of the comment at translation time (see app. B).
The delimiter '.' (point) is used to denote a field reference as well as a decimal point (see 3.1.1).

2.4. Identifiers

Space and New Line are not allowed inside an identifier, but the symbol '_' may be used.
The words for compound symbols (see 2.2.1. and 2.3) can never be used as identifiers.

Examples:

goto go_to	Both are interpreted as the delimiter 'goto'.
go to	An erroneous construction consisting of two identifiers.
13do a7:=	The number 13, the delimiter 'do', the identifier a7, and the delimiter :=
begin of_line : =	An erroneous construction consisting of the delimiter 'begin', the identifier 'of-line', the delimiter : and the delimiter =

2.4.3.

Algol 6 adds field variables, zones and zone arrays.

2.5. NUMBERS

Algol 6 numbers differ from Algol 60 in distinguishing between two types of integers and that the number range is limited.

2.5.4. Types

Integers are either of type integer or of type long, depending on the value. All other representable numbers are of type real.

2.5.5. Integer and long literals

Integers and longs may not exceed the interval

-140 737 488 355 327 <= integer <= 140 737 488 355 327.

If the literal is within the interval

-8 388 607 <= integer <= 8 388 607

it is classified as being of type integer. Outside this interval it is classified as being of type long (cf. section 3.3.4).

2.5.6. Real literals

The real may not have more than 14 significant digits or 14 decimals. The exponent part may not exceed the interval -1000 < exponent < 1000. The total number is confined to the range -1.6'616 ≤ number ≤ 1.6'616.

The number is converted to internal binary form using the same methods as the procedures read and readall. The relative error of the result is about 3'-11.

2.6. Strings

<string literal> ::= <text string>|<layout string>
<text string> ::=
<:<any sequence of text symbols not containing ':>' or '<:'>:>

Layout strings are described in 9.78, write. A text symbol is a character belonging to the classes basic, graphic, or in text (see 2.0.1) or it is a positive integer of at most 3 digits enclosed in < >. The latter construction has precedence over the character by character interpretation, and represents the character with the integer as internal value. The value must obey 0 < value < 128. Notice that 'nested' strings are not allowed. The general string concept is described in 3.6.

Examples

<:a<b <99>>d:> will be printed by a running program as
a<b c>d

<<_-d.ddd'+d> is a layout string.

2.7. Quantities, kinds and scopes

Algol 6 adds three kinds of quantities: zone, zone array, and field variable.

2.8. Values and types

The value of a zone is a set of values called the zone descriptor, plus a set of values in the zone buffer area, plus a set of values called the share descriptors (see 5.5).

The value of a zone array is the set of values of the corresponding subscripted zones.

Algol 6 distinguishes between 4 types: integer, long, real and boolean.

A field variable possesses an integer value, but has an associated type denoting the type of a field. A field is either a variable field or an array field. Fields are subsets of arrays or zones. Variable fields possess a single value. Array fields are one dimensional arrays.

3. EXPRESSIONS

Algol 6 adds string expressions and zone expressions to the reference language. The full definition becomes:

```
<expression> ::= <arithmetic expression>|<boolean expression>|
                 <designational expression>|
                 <string expression>|<zone expression>
```

3.1. Variables and fields

Algol 6 adds record variables, field variables, and fields to the reference language

3.1.1. Syntax

Note that [ ] is replaced by ( ).
The full syntax becomes:

```
<variable identifier> ::= <identifier>
<simple variable> ::= <variable identifier>
<simple field variable> ::= <identifier>
<array field variable> ::= <identifier>
<field variable> ::= <simple field variable>|
                     <array field variable>
<subscript expression> ::= <arithmetic expression>
<subscript list> ::= <subscript expression>|
                     <subscript list>,<subscript expression>
<array identifier> ::= <identifier>
<zone identifier> ::= <identifier>
<zone array identifier> ::= <identifier>
<zone expression> ::= <zone identifier>|
     <zone array identifier>(<subscript expression>)
<field base> ::= <array identifier>|<zone expression>|
                 <array field>
<array field> ::= <field base>.<array field variable>
<variable field> ::= <field base>.<simple field variable>
<record variable> ::= <zone identifier>(<subscript expression>)|
     <zone array identifier>(<subscript expression>,
     <subscript expression>)
<subscripted variable> ::= <array identifier>(<subscript list>)|
     <array field>(<subscript expression>)
<variable> ::= <simple variable>|<variable field>|
     <subscripted variable>|<record variable>|<field variable>
<field reference> ::= <array field>|<variable field>
<field> ::= <field reference>
```

3.1.2. Examples

See 9.22 with subsections and 9.79.

3.1.3. Semantics (of record variables and fields)

Record variables designate values which are components of zone buffer areas. The subscript expressions are evaluated like subscripts of ordinary subscripted variables.

In case of a zone array with subscripts, the first subscript expression selects a zone from the zone array. This subscript must obey

1 $\le$ subscript $\le$ number of zones declared in the array.

The last subscript selects a variable within the zone record, which in turn is a set of consecutive variables of the selected zone buffer area. This subscript must obey

1 $\le$ subscript $\le$ number of variables currently in the record.

When an expression is assigned to a record variable, the location (see 4.2.3) of the selected buffer element is not influenced by possible changes of the record caused by procedure calls in the right hand expression.

Fields are subsets of arrays or zones. A field consists of a number of bytes (see 3.1.6) located within an array or a zone. The type of a field is defined in the declaration of the field variable (cf. section 5.7. Field declarations).

3.1.4. Subscripts (to array fields)

3.1.4.3.

An array field is always considered one dimensional. The ordering of the bytes in the field base and in the array field follows the lexicographical ordering (cf. 5.2.6. Lexicographical ordering). The subscript bounds are defined by means of the byte bounds (cf. 5.2.7). The byte bounds for the array field are obtained by subtracting the value of the array field variable from the byte bounds of the field base (possibly an array field). An element must be located within the field base.

3.1.5. Type of record variables and field variables

A record variable is of type real.
A field variable is of type integer.

3.1.6. Ranges of values. Type length. Binary patterns

Depending on the type, each variable is represented by an integral number of bytes. Each byte is of 12 bits. The number of bytes representing a variable is called the type length. The type length may some times be expressed in bits.

3.1.6.1. Booleans are represented as 12 bits quantities. The type length of a boolean variable is 1 byte. The binary pattern of a boolean is extended with zeroes to the left whenever needed. The last of the 12 bits is 0 when the boolean is false, 1 when it is true.

The logical constants 'true' and 'false' and the result of applying the relational operators will always be 12 zeroes for false, 12 ones for true. Other binary patterns may be obtained by applying the operators add and shift. The 5 logical operators work on all 12 bits in parallel.

3.1.6.2. Integers are represented in 24-bits, 2's complement, binary form. This gives the range:

-8 388 608 <= integer <= 8 388 607.

The type length of an integer variable is 2 bytes, and the binary pattern of an integer is the 24 bits of its representation extended with zeroes to the left whenever needed. The binary patterns are used in connection with the operators add, extract, and shift.

3.1.6.3. Longs are represented in 48 bits, 2's complement, binary form. The range of longs should be confined to:

-140 737 488 355 327 <= long <= 140 737 488 355 327.

The type length of a long variable is 4 bytes, and the binary pattern of a long is the 48 bits of its representation. The binary patterns are used in connection with the operators add, extract, and shift.

3.1.6.4. Reals are represented as 48-bits built-in floating point numbers. This gives the following range of non-zero real values:

    1.6'-617 < abs(real) < 1.6'616

The precision of real values correspond to 35 significant bits. Thus one unit added to the last binary place will correspond to a relative change of the number of between 6'-11 and 3'-11.

The type length of a real variable is 4 bytes. The 3 first bytes are used for the number part and the last byte for the exponent part of the real.

The binary pattern of a real consists of a 36-bits, 2's complement, number part followed by a 12-bits, 2's complement, exponent part so that the real value is:

    number*2**exponent.

The number is either 0 or in the range -1 ≤ number < -0.5, 0.5 ≤ number < 1. The exponent is in the range -2048 ≤ exponent ≤ 2047. The exponent of 0.0 is -2048, but other exponents might be obtained by the operator 'add'.

If r is a floating point zero with an exponent <> -2048, the relation r = 0 will be false because the operands are compared bit by bit. The relations r ≤ 0 or r ≥ 0 will both be true, however. Operations like r + b cannot be expected to give b (see ref. 4).

3.1.7. Reals used as semi-long integers

As there is neither built-in long multiplication nor built-in long division, programs using many of these operations on large integers may be speeded up a little by representing them as real variables.
This can be done with full accuracy as long as all results are kept in the range

    -2**35 = -34 359 738 368 ≤ real ≤ 34 359 738 367 = 2**35 - 1

If the results exceeds this range, the last bits of the semi-long integer are lost.

A kind of integer division may be obtained by a real division followed by a cut-off of decimals caused by the addition of a large constant. For results in the range 0 ≤ result ≤ 2**34, this is done as follows:

```
roundconstant:= 2**34;
result:= r1/r2 + roundconstant - roundconstant;
```

Safety against loss of accuracy may be obtained by scaling the semi-long integers so that loss of accuracy will cause a floating point overflow. The scale factor f is chosen so that f*2**35 = 2**2048 and f*(-2**35) = -2**2048. This is fulfilled by f = 2**2013. Addition and multiplication with check for loss of accuracy may be performed like this:

```
r1:= i1*f;       r2:= i2*f;
r1 + r2          r1/f*r2
```

3.1.8. Fields

Fields are subsets of arrays and zone records. A field variable is a pointer indicating a field within an array, a zone record, or an array field. The type of the field depends only of a type declared together with the field variable (cf. section 5.7. Field declarations). All the bytes of a variable field must be located within the field base.

3.2. Function designators

3.2.1. Syntax

The syntax is changed slightly. See 4.7, procedure statements.

3.2.4. Standard functions

The standard functions abs and entier are replaced by operators of the same name (see 3.3). This implies that variables with the name abs or entier cannot be declared, and that abs or entier cannot be used as an actual parameter specified as real procedure or integer procedure.

The numerical standard functions of algol 6 are listed below. They are described in detail in chapter 9.

arcsin	cos	random	sin
arctan	exp	sgn	sinh
arg	ln	sign	sqrt

3.2.5. Transfer functions

Entier is replaced by an operator (see 3.2.4), and the operators round, extract, extend, add, real, long, and string take care of other type transfers.

3.3. Arithmetic expressions

3.3.1. Syntax

Algol 6 adds the operators mod, shift, add, extract, abs, entier, round, real, long, and case to the reference language. The full syntax becomes:

```
<adding operator> ::= +|-
<multiplying operator> ::= *|/|//|mod
<pattern operator> ::= shift|add|extract
<monadic operator> ::= abs|entier|round|extend|real|long
<primary> ::= <unsigned number>|<variable>|<function designator>|
      (<arithmetic expression>)|
      <monadic operator><primary>|real <string primary>|
      long <string primary>
<factor> ::= <primary>|<factor>**<primary>|
      <factor><pattern operator><primary>|
      <boolean basic> extract <primary>
<term> ::= <factor>|<term><multiplying operator><factor>
<simple arithmetic expression> ::= <term>|
      <adding operator><term>|
      <simple arithmetic expression><adding operator><term>
<if clause> ::= if <boolean expression> then
<case clause> ::= case <arithmetic expression> of
<arithmetic expression list> ::= <arithmetic expression>|
      <arithmetic expression list>,<arithmetic expression>
<arithmetic expression> ::= <simple arithmetic expression>|
      <if clause><simple arithmetic expression> else
      <arithmetic expression>|
      <case clause>(<arithmetic expression list>)
```

3.3.2. Examples

Primaries:

```
long(if b then <:abc:> else <<dd.d0>)
abs round ra(i)
entier cos(y+z)
```

Factors:

```
round r shift (-6) add j
(a < b) extract 1
```

Arithmetic expressions:

```
case i+j of(i mod j,if b then r**j else i,case i of(j))
if b then (case i of(j,r)) else case i of (1,5)
```

3.3.3. Semantics

The semantics of the new operators are given in chapter 9. Field variables used outside field references are handled as variables of type integer.

3.3.4. Operators and types

The types of the new operators are given in chapter 9.

The result of applying ** is always of type real, even if both operands are of type integer or long.

The operators +, -, and * yield an integer value if both the operands are of integer type, a real value if at least one is of real type, and a long value otherwise.

The operator / always yields a real value.

The operators // and mod are defined for two operands of type integer or long. They yield an integer value if both operands are integer and a long value otherwise.

The result of

<if clause><simple arithmetic expression> else
<arithmetic expression>

is of type integer if both expressions are of type integer, of type real if at least one expression is of type real, and of type long otherwise.

The result of

<case clause>(<arithmetic expression list>)

is of type integer if all expressions in the list are of type integer, of type real if at least one expression is of type real, and of type long otherwise.

3.3.5. Precedence of operators

Function calls in an expression may cause 'side-effects', but the result will correspond to a strict left to right evaluation of the expression, so that side-effects only may influence variables to the right of the function call.

According to the syntax given in section 3.3.1 the following rules of precedence hold:

first:	abs	entier	real	round	long	extend
second:	**	add	extract	shift		
third:	*	/	//	mod		
fourth:	+	-				

3.6.6. Arithmetic of real, long, and integer quantities

The operations + - * / ** (for integer or long exponents) are performed by the built in floating point operations whenever the result is of type real, and by the fixed point operations whenever the result is of type integer. Whenever the result is of type long + and - are performed by the built in double length operation, whereas the operation * is performed by a subroutine.

The operations // and mod are performed by the built in fixed point division whenever the result is of type integer, and by a subroutine whenever the result is of type long.

When necessary integer operands are floated by means of the built in float operation or converted to a long by extension of the sign. Conversion of operands of type long to type real is performed by a subroutine.

The range of values of type real and integer is given in 3.1.6. The action when the range of reals is exceeded, is controlled at run time by means of the two standard integer variables 'overflows' and 'underflows' (see chapter 9). The action when the range of integers or longs is exceeded, is determined at translation time by means of the translation parameter 'spill' (see 5.4 and app. B).

The precision of real arithmetic may be decreased from 36 bits to 33 bits. This option is controlled at run time by means of the procedure 'system'. The results of the numerical standard functions are distorted correspondingly when the low precision is selected.

3.4. Boolean expressions

3.4.1. Syntax

Algol 5 adds the operators case, add, and shift to the reference language. The full syntax becomes:

```
<relational operator> ::= <|<=|=|>=|>|<>
<and> ::= and|&
<or> ::= or|!
<relation> ::= <simple arithmetic expression>
     <relational operator><simple arithmetic expression>
<boolean pattern operator> ::= add|shift
<boolean basic> ::= <logical value>|<variable>|
     <function designator>|
     <boolean basic><boolean pattern operator><primary>|
     (<boolean expression>)
<boolean primary> ::= <boolean basic>|<relation>
<boolean secondary> ::= <boolean primary>|
     -,<boolean primary>
<boolean factor> ::= <boolean secondary>|
     <boolean factor><and><boolean secondary>
<boolean term> ::= <boolean factor>|
     <boolean term><or><boolean factor>
<implication> ::= <boolean term>|
     <implication>=><boolean term>
<simple boolean> ::= <implication>|
     <simple boolean>==<implication>
<boolean expression list> ::= <boolean expression>|
     <boolean expression list>,<boolean expression>
<boolean expression> ::= <simple boolean>|
     <if clause><simple boolean> else <boolean expression>|
     <case clause>(<boolean expression list>)
```

3.4.2. Example

```
if b add 1 shift 3 then (case i of(true,b or c) else
case j of((u=v) shift 1,false)
```

3.4.3. Semantics

The semantics of the new operators are given in chapter 9.

3.4.4. Types

The types of the new operators are given in chapter 9.

3.4.6. Precedence of operators

The priority of add and extract is the priority of **, i.e. higher than the relational operators.

3.4.7. Arithmetic of boolean quantities

The representation of booleans and some rules for boolean arithmetic is given in 3.1.6. Here, we add the rules for the relational operators:

< <= >= >	are in most cases executed as a subtraction (floating point or fixed point) of the two operands. Thus, you must be prepared for overflow, underflow, or spill.
= and <>	are always performed as a bit by bit comparison of the two operands. This may for instance be utilised to compare two text strings packed into real variables without risk of overflow (see example 3 of 9.41).

3.5. Designational expressions COMMONLY KNOWN AS LABELS

3.5.1. Syntax

Integers are not permitted as labels. The designational expressions are extended with case constructions as described in 9.8.

3.5.6. Switch versus case statement

Switches are implemented fully in algol 6, but we recommend the use of case statements (see 9.8) instead of 'goto sw(i)'. Case statements are much faster and may give a clearer program.

3.6. String expressions

3.6.1. Syntax

```
<formal string> ::= <identifier>
<string primary> ::= <formal string>|<string literal>|
     string <arithmetic expression>|(<string expression>)|
     <string primary> add <primary>
<string expression list> ::= <string expression>|
     <string expression list>,<string expression>
<string expression> ::= <string primary>|
     <if clause><string primary> else <string expression>|
     <case clause>(<string expression list>)
```

3.6.2. Examples

```
if b then <:ok:> else <:error:>
case i of (<:first:>,<:second>,string ra(increase(j)))
if b then (case i of (string r,fs))
     else case i of(<:ab:>,<<d.dd>)
```

3.6.3. Semantics

A string expression is a rule for computing a string value. The principles of evaluation are analogous to the evaluation of an arithmetic expression. The semantics of the operator 'string' are given in 9.70 and of 'add' in 9.2.

String expressions are used as actual parameters and as arguments of the operator 'real' (see 9.55).

The value of a string expression is

a short text string	(at most 5 characters, for example <:abcde:>)
a long text string	(a literal text string of more than 5 characters, for example <:result:>),
a layout string	(for example <<dd.dd'+d>), or
a text portion	(6 characters none of which are Nulls. This cannot occur as a literal text, but may be obtained by the operators 'string' or 'add', for example <:abcde:>add 92).

When a standard procedure references a string parameter and obtains a text portion as the result, it will accept these 6 characters as the first part of the string and reference the parameter again and again to obtain the next text portions. When a short or a long text string is obtained, the string end is met. This rule implies that the string parameter must have side-effects to supply new text portions when it is referenced repeatedly. The standard procedure 'increase' assists you with this task as explained in example 2 of 9.70.

3.6.4. Types

The argument of the operator 'string' must be of type real or long. A formal string must be a formal parameter specified as string.

3.6.5. Binary pattern

The binary pattern of a string value is 48 bits with the values given below.

Text portion and short text string

The characters of the text string (omitting the string quotes) are represented as their internal value (see 2.0.1) and packed as 8-bit bytes from left to right. The 48 bits are filled up to the right with zeroes.

Long text string

The last 24 bits contain a one followed by some undefined bits. The first 24 bits contain segm shift 12 add rel. The characters of the text string are stored as text portions on the backing storage area which is occupied by the algol program. The first text portion representing the first 6 characters is found on segment 'segm' word rel//2-1 and rel//2 (the 256 words of a segment are numbered 0, 1, 2, ...). The next text portions are found in word rel//2-3 and rel//2-2 and so on, until 48 bits representing a new long text string are found or until 48 bits representing a short text string are found. The first possibility specifies the continuation of the string on a new segment, the latter possibility signals the string end.

Layout string

The first 24 bits represent the spaces of the layout as follows: First, a 1 followed by a 1 for each leading space of the layout. Second, one 0. The following bits correspond to the digit positions of the number part (z, f, d, and 0). A bit is 1 if the corresponding digit position is followed by a space, otherwise 0.

The last 24 bits contain:

```
bit 0      0
bit  1- 5 b = number of significant digits (z, b, f, and d).
bit  6- 9 h = number of digit positions before the point.
bit 10-13 d = number of digit positions after the point.
bit 14-15 pn= first letter of number part (z=10, f=01, d=00, b=11).
bit 16-17 fn= sign of number part (+ =10, - =01, no sign = 00).
bit 18-19 s = number of digits in exponent.
bit 20-21 pe= first letter of exponent part (z=10, f=01, d=00).
bit 22-23 fe= sign of exponent part coded as fn.
```

3.7. Zone expressions

3.7.1. Syntax

```
<zone identifier> ::= <identifier>
<zone array identifier> ::= <identifier>
<zone expression> ::= <zone identifier>|
    <zone array identifier>(<arithmetic expression>)
```

3.7.2. Examples

in polyfase(output) polyfase(input(i))

3.7.3. Semantics

The value of a zone expression is a zone. Zone expressions are used as actual parameters.

The arithmetic expression is evaluated as a subscript expression. It selects a zone from the zone array. The subscript must obey

$1 \leq$ subscript $\leq$ number of zones declared in the array.

4. STATEMENTS

4.1.1. Syntax

Algol 6 adds the case statements described in 9.8. The definition of a statement becomes:

<statement> ::= <unconditional statement>|
<conditional statement>|<for statement>|
<case statement>

A procedure may be translated alone, and everything until the first begin or external is skipped, so the definition of a program becomes:

<program> ::= <block>|<unlabelled compound>|
external <procedure declaration>; end

4.2. Assignment statements

4.2.3. Semantics

Note that variables are extended with record variables, variable fields and field variables (see 3.1). The location of a zone buffer element designated by a record variable is not influenced by expressions to the right of the record variable, even if these change the position of the record within the zone buffer. Note the reformulation of 4.2.3.1 and 4.2.3.3. The location of a variable is an absolute address in the RC 4000.

4.2.3.1. The locations of all variables, including subscripted variables, record variables, and variable fields, occurring in the left part are evaluated from left to right.

(4.2.3.2. The expression of the statement is evaluated.)

4.2.3.3. The value of the expression is assigned to all the left part variables with locations as evaluated in step 4.2.3.1 in sequence from right to left.

4.2.4. Types

Field variables may be used as variables of type integer.

Long is considered as a new arithmetic type. Conversion procedures exist between all three types. The conversion of a real value to an integer or long and the conversion from a long value to an integer are performed so that spill alarm (see appendix B) may occur.

4.6. For statements

4.6.1. Syntax

Only a simple variable or a field variable can occur as the controlled variable of a for statement.

4.6.4.2. Step-until-element

In the following algorithm, localB is an anonymous variable, while A, B, and C represent the expressions of A step B until C. V is the controlled variable. The step-until-element is executed in this way:

```
        V:= A; localB:= B;
    L1: if (V-C)*localB > 0 then goto Element_exhausted;
        Statement S;
        localB:= B; V:= V + localB;
        goto L1;
```

4.6.5. The value of the controlled variable upon exit

Upon exit from a for statement, the value of the controlled variable is defined by the algorithm in 4.6.4.2 above and in 4.6.4.1 and 4.6.4.3 of the Revised Report (ref. 3).

4.6.6. Goto leading into a for statement

Any occurrence outside a for statement of a label which labels a statement inside the for statement is forbidden.

4.7. Procedure statements

4.7.1. Syntax

The expressions of algol 6 include string expressions, variable fields, and zone expressions, which may occur as actual parameters. An actual parameter is:

```
<actual parameter> ::= <zone array identifier>|
        <expression>|<array identifier>|<array field>
        <switch identifier>|<procedure identifier>
```

The 'fat comma' defined by)<letter string>:(may not contain compound symbols.

4.7.2. Examples

```
clear(a)begin_of clearing:(i) end_of clearing:(j)
```

4.7.3. Semantics of zone expressions, array fields and field variables

The zone of a zone expression is always evaluated before the procedure is entered.

An array field is evaluated before the procedure is entered. The evaluation is made like this:

a) The bound bytes are computed as shown in section 5.7.5.
b) The lower bound byte is adjusted relative to the value found above. The adjustment is made as follows:

```
lower_bound_byte:= (lower_bound_byte - 1)//type_length*
     type_length + if lower_bound_byte <= 1 then 1
     else typelength + 1
```

c) A description of a one-dimensional array of the resulting type and with these bound bytes is set up local to the procedure.
   If the procedure uses this array as an actual array field parameter in subsequent procedure calls, this cutting may be performed again. Thus, from a certain step, the bytes of an array may be unaccessible from the procedures, if the values of the array field variables are not chosen appropriately.

A parameter specified as a field variable may correspond to an actual parameter of type integer. A field variable as an actual parameter behaves as a variable of type integer.

4.7.5. Restrictions

A value parameter specified as type integer or type real may correspond to an actual parameter of type real or integer. Value arrays and value labels are not allowed (see 5.4).

A formal parameter specified as real array may actually be a zone expression. In this case, the array elements are that part of the zone buffer which is selected as the zone record at the moment of the call.

In all other cases, the compiler requires a strict agreement between specification and actual kind and type (see however 4.7.5.3). All parameters in algol procedures must be specified.

4.7.5.2.

A formal name parameter which occurs as a left part variable in an assignment statement within the procedure, may actually be an expression which is not a variable (a constant for instance). In this case, the assignment takes place to a fictious variable. If such an actual parameter is a constant, the future value will be taken from this fictious variable, and if it is an expression, the assignment disappears into thin air.

4.7.5.3.

An actual parameter which is an array identifier can only correspond to a formal array parameter with the same number of subscripts or with one subscript. In the latter case, the lexicographical ordering of the array elements is used as explained in 5.2. An array field is considered as a one-dimensional array (see 4.7.3).

4.7.9. Recursive procedures

Recursive procedures are handled fully in algol 6, note however the possible 'cutting' of array parameters which are actually array fields. If a variable is declared 'own' in a procedure body and the procedure is called recursively, the same own variable is used in all the dynamic incarnations of the procedure. An application of this is shown in example 3 of 6.3.4.

5. DECLARATIONS

Algol 6 adds the declarations of field variables, long variables and arrays, zones, and zone arrays. All programs may be thought of as surrounded by one common block (the standard identifier block). The declarations of this block are given in the backing storage catalog of the RC 4000. New procedure declarations are inserted in this block when external procedures are translated (see 9.20). Procedures expressed in machine language, simple variables, and zones may be inserted in the standard identifier block as described in ref. 10.

Initial values, owns

Only simple variables may be declared own in Algol 6. Own booleans are initially false, own integers are initially 0, and the binary pattern of own reals and longs is initially 0. All other variables have undefined contents just after their declaration (for zones, see 9.79).

For owns and recursive procedures, see 4.7.9.

5.1. Type declarations

5.1.1. Syntax

A new type, long, is introduced. The syntax for type becomes:

<type> ::= real|long|integer|boolean

5.1.3. Semantics

The range and representation of variables are given in 3.1.

In arithmetic expressions, any position which can be occupied by an integer or a real declared variable may be occupied by a long declared variable.

5.2. Array declarations

5.2.1. Syntax

Own arrays are not allowed. Note that long is a new type.

5.2.4. Lower and upper bounds

Note that at least one element must be declared in an array and that all identifiers in the bounds must be non-local.

5.2.6. Lexicographical ordering

The elements of an array are stored in a sequence, and a multi-dimensional array declared

Am(low:up1,low2:up2,...,lown:upn)

may in certain connections (specified in 5.2.6.1 and 5.2.6.2) be considered as a one-dimensional array

Ao(low:up).

Whenever the mapping of Am on Ao makes sense, the elemment

Am(i1,i2,...,in)

may be found as

```
Ao(...((i1*c2+i2)*c3+i3)*...+in)
```

where

```
c2 = up2 - low2 + 1, c3 = up3 - low3 + 1, and so on.
```

This mapping of the elements is called the lexicographical ordering because it is a linear ordering of the elements obtained by varying the first subscripts at the slowest rate.

The values of low and up may be seen to be:

```
low = ...((l1*c2 + l2)*c3 + l3)* ... + ln
up  = ...((u1*c2 + u2)*c3 + u3)* ... + un
```

It may also be seen that the (possibly fictive) element Am(0,0,...,0) is the same as Ao(0).

5.2.6.1. Multi-dimensional array as actual parameter

A multi-dimensional array may occur as an actual parameter where the corresponding formal is a one dimensional array. The mapping above is used in that case.

5.2.6.2. Multi-dimensional array as field base

Whenever a multi-dimensional array is used in a field reference as the (ultimate) field base, the byte numbering and addressing described in 5.2.7 and 5.2.8 is found by mapping the multi-dimensional field base on a one-dimensional field base according to the rules above.

5.2.7. Bound bytes and byte numbering

Each element of an array is represented by a number of bytes. This number is the type length explained in section 3.1.6.

The first byte in an array is called the lower bound byte and the last one the upper bound byte. Let an array be declared

```
A(low:up)
```

then

```
lower bound byte = (low - 1)*type length + 1
upper bound byte = up*type length.
```

The bytes of an array are numbered relative to the rightmost byte in the (possibly fictive) element A(0). The element A(i) contains the bytes

```
(i - 1)*type length + 1 <= byte number <= i* type length.
```

5.2.8. Word boundaries and addresses

When an array is declared, it is created so that the word boundaries are between an even numbered byte and its odd numbered successor.

An array element, A(i), is addressed within the array by the byte with the number i*type length.

5.4. Procedure declarations

5.4.1. Syntax

The set of possible specifiers is extended so that longs, field variables, zones and zone arrays, may be specified and the syntax for specifier becomes:

```
<specifier> ::= string|<type>|array|<type> array|label|switch|
                <procedure>|<type> procedure|<type> field|
                array field|<type> array field|zone|zone array
```

Note that long is a new type.

5.4.5. Specifications

All parameters must be specified. Only parameters specified real, long, integer, or boolean may occur in the value part.

An actual field variable may correspond to a formal integer and vice versa.

An actual array field may correspond to a formal array of the same type.

An actual zone may correspond to a formal real array.

An actual real may correspond to a formal integer value and an actual integer may correspond to a formal real value.

Except for these possibilities, the kind of an actual parameter must correspond exactly to the kind and type of the specification.

5.4.6. Code as procedure body

Procedures may be expressed in machine language and introduced into the standard identifier block (see introduction to chapter 5) as it is explained in ref. 10.

Algol procedures may be translated alone (see 9.20).

5.5. Zone declarations

5.5.1. Syntax

```
<length> ::= <arithmetic expression>
<shares> ::= <arithmetic expression>
<block proc> ::= <procedure identifier>
<zone segment> ::= <zone identifier>(<length>,<shares>,<block proc>)|
                   <zone identifier>,<zone segment>
<zone list> ::= <zone segment>|<zone list>,<zone segment>
<zone declaration> ::= zone <zone list>
```

5.5.2. Examples.

```
zone master(2*bl,2,stderror)
zone m1,m2(a,b,c),m3(900,3,pr)
```

5.5.3. Semantics

A zone declaration declares one or several identifiers to represent zones. The arithmetic expressions in the declaration are evaluated once for each entrance into the block. Each zone consists of:

a buffer area
a zone descriptor
one or more share descriptors (often just called shares)

Inside the block, a zone identifier may occur as an actual parameter, as a constituent of a record variable, or as a field base (cf. 3.1).

Buffer area

The length of the buffer area for any zone is given by <length> in the first parenthesis following the zone identifier.

Each element of the buffer area may be used as a real variable as explained for zone record below. The elements are in some connections identified by a byte number in the range 1 <= byte number <= 4*length.

Zone descriptor

A zone descriptor consists of the following set of quantities, which specify a process or a document (see ref. 1) connected to the zone and the state of this process:

process name	A text string specifying the name of a process or a document.
mode and kind	An integer specifying mode and kind for a document (see 9.41, open).
logical position	A set of integers specifying the current position of a document.
give up	An integer specifying the conditions under which <block proc> is to be called.
state	An integer specifying the latest operation on the zone.
record	Two integers specifying the part of the buffer area nominated as the zone record.
used share	An integer specifying a share descriptor within the zone.
last byte	An integer specifying the end of a physical block on a document.
block procedure	The procedure <block proc> in the first parenthesis following the zone identifier.

The normal use of these quantities is explained in details in chapter 6.

Share descriptor

Each zone contains the number of share descriptors given by <shares> in the first parenthesis following the zone identifier. The share descriptors are numbered 1, 2, ..., <shares>.

A share descriptor consists of a set of quantities which describe an external activity sharing a part of the buffer area with the running program. An activity may be a parallel process transferring data between a document and the buffer area, or it may be a child process executed in the buffer area under supervisory control of the algol program. Section 6.4 explains these possibilities.

The set of quantities forming one share descriptor is:

share state	An integer describing the kind of activity going on in the shared area.
shared area	Two integers specifying the part of the buffer area shared with another process by means of the share descriptor.
operation	Specifies the latest operation performed by means of the share descriptor.

Zone record

A number of consecutive bytes of the buffer area may at run time be nominated as the zone record. The bytes of the zone record may be available as record variables, which may be thought of as a kind of real subscripted variables. The record variables are numbered 1, 2, ..., <record length> and referenced as described in 3.1. All bytes of the record may be referenced by means of field references, as the zone may be used as a field base.

5.5.4. Types

The two expressions <length> and <shares> must be of type integer. The procedure <block proc> must be declared like this:

```
procedure <block proc> (z,s,b); zone z; integer s,b;
```

5.5.5. Scope

All identifiers occurring in <length> and <shares> must be non-local to the block. However, <bloc proc> may also be local.

At the time of exit from the block (through end, or by a goto statement), the activities described by the share descriptors are terminated as follows: A communication with a parallel process is completed by means of the monitor function wait answer (see ref. 1). A running child process is stopped (but not removed, see ref. 1).

5.5.6. Standard zones

Two zones, 'in' and 'out', are available without declarations. They are described in 9.28 and 9.42.

5.6. Zone array declarations

5.6.1. Syntax

```
<zones> ::= <arithmetic expression>
<length> ::= <arithmetic expression>
<shares> ::= <arithmetic expression>
<block proc> ::= <procedure identifier>
<zone array list> ::= <zone array list>,<zone array list>|
    <zone array identifier>(<zones>,<length>,<shares>,
    <block proc>)
<zone array declaration> ::= zone array <zone array list>
```

5.6.2. Examples

```
zone array inmerge(3,2*600,2,stderror),outmerge(3,2*600,2,stderror)
```

5.6.3. Semantics

A zone array declaration declares one or more identifiers to represent one-dimensional arrays of zones. The arithmetic expressions in the declaration are evaluated once for each entrance into the block. Each zone array consists of as many zones as specified by <zones>. All these zones are declared with <length>, <shares>, and <block proc> as specified (cf. section 5.5). The zones of a zone array are numbered 1, 2, ..., <zones>.

Inside a block, a zone array identifier may occur as an actual parameter, as a constituent of a subscripted zone occurring as a parameter (cf. 3.7), or as a constituent of a record variable (cf. 3.1).

5.6.4. Types

<zones> must be of type integer. See section 5.5.4 for <length>, <shares>, and <block proc>.

5.6.5. Scope

All identifiers occurring in <zones> must be non-local to the block. See section 5.5.5 for <length>, <shares>, <block proc>, and the exit from the block.

5.7. Field declarations

5.7.1. Syntax

```
<field list> ::= <field variable>|<field variable>,<field list>
<variable field declaration> ::= <type> field <field list>
<array field declaration> ::= <type> array field <field list>|
                              array field <field list>
<field declaration> ::= <variable field declaration>|
                        <array field declaration>
```

Note that long is a new type.

5.7.2. Examples

See 9.22 and subsections.

5.7.3. Semantics

A field declaration serves to declare one or several identifiers as field variables. Field variables are integers and may be used whereever an integer variable may be used.

A variable field declaration declares simple field variables and an array field declaration declares array field variables. The type declared together with the field variables, the associated type, has no meaning outside field references.

All field variables declared in one declaration have the same associated type. If no type declarator is given in an array field declaration the type real is understood.

5.7.4. Location of a variable field

A variable field is located within an array, a zone record, or an array field. The denotation of a variable field is shown in section 3.1. The variable field consists of as many bytes as the type length of the associated type shows. A variable field cannot occupy bytes outside the bound bytes (cf. section 5.2.7 and section 3.1.4.3).

The location of a variable field is determined by a byte number equal to the value of the simple field variable. This byte number is used as an address of the field. Boolean fields are addressed by their byte number. Integer, long, and real fields are synchronized with the word boundaries (cf. section 5.2.8) of the RC 4000. Integer fields are addressed by one of the 2 bytes forming the integer word. Long and real fields are addressed by one of the 2 bytes in the right hand word. The address must be >= lower bound byte + type length - 1 and it must be <= upper bound byte.

5.7.5. Location and bounds of an array field

An array field is located within the field base. The byte number referring to a certain piece of data in the array field is found by subtracting the value of the array field variable from the corresponding byte number in the field base. If the field base is an array field, this rule may be used recursively.

The bound byte numbers are given by the formula:

bound byte of array field =
bound byte of field base - value of array field variable.

A subscripted element in an array field is addressed according to the rule in section 5.2.8. The address of a subscripted element must be >= lower bound byte + type length - 1 and it must be <= upper bound byte.

6. INPUT/OUTPUT SYSTEM

This chapter describes the use of zones for input/output and for programming of operating systems. Details of the various procedures are given in section 9.

Let us start with a typical example of output to a peripheral device specified by the algol program:

```
begin zone pr(2*128,2,stderror);
comment declare a zone which will buffer the output. Two
    buffers of 128 elements of 4 bytes each (or 128*6 charac-
    ters) are used here. The procedure stderror is called when
    the device causes trouble;

open(pr,4,<:bs53:>,0);
comment specify the output device, here:
    backing storage area bs53;

write(pr,<:results::>,...);...
comment output the results;

close(pr,true);
comment terminate the output, empty the buffers;

end;
```

Exactly the same scheme would work for character input if write(...) was replaced by read(...). If the device is a magnetic tape, the tape must be positioned before input or output can start. That is done by means of the procedure setposition.

Input and output are buffered in RC 4000. In the example above, this means that 128*6 characters are packed in the zone buffer before they are transferred to the backing storage. If you forget to close the zone, or if the run is terminated with an alarm, the last buffer of characters is never transferred to the device. When you output to the standard zone 'out', the File Processor will take care of printing the last buffer, even if your program is terminated with an alarm.

6.1. Documents

The high level procedures assume that all peripheral devices scan documents. For instance, a document scanned by a paper tape reader is a roll of paper tape, a document scanned by a magnetic tape station is a reel of magnetic tape. The documents are at run time addressed by names appearing as text strings in algol.

A document may be thought of as a string of information, either a string of 8-bit characters or a string of real variables (elements). The string is on some documents broken into physical blocks (e.g. on magnetic tapes and backing storage areas). The procedures for input/output on character level and record level keep track of the current logical position of the document. The logical position points to the boundary between two characters or two elements of the document. During normal sequential use of the document, the logical position moves along the document corresponding to the calls of the input/output procedures.

For documents consisting of physical blocks, the logical position is given by a position within the physical block, plus a block number, plus (for magnetic tapes) a file number. Note that the block number is ambiguous in the case where the logical position points to the boundary between two physical blocks. This ambiguity is resolved explicitly in the description of the individual procedures: The term 'the logical position is just after or just before a certain item' implies that the block number is the block number of that item.

The following sections give a survey of some documents and the way they transfer information to and from the zone buffer. The rules for protection of documents and further details are found in ref. 1, ref. 5, and ref. 8. The kinds mentioned below are explained in section 9.41, open.

6.1.1. Internal process (kind 0)

An internal process (another program executed at the same time as your job) may receive or generate a document. If the process just transmits the information to or from a peripheral device, the rules below for that device will hold for the communication with the document too. The kind specified in 'open' should then be the kind of the document.

The internal process may also handle the information in its own way, and then no general rules can be given, but usually, the end of the document is signalled as explained in section 6.3.3.

6.1.2. Backing storage (kind 4)

The backing storage consists of a drum or a disc or both. You have no direct access to the entire backing storage, but only to documents which are backing storage areas consisting of a number of consecutive segments. Each segment contains 512 bytes (or 128 real variables). The segments are numbered 0, 1, 2, ... within the area, and the block numbers mentioned above are exactly these segments numbers. File numbers are senseless.

One or more segments may be transferred directly as bit patterns to or from the core store in one operation. The number of segments transferred is the maximum number that fits into the share used.

The physical backing storage may be a drum or a disc. Details about the various types of devices may be found in ref. 8.

6.1.3. Typewriter (kind 8)

A typewriter may be used both for input and output. The sequence of characters input forms one document (infinitely long), and the sequence of characters output forms another document. File number and block number are senseless on a typewriter.

One input operation transfers one line of characters (including the terminating New Line character) to the share. If the share is too short, less than a line is transferred, but that is an abnormal situation. The characters are packed in ISO 7-bit form with 3 characters to one word, and last word is filled up with nulls. One output operation transfers characters packed in the same form to the typewriter. Several lines may be output by one operation.

6.1.4. Paper tape reader (kind 10)

A document consists of one roll of paper tape. It may be read in various modes: with even parity, with odd parity, without parity, or with transformation from flexowriter code to ISO code. File number and block number are senseless for a paper tape.

One input operation will usually fill the share with characters packed 3 per word, but fewer characters may also be transferred, for instance at the tape end. In such cases, the last word is filled up with null characters. The characters are not necessarily ISO characters, that depends on the meaning you assign to them.

The RC 2000 tape reader can read about 2000 characters a second.

6.1.5. Paper tape punch (kind 12)

A document is from the programs point of view infinitely long, even when the operator divides the output into more paper tapes. A paper tape may be punched in various modes: with even parity, with odd parity, without parity, or with transformation from ISO code to flexowriter code. File number and block number are senseless for a tape punch.

One output operation may punch any number of characters packed 3 per word. In all modes, except the mode without parity, only the last 7 bits of the characters are output and extended with a parity bit.

The RC 150 tape punch can punch about 150 characters a second.

6.1.6. Line printer (kind 14)

A document is from the programs point of view infinitely long. File number and block number are senseless on a printer.

One output operation may print any number of characters packed 3 per word. The characters must be in ISO 7-bit code.

A line printer can print 7 to 17 lines a second.

6.1.7. Card reader (kind 16)

A document is one deck of cards. The card reader may read in various modes as descibed in ref. 8.

One input operation will fill the share with an integral number of cards.

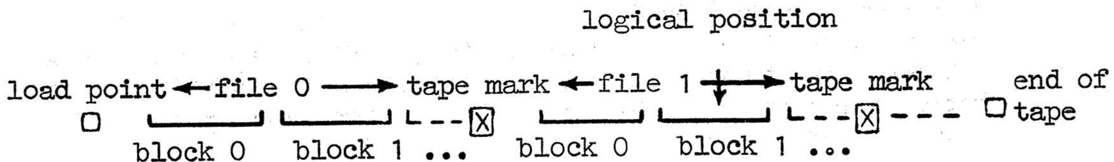
Usually jobs let the operating system read all necessary card decks before they are started. The cards may then be read as a normal ISO text stored on backing store (see ref. 7 for further details).

6.1.8. Magnetic tape (kind 18)

A document is one reel of tape. It consists of a sequence of files separated by a single file mark. Each file consists of physical blocks possibly with variable lengths. The blocks may be input or output in even or odd parity. The files and blocks are numbered 0, 1, 2, ... as shown in the figure.

One operation transfers one physical block to or from a share. If an input block is longer than the share, only the first part of the block is transferred.

A magnetic tape document:



Two kinds of tape stations exist: 7-track stations where a block consists of a sequence of 6-bit bytes; one word of the share is here transferred as 4 6-bit bytes. 9-track stations where a block consist of a sequence of 8-bit bytes; one word of the share is here transferred as 3 8-bit bytes. This difference causes no trouble as long as the tapes are written and read on RC 4000. But if you try to move a 7-track tape to another computer (or to an off-line converter), you should remember that the read and write procedures of algol work with 8-bit characters packed 3 to a word, which means that the physical 6-bit bytes on the tape have a strange relation to the logical 8-bit characters. You may, however, read or write 6-bit characters by means of the operators shift, add, extract and the procedures inrec6, outrec6.

The share length you use for output to a magnetic tape determines the physical block length. As the blocks are separated by block gaps, the share length has influence on the amount of information the tape can hold and also on the maximum transfer speed. With a density of 556 bpi (bytes per inch), a share length of 60 elements (240 bytes) will generate blocks of about 3/4 inch (more or less depending on the kind of the station). If the block gap is 3/4 inch, half of the tape is used for blocks and half for block gaps. The data is transferred with 0.38 times the maximum tape speed, if block gaps take 1.6 the time of blocks of the same length. If you used a share length of 600 elements (2400 bytes), 10/11 of the tape would be used for data and the transfer rate would be 0.86 of the maximum possible.

Details on actual transfer rates and possible densities is found in ref. 8 and the device manuals.

6.1.9. Devices without documents

Some peripheral devices, for instance the clock, do not scan documents, and they cannot be handled by the high level zone procedures. However, the primitive input/output level may handle such devices too.

6.2. High level zone procedures

The following standard identifiers are known as the high level zone procedures, because they work with a built in strategy for handling of peripheral devices (see 6.3). This built in strategy tries to make the documents appear as uniform as possible. For instance, the 'end of file' and 'end of document' conditions are transformed into End of Medium characters, which are detected easily by the normal use of the read procedures.

open	(see 9.41). Connects a document to the zone and divides the buffer area into shares of equal size.
close	(see 9.14). Terminates the current use of a zone including emptying of output buffers and possibly releasing of the document.
setposition	(see 9.58). Terminates the current use of a zone including emptying of output buffers. A magnetic tape or a backing storage area is then positioned to the file and block specified. The positioning takes no time on a backing storage area, but it may involve a lot of tape moving operations for a magnetic tape.
getposition	(see 9.23). Gets the file and block number corresponding to the current logical position of the document.
read	(see 9.51). Inputs a sequence of numbers given in character form on a document.
readchar	(see 9.53). Inputs one non-blind character from a document.
readstring	(see 9.54). Inputs a text string given as characters on a document.
readall	(see 9.52). Inputs a mixture of numbers, single characters, and text strings from a document.

repeatchar (see 9.56). Makes the latest character read from the document available for reading once more.

intable (see 9.32). Exchanges the current input alphabet with an alphabet specified in the program.

tableindex (see 9.75). Used in connection with intable to define the alphabet.

write (see 9.78). Prints texts, numbers, and single characters on a document.

inrec6 (see 9.31). Gets a sequence of bytes from a document and makes them available as a zone record.

outrec6 (see 9.46). Creates a zone record with an initially undefined content. The program may then assign values to the record variables, and later the record will be output to the document as a sequence of bytes.

swoprec6 (see 9.72). Gets a sequence of bytes from a backing storage area and makes them available as a zone record. The program may then modify the record, which later is transferred back to the backing storage area.

changerec6 (see 9.10). Regrets the former record and replaces it by a new one. The function of changerec6 depends on which of the procedures inrec6, outrec6, or swoprec6 was called most lately, i.e. the use of the document.

inrec, outrec, swoprec, and changerec are the Algol 5 versions of inrec6, outrec6, swoprec6, and changerec6. They differ from the latter in that the record length is measured in elements of 4 bytes each.

invar (see 9.33). Gets a sequence of bytes from a document as inrec6, but the number of bytes is given as the first word in the record. A check sum stored in the second word may be checked.

outvar (see 9.48). Creates a zone record of a specified length and stores data from an array or an other record. The length is stored in the first word of the record. A checksum is generated and stored in the second word of the record.

changevar (see 9.11). Changes the length of an existing record generated by means of outvar. The checksum is computed.

checkvar (see 9.13). Generates a checksum in an existing record.

The records generated by inrec6, outrec6, swoprec6, changerec6, and the corresponding Algol 5 versions of the same procedures are often referred to as fixed length records, although they may be of varying length. The records generated by invar, outvar, and changevar are referred to as var-records, and these procedures including the checkvar are referred to the var-procedures. All 12 procedures are referred to as the record handling procedures opposed to the character handling procedures read, readchar, readstring, readall, repeatchar, write, outtext, outchar, and outinteger.

Two standar zones, 'in' (9.28) and 'out' (9.42), exist. 'In' is used for input on character level, 'out' is used for output by means of write. The documents connected to in and out are determined when the run starts.

6.3. Buffering and checking

This section explains the algorithms used by the high level zone procedures for buffering and checking of the information on a document.

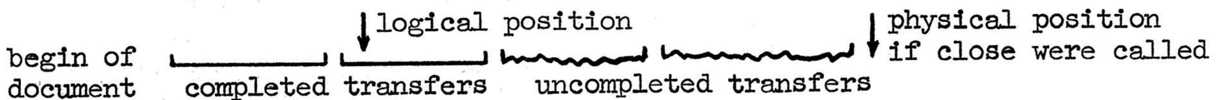
6.3.1. Multishare input/output

The amount of information transferred to or from a share in one operation is called a block. On a magnetic tape, a block is a physical block or a tape mark. On a backing storage area, a block is one or more segments. On a paper tape reader, a block is usually one share of characters.

Input

During input from a document via a zone with sh shares, the system uses one of the shares for unpacking of information and the remaining sh-1 shares for uncompleted input of later blocks. The following picture shows the state of the blocks of the document.

Input, sh = 3

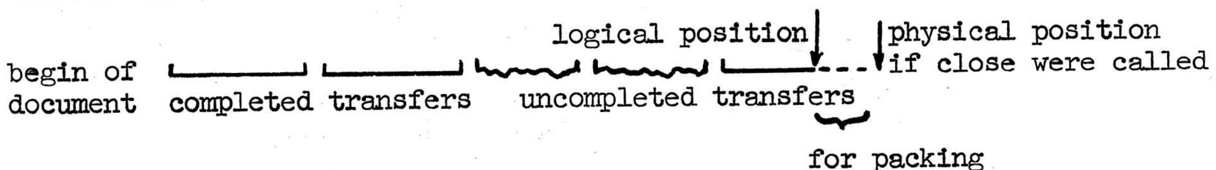


Note that when the document is closed, the physical position of the document is far ahead of the logical position. This is particularly important at the end of magnetic tapes where the 'waved' blocks may be absent and the tape then comes off the reel.

Output

During output to a document via a zone with sh shares, one share is used for packing of information, and 0 to sh-1 of the remaining shares are used for uncompleted output of previous blocks. The following picture shows the state of the blocks in the output stream.

Output, sh = 3



Note that when the document is closed, the physical position is just after the block corresponding to the logical position.

Swoprec

The procedure swoprec utilizes the shares as follows: One share is used for packing and unpacking of information. If sh > 1, another share is used for uncompleted output. Remaining shares are used for uncompleted input of later blocks.

Choice of sh

The advantage of the multishare input/output is that differences in speed between the program and the device may be smoothed to any degree. The most frequent choice is between single or double buffer input/output. The following rule of thumb may help you to choose in cases where you scan a document sequentially:

th = time spent by the program with handling of the information in a block
td = time spent by the device with transfer of a block
td + th is the total time in single buffer mode (sh = 1)
max(td,th) is the total time in double buffer mode (sh = 2)

If th varies from block to block, the situation is more complicated and sh > 2 may pay.

The following rule of thumb concerns the sequential use of swoprec:

th + 2*td is the total time per block with sh = 1
max(th,td) + td is the total time per block with sh = 2
max(th,2*td) is the total time per block with sh = 3

You should always use single buffering on printer, plotter, and punch, except if you know for sure that your job is not stopped and started by the operating system. The reason is that an output operation is terminated halfway when the job is stopped, but with sh > 1 the next output operation is started before the first is checked and output again.

You should always use single buffering for typewriter output, because the operator at any moment may stop the output operation to send a console message.

Message buffers occupied

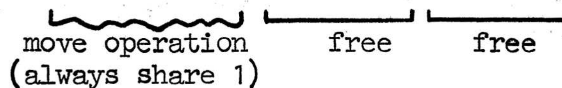
Input/output by means of sh shares occupies permanently sh-1 of the message buffers available for the job (see ref. 1). From the moment setposition has been called for a magnetic tape and until the first input/output operation is performed, one message buffer is occupied (even when sh = 1).

6.3.2. Algorithms for multishare input/output

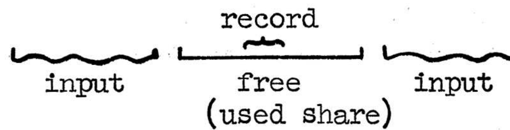
You must know about these algorithms if you want to interfere with the system in the block procedure of the zone (examples of block procedures are given in 6.3.4). Section 9.26 and 9.24 explain more about the variables in a zone. Ref. 1 and ref. 8 explain the rules behind the communication with devices. Below sh denotes the number of shares in the zone.

Snapshots of shares in typical situations (sh = 3)

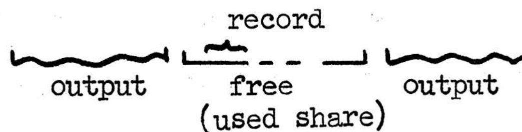
Just after setposition on a magnetic tape:



After inrec:



After several outrecs:



Change of block at input

```
rep: if share_state(used_share) = free then
     begin start transfer(input);
       used_share:= used_share mod sh + 1;
       goto rep
     end;
     comment now all shares are busy with transfers except after
       a positioning;
     wait_transfer(used_share); comment share state becomes free.
       The operation checked might be a positioning operation;
     last_byte:= top_transferred(used_share) - 1;
     comment now the share contains data from record base to last
       byte;
```

Change of block at output

```
     if share_state(used_share) <> free then
     begin wait_transfer(used_share);
       comment a positioning operation might be uncompleted;
     end;
     start transfer(output);

     used_share:= used_share mod sh + 1;
     comment one or more shares behind used_share are busy with
       transfers;
     wait_transfer(used_share);
     comment share state becomes free and the share may be filled
       from record base to last byte;
```

Start transfer (operation)

This procedure works only on used_share. It sets a part of the message and sends it:

```
first absolute address of block:= abs address of first shared;
segment number of message:= segment count;
update segment count for next transfer;
operation in message:= operation;
comment the mode is left unchanged;
send message;
share state:= uncompleted transfer;
```

Wait transfer

This procedure waits for the answer from a transfer or tape positioning, checks it, and performs the standard error actions (error recovery). Finally it may call the block procedure of the zone. In details this works as follows:

```
record_base:= abs address of first_shared(used_share) - 1;
last_byte:= abs address of last_shared(used_share) + 1;
record_length:= last_byte - record_base;
st:= share_state(used_share);
if st <> running child process then
share_state(used_share):= free;
if st <> uncompleted transfer then goto return;

wait_answer(st); if kind = magnetic tape then
begin
  if some words were transferred then block_count:=
    block_count + 1;
  if tape_mark sensed and operation is input or output mark
    then begin file_count:= file_count + 1; block_count:= 0
         end
end;
compute logical status word; comment the logical status word
  is 24 bits describing the error conditions of the transfer,
  see 6.3.3;
top_transferred(used_share):= if operation = io then
1 + address_of last_byte_transferred else
  first_shared(used_share);
users_bits:= common_ones in logical status and give up mask;
remaining_bits:= logical_status - users_bits;
Perform standard error actions for all ones in remaining bits
  (see 6.3.3).
if a hard error is detected then
logical_status:= logical_status + 1;
if hard_error is detected or users_bits <> 0 then
begin b:= top_transferred(used_share) - 1 - record_base;
  let record describe the entire shared area from first shared
    to last shared;
  save:= zone_state;
  if operation = input and tapemark and b = 0 then b:= 2;
  blockproc(z,logical_status,b);
  zone_state:= save;
  if b < 0 or b + record_base > last_byte then index_alarm;
  top_transferred(used_share):= b + 1 + record_base;
return: end;
```

6.3.3. Standard error actions

Each standard error action is mainly concerned with a single bit of the remaining bits in the logical status word. The logical status word is 24 bits generated at the end of an operation on the document. The first bits until 1 shift 12 are taken directly from the monitor, which takes most of the bits directly from the hardware. The last bits are a transformation of the result supplied by the monitor, while bits 1 shift 8, 1 shift 7, and 1 shift 6 are generated by the wait transfer routine (see 6.3.2). The meaning of the bits is as follows:

Logical status word

1 shift 23: Intervention. The device was set in local mode during the operation, presumably because the operator changed the paper or the like.

1 shift 22: Parity error. A parity error was detected during the block transfer.

1 shift 21: Timer. The operation was not completed within a certain time defined in the hardware.

1 shift 20: Data overrun. The high speed channel was overloaded and could not transfer the data.

1 shift 19: Block length. A block input from magnetic tape was longer than the buffer area allowed for it.

1 shift 18: End of document. Means various things, for instance: Reading or writing outside the backing storage area was attempted, the paper tape reader was empty, the end of tape was sensed on magnetic tape, the paper supply was low on the printer. See ref. 1 and ref. 8 for further details.

1 shift 17: Load point. The load point was sensed after an operation on the magnetic tape.

1 shift 16: Tape mark or Attention. A Tape mark was sensed or written on the magnetic tape or the attention button was pushed during typewriter i/o.

1 shift 15: Write-enable. A write-enable ring is mounted on the magnetic tape.

1 shift 14: Mode error. It is attempted to handle a magnetic tape in a wrong mode (NRZ or PE).

1 shift 13: Read error. Occurs on card reader. See ref. 8.

1 shift 12: Card reject. Occurs on card reader. See ref. 8.

1 shift 8: Stopped. Generated by the check routine when less than wanted was output to a document of any kind or zero bytes were input from a backing storage area.

1 shift 7: Word defect. Generated by the check routine when the number of characters transferred to or from a magnetic tape is not divisible by the number of words transferred, i.e. when only a part of the last word was transferred.

1 shift 6: Position error. Generated by the check routine after magnetic tape operations, when the monitors count of file and block number differs from the expected value in the zone descriptor (see 9.26, getzone).

1 shift 5: Process does not exist. The document is unknown to the monitor.

1 shift 4: Disconnected. The power is switched off on the device.

1 shift 3: Unintelligible. The operation attempted is illegal on that device, e.g. input from a printer.

1 shift 2: Rejected. The program may not use the document, or it should be reserved first.

1 shift 1: Normal answer. The device has attempted to execute the operation, i.e. '1 shift 5' to '1 shift 2' are not set.
1 shift 0: Hard error. The standard error action has classified the transfer as a hard error (see 6.3.2), i.e. the error recovery could not succeed.

The standard error action for 'stopped' cannot be performed successfully if 'users bits' (see 6.3.2) contain any one of the following bits: 1 shift 22, 21, 20, 19, 18, 16, 7, 5, 4, 3, or 2. As a consequence, the stopped-bit is ignored by the standard error actions in this case.

The bit 'normal answer' is always ignored, the remaining standard error actions depend on the document kind given in 'open' as shown below. This kind has not necessarily any relation to the actual physical kind. Situations not covered by the description are hard errors.

As an appendix to this section, you will find a quick index on how the standard error actions work for the different devices and status bits. You will also find the translation of the status bits to the messages from FP when the Algol program stops because of device errors (stderror is called).

Below follows a more elaborate description of the actions.

Details of handling of device status

Kind 0, internal process

Any status bit except '1 shift 18', end document, '1 shift 8', stopped, and '1 shift 1', normal answer, is treated by calling the block procedure. The special actions to be taken must be defined by a special agreement between your program and the internal process.

End of document: This will only make sense during input. If anything has been input, the bit will be ignored. Otherwise the empty block will be replaced by 2 bytes containing the text <:<25><25><25>:>. If this bit appears during any other operation, it will cause the block procedure to be called.

Stopped (during output): The output operation will be repeated for the remaining part of the buffer. This action may compensate for differences in share sizes in your program and in the internal process.

Kind 4, backing storage area

The monitor usually repeats defect transports to or from backing storage areas. Therefore most error bits are treated as hard errors. Only the bits '1 shift 18', end of area, '1 shift 8', stopped, '1 shift 5', process does not exist, and '1 shift 2', rejected are given special treatment.

End of document (i.e. area): If this happens during input, and if nothing has been transferred, the empty block is replaced by 2 bytes containing the text <:<25><25><25>:>, otherwise the bit is ignored. During output, the standard action is to try to extend the area (not at all possible in system 2). If it is impossible to extend, the block procedure is called, otherwise the output operation is repeated.

Stopped: This status may appear both during input and during output. The transfer is repeated except if it has been overruled by the action for end of area, or the two actions below.

Appendix to 6.3.3. Standard error actions

The status bits remaining after extraction of the user's bits get a special treatment depending on the kind used in the call of open (see section 9.41).
This treatment is shown in shorthand in the table below.

identification of status bit			actions for the different kinds							
			magtape	card reader	line printer	paper tape	paper tape	typewriter	bs-area	internal
bit no.	algol equivalent	name	18	16	14	12	10	8	4	0
Hardware generated bits 0	1 shift 23	intervention	ignore	ignore	ignore	ignore	ignore	ignore	give up	give up
1	1 shift 22	parity error	repeat 5 times	ignore	give up	give up	ignore	give up	give up	give up
2	1 shift 21	timer	give up	give up	give up	give up	give up	give up or ignore	give up	give up
3	1 shift 20	data overrun	repeat all	give up	error	error	error	error	give up	give up
4	1 shift 19	block length error	give up	error	error	error	error	error	error	give up
5	1 shift 18	end document	give up	ignore or EM	change	change	ignore or EM	give up	extend or EM	give up or EM
6	1 shift 17	load point	ignore	ignore	error	error	ignore	error	error	give up
7	1 shift 16	tapemark or attention	EM	ignore	error	error	ignore	ignore	error	give up
8	1 shift 15	writing enabled	ignore	error	error	error	error	error	give up	give up
9	1 shift 14	mode error	give up	error	error	error	error	error	give up	give up
10	1 shift 13	read error	error	ignore	error	error	ignore	error	give up	give up
11	1 shift 12	card reject	error	ignore	error	error	ignore	error	give up	give up

Software generated bits	12	1 shift 11	checksum error *)	error	error	error	error	error	error	error	error
	13	1 shift 10	bit 15	error	error	error	error	error	error	error	error
	14	1 shift 9	bit 14	error	error	error	error	error	error	error	error
	15	1 shift 8	stopped	repeat all or ring	error	repeat rest	repeat rest	error	repeat rest	repeat all	repeat rest
	16	1 shift 7	word defect	repeat 5 times	error	error	error	error	error	error	error
	17	1 shift 6	position error	give up	error	error	error	error	error	error	error
	18	1 shift 5	does not exist	mount	give up	give up	give up	give up	give up	create	give up
	19	1 shift 4	disconnected	mount	give up	give up	give up	give up	give up	give up	give up
	20	1 shift 3	unintelli-gible	give up	give up	give up	give up	give up	give up	give up	give up
	21	1 shift 2	rejected	reserve	give up	give up	give up	give up	give up	create	give up
	22	1 shift 1	normal	ignore	ignore	ignore	ignore	ignore	ignore	ignore	ignore
	23	1 shift 0	hard error								

* The 'checksum error' bit is not generated by the Algol check routines, but by invar (see section 9.33). It will not be explained in this section.

The entries 'give up' and 'error' in the table above mean that the hard error bit will be set and the give up action - your block procedure - will be called. If the entry says 'give up', it means that this status bit may occur for the kind specified, but no standard action has been invented. If it says 'error', it may mean that you have opened with a wrong kind or that the system has been misused in some other way.

The entry 'ignore' means that no action is taken for this status. This may either be because the status is normal for the device (write enable for magtape or normal answer) or because it occurs together with another status.

Process does not exist: An area process is created. If the creation is not successfull, the action gives up and calls the block procedure. If the operation is output, the area process is reserved for exclusive access. If this is not possible, the action gives up and calls the block procedure. Now the transfer is repeated.

Rejected: Handled exactly as process does not exist.

Note that the status messages process does not exist or rejected may be caused by the fact that you have exceeded your area claims.

Kind 8, typewriter

Among the status bits concerning the hardware only the timer status, '1 shift 21' has been given special treatment. The ignored hardware bits will either generate disconnected status, i.e. '1 shift 4' or '1 shift 8', stopped during output.

Timer: If this status happens as a result of an output message, the block procedure is called. After an input operation it is ignored if anything has been input, otherwise the input operation is repeated.

Stopped (during output): If this bit is generated together with the ignored bits, the rest of the buffer is output.

Kind 10, paper tape reader

Only end document status '1 shift 18' gets a special treatment from the check system. If a parity error occurs, the monitor will substitute the defect character by a substitute character, decimal value 26. Intervention status is ignored.

End of document (i.e. end of paper tape): If anything has been input the status is ignored, otherwise a block of 2 bytes containing <:<25><25><25>:> is simulated.

Kind 12, paper tape punch

If something has been punched with parity error, the action is to give up, and call the block procedure. The same thing happens after a timer status as this usually is caused by the punch running out of paper tape without having given end document status. This is either caused by hardware malfunction or by misuse of the punch.

End of document (i.e. no more tape): A message is sent to the parent, requesting that the paper is changed in the punch and that the job is stopped until the operator tells that he has done so.

Stopped (during output): The remaining part of the share is output.

Kind 14, line printer

If a parity error occurs during printing, the standard action is to give up. The end of document status means that the paper has run out.

End of document (i.e. no more paper): A message is sent to the parent requesting that the paper is changed and that the job is stopped until the operator tells that he has done so.

Stopped (during output): The remaining part of the share is output.

Kind 16, card reader

A parity error status, signalling an error in the conversion, is ignored by the standard error actions, as the monitor substitutes the wrong combination by a substitute character corresponding to the conversion (see details in ref. 8). The end of document status shows end of card deck.

End of document (i.e. end of deck): If anything has been input, the status is ignored, otherwise a block of 2 bytes containing <:<25><25><25>:> is simulated.

Kind 18, magnetic tape

The actions for magnetic tapes are made so that a tape may be unloaded and remounted during the run without harming the job using the tape. Label check is not included, it is expected that the operating system (or the machine staff) performs this. The action on mode error is to give up and call the block procedure.

Parity error: The stopped bit is ignored in this case. An input operation is repeated up to 5 times, but if the parity error persists, the error is a hard one. An output operation is repeated up to 5 times, preceded by 1 erase operation the first time, 2 erase operations the second, and so on. If the parity error persists, the standard actions give up and call the block procedure.

Word defect: The actions are as for parity error. Note that if you suppress the word defect action by setting '1 shift 7' in your give up mask, you can read tapes not written on the RC 4000 or tapes written with trail <> 0 (see open, 9.41). Of course your block procedure will be called each time the bit occurs. In case of word defect, unused character positions are filled with binary nulls.

Tapemark: Tapemark is ignored after a sense or a move operation. If tapemark occurs after an input operation, the standard action is to simulate a block of 2 bytes containing <:<25><25><25>:>.

Stopped (during output): If the 'ring' bit is set, the output is repeated. Otherwise a message is sent to the parent requesting a write enable ring to be mounted. When the job is restarted after mounting of the ring, the output is repeated.

Does not exist: This bit is ignored after a sense operation or a move operation. In other cases, a mount-tape-message is sent to the parent. Next, the tape is reserved for exclusive access and if this goes wrong, the mount-tape-message is sent again. Third, the tape is positioned according to file and block count and the operation is repeated.

Rejected: Handled as 'does not exist', except that the mount-tape-message is not sent.

Parent message

The parent (i.e. the operating system for your job) may either handle a message according to its own rules, or it may pass the request on to the operator. The job may ask the parent to stop the job temporarily until the operation has been performed. The exact rules depend on the operating system in question.

6.3.4. Block procedure

Call situation

The high level zone procedures may call the block procedure after input and output operations and after move operations and output mark operations on magnetic tapes. After such an operation, the call will take place in these cases:

1. When some of the bits set in the give up mask occurred in the logical status word.
2. When the standard error actions classified the situation as a hard error (give up).

The block procedure is called with 3 parameters:

blpr(z,s,b)

- z is the zone. The record of z is the entire shared area available for the transfer.
- s is an integer containing the logical status word.
- b is the number of bytes transferred in the operation.

You can tell the difference between the call reasons by means of the last bit in the logical status word.

Purpose and return

In the block procedure, you can do anything to the zone by means of the primitive zone procedures and the high level zone procedures (in the latter case you must be prepared for a recursive call of the block procedure, for instance as shown in example 3 below).

To make sense, the effect of the work should be an improved check or error recovery of that operation which caused the block procedure to be called. You may also avoid a standard error action by means of the give up mask and instead perform your own checking of the transfer.

You signal the result of the checking back to the high level zone procedure by means of the final block length, b. The value of b has no effect when an output operation is checked, but after an input operation you may signal a longer or a shorter block or even an empty block (b = 0). However, the value of b at return must correspond to a block which is inside the shared area specified by the value of used_share at return. Otherwise, the run is terminated with an index alarm. Further details may be found in 6.3.2.

Example 1, rejecting part of a block

A block procedure which tries to repair an input block after persistent parity error looks like this:

```
procedure repair(z,s,b); zone z; integer s,b;
if s shift (-22) extract 1 = 1 then
begin comment handling of persistent parity error;
  integer to,from;
  to:= 0;
  for from:= 1 step 1 until b//4 do
  if z(from) is o.k. then
  begin to:= to + 1;
    z(to):= z(from);
  end;
  comment the defect items of the block are squeezed out.
    The new length is signalled back;
  b:= to*4;
end
else stderror(z,s,b);
```

The zone should be opened with a give up mask of 0.

Example 2, copy input

A block procedure which copies everything read from 'z' to 'test' may look like this:

```
procedure copy(z,s,b); zone z; integer s,b;
if s extract 1 = 1 then stderror(z,s,b) else
begin comment this code also works for b = 0;
  outrec6(test,b);
  tofrom(test,z,b);
end;
```

The zone must be opened with a give up mask of 2 (normal answer). Inrec6, invar and read take action on nothing transferred (maybe stopped).

Example 3, label checking on magnetic tape

This example has no relevance in system 3, if your parent (operating system) is Boss.

The safety of magnetic tape positioning can be improved by means of file labels. Each of the logical files on the tape are separated by two tape marks surrounding one label block. This block contains the logical file number in text form.

The positioning to block 0 of a logical file (counted 1, 2, ...) is started with this procedure:

```
procedure logpos(z,f); zone z; integer f;
setposition(z,f*2 - 2,0);
```

The procedure cannot check the label, because simultaneous positioning then would be impossible. Instead the block procedure may check the label:

```
procedure labelcheck(z,s,b); zone z; integer s,b;
if s extract 1 = 1 then stderror(z,s,b) else
begin integer array ia(1:20); integer op,f,bl,lab;
  own boolean next;
  comment next indicates whether the procedure was called
    from labelcheck itself;
  getzone6(z,ia); getshare6(z,ia,ia(17));
  comment the operation checked is used_share, which now is
    moved to ia;
  op:= ia(4) shift (-12) extract 12;
  if (op = 0 or op = 8) and -, next then
  begin comment positioning operation not called from
    labelcheck was completed;
    next:= true;
    getposition(z,f,bl); setposition(z,f,bl);
    if read(z,lab,op) <> 1 or lab <> f//2 + 1 then
    system(9,f//2 + 1,<:<10>position:>);
    comment if label did not contain exactly one number or
      the file number recorded is wrong, the run is terminated
      with an alarm;
    setposition(z,f+1,0); b:= 0; next:= false;
  end
end;
```

The zone must be opened with a give up mask of 2 (normal answer).

6.4. Primitive Level, Operating System

When you use zones on the primitive level, you can change the values of the zone descriptor and the share descriptors (see 5.5) in nearly any way. In this way you may handle the peripheral devices in non-standard ways. You may also use the full principle of sharing buffer area with other processes to create child processes and let the algol program work as an operating system to these child processes.

The following 7 standard procedures are known as the primitive level zone procedures:

getzone6	(see 9.27). Transfers the contents of a zone descriptor to an array.
setzone6	(see 9.62). Transfers the contents of an array to a zone descriptor.
getshare6	(see 9.25). Transfers the contents of a share descriptor in a zone to an array.
setshare6	(see 9.60). Transfers the contents of an array to a share descriptor in the zone.
monitor	(see 9.49). This procedure is the algol equivalent to all the functions of the monitor. It starts and stops communication with peripheral devices, it creates, starts, stops, and removes child processes, etc.
blockproc	(see 9.6). Calls the block procedure of a given zone.
check	(see 9.12). Checks a transfer to or from a document in the way used by the high level zone procedures.

6.4.2. Document driver

You may let the algol program control a document to which other processes in the computer send output:

1. Use entry 20 (or entry 24) in 'monitor' to wait for messages sent to the algol program. The sender of the message assumes that the algol program is a document.
2. Copy the block of information described in the message into a zone buffer area by means of 'system', entry 5, or use entry 70 in 'monitor'.
3. Send the answer to the message by means of entry 22 in 'monitor'.
4. Output the block of information to the document.

Under special circumstances, for instance when the algol program is the operating system for these other processes, it is possible to control input and output from a document, even without copying the block of information from one buffer to another. That is possible because both the sender process and the buffer for the document may be parts of the same zone buffer area.

6.4.3. Operating system

You may let the algol program create, start, stop, and remove a child process in this way:

1. Use entry 56 in 'monitor' to create the child process in a zone buffer area. It may be necessary to use entry 72 in 'monitor' to set your own catalog base in order to define the base of the process name.
2. Include the process as a user of some peripheral devices by means of entry 12 in 'monitor', and give the process access to the backing store by means of entry 78 in 'monitor'.
3. Initialise the child process area with a suitable binary program, for example the File Processor code which may be read directly from the backing storage area fp into the zone buffer area.
4. Set the machine registers of the child process by means of entry 62 in 'monitor'. See ref. 2 and ref. 6, if FP is used.
5. Start the child process by means of entry 58 in 'monitor'. Now, the child process starts executing the instructions of the binary program. We say that it runs in parallel with the other processes in the computer (including your algol program). If FP is the executive system, the user base is communicated so that this is the catalog base at which the child process was started. FP will as its first action set the catalog base to standard.
6. When you want to stop the child, use entry 60 in 'monitor'. ACCORDING TO 'MAINTENANCE LIST' NOV 77 MONITOR (60) DOESN'T WORK.
7. Wait for the completion of the stop by means of entry 18 or 24 in 'monitor'. Now, all modifications of the child process area have ceased, and you may for instance store the area on the backing storage, use the area for something else, later reestablish the process area and start the child again by means of entry 58 in 'monitor' so that it continues as if nothing had happened.
8. When you want to get rid of the child and withdraw its resources, you use entry 64 of 'monitor'. Remember the process must be stopped first.

In order to make an operating system which handles several child processes, serves as a driver for peripheral devices, and communicates with the operator, you have to mix the principles of 6.4.1, 6.4.2, and 6.4.3. In this mixing, entry 24 of 'monitor' is very useful to help the program serving the first arriving event first. An event is here the arrival of a message or an answer, or the completion of a stop.

7. SYSTEM CONTROL, ETC.

This chapter gives a brief introduction to the 6 standard identifiers which control global conditions of the running algol program.

<u>blocksread</u> (see 9.7). This integer variable is increased by one each time a segment of the running algol program is transferred from the backing storage to the core store. It can assist you in balancing the use of the core store.

overflows (see 9.49). This integer variable controls the action on floating point overflow.

underflows (see 9.77). This integer variable controls the action on floating point underflow.

system (see 9.73). This procedure controls the floating point precision (mantissa of 36 or 33 bits). The procedure may also supply information about the surroundings (the console, the parent, the state of the message queue), it may move any area of the core store into an array, it may give the length of the available core, and it may terminate the run with an alarm message.

systime (see 9.74). This procedure gives access to the real time clock in the monitor and to the CPU time used by the job. Further it may convert real time to date and clock.

stderror (see 9.69). This procedure terminates the run with an error message specifying an error condition on a peripheral device. It is used as the block procedure of zones where you don't care for device errors.

8. THE ALGOL SYSTEM

This chapter describes the way the algol compiler and the running program fits into the RC 4000 multiprogramming system.

8.1. Translation

The compiler works in your job process and you start the translation by means of an FP-command specifying the source text, the compilation variants, and the file where the resulting object program should end (see app. B).

The result of the translation is either a complete, self-contained, binary program or a binary external procedure. In the first case, the program may be executed as described in 8.3; in the second case, the procedure may be used as a standard procedure in later translations. If you permanent the program or the procedure (give it scope user or scope project), you can use it in later jobs.

8.1.1. The compiler

The compiler occupies about 13000 instructions divided into 12 passes, either on backing storage or on magnetic tape. In the first case, it may be used for simultaneous translation in several job processes.

The 12 passes of the compiler perform the following tasks: Pass 0 is a common administration routine. Pass 1 to 8 perform the translation into binary code by means of 8 scans of the source program. The intermediate program text is stored in the place later occupied by the binary program. Pass 9 rearranges the binary program, inserts references to standard procedures, and includes the code for the standard procedures used in the program. Pass 10 includes the run time administrative system (RS). When an external procedure is translated, pass 9 only rearranges the binary procedure and RS is not included. Pass 11 does not exist, but a pass 12 may make crossreferences of where the different names are used.

8.1.2. Storage requirements, etc.

The compiler requires a job with a core area of 12 000 bytes with 4 message buffers and with 6 area processes (4 if current input and output are not backing storage).

The minimum core area may cause the translation to terminate with the alarm 'stack'. This is due to the limited size of the table of identifiers in pass 2 and 5, and the table of labels, case elements, and procedures in pass 8. A greater core area will remedy the problem: just 1000 bytes more give room for about 250 identifiers.

8.1.3. Speed, length of object code

After basic time of 2 seconds (compiler on drum), the total translation speed is about 1000 characters/second or 500 final instructions per second for an average program.

The final program consists of the code corresponding to the source text, plus 7 segments for RS, plus the length of the standard procedures incorporated. The length of the code corresponding to the source text is about 1.5 the length of the source text.

8.1.4. Error checking

The compiler performs extensive syntax and type checking, but a few errors may pass undetected as described in C.2.2.

Except for some rare errors concerning communication with the surrounding system, no error can stop the compilation, and most of the errors will be detected in the first translation. Suitable mechanisms are included to prevent one error from generating several error messages.

Whenever the translation has worked to the end, the program may be executed until the first point where a syntax error was detected or until the first point where an undeclared or doubly declared identifier is used. The run is then terminated with the message 'syntax line...'.

8.2. Assembly, index, spill

Pass 9 performs the assembly of standard procedures into the main program and if these standard procedures reference other standard procedures the assembly continues recursively. All standard identifiers must exist in the catalog at this stage.

At run time, subscript check will be omitted during the execution of all program parts compiled with index.no. All standard procedures mentioned in chapter 9 may be thought of as compiled with index.yes.

If the main program is compiled with spill.yes, a partial check of integer overflow is performed in procedures compiled with spill.no. If the main program is compiled with spill.no, integer overflow at multiplication will still be detected in subroutines compiled with spill.yes. None of the standard procedures of chapter 9 can cause an integer overflow.

8.3. Execution

A binary object program is executed in the job process and started by means of an FP-command as described in app. B. The program must at that moment exist in a backing storage area.

8.3.1. Segmentation

The object program consists of independent program segments of 512 bytes. Whenever the running program demands a program segment which is not in the core store, it is transferred from the backing storage possibly replacing another segment in the core store. The number of segments held in the core store is increased gradually until the limit posed by the variables is met. If more variables are declared, some segments will be released from the core store.

This scheme works satisfactorily as long as the program segments involved in the current part of the algorithm are kept in the core store. Under these circumstances a jump to another segment is performed in 7 microseconds, while a jump within one segment is performed in 3 microseconds.

When the number of variables is increased so that the active segments cannot stay in core, the program can still run, but a jump to another segment will often cause a transfer from the backing storage resulting in a jump time of 18 000 microseconds. Section 9.7 shows how these situations may be detected. You will see from this that it is very important to avoid crowding the job area with variables. As a rule, you should have room for at least 8 segments in the core store, corresponding to 4000 bytes.

As further aid, the compiler may print a list of line numbers corresponding to the segment boundaries in the object program. The list is printed if the compiler is called with details.8.8. (see app. B).

8.3.2. Storage requirements

During program execution, the job area is organized in this way:

Length in bytes:	Contents
1500	File Processor
600	RS — RUNNING SYSTEM
Depends on program	Own variables for entire program.
2*L	Segment table, L=total number of program segments.
minimum:1024 reasonable:4096	Room for program segments currently in core store.
	Room for variables, arrays, zones.
1024	Buffers for in and out.

When the program is called with the parameter 0 (see app. B), the space occupied by File Processor and buffers for in and out becomes 16 bytes.

The space occupied by variables at any moment of the execution is the sum of the reservations made at entries to all the blocks and procedure bodies which are active.

Lengths of core store are usually given in bytes (one byte = 12 bits), sometimes in words or double words (4 bytes = 2 words = 1 double word).

The reservations made at block entry may be derived from the declarations of the block as follows:

Quantity:	Number of bytes reserved:
Simple boolean variable, field variable, simple integer variable	2
Simple long variable, simple real variable	4
Array segment	2*(number of array identifiers + 1 + number of subscripts) + space for total number of array elements.
Array element, boolean	1
Array element, integer	2
Array element, real or long	4
Zone	50 + 24*number of shares + 4*bufferlength.
Zone array	2 + space for all the zones.
Working locations	Depends on structure of program, usually about 10 for each block.
Block, procedure body	2*number of statically surrounding blocks + (if normal block then 4 else if type procedure then 14 else 10);
Parameter	8 if the actual parameter is constant, 4 otherwise.

8.3.3. Message buffers, area processes, etc.

The job process must have been created with a sufficient number of message buffers and area processes. The number of message buffers occupied at any moment during the execution of the program is derived as follows:

Reserved for RS	1
Each n-shared zone used for high level input/output ('in and 'out' count as 1-shared zones)	n-1
Each zone busy with positioning a magnetic tape (then it is not used for input/output)	1
Zones used on primitive level, each share describing an uncompleted transfer.	1

The number of area processes occupied at any moment is 2 + the number of backing storage areas opened for input/output. Remember to include possible area processes used by 'in' and 'out'.

8.3.4. Execution times

Are given in app. A.

8.3.5. Error checking

The dynamic error checking and the error message are given in app. C.

9. ALPHABETIC LIST OF NEW ELEMENTS

This chapter gives a detailed description of all the standard identifiers supplied as a part of the compiler and all the delimiters not found in algol 60.

The syntax of operators and other delimiters is described rather informally. Take the last two lines of 'add' (9.2) as example:

<boolean> add <primary> is of type boolean
Priority as **.

This shows that 'add' has two operands, the left is of type boolean, the right of type integer. The result of applying 'add' to these two operands is of type boolean. The term 'priority as **' means that in the rules of precedence for evaluation of an expression, 'add' and '**' appear on the same level and they are executed in sequence from left to right within the expression.

The description of procedures follows a different scheme. Take 'inrec6' (9.31) as an example:

Call: inrec6(z,length)

inrec6	(return value,integer). The ...
z	(call and return value,zone). The ...
length	(call value, integer, long, or real). The ...

This shows that inrec6 is called with two parameters. The first, z, must be a zone. The contents of the zone at call time is significant and it is changed at return from the procedure. The second, length, must be an integer, a long or a real. The value of length at call time is significant. It is not changed at return. Finally, inrec6 is shown to have an integer value at return.

The parameters may actually be expressions, of course. Unless something else is mentioned, it is a tacit assumption that all the parameters are evaluated once, but not necessarily in sequence from left to right. Especially, if something is assigned to a parameter, the assignment may or may not be delayed until all the parameters have been evaluated (see for instance read, 9.51). Note, that the evaluation of a string parameter will access the actual parameter repeatedly until the string end is supplied (see 3.6).

9.1. Abs

This monadic operator yields the absolute value of integer, long, or real expression.

Syntax:
abs <integer> is of type integer.
abs <long> is of type long.
abs <real> is of type real.
Priority higher than **.

Examples: abs r   abs sin(x)   abs(0.5+sin(x))

9.2. Add

This dyadic operator is used for packing of integer values into a real, long, integer, or boolean value.

Syntax:

<real>	add <primary>	is of type real.
<long>	add <primary>	is of type long.
<integer>	add <primary>	is of type integer.
<boolean>	add <primary>	is of type boolean.
<string>	add <primary>	is of type string.

Priority as **.

If the right hand operand is real or long it is rounded (in the sense of 9.57) to an integer. Now both operands are treated as binary patterns (see 3.1 and 3.6) and the right hand integer is added to left hand operand to obtain the binary pattern of the result. If the result is a boolean, it is cut to 12 bits. The addition is binary addition in 24 bits with rightmost bit added to rightmost bit. No carry is propagating into a possible left hand word.

Example 1:
Let i+1 and j be integers between 0 and 63. They may be packed into one boolean variable in this way:

```
b:= false add (i+1) shift 6 add j;
```

If j were negative, the statement would not work as intended.

Example 2:
Two signed integers may be packed into one real in this way:

```
r:= 0.0 shift 24 add i1 shift 24 add i2;
```

Note that the binary pattern of a negative number has zeroes in front of the 24 ordinary bits, and that no carry will propagate into the i1-part of r.

Example 3:
The last bit of an integer 'j' may be tested in this way:

```
if false add j then ..
```

9.3. Real procedure arcsin

Call: arcsin(r)

arcsin	(return value, real). Is the mathematical function arcsine of the argument r. $-\pi/2 < arcsin < \pi/2$.
r	(call value, real, long, or integer). $-1 \le r \le 1$ must hold.

Accuracy:

r = 1, -1	gives an absolute error of 3'-12
r = 0	gives arcsin= 0
0 < abs r < 0.5	gives a relative error below 1.1'-10
0.5 ≤ abs r < 1	gives a relative error below 1.6'-10

9.4. Real procedure arctan

Call: arctan(r)

arctan	(return value, real). Is the mathematical function arctangent of the argument r. $-\pi/2 \leq$ arctan $\leq \pi/2$.
r	(call value, real, long, or integer).

Accuracy:

r = 0	gives arctan = 0
r <> 0	gives a relative error below 1.5'-10

9.5. Real procedure arg

Call: arg(u,v)

arg	(return value, real). Is the argument in radians of the complex number u+i*v. $-\pi <$ arg $< \pi$. If u < 0 and v = 0, arg is positive.
u	(call value, real, long, or integer).
v	(call value, real, long, or integer).

Accuracy:

v = 0 and u ≥ 0	gives arg = 0.
v <> 0 or u < 0	gives a relative error below 1.8'-10.

Example:
Let a and b be the lengths of two sides of a triangle, and let C be the angle between them (in radians). The angle B, opposite to b, is then computed by:

```
B:= arg(a-b*cos(C),b*sin(C));
```

9.6. Procedure blockproc

Executes a call of the block procedure associated with a given zone. Blockproc makes it possible in pure algol to obtain an effect like check (used by inrec, read, write, etc.), which only knows the zone, but still manages to call the block procedure (see 6.3.4.).

Call: blockproc(z,s,b)

z	(call and return value, zone). Specifies the procedure to be called.
s	(call and return value, integer). The value of s is supposed to be a logical status word.
b	(call and return value, integer). The value of b is supposed to be the number of bytes in a block transfer.

Let pr be the block procedure of z. Then the following call will be executed:

```
pr(z,s,b)
```

9.7. Integer blocksread

This standard variable is increased by one each time a segment of the algol program is transferred from the backing storage. This enables you to estimate the length of the program loops and balance the use of the core store. The value of blocksread is printed at program end (see appendix 2).

Example 1:

If you feel that your program is running very slowly, the first thing to do is to insert a piece of code around the inner loop:

```
blocksread:= 0;
The inner loop;
write(out,blocksread//55);
```

The number printed is then the number of seconds spent in transferring program segments from drum to the core store. If this explains the trouble, there are only two solutions: 1) to change the program so that fewer variables are declared, or 2) to run the job in a greater core area. In this example the integer printed after the end message is not the total number of segment transfers during the run, but it shows the number of transfers since the latest time blocksread was set to 0.

Example 2:

In many cases a program can run with an array of varying length. One example is the first phase of a magnetic tape sorting. Here you save tape passes in the second phase by increasing the array available for the first phase. But if you increase too much, the first phase will become very slow because of frequent program transfers.

The following program shows how this can be balanced by the algorithm itself. The idea is to reserve an array of maximum size (see system, 9.73) and then decrease the length of the array whenever segments are transferred in the inner loop.

```
        n:= max;
rep:    begin array ia(1:n);
        s:= blocksread;
        The inner loop;
        if blocksread > s then n:= n - 128;
        end;
        goto rep;
```

It is much more difficult to do the same thing starting by a short array.

9.8. Case

This delimiter occurs in case-expressions and case-statements, which are generalisations of if-expressions and if-statements. The case-constructions use an integer to select among several expressions or statements.

Syntax: case <arithmetic expression> of begin <statement list> end
is a statement.
case <arithmetic expression> of (<expression list>)
is an expression.
They have the same syntactical positions as if-statements and if-expressions.
The elements of an expression list must either be all arithmetic, all boolean, all string, or all designational.

The statements of a statement list are separated by semicolons and numbered 1, 2, 3, ... The expressions of an expression list are separated by commas and numbered 1, 2, 3, ...

A case-construction is executed as follows: First, evaluate the arithmetic expression and if necessary round it to an integer. Next, select the list element corresponding to the result. If no such list element exists, the run is terminated. If the selected element is a statement, execute it and continue the execution after the complete case-statement (provided that a goto was not executed). If the selected element is an expression, evaluate it and take the result as the value of the case-expression.

If the elements in an expression list are arithmetic, the type of the resulting value will be:

integer	if all elements are integer
long	if one or more elements are long, but none real
real	if one or more elements are real.

Example 1, initializing a table.

An array may be initialized with the values 3, 5, 0, 1, 1, 2 in this way:

```
for i:= 1 step 1 until 6 do
ia(i):= case i of (3,5,0,1,1,2);
```

Example 2:

The logical status word occuring as a parameter to block procedures may be displayed in this way:

```
for i:= 0 step 1 until 23 do
if logical_status shift (-i) extract 1 = 1 then
write(out,case 24-i of
(<:local:>,<:parity:>,<:timer:>, ...));
```

Example 3:

See example 2 of readchar, 9.53.

9.9. Integer procedure changerec

Regrets the latest call of inrec, outrec or swoprec and makes a record of a new size available. The procedure is the Algol 5 version of changerec6.

Call: changerec(z,length)

changerec	(return value, integer). The number of elements of 4 bytes each left in the present block for further calls of inrec, outrec or swoprec.
z	(call and return value, zone). The name of the record.
length	(call value, integer, long or real). The number of elements of 4 bytes each in the new record. Length must be >= 0.

For further details see 9.10.

9.10. Integer procedure changerec6

Regrets the latest call of inrec6, outrec6 or swoprec6 and makes a record of a new size available.

Call: changerec6(z,length)

changerec6	(return value, integer). The number of bytes left in the present block for further calls of inrec6, outrec6 or swoprec6.
z	(call and return value, zone). The name of the record.
length	(call value, integer, long or real). The number of bytes in the new record. Length must be >= 0. If length is odd, one is added.

Zone state:
The zone must be in one of the states 5, 6 or 7, i.e. after record input, after record output, or after record swop (see getzone6, 9.27), and it is left in the same state.

Blocking:
Changerec6 can be used to regret a former call of the procedures for record handling.
This happens in the following way:

1) Check that 5 <= zone state <= 7. Set the record length to 0 (zero) and the logical position just before the record base.

2) Start the record procedure indicated by the zone state with the same parameters as changerec6. I.e. if zonestate = after record input then inrec6 (z,length) else if zone state = after record output then outrec6(z,length) else swoprec6(z,length).

The terms zone state, record length, and record base are explained in section 9.27, getzone6.
If there is room in the current block for the new record size, a call of changerec6 will not change block. In this case data in elements available both before and after the call are unchanged.

If you are not aware of the rest length in the used share, you must be prepared for a block change if the length in the call of changerec6 is greater than that of the previous call of a record procedure.
The blocking is explained in more detail in 9.31, inrec6, 9.46 outrec6, and 9.72 swoprec6.

Example 1:
Output of records with variable length.

Records with variable length, where the length is stored in the first word (2 bytes), may be output like this:

```
rep:   outrec6(z,maxlength);
       ....; Fill the buffer and compute the actual length.
       z.firstword:= actuallength;
       changerec6(z,actuallength);
       if ..... then goto rep;
```

Compare this with example 1 of outrec6, section 9.46, where the actual length is known before the call of outrec6.

Example 2:
See example 2 of invar, 9.33.

9.11. Integer procedure changevar

Is used in connection with outvar, as it replaces a record placed in z by means of outvar with another, maybe of a new length. The call changevar(z,z) always works so that indices available both before and after the call refer to the same piece of data - even though a block change may have happened.

Call: changevar(z,A)

changevar	(return value, integer). The number of bytes available for further calls of outvar before change in block takes place exactly as for outrec6.
z	(call and return value, zone). The zone used for output.
A	(call value, real array). An array containing the record to replace the current zone record. The first word of the element with lexicographical index 1 must contain the new record length in bytes. If it is odd, 1 is added.

Zone state:
The zone state must be after record output (state 6, see 9.27, getzone6), and the latest record may have been placed by means of outrec6, outvar or the like.

Blocking:
Changevar tests whether the next record may reside within the current block, and changes the block if this is not the case. The old record is not output. The call changevar(z,z) gets a special treatment, as the second parameter will be saved if it cannot reside in the zone buffer

while the block is changed. The blocking and the function is explained in more detail in section 9.48, outvar.

Record Format, counting of records:

The record format is explained in section 9.33, invar. The free zone parameter (see 9.27) is decreased by one if the new length is 0 (null). Otherwise it is not changed.

Example, sequential file updating by merging.

Certain systems maintain their master files by merging, an old master file with a transaction file giving a new master file. We assume that the files are sorted in ascending order with respect to a key field, that the files end with an end-record with the key equal to the maximum value for longs, and that the records are var-records.

The following algorithm allows several transactions to the same master record. It also allows transactions to a new master record, supposed that the new record precedes the transaction record. The algorithm can easily be extended to more than 3 files.

```
begin comment merging algorithm;
  zone old, trans, new(..., ..., stderror);
  integer action, creation, removal, changes, guessed_len;
  long first, infinity;
  integer field length, type; long field key;

  length:= 2; ... infinity:= extend(-1) shift (-1); ...
  comment The initialisation of the type identifications
  'creation', 'removal', and 'changes' as well as the field
  variables 'key' and 'type' depend on the record format. The
  initialisation of 'infinity' assumes that the key is > 0. The
  value of 'guessed_len' may lie between the minimum length
  and the maximum length of the record. If it is the minimum
  length, blockchanges are postponed as long as possible, and
  if it is the maximum length, intermediate savings during
  changevar is avoided;

  open(old ..... maybe also setposition on the documents;
  invar(old); invar(trans); outrec6(new, guessed_len);
  new.length:= guessed_len; new.key:= infinity;

rep: ; comment The following code determines an action number
  which may be thought of as a binary number 1 <= action <= 7,
  where 1 means new contains the lowest key, 2 means trans con-
  tains the lowest key, 4 means old contains the lowest key;

  first:= old.key; action:= 4;
  if trans.key = first then action:= 4 + 2
  else if trans.key < first then
    begin first:= trans.key; action:= 2;
    end;

  if new.key = first then action:= action + 1;
  else if new.key < first then action:= 1;

  case action of
  begin
```

```
          begin comment output the ready record;
            outrec6(new.guessed_len);
            new.key:= infinity; new.length:= guessed_len;
          end 1;

          begin comment the transaction should be a creation;
            if trans.type <> creation then error;
            trans.type:= ... perform necessary changes in trans;
            changevar(new, trans); invar(trans);
          end 2;

          begin comment the transaction must be a removal or a change;
            if trans.type = creation then error;
            if trans.type = removal then new.key:= infinity else
            begin
              ... perform changes in new, perhaps make
                  new.length:= new_len; changevar(new, new);
              .....
              checkvar(new);
            end;
            invar(trans);
          end 3;
          begin comment no transactions to this record
old_to_new:
            changevar(new,old); invar(old);
          end 4;

          begin comment 2 records with the same key exist. This is
            a serious error;
            alarm;
          end 5;

          begin comment let the transaction wait until we have been
            through the logic once more;
            goto old_to_new;
          end 6;

          begin comment if all three keys are equal to infinity, we
            have finished;
            if old.key = infinity then goto mergeend;
            alarmcall;
          end 7

        end actions;
        goto rep;

mergeend: ..... Put a correct end record into new, maybe check the
        end records of old and trans, close the zones properly.
      end;
```

If the number of transactions is not small compared with the number of records in old, the checkvar-call concluding action 3 should be moved so that it is performed just prior to the outrec-call in action 1. Note that in this algorithm the number of new records is not counted in the free zone parameter (see 9.27), as outvar is never called.

9.12. Procedure check

This procedure waits for and checks an answer from a transfer in exactly the same way as high level zone procedures check their transfers.

Call: check(z)

z (call and return value, zone). The operation given in used share of z (see 9.27, getzone6) is waited for and checked.

The algorithm is given in 6.3.2, wait transfer. Section 6.3.3 describes the standard error actions.

9.13. Integer procedure checkvar

This procedure calculates the record checksum of a record with the format of a variable length record as generated by outvar (see 9.48). The checksum is stored in the second word of the record. The procedure is intended for use in the very special cases where the checksum is destroyed or becomes invalid or where a checksum is needed later on.

Call: checkvar(z)

checkvar (return value, integer). The checksum which was stored in the record before call of checkvar.

z (call and return value, zone). Specifies the record for which the checksum must be calculated, and where it is stored.

Zone state:

The record length given in the first word of the record must be greater than or equal to 4 and equal to the record length of the zone descriptor (see 9.27). The zone state is irrelevant and unchanged.

No transfer is initiated by checkvar.

Example 1, simulating an end-record.

An end record may be generated in the block procedure when tapemark is sensed

```
procedure endfile(z,s,b);
  zone         z     ;
  integer          s,b ;
  if s extract 1 = 1 then stderror(z,s,b)
  else if b > 0 then
  begin integer array descr(1:20);
    integer field reclen,firstword;
    reclen:= 32; firstword:= 2;
    getzone6(z,descr);
    b:= descr.reclen:= z.firstword:= length;
    ..... set other parameters in the record;
    checkvar(z);
    setzone6(z,descr);
  end;
```

The zone should be opened with giveup mask 1 shift 16.

Example 2:

See example 2 of invar, 9.33.

9.14. Procedure close

Terminates the current use of a zone and makes the zone ready for a new call of open. Close may also release a device so that it becomes available for other processes in the computer.

Call: close(z,rel)

z (call and return value, zone). Specifies the document, the position of the document, and the latest operation on z.

rel (call value, boolean). True if you want the document to be released, false otherwise.

Close terminates the current use of the zone as described for setposition, 9.58. If the document is a magnetic tape which latest has been used for output (state 3 and 6, see getzone6, 9.27), a tape mark is written.

Finally, close releases the document if rel is true. Releasing means for a backing storage area that the area process description inside the monitor is released for use by other zones of yours. The area itself is not removed and you may later open it again.

In case of a magnetic tape, two kinds of release exist: If rel is true and the binary pattern is false add 1, the tape will be released, which means that the tape is not needed later in the run. Release of a work tape means that the tape is made available to other users. If rel is true with another binary pattern, the tape may be unmounted now (for instance if tape stations are sparse), but it will be needed later in the run. In both cases a message is sent to the parent asking for release or suspension of the tape.

Releasing means for other documents, that the corresponding peripheral device is made available for other processes.

Zone state

The zone may be in any state when close is called. After the call the zone is in state 4, after declaration meaning that it must be opened before it can be used for input/output again.

Example 1:

A backing storage area which you want to open more times should not be released, because that may allow other processes to remove it or output to it. Avoid it in this way:

```
open(master,4,<:bs52:>,0);
for ... do outrec6(master,     ...
close(master,false);
open(trans,4,<:bs52:>,0);
...
```

Example 2:

Let z1 and z2 be two zones which describe magnetic tapes. If you want to close them and rewind them, proceed in this way:

```
setposition(z1,0,0); setposition(z2,0,0);
close(z1,false); close(z2,false);
```

The rewindings are then performed in parallel and completed when close is called.

9.15. Real procedure cos

Call: cos(r)

cos (return value, real). The mathematical function cosine of the argument r. $-1 \le \cos \le 1$.

r (call value, real, long or integer). The argument in radians.

Accuracy:

abs(r) < $\pi/2$ gives a relative error below 1.2'-10

abs(r) ≥ $\pi/2$ To the relative error of 1.2'-10 must be added the absolute error of the argument, r*3'-11. This means that cos is completely undefined for abs(r) > 3'10, and then the result is always 0.

Example:

Let d be an angle in degrees. The cosine of d is then

```
cos(3.1415 9265 359/180*d)
```

9.16. Entier

This monadic operator transfers an expression of type real to the largest integer not greater than the real expression. The operation may cause integer overflow.

Syntax: entier <real> is of type integer
Priority higher than **.

9.17. Long procedure exor.

Performs the function 'exclusive or' on two 48 bit entities a and b. If bit patterns (see 3.1.6) of a and b are shorter than 48 bits, they are extended by repetition of the sign bit.

Call: exor(a,b)

exor (return value, long). Bit pattern equal to -, (a==b) performed bit by bit after possible extension of the parameters a and b.

a,b (call value, short string (text portion), real, long, integer or boolean). The two parameters do not have to be of the same kind. They are - if necessary - extended and they are handled as described below.

Handling of a and b according to kind:

String: It is tested that a string parameter describes a text portion or a short string (see 3.6.3). This is a 48 bit entity.

Real: A real is represented by 48 bits, no conversion.

Long: A long is represented by 48 bits, no conversion.

Integer: An integer is extended to a long as if the operator extend (see 9.19) had been applied.

Boolean: A boolean is considered as a short integer. The 12 bit boolean pattern is extended to a 48 bit long according to the algorithm

```
int:= boo extract 12;
if int > 2047 then int := int - 4096;
param:= extend int;
```

The rules for extension imply that actual parameters with values true, -1, and extend (-1) are equivalent. Note that the rules also imply that the effect of an integer with the value 2048 differs from the effect of a boolean with the value false add 2048.

Example:
In certain data transmission problems, a check character, which is a longitudinal parity check of a data block is needed. If the block is of more than 6 characters, the algorithm for finding the check character may look somewhat like this:

```
longfield:= firstword + 2;
checkword:= z.longfield;
for longfield:= longfield + 4 step 4 until lastword do
    checkword := exor(checkword,z.longfield);
if longfield - 4 <> lastword then
    checkword:= exor(checkword,z.lastword);
checkword:= exor(checkword,checkword shift (-24));
checkword:= exor(checkword extract 8, checkword shift (-8));
checkchar:= exor(checkword,checkword shift (-8)) extract 8;
```

9.18. Real procedure exp

Call: exp(r)

exp	(return value, real). The exponential function of the argument r, e**r.
r	(call value, real, long, or integer). r < 1000.

Accuracy:

r = 0	gives exp = 1.
r < -1000	gives exp = 0.
abs(r) < ln(2)/2	gives a relative error below 8.5'-11.
(n-0.5)*ln(2) <= abs(r) <= (n+0.5)*ln(2)	gives a relative error below 1.2'-10 + n*2'-11.

Alarm A value of r greater than 1000 will terminate the run.

9.19. Extend

This monadic operator operates on an integer expression and converts it into a 48 bit long.

Syntax: extend <integer> is of type long
priority higher than **

Example:
As operations on integers give integer values, an unwanted integer overflow may occur when two integers are multiplied. This may be avoided if the operator extend is applied on one of the operands.

```
totals:= extend pieces * price
```

This is of course only relevant if totals reasonably can exceed 8 000 000 and is a long.

9.20. External

This delimiter replaces the first begin of the program when an algol procedure is translated alone.

Syntax: external <procedure declaration>; end is a program.
A maximum of 7 parameters is allowed.

A procedure translated in this way becomes a standard procedure, which means that other algol programs may call the procedure without having to declare it. The name of the procedure is the name of the backing storage area in which it was translated. All standard identifiers used from the procedure must be present in the catalog when the procedure is translated, but the actual code determining these standard identifiers is not copied until the procedure itself is copied into an ordinary algol program.

The name of an external procedure may not contain capital letters, because they are forbidden in names of backing storage areas.

Example:

A standard function 'tg' may be compiled in this way:

```
tg=algol;    File processor commands, see ref. 2 and ref. 8.
external real procedure p(r); value r; real r;
begin real v;
  v:= cos(r);
  p:= if v <> 0 then sin(r)/v else '600
end; end
scope user tg; File processor command.
```

From another program it may be used like this:

```
write(out,(1+tg(B/2))/(1-tg(B/2)));
```

Assume that the procedures cos and sin are replaced with better versions. These new versions will automatically be used whenever tg is used during the translation of an algol program.

9.21. Extract

This dyadic operator is used for unpacking of integer values from a real, long, integer, or boolean value.

Syntax:
<real> extract <primary> is of type integer.
<long> extract <primary> is of type integer.
<integer> extract <primary> is of type integer.
<boolean> extract <primary> is of type integer.
Priority as **.

Extract treats the left hand operand as a binary pattern (see 3.1), the right hand primary is rounded to an integer if it is of type long or real (see 9.57) and now extract extracts a number of the rightmost bits as indicated by the value of the primary. These bits are extended with zeroes in front if necessary. The resulting value is the integer with these bits as its binary pattern. The result is undefined if the, possibly rounded, primary has a value below 0 or above 24.

Example 1, simple splitting.

A boolean may be split into two integers in this way:

```
i2:= b extract 6; i1:= b shift (-6) extract 6;
```

Both integers will be in the range 0 to 63.

Example 2, splitting with sign.

A real may be split into two signed integers in this way:

```
i1:= r shift (-24) extract 24; i2:= r extract 24;
```

Usually a signed integer is packed and split in this way

```
comment -32 <= i <= 31;
r:= r shift 6 add (i+32);
...
i:= r extract 6 - 32;
```

Example 3, splitting of text into characters.

A text string stored in the integer array ia may be split into a sequence of characters stored as integers in the array char in the following way:

```
comment c is the current index within char,
        s counts positions within ia(i);
s:= i:= 0;
for c:= 1,c+1 while ch <> 0 do
begin
  if s <> 0 then s:= s + 8 else
  begin s:= -16; i:= i + 1; t:= ia(i) end;
  char(c):= ch:= t shift s extract 8;
end;
```

A faster version, which always splits ia(i) into 3 characters even if one of them is the stop character (0), works like this:

```
comment c is current index within char, t contains ia(i);
t:= c:= -2; i:= 0;
for c:= c + 3 while t extract 8 <> 0 do
begin
  i:= i + 1; t:= ia(i);
  char(c):= t shift (-16) extract 8;
  char(c+1):= t shift (-8) extract 8;
  char(c+2):= t extract 8;
end;
```

Example 4, scaling of reals.

An array of reals may be scaled so that all elements are in the range $-1 < r < 1$ in the following way. The mantissas are not touched so that full accuracy is maintained. The main problem in the algorithm is the handling of the sign of the exponent.

```
max:= -2048;
for i:= 1 step 1 until n do
begin
  e:= ra(i) extract 12; if e >= 2048 then e:= e - 4096;
  if e > max then max:= e;
end;
comment max is now the maximal two's exponent;
for bf:= 4*n step -4 until 4 do
begin
  e:= ra.bf extract 12;
  if e >= 2048 then e:= e - 4096; e:= e - max;
  if e < -2048 then ra(bf shift (-2)):= 0
                else ra.bf:= false add e;
end;
```

The subscript expression bf shift (-2) is slightly faster and a little shorter in the translated code than bf//4.

9.22. Field

This delimiter is used to declare or specify field variables (see 5.7, 4.7.1, and 3.1), Field variables are pointers allowing reference to fields of various kinds (variable field and array field) within arrays or zone records.

The syntax and semantics are explained in sections 3.1.1, 3.1.3, 3.1.4.3, 3.1.5, 3.1.8, 4.2, 4.7.1, 4.7.5, 5.2.6, 5.2.7, 5.7, and the subsections. Here a more informal description will be given.

Field variable declaration and specification:

```
<type> field <field list>
<type> array field <field list>
array field <field list>
```

The declaration 'array field <field list>' declares real array field variables. A field list is a list of identifiers separated by commas.

Field variables are used in field references or they may be used as integer variables.

9.22.1. The field concept and Algol 6.

A file may be seen as a set of records and a record may be seen as a set of fields, where a field is the smallest entity which in some connection is considered as a unit of data. The terms field, record, and file can only get a meaning when they are defined together with a specific data set and the operations on it. Therefore there is no logical conflict in subdivision of the levels in the hierarchical relation

$$\text{field} \subseteq \text{record} \subseteq \text{file}.$$

The Algol 6 field concept is defined along these lines but it is extended in that it is applicable to arrays of any type as well as (zone) records. The Algol 6 fields may be variables of the types boolean, integer, long, or real, or they may be arrays of the same types. Array fields are one dimensional.

Please note the difference between the concepts 'field variable' meaning the pointer and 'variable field' meaning the piece of data pointed at.

Fields are selected by means of a byte-address within the array or record. The length of the selected field is defined by a type associated with the field variable. To use field variables it is necessary to know how elements are stored and represented in an array.

In a real or a long array each element takes up 4 bytes. In an integer array the elements take up 2 bytes each, and in a boolean array one byte per element is used. The number of bytes taken by an element is called the <u>type length</u>.

<u>Example 1:</u>

If a program contains the declarations

```
real array RA(1:3); long array LA(1:3);
integer array IA(1:5); boolean array BA(1:11);
```

then the byte numeration is according to this scheme:

RA(1)				RA(2)				RA(3)			
1	2	3	4	5	6	7	8	9	10	11	12
LA(1)				LA(2)				LA(3)			

IA(1)		IA(2)		IA(3)		IA(4)		IA(5)	
1	2	3	4	5	6	7	8	9	10

BA(1)	BA(2)	BA(3)	BA(4)	BA(5)	BA(6)	BA(7)	BA(8)	BA(9)	BA(10)	BA(11)
1	2	3	4	5	6	7	8	9	10	11

Δ

The general rule for the byte numeration, which applies to arrays with any number of indices and with lower bound <> 1 is explained in section 5.2.6. In short we have:

<u>The byte with number 0 (zero) is the last byte in the (possibly fictive) element with subscripts (0, 0, ... , 0).</u>

<u>Example 2:</u>

If a program contains the declaration

```
real array B(1:2,0:1),
```

the array may be sketched like this

lexicographical index

	0	1	2	3	4	5
	B(0,0)	B(0,1)	B(1,0)	B(1,1)	B(2,0)	B(2,1)
byte no.	... 0	1 2 3 4	5 6 7 8	9 10 11 12	13 14 15 16	17 18 19 20
	non existent		word		boundaries	

Fields of integer, real and long type are synchronized with the word boundaries of the RC 4000. For arrays which are not formal, (see 9.22.3) the word boundaries are between an even numbered byte and its odd numbered successor.

This implies that field variables with these types associated should preferably assume even values. If such a field variable is assigned odd values it will denote the same variable as it would if one had been added. But as odd plus odd is even, you may easily make an error if odd field references are used. This is especially the case when you use array fields as actual parameters in procedure calls (see further in section 9.22.3).

9.22.2. Field variables and field references.

Field variables are assigned values in the same way as integers. The value assigned is an integer. In fact field variables used outside field references behave as normal integers independent of the type associated.

Example 3:

```
    for balance:= 4 step 48 until 340 do
      sum:= A.balance + sum;
```

Associated with a field variable is - as already indicated - a type. The associated type may be real, long, integer, or boolean. A field variable may denote a variable or an array of the associated type lying within an array of any type or within a zone record.

In the field reference A.f, we refer to A as the field base and to f as the field variable.

The location of the field referenced by a denotation like this depends on

1) the value of f
2) the type associated with f
3) the lexicographic order of the elements in A

The byte to which a field reference points will be called the reference byte.

In field references simple field variables denote variables of the type associated with the field variable, and array field variables denote fields that are arrays. The type of the array field is the type associated with array field variable.

In a variable field reference the reference byte should be the rightmost byte in the field referenced.

Example 4:

If the program in example 2 contains the declarations

```
        integer field balance,fun;
        real field rate;
```

and the initialization

```
        balance:= 4; rate:= 20; fun:= 10;
```

then

B.rate refers to the real element B(2,1)

B.fun refers to the integer variable placed in the first half of B(1,1), i.e., the value of B.fun is the same as the value of B(1,1) shift (-24) extract 24.

B.balance refers to the non-existing integer element with byte numbers 3 and 4. Therefore the program with this reference will be terminated with field alarm.

A simple real or long field variable may point at a field consisting of two words from adjacent variables of the array. Thus if rate = 18, it refers to a real variable consisting of the last half of B(2,0) and the first half of B(2,1).

△

If A is an array of any type or a zone record, and if F is an array field identifier with any associated type, then

the reference A.F denotes a one dimensional array where the reference byte defines the - possibly fictive - element A.F(0).

The bounds of the array field are defined so that

bound byte in field = bound byte in base - field variable.

The bound bytes are the first and last bytes in the field. The bound bytes are not necessarily accessible by indexing.

Special rules apply when array field references are used as parameters to procedures. These rules are discussed in section 9.22.3.

Example 5:

If a program contains the declarations

```
long array A(1:3);
integer array field iaf;
```

then the assignment

```
iaf:= 0;
```

will define an integer array A.iaf with the same locations as A. The array A.iaf may be sketched like this:

	A.iaf(1)		A.iaf(2)		A.iaf(3)		A.iaf(4)		A.iaf(5)		A.iaf(6)	
0	1	2	3	4	5	6	7	8	9	10	11	12
	A(1)				A(2)				A(3)			

We see that if the long array is declared with lower bound 1, the integer array will also have its lower bound 1, when the field variable has the value 0. If we have the assignment

```
iaf := 6;
```

the integer array is allocated like this:

	A.iaf(-2)		A.iaf(-1)		A.iaf(0)		A.iaf(1)		A.iaf(2)		A.iaf(3)	
0	1	2	3	4	5	6	7	8	9	10	11	12
	A(1)				A(2)				A(3)			

The index bound for the integer array are as if it had been 'declared'

integer array A.iaf(-2:3).

We may use a field reference to define a field in an array field, for instance with iaf = 0, we may refer to the first half of A(2) by means of an integer field subtotals = 6. The reference looks like this

A.iaf.subtotals

With iaf = 6, this reference denotes the last half of A(3). △

9.22.3. Fields as parameters to procedures

Note that field variables may be used as actual parameters to procedures. They behave as integers. Formal parameters may be specified as field variables, but they must not be called by value. The actual must be integer. In the procedure body they act as field variables.

Variable fields as actual parameters are handled in the same way as subscripted variables. This means that if the corresponding formal is not called by value, the field will be evaluated each time the formal is referred (Jensen's Device).

Array fields are evaluated and a description of the array field as an array is set up before the procedure is entered. This description, local to the procedure is made so that references to the array parameter are just as effective as references to an array declared local in the procedure body.

If you restrict yourself to using actual array field references where the reference byte index is a multiple of the type length, and the field base is one dimensional with lower bound 1, you will hardly run into trouble.

Otherwise the formal array may be 'cut' in order to ease and secure index check in the procedure body. The 'cutting' is made so that the number of bytes between the reference byte of the array field and the first accessible byte of the formal array is a multiple of the type length. The term 'between' is to be understood so that

(address(ref byte) - address(lower bound byte) - 1) mod typelength = 0

is true.

Example 6:

Consider a program like this:

```
...
long array LA(1:2);
long array field laf;
procedure test (la);
  long array    la ;
  begin boolean field bf; integer i;
    ....
    ... la(i) ... la.bf ...
  end;
...
test(LA.laf);
...
```

For some values of laf, the accessible parts of the formal array la may be sketched like this

laf									interval of i	interval of bf
-4	5	6	7	8	9	10	11	12	2:3	5:12
-3		5	6	7	8	9	10	11	2:2	5:11
-2			5	6	7	8	9	10	2:2	5:10
-1				5	6	7	8	9	2:2	5:9
0	1	2	3	4	5	6	7	8	1:2	1:8
1		1	2	3	4	5	6	7	1:1	1:7
2			1	2	3	4	5	6	1:1	1:6
3				1	2	3	4	5	1:1	1:5
4	-3	-2	-1	0	1	2	3	4	0:1	-3:4

Bytes with equal locations are shown in the same column.

The reference byte numbers corresponding to indexing in la are underlined, and the bytes accessible by direct indexing are double overlined. The word boundaries are shown as lines going from line to line. Δ

In arrays which are actually fields, the word boundaries are only between an even numbered byte and its odd numbered successor, if the value of the field variable is even.

If an actual array in a procedure call is a multiple fielded array or record, only the type length associated with the last array field variable is used in a possible 'cutting' of the lower bound.

9.23. Procedure getposition

Gets the block and file number corresponding to the current logical position of a document.

Call: getposition(z,File,Block)

z	(call value, zone). Specifies the document, the position of the document, and the latest operation on z.
File	(return value, integer). Irrelevant for documents other than magnetic tape. Specifies the file number of the current logical position (see 6.1). Files are counted 0, 1, 2, ...
Block	(return value, integer). Irrelevant for documents other than magnetic tape and backing storage. Specifies the block number of the current logical position (see 6.1). Blocks are counted 0, 1, 2, ...

Getposition does not change the zone state and it may be called in all states of the zone. If the zone is not opened, the position got will be undefined, however. The position is also undefined after a call of close.

Example 1:
During the generation of a magnetic tape, you may note the position of a particular record and later return to that block:

```
outrec6(z,10);
getposition(z,f,b);
outrec6(z,10);
setposition(z,f,b); inrec6(z,10);
```

If you want to get the same record again, you may use getzone6 (see 9.27) to get the position within the block, or you may use the value of inrec6 or outrec6 to denote the position within the block.

9.24. Procedure getshare

Moves the contents of a share descriptor into an integer array for further inspection. The procedure is the Algol 5 version of getshare6.

Call: getshare(z,ia,sh)

z	(call value, zone). Specifies the share together with sh.
ia	(return value,integer array,length ≥ 12).
sh	(call value, integer). The number of a share within z. The contents of the share descriptor are moved to the first element of ia and on.

Works as getshare6 (see section 9.25) except that getshare computes first shared and last shared as a buffer index instead of as a byte index. The buffer index is equal to (byte index+3)//4.

9.25. Procedure getshare6

Moves the contents of a share descriptor into an integer array for further inspection. The procedure is designed for the primitive level of input-output, where you implement your own blocking strategy for the peripheral devices, and for use in the block procedures where you want to interfere with the standard handling of devices. Skip it if you are satisfied with the high level zone procedures.

A share descriptor consists of 12 pieces of information, most of them with names originating from their use in high level zone procedures. The explanation below requires some knowledge of handling of peripheral devices (see ref. 8).

The share descriptor contains certain absolute addresses of bytes within the zone buffer. The reason for this and the relation between the absolute address and the usual byte index are given for the procedure getzone6.

Call: getshare6(z,ia,sh)

z (call value, zone). Specifies the share together with sh.

ia (return value, integer array, length ≥ 12). The following list assumes that 'ia' has been declared as ia(1:12).

sh (call value, integer). The number of the share within z. The contents of the share descriptor are moved to the first element of ia and on.

ia(1) Share state. Describes what the share is used for:
= message buffer address for an uncompleted transfer or a stopping child process.
= -process description address for a running child process.
= 0 for a free share. See below.
= 1 for a ready share. See below.

ia(2) First shared. Byte index for the first element available for a block transfer which uses this share and was started by a high level zone procedure.

ia(3) Last shared. Byte index for the last element available for a block transfer which uses this share and was started by a high level zone procedure.

ia(4) to ia(11) Message. A high level zone procedure leaves the latest message sent by means of this share in the message part of the share descriptor. A message describing a block transfer is composed like this,

ia(4) operation shift 12 + mode
ia(5) first absolute address of block
ia(6) last absolute address of block
ia(7) segment number (only significant for backing storage)

ia(12) Top transferred. The absolute address of the byte just after the latest block transferred by means of this share. Top transferred may differ from ia(6) + 1 after an input operation, for instance.

Free and ready share

The output procedures do not distinguish between a free and ready share, but whenever an input procedure tries to get a new block of information, it assumes that a ready share contains a block of information already and that a free share must be filled with a block from the device.

Example 1:

Let z be declared as z(300,3,stderror) with base buffer area = 29 999, (see definition in 9.27) and assume that you have opened the zone. The calls getshare6(z,ia,1), getshare6(z,ia,2), and getshare6 (z,ia,3) will now yield the following results in typical situations (X designates an undefined value):

	ia(1)	ia(2)	ia(3)	ia(4)	ia(5)	ia(6)...	ia(12)
When the first block of input is being processed:							
used share	0	1	400	input	30 000	30 398	30 276
share2	>0	401	800	input	30 400	30 798	X
share3	>0	801	1200	input	30 800	31 198	X
When the first block of output has been produced:							
share1	>0	1	400	output	30 000	30 350	X
used share	0	401	800	X	30 400	30 798	X
share3	0	801	1200	X	30 800	31 198	X
Just after setposition for a magnetic tape:							
used share	>0	1	400	move	position	30 398	X
share2	0	401	800	X	30 400	30 798	X
share3	0	801	1200	X	30 800	31 198	X

9.26. Procedure getzone

Moves the contents of a zone descriptor into an integer array for further inspection. The procedure is the Algol 5 version of getzone6 and works as getzone6 except that the record length is given in buffer elements instead of bytes.

A buffer element consists of 4 bytes.

Last byte of a buffer element, the reference byte, has the absolute address:

base buffer area + 4 * buffer index.

Call: getzone(z,ia)

z	(call value, zone). The contents of the zone descriptor are moved to the first element of ia and on.
ia	(return value, integer array, length $\geq$ 20).

Getzone should only be used if the zone has only been used for inrec, outrec or swoprec. As it cuts the record length to an integral number of elements, it may give misleading results if the Algol 6 procedures inrec6, outrec6, swoprec6, changerec6, invar, outvar, or changevar have been used.

For further description see 9.27, getzone6.

9.27. Procedure getzone6

Moves the contents of a zone descriptor into an integer array for further inspection. The procedure is designed for the primitive level of input-output, where you implement your own blocking strategy for the peripheral devices, and for use in the block procedures where you want to interfere with the standard handling of the devices. Skip it if you are satisfied with the high level zone procedures.

A zone descriptor consists of 20 pieces of information, most of them with names originating from their use in high level zone procedures.

The zone buffer is just a sequence of real variables - from the point of view of the algol program - but other processes (peripheral devices, etc.) regard it rather as a sequence of bytes, each being identified by its absolute address.

If you want to communicate with other processes on the very primitive level (procedure monitor), you cannot avoid the absolute addresses. They are related to the usual byte index in this way:

The reference byte of a field in the zone has the absolute address:
base buffer area + byte index.

This expression also defines the quantity 'base buffer area' as the absolute address of the byte preceding the zone buffer area. The value of 'base buffer area' and certain other byte addresses are available by means of getzone6.

Call: getzone6(z,ia)

z (call value, zone). The contents of the zone descriptor are moved to the first element of ia and on.

ia (return value, integer array, length ≥ 20). The following list assumes that ia has been declared as ia(1:20).

ia(1) Mode shift 12 + kind. Values and significance are explained under the procedure open.

ia(2) to ia(5) Process name. The name of the process (document) with which the zone communicates for the moment. The name is extended to 12 characters using null characters for fill.

ia(6) Name table address. The corresponding variable in the zone descriptor is used by the monitor to speed up the search for the process given by the process name.

ia(7) File count. Only significant for magnetic tape handling. See explanation below.

ia(8) Block count. Only significant for magnetic tape handling. See explanation below.

ia(9) Segment count. Only significant for handling of backing storage areas. See explanation below.

ia(10) Give up mask. See 6.3.

ia(11) Free parameter. Is used by the Fortran read/write system and by the var-procedures. See explanation below.

ia(12) Partial word. Used by the procedures for input-output on character level to unpack or pack characters. See explanation below.

ia(13) Zone state. Used by high level zone procedures to keep track of the latest operation on the zone. See below.

ia(14) Record base. The absolute address of the byte preceding the first byte of the present record. During character input or output the record may be regarded as the word in the zone buffer in which the partial word will end or from which it came.

ia(15) Last byte. Absolute address of the last byte of current block. During output the block matches the shared area used for the moment, during input the block matches the block transferred from the device.

ia(16) Record length. Number of bytes in the present record. Notice that the record length is 0 during character input or output.

ia(17) Used share. Number of a share within z. Used share will in high level zone procedures be the share in which items are stored for the moment or from which they are fetched.

ia(18) Number of shares. The value given in the zone declaration.

ia(19) Base buffer area. See above.

ia(20) Buffer length. The value given in the zone declaration, i.e. measured in double words.

File count, block count

In the high level zone procedures of algol the two variables, file count and block count, are used in two ways: When a tape positioning is initiated, file and block count denote the wanted final position. When a block transfer has been checked, file and block count denote the physical position corresponding to the end of that block.

Segment count

The current value of segment count is used as the 4th word of every message sent to a device by the high level zone procedures. It will only have significance when the message is sent to a backing storage process, however. As soon as the message is sent, segment count is updated to correspond to a transfer of the next block from the backing storage.

Free parameter

The so called free parameter may contain anything if the zone is not used by the Fortran read/write system or by the procedures changevar, invar and outvar. It is set to zero when the zone is declared. The var-procedures use this parameter as a counter of logical records generated or read by the procedures. The var-procedures are described in the sections 9.11, 9.33 and 9.48. The Fortran read/write system uses the last bit of this parameter to signal if the latest call of read or write used format or format0. A one in the last bit means that format0 was used and a null means that format was used. See ref. 9 for further details.

Partial word

One element of the zone buffer consists of two words. Each of the words contains 3 characters like this: ch1 shift 16 + ch2 shift 8 + ch3. Partial word may after the call of a procedure on the character level contain this:

After input:	After output:
ch2 shift 16 + ch3 shift 8 + 1	1
ch3 shift 16 + 1 shift 8	1 shift 8 + ch1
1 shift 16	1 shift 16 + ch1 shift 8 + ch2

Zone state

The action of a high level zone procedure will in general depend on the latest operation upon the same zone.

zone state = 0 positioned after open.
1 after character reading.
2 after repeatchar.
3 after character printing.
4 after declaration.
5 after record input.
6 after record output.
7 after record swop.
8 after open on magnetic tape,
≥ 9 after some procedures not described in this manual.

The procedure setposition expects the zone state to be 0, 1, 2, 3, 5, 6, 7, or 8 and leaves the zone state = 0.
The procedure open expects the zone state to be 4 and leaves the zone state = 0 or 8.
The procedure close leaves the zone state = 4.
The procedures inrec, inrec6, and invar expect the zone state to be 0 or 5 and leave the zone state = 5.
The procedures outrec, outrec6, and outvar expect the zone state to be 0 or 6 and leave the zone state = 6.
The procedures swoprec and swoprec6 expect the zone state to be 0 or 7 and leave the zone state = 7.
The procedures read, readall, readchar, and readstring expect the zone state to be 0 or 1 and leave the zone state = 1.
The procedures write, outchar, outinteger and outtext expect the zone state to be 0 or 3 and leave the zone state = 3.

Example 1:
Let z be declared as z(2*128,2,stderror) and opened as the backing storage area <:sldata15:>. After 130 calls of 'outrec6(z,4)' the call getzone6(z,ia) will yield something which only depends on the value of base buffer area:

variable	contains
ia(1),modekind	4
ia(2)-ia(5), process name	<:sldata15:>,0
ia(6),name table address	Some address
ia(7),file count	0
ia(8),block count	0
ia(9),segment count	1(prepared for output of the next segment)
ia(10),give up mask	As defined by open
ia(11),free parameter	0
ia(12),partial word	1
ia(13),zone state	6(after outrec6)
ia(14),record base	30 515(base buffer + 4*128 + 4)
ia(15),last byte	31 023(base buffer + 4*256)
ia(16),record length	4
ia(17),used share	2(one block output already)
ia(18),number of shares	2
ia(19),base buffer area	29999
ia(20),buffer length	256

Example 2, character output to core store:

Numbers may be transformed to character form by means of write. The only problem is that you do not want to output the characters on a device, but rather keep them in long variables as text portions. This is possible by means of getzone6,setzone6.

```
      begin zone convert(10,1,stderror); integer array ia(1:20);
      open(convert,0,<:dummy:>,0);
rep:  write(convert,<<ddd.dd'dd>,the number to be converted,false,2);
      comment the partial word has been forced into the buffer  by the
              2 null characters;
      getzone6(convert,ia);
      ia(12):= 1; ia(14):= ia(19); ia(16):= 40;
      setzone6(convert,ia);
      comment Now  the  record  contains the number in character form.
              Record  base  and  partial word are ready for converting
              the next number;
      x1:= long convert(1); x2:= long convert(2); ...
      goto rep;
```

Example 3, improved setposition

The procedures getposition, setposition do only enable a device to be positioned at the beginning of a block. You may reasume reading from the middle of a block on magnetic tape or backing storage in this way:

```
comment generalised getposition;
read(z,...);
getposition(z,pos1,pos2); getzone6(z,ia);
comment pos3 is the relative position within the used share;
pos3:= ia(14) - ia(19) - ia(20)*4//ia(18)*(ia(17) - 1);
pos4:= ia(12);

comment generalised setposition,  perhaps  with  the device con-
        nected to another zone;
setposition(z1,pos1,pos2); readchar(z1,c);
comment now the device is positioned  and  the  first  block  is
        read into the first share;
getzone6(z1,ia);
ia(14):= pos3 + ia(19); ia(12):= pos4; setzone6(z1,ia);
read(z1,...);
```

9.28. In

The standard identifier 'in' is a preopened zone available for input on character level. The actual file connected to the zone is determined by the file processor command which started the program (see App. B).

The call <program> will let 'in' be the current input file of the file processor. The call <program><text file> will let 'in' be the file <text file>. The call <program><integer> makes 'in' unavailable (but frees some space in the job area).

When the program terminates the latest operation on 'in' must have been a call of a character reading procedure.

9.29. Integer procedure increase

Used in connection with a variable text as parameter to write, open, etc.

Call: increase(i)

increase (return value, integer). The procedure performs:
increase:= i; i:= i + 1;
but i is only evaluated once.

i (call and return value, integer).

Example: See example 2 of string.

9.30. Integer procedure inrec

This is the Algol 5 version of inrec6. Inrec gets a sequence of elements of 4 bytes each from a document and makes them available as a zone record.

Call: inrec(z,length)

inrec (return value, integer). The number of elements each of 4 bytes left in the present block for further calls of inrec.

z (call and return value, zone). The name of the record. Determines further the document, the buffering, and the position of the document (see 6.1).

length (call value, integer, long, or real). The number of elements of 4 bytes each in the new record. Length must be ≥ 0.

For further description see 9.31, inrec6.

Inrec may be used with advantage, if the document is considered to contain reals.

Example:

Records of variable length may be handled in the Algol 5-way by means of inrec and outrec, but you should be careful: For magnetic tapes the record length should be checked in the block procedure to make sure that they match the block length. For backing storage areas the unused elements at the block end must be skipped (outrec clears them).

Suppose the record length is stored as the first element of the record. The record may then be fetched in this way for all devices:

```
rep: remaining:= inrec(z,1); length:= z(1);
     if length <= 0 then
     begin inrec(z,remaining);
       comment unused elements are skipped;
       goto rep
     end;
     inrec(z,length - 1);
```

Another solution is to call the block procedure after all normal answers and let it adjust or check the length. Note that the relation length = 0 instead of length $\leq$ 0 would not work because a backing storage area is filled up with binary zeroes (cf. 3.4.7 and 3.1.6).

9.31. Integer procedure inrec6

Gets a sequence of bytes from a document and makes them available as a zone record. The document may be scanned sequentially by means of inrec6, because the next call of inrec6 gets the elements just after those got now.

Call: inrec6(z,length)

inrec6	(return value, integer). The number of bytes left in the present block for further calls of inrec6.
z	(call and return value, zone). The name of the record. Determines further the document, the buffering, and the position of the document (see 6.1).
length	(call value, integer, long, or real). The number of bytes in the new record. Length must be ≥ 0. If length is odd, 1 is added to the call value.

Zone state.

The zone z must be open and ready for record input (state 0 or 5, see 9.27), i.e. the zone may only have been used by inrec6, invar or the like since the latest call of open or setposition. To make sense, the document should be an internal process, a backing storage area, a typewriter, a paper tape reader, a card reader, or a magnetic tape. In the latter case setposition(z,...) must have been called after the call of open(z,...).

Blocking

Inrec6 may be thought of as transferring the bytes just after the current logical position of the document and changing the logical position to after the last byte of the record.

However, all bytes of the record are taken from the same block, so if the record cannot be taken from the current block, the block is changed as described in 6.3. Then the record becomes the first bytes of that block, but if it still cannot hold the record the run is terminated (empty blocks are completely disregarded).

Records of length 0 need a special explanation: if not even a single word is left in the block, the block is changed and the logical position points to just before the first word of the new block.

Note that inrec6 changes the blocks in such a way that a portion at the end of a block may be skipped. So be careful to read a backing storage area with the same share length as that with which it was written, otherwise, wrong portions might be skipped at reading.

Example 1:

A simple scan of a file on a magnetic tape in double buffer mode may be programmed in this way (all records are assumed to be of 20 bytes):

```
begin zone file(2*128,2,endfile);
procedure endfile(z,s,b); zone z; integer s,b;
if s extract 1 = 1 then stderror(z,s,b) else
if b > 0 or s shift (-18) extract 1 = 1 then goto endscan;
```

```
          open(file,18,<:mt600304:>,1 shift 18+1 shift 16)
          setposition(file,1,0); comment skip the label in file 0;
     rep: inrec6(file,20);
          y:= long file(1) + file.intf;
          goto rep;
endscan:  close(file,true);
```

The scan is terminated by the procedure endfile which is called at tape mark (1 shift 16), end of tape (1 shift 18), and all hard errors. After the positioning (but before the first input operation) end file may be called with tape mark indication. In this case however, b = 0, while b > 0 after input of a tape mark.

The same piece of code would work for an area on the backing store if the file was generated with a share length of 128 elements of 4 bytes and if the second and third parameter to open were changed.

Example 2:

Two files of 100 byte records on magnetic tape are arranged in ascending order (sorted with respect to the key indicated by the integer field keyf). They may be merged into one file in this way:

```
begin zone result(2*256,2,stderror);
  zone array in(2,2*256,2,endfile);
  procedure endfile(z,s,b); zone z; integer s,b;
  if s extract 1 > 0 then stderror(z,s,b) else
  begin b:= 100; z.keyf:= large;
    comment the procedure simulates the presence of a record
            with a very large key;
  end;

    open(in(1),...,1 shift 16); ... setposition ...
    large:= (-1) shift (-1);
    inrec6(in(1),100); inrec6(in(2),100);
    for k:= if in(1).keyf < in(2).keyf then 1 else 2
    while in(k).keyf < large do
    begin
      outrec6(result,100);
      tofrom(result,z(k),100);
      inrec6(in(k),100);
    end;
    close(result, ...);
```

Example 3, block reading.

You may read a magnetic tape file or backing storage area block by block in this way:

```
for b:= inrec6(z,0) while b > 2 do
begin
comment b is now the block length in bytes the standard ac-
tions simulate one word containing <:<25><25><25>:> at tapemark
and end of area;
inrec6(z,b);
comment the block is now available as one record;
... ;
end;
if z.firstword <> long <:<25><25><25>:> shift (-24) extract 24
then error;
```

9.32. Procedure intable

Exchanges the current input alphabet used by all the read procedures on character level.

Call: intable(alpha)

alpha (call value, 0 or an integer array of one dimension). A zero signals that the standard alphabet be used. An integer array contains the new alphabet in table form as described below.

1. alpha is an integer array:

The actual contents of alpha are used in all calls of read procedures until a new alphabet is selected. This means that any change in the contents of alpha may have effects on the character reading. If a read procedure is called at a place where alpha is undeclared, an undefined alphabet is used.

To each character 'c' delivered by the peripheral device is associated a Class and a Value, determined by the read procedures in this way:

alpha(c+table_index) = Class shift 12 + Value extract 12

Class is an integer, $0 \le Class \le 4095$. Value is an integer, $-2048 \le Value \le 2047$. The character 'c' is an integer, $0 \le c \le 255$. The ISO characters utilize only half of this interval. The standard integer 'table_index' is normally 0, but you may use it to modify the alphabet.

The class determines how the value corresponding to a character is handled:

Class = 0,	blind: The character is skipped by all read procedures.
Class = 1,	shift character: The value is assigned to table_index and the character is looked up again in the alphabet to determine Class and Value.
Class = 2,	digits: The character is a decimal digit the value of which is Value - 48. To make sense, $48 \le Value \le 57$ should be fulfilled.
Class = 3,	signs: The character is the sign of a decimal number. Value = 43 means +, Value = 45 means -.
Class = 4,	decimal point: The character may be used as a decimal point.
Class = 5,	exponent mark: The character may be used as the ' of Algol.
Class = 6,	letters: The character may be used as part of a text but not as part of a number.
Class = 7,	delimiter: The character cannot be part of a text or a number.
Class = 8,	terminator: The character is a delimiter as class 7, but it will terminate a call of readall. If value is 25, it will immediately terminate a call of read or readstring.
Class > 8,	other delimiters: The character is handled as class 7.

2. Alpha is 0:

The standard alphabet given in section 2.0.1 is used until a new alphabet is selected. The value of table_index has no influence on the alphabet. When the run starts, the standard alphabet is selected automatically.

You should not hesitate to use a special alphabet table: The character reading will be speeded up compared to what you could do in algol with the standard alphabet, and the input algorithm becomes clearer. There are two drawbacks: 1) The table takes space, but remember that 2*128 integers correspond to one segment of a program (10 to 20 lines), and that much is easily saved in the central loop of the input program. 2) The table is cumbersome to initialise (even with the method of example 2 below), but we believe that is inevitable. In many cases it will be an advantage to make a procedure which initialises the table with a standard alphabet and then add modifications to this table.

Example 1, number variants:

Assume you want to read numbers coded in ISO form but with space regarded as blind information and without exponent part. You may then proceed like this:

```
comment initialise table with the ISO alphabet;
table(32):= 0; table(39):= 7 shift 12 + 39; ...
comment define space, apostrophe, and all other characters;
intable(table); table_index:= 0;
read(z,...);
```

Example 2, flexowriter conversion.

It is possible to use the read procedure for input represented in flexowriter code if the underlining may be disregarded. The shift characters, class 1, may take care of the case shift characters. An alphabet table of 2*128 elements is required. One way of initialising the table goes like this:

```
for i:= 0 step 1 until 255 do table(i):=
(case i+1 of     (0,2,2,2,2, 2,2,2,2,2,          CLASS
                  0,8,8,6,0, 0,2,7,6,6,
                  ...) shift 12 +
(case i+1 of     (32,49,50,51,52, 53,54,55,56,57,
                  0,12,25,125,0, 0,48,60,115,116,
                  ...             ));
```

Upper case and Lower case require

```
table(60):= 1 shift 12 + 128;
table(60 + 128):= 0;
table(58 + 128):= 1 shift 12 + 0;
table(58):= 0;
```

Note, that if the input was flexowriter paper tapes which were read in ISO-mode, the parity hole would not be the flexowriter parity hole, and as a consequence a different alphabet table would be needed.

Example 3: See example 3 of readall.

9.33. Integer procedure invar

This procedure together with outvar, changevar, and checkvar are intend for easy handling of records of variable length. Every record must contain its own length in bytes in its first word, the length word. Invar makes the next record written by means of outvar available as a zone record. A record checksum in the second word may be checked, and the number of records are counted in the so called free parameter in the zone descriptor (see 9.27). This procedure may call the block procedure with the status 1 shift 11, checksum error, if the record length wanted is < 4 or > remaining bytes in the block or odd or if the checksum is calculated and not equal to the value of the second word in the record.

Call: invar(z)

invar (return value, integer). The number of bytes left in the present block.

z (call and return value, zone). The name of the record. Determines the document, the buffering, and the position of the document (see 6.1).

Zone state.

The zone z must be open and ready for record input (state 0 or 5), i.e. the zone may only have been used by invar or the like since the latest call of open or setposition. The free parameter (see 9.27) in the zone descriptor is used to count the number of records accepted by invar. The value of this parameter is interpreted as check_wanted shift 23 + record count where check_wanted = 1 means that a checksum is calculated by invar and checked against the second word in the record. See below for further details.

Blocking

You may think of invar in the way that the procedure tastes the value of the first word just after the current logical position of the document. Now invar exposes as many bytes as the length word indicates, including the two bytes of this word.

However all bytes must be taken from the same block. If this is not possible, the block procedure of the zone is called. See further on length errors below.

If the length word is null, it is skipped and the next word from the document is tried as length word. When there are no more in a block, the block is changed. This covers skipping of blank block tails that may be generated by outvar when the kind of the document is backing storage (see 9.48).

Length errors. Checksum

If the length word is <> 0, it is expected to be even, >= 4 and <= the bytes remaining in the present block. If not all three conditions are fulfilled, invar will give up and call the block procedure (see below).

When the length word has passed the tests above, the contents of the second word may be tested as a check sum of the record. If check is wanted (see zone state above), invar tests if the sum of all words in the record taken modulo 2**24 is equal to -3. If not, invar calls the block procedure.

Block procedure, call conditions.

Invar may call the block procedure in two different situations:

a) The length word is not sensible (see above).
b) Record sumcheck is wanted, and the sum is not ok (see above).

The call conditions for the parameters to the block procedure are:

z: The zone state is after record input. The defect record is not counted in the free parameter. The record starts just before the length word and depends on the length word like this:

```
record length:=
  if length word < 4 or length word > remaining then
  remaining else if recordlength is odd then lengthword
  + 1 else length word.
```

Remaining means the number of bytes remaining in the present block including the length word. The terms zonestate, free parameter, and record length is explained in 9.27, getzone6.

s: The status word parameter has the value, 1 shift 11.

b: The bytes transferred parameter is equal to the record length, described above.

After return from the block procedure, invar restarts its algorithm by fetching the next logical record. A defect record will thus be skipped if the block procedure simply ignores the call.

Example 1:

Your block procedure may test whether situation a) or situation b) above has caused the block procedure to be called. This may be done as follows:

```
procedure blpr(z,s,b); zone z; integer s,b;
begin
  .....
  if s = 1 shift 11 then
  begin integer field lengthword; lengthword:= 2;
    if b < 4 or b <> z.lengthword then
    begin comment length error;
    end
  else
    begin comment checksum error; ...
    end;
  end
  .....
end;
```

Example 2, attempt to repair a defect record.

When you read a file from magnetic tape written by means of outvar, you may try to make sense of blocks with parity error and where the standard actions have given up.

This will only be waste of machine power if all records are needed in a run. In such case it is better to give up once status errors occur. It must be recognized, however, that problems exist where it is essential to make as much sense out of a file as possible in one run and then try to pick up the defect records in a later run.

A block procedure which counts the number of wrong 'records' and only gives up when this number is too large may look something like this:

```
          procedure afterparity(z,s,b); zone z; integer s,b;
          begin own integer faults; integer field length;
            length:= 2;
            if s = 1 shift 11 then
            begin
              faults:= faults + 1;
              if faults > max then stderror(z,s,b);
              if b <> z.length then
    drop:     write(out,<:record dropped expected::>,z.length,
                <: dropped::>,b,<: bytes<10>:>)
              else
              begin comment maybe only checksum error;
                if b < min length or b > max length then goto drop;
                .... now check the contents of the possible record if it
                does not seem to be sensible then drop it else set a mark
                that it may be erroneous and
                checkvar(z); changerec6(z,0);
                comment force a new checksum into the record and regret
                the record so that invar may take it once more;
              end;
              end
              else if logand(s,-1-(1 shift 22 + 1 shift 15 + 3) <> 0
              then stderror(z,s,b);
              comment give up if hard error except in connection with
              parity error, ring indication and normal answer;
            end;
```

When the zone with this block procedure is opened, the give up mask should not contain the parity error bit, as the standard action, 5 re-readings, is wanted for parity error. The bit for checksum wanted should be set in the free zone parameter (see getzone6, 9.27):

```
          open(z,18,<:...:>,0);
          setposition(z,1,0);
          getzone6(z,ia); ia(11):= 1 shift 23;
          setzone6(z,ia);
    rep:  invar(z);
          ..... handle the record, note the error-mark;
          goto rep;
```

Example 3:

See example of changevar.

9.34. Real procedure ln

Call: ln(r)

ln	(return value, real). The Napierian logarithm of r. r > 0.
r	(call value, real, long, or integer).

Accuracy:

r = 1	gives ln = 0
0.5 < r < 2	gives absolute error below 2.2'-10
0.25 ≤ r ≤ 0.5 or 2 ≤ r ≤ 4	gives relative error below 1.8'-10
r < 0.25 or 4 ≤ r	gives relative error below 1.2'-10

Alarm: The run is terminated if r ≤ 0.

9.35. Long procedure logand

Performs the function logical and (logical multiplication) on two 48 bit entities a and b. If the type length of a and/or b is smaller than 48 bits, they are extended by repetition of the sign bit.

Call: logand(a,b)

logand	(return value, long). Bitpattern equal to (a and b) performed bit by bit after a possible extension of the parameters a and b.
a,b	(call values, short string (text portion), real, long, integer, or boolean). The two parameters do not have to be of the same kind. They are - if necessary - extended and they are handled as described below.

Handling of a and b according to kind:

String:	It is tested that a string parameter describes a text portion or a short string (see 3.6.3). This is a 48 bit entity.
Real:	A real is represented by 48 bits. No conversion.
Long:	A long is represented by 48 bits. No conversion.
Integer:	An integer is extended to a long as if the operator extend (see 9.19) had been applied.
Boolean:	A boolean is considered as a short integer. The 12 bit boolean is extended to a 48 bit long according to the algorithm:

```
int:= boo extract 12;
if int > 2047 then int:= int - 4096;
param:= extend int;
```

The rules for extension imply that actual parameters with the values true, -1, and extend (-1) are equivalent. Note that the rules also imply that the effect of an integer with the value 2048 differs from the effect of a boolean with the value false add 2048.

Example:
See example 2 of invar.

9.36. Long procedure logor

Performs the function logical or (logical addition) on two 48 bit entities a and b. If the type length of a and/or b is smaller than 48 bits, they are extended by repetition of the sign bit.

Call: logor(a,b)

logor (return value, long). Bit pattern equal to (a or b) performed bit by bit after a possible extension of the parameters.

a,b (call values, short string (text portion), real, long, integer, or boolean). The two parameters do not have to be of the same kind. They are - if necessary - extended and they are handled as described for logand.

9.37. Long

This operator changes the type of a string expression or a real primary to type long. The binary pattern of the operand is unchanged. Note that this use of the delimiter long is totally different from its use in a declaration or specification.

Syntax: long <string> is of type long
long <real> is of type long
Priority higher than **

The binary pattern of a string is described in 3.6.5. The binary pattern of a real is described in 3.1.6.

9.38. Message

This delimiter may print a message during the translation of a program.

Syntax: message follows the same rules as comment.

The text between message and semicolon is printed on current output if message.yes was used in the translation parameter list (see app. B).

Example:
You can save the listing of a long algol program and still keep track of the line numbers. Put 1 or 2 messages on each page of the program (for instance as page head) and translate it with: algol message.yes. The messages are then printed with their line numbers attached and you can easily find any other line given its line number.

9.39. Mod

This dyadic operator yields the remainder corresponding to an integer division.

Syntax: <integer> mod <integer> is of type integer.
<long> mod <integer> is of type long.
<integer> mod <long> is of type long.
<long> mod <long> is of type long.
Priority as //.

The value of i mod j is defined as

```
i - i//j*j
```

Note that the sign of i mod j is the same as the sign of i.

Example, cyclical counting.
Counting i = 1,2,3,1,2,3,1,... may be done in this way:

```
i:= i mod 3 + 1;
```

A longer but slightly faster version is:

```
i:= if i = 3 then 1 else i + 1;
```

9.40. Integer procedure monitor

This procedure is the algol equivalent of the monitor procedures. You may use it to handle peripheral devices in a non-standard way and to program operating systems and executive functions in algol.

In most cases the algol procedure will only transform the parameters to the form required by the monitor, and the description below describes mainly this transformation. You will have to consult the manual of the multiprogramming system (ref. 1) and the monitor 3 manual (ref. 5) for the details and the ideas behind each entry.

Call: monitor(fnc,z,i,ia)
monitor (return value, integer). In most cases the result of the corresponding call of a monitor procedure.

fnc (call value, integer). A function code specifying the monitor procedure to be called.

z (call and return value, zone). The zone descriptor contains in most cases the name of the process or catalog entry concerned.

i (call and return value, integer). Used for various purposes, e.g. device number, message buffer address.

ia (call and return value, integer array). Used for various purposes, e.g. tail of catalog entry, contents of answer. Various lengths of ia are required in the various cases.

Certain of the procedures are only applicable to system 3. They are marked with (sys. 3).

In most cases only some of the last 3 parameters are actually used by the procedure. The value of fnc determines always the function as follows:

fnc = 4, process description:

monitor	result, i.e. process description address or 0.
z	(call value). Contains the process name.

fnc = 6, initialise process:

monitor	result, i.e. 0 means process initialised, 1,2,3 means not initialised.
z	(call value). Contains the process name.

fnc = 8, reserve process:

monitor	result, i.e. 0 means process reserved, 1,2,3 means not reserved.
z	(call value). Contains the process name.

fnc = 10, release process:

z	(call value). Contains the process name.

fnc = 12, include user:

monitor	result, i.e. 0 means included, 2,3,4 means not included.
z	(call value). Contains the process name.
i	(call value). Device number.

fnc = 14, exclude user:

monitor	result, i.e. 0 means excluded, 2,3,4 means not excluded.
z	(call value). Contains the process name.
i	(call value). Device number.

fnc = 16, send message:

monitor	buffer address, 0 if the buffer claim is exceeded.
z	(call value). Contains the process name.
i	(call value). The number of a share within z. The share state must at call time be 0 or 1, at return time it is the buffer address. The message sent is given in the share descriptor. (See 9.25, getshare6). Note that you may change the message in the share by means of the procedure setshare6, 9.60.

fnc = 18, wait answer:

monitor	result, i.e. 1 means a normal answer, 2,3,4,5 means dummy answers.
z	(call value). Determines together with 'i' the buffer address.
i	(call value). The number of a share within z. The share state must be the buffer address at call time, at return time it is 0.
ia	(return value, length $\geq$ 8). The answer is stored here.

fnc = 20, wait message:

monitor	result, i.e. positive for a normal message, negative for a message from a removed process.
z	(return value). The process name is stored here.
i	(return value). Buffer address.
ia	(return value, length $\geq$ 8). The message is store here.

fnc = 22, send answer:

i	(call value). Buffer address.
ia	(call value, length $\geq$ 9). The first 8 elements contain the answer, the 9th element contains the result.

fnc = 24, wait event:

monitor	result, i.e. 0 for a message, 1 for an answer.
z	(return value). The name of the sending process is stored here if a message was received.
i	(call and return value). Last and next buffer address.
ia	(return value, length $\geq$ 8). If a message is received, it is stored here.

An event may either be a message sent to the job or an answer described as the share state of some zone (possibly 'in' or 'out').

fnc = 26, get event:

i	(call value). Buffer address pointing to a message. An answer cannot be released in this way - use wait answer instead.

fnc = 40, create entry:

monitor	result, i.e. 0 means entry created, 1,2,3,4,5,6 means entry not created.
z	(call value). Contains the entry name.
ia	(call value, length $\geq$ 10). Contains the tail of the entry.

fnc = 42, lookup entry:

monitor	result, i.e. 0 means entry looked up, 2,3,6 means not looked up.
z	(call value). Contains the entry name.
ia	(return value, length $\geq$ 10). The tail of the entry is stored here.

fnc = 44, change entry:

monitor	result, i.e. 0 means changed, 1,2,3,4,5,6 means entry not changed.
z	(call value). Contains the entry name.
ia	(call value, length $\geq$ 10). Contains the new tail of the entry.

fnc = 46, rename entry:

monitor	result, i.e. 0 means entry renamed, 1,2,3,4,5,6 means entry not renamed.
z	(call value). Contains the present entry name.
ia	(call value, length $\geq$ 4). Contains the new entry name.

fnc = 48, remove entry:

monitor	result, i.e. 0 means entry removed, 1,2,3,4,5,6 means entry did not exist or entry is not removed.
z	(call value). Contains the entry name.

fnc = 50, permanent entry:

monitor	result, i.e. 0 means entry made permanent, 1,2,3,4,5,6 means entry not permanent.
z	(call value). Contains the entry name.
i	(call value). Catalog key.

fnc = 52, create area process:

monitor	result, i.e. 0 means area process created, 1,2,3,4,6 means process not created.
z	(call value). Contains the process name.

fnc = 54, create peripheral process:

monitor	result, i.e. 0 means process created, 1,2,3,4,5,6 means process not created.
z	(call value). Contains the process name.
i	(call value). Device number.

fnc = 56, create internal process:

monitor	result, i.e. 0 means process created, 1,3,6 means process not created.
z	(call value). Contains the process name. The process will be created in the buffer area of z.
ia	(call value, length $\geq$ 6 in sys2, $\geq$ 9 in sys3). Contains the parameters in this way:

in system 2:

1st element	buffer index for start of process
2nd element	buffer index for last of process
3rd element	buffer claim shift 12 + area claim
4th element	internal claim shift 12 + function mask
5th element	catalog mask
6th element	protection register shift 12 + protection key

in system 3, the first 4 elements are as for system 2. the following are:

5th element	protection register shift 12 + protection key
6th element	lower limit of max base
7th element	upper limit of max base
8th element	lower limit of std base
9th element	upper limit of std base

fnc = 58, start internal process:

monitor	result, i.e. 0 means process started, 2,3,6 means process not started.
z	(call value). Contains the process name. The process must have been created inside the zone buffer.
i	(call value). The number of a share within z. The share state must at call time be 0 or 1, at return time it is - process description address.

fnc = 60, stop internal process:

monitor — result, i.e. 0 means stop initiated, 3,6 means stop not allowed.

z — (call value). Determines together with i the process.

i — (call value). The number of a share within z. The share state must at call time be - process description address. At return time it is the buffer address. Notice that the process name in z is irrelevant.

fnc = 62, modify internal process:

monitor — result, i.e. 0 means process modified, 2,3,6 means modification not allowed.

z — (call value). Contains the process name.

ia — (call value, length $\geq$ 6). Contains the modified registers.

fnc = 64, remove process:

monitor — result, i.e. 0 means process removed, 1,2,3,5,6 means removal not allowed.

z — (call value). Contains the process name.

fnc = 68, generate name:

monitor — result, i.e. 0 means name generated, 1,2 means name not generated.

z — (return value). The generated name is stored here.

fnc = 70, copy core area:

monitor — result of the copying, 0 meaning area copied, 2 or 3 area not copied.

z — (call value). Contains the area to or from which the copying will take place. The limits of the copying are given by the zone parameters record base and last byte.

i — (call value). The buffer address of the input or output message defining sender's copy area.

ia — (return value, length >= 9). Contains information about the copying almost ready to be used by send answer:

1st element	should then be set by the user
2nd element	if result <> 0 then 0 else bytes copied
3rd element	if result <> 0 then 0 else chars copied
9th element	if result = 3 then 3 else 1.

fnc = 72, set catalog base (sys. 3):

monitor — result, 0 means catalog base set, 2,3,4,6 means catalog base not set.

z — (call value). Contains the name of a child process or a null-name, meaning own process.

ia — (call value, length >= 2). Contains the base to be set.

1st element	lower limit of the base
2nd element	upper limit of the base

fnc = 74, set entry base (sys. 3):

monitor	result, 0 means entry base set, 2,3,4,5,6,7 means entry base not set.
z	(call value). Contains the entry name.
ia	(call value, length >= 2). Contains the entry base to be set, as for the fnc = 72, set catalog base.

fnc = 76, lookup head and tail (sys. 3):

monitor	result, 0 means entry looked up, 2,3,6 means entry not looked up.
z	(call value). Contains the entry name.
ia	(return value, length >= 17). The entry looked up.

fnc = 78, set backing store claims (sys. 3):

monitor	result, 0 means claims set, 1,2,3,6 means claims not set.
z	(call value). Contains the name of a child process.
ia	(call value, length >= 4 + 2*no of keys). The first 4 elements contain the name of the bs document

5th element   entry claim, key 0
6th element   segment claim, key 0
....
(5+2*max key)th element   entry claim, max key
(6+2*max key)th element   segment claim, max key

fnc = 80, create pseudo process (sys. 3):

monitor	result, 0 means pseudo process created, 1,2,3,6 means pseudo process not created.
z	(call value). Contains the name of the pseudo process.

fnc = 82, regret message (sys. 3):

monitor	no result from this operation. Misuse will give break 6.
z	(call value). Determines together with 'i' the buffer address of the message to be regretted.
i	(call value). The number of a share within z. The share state must be the buffer address at call time. At return it is 0.

fnc = 90, permanent entry in auxiliary catalog (sys. 3):

monitor	result, 0 means entry made permanent, 2,3,4,5,6,7 means entry not made permanent.
z	(call value). Contains the entry name.
i	(call value). The catalog key.
ia	(call value, length >= 4). Contains the name of the bs document.

Parameters not mentioned in the description are neither used nor changed for that value of fnc. If the requirements stated above are not fulfilled, or if the situation termed 'parameter error' in ref. 1 or ref. 5 occurs, the run will be terminated with an alarm. Values of fnc not mentioned above will also terminate the run.

Example 1, create a backing storage area.

A backing storage area sldata3 of s segments may be created and then used like this:

```
begin zone z(512,1,stderror); integer array tail(1:10);
open(z,4,<:sldata3:>,0);
comment. The zone contains now the document name. The document
         is not initialised in case of kind = 4;
tail(1):= s; tail(2):= 1; comment preferably a disc area;
for i:= 3 step 1 until 10 do tail(i) := 0;
if monitor(40)create_entry:(z,0,tail) > 0 then goto error;
outrec(z,...);
```

In system 2, the area may be made permanent with some key, so that it can survive the job:

```
if monitor(50)permanent_entry:(z,key,tail) > 0 then goto error;
```

Example 2, scope user of an area (system 3).

The scope user function consists of 2 steps. First the area is made permanent with catalog key 3. Now, as key is >= min global key (see ref. 5), the entry base may be set to the user base of the process.

Let the zone z be opened to the area to be scoped.

```
system(11)bases:(i,ia);
ia(1):= ia(5); ia(2):= ia(6); comment fetch the user base;
if monitor(50)permanent entry:(z,3,ia) <> 0 then goto error;
if monitor(74)set_base:(z,0,ia) <> 0 then goto error;
```

Example 3, find scope of an entry (system 3).

As the catalog base of an internal process and of a catalog entry may use almost the full integer range (see 2.5.5), they must be handled as longs when relations between them are calculated, in order to prevent overflow.

```
system(11)bases:(i,bases);
if monitor(76)head_and_tail:(z,0,entry) <> 0 then goto error;
case entry(1) extract 3 + 1 of
begin

  comment key 0, maybe temp:;
  if extend entry(2) = extend bases(3)
  and extend entry(3) = extend bases(4)
  then scope:= 1 else scope:= 6;

  comment key 1:; scope:= 6;

  comment key 2, maybe login:;
  if extend entry(2) = extend bases(3)
  and extend entry(3) = extend bases(4)
  then scope:= 2 else scope:= 6;
```

```
        begin comment key 3, user, project, or system;
          l1:= entry(2); l2:= entry(3);
          if l1 = extend bases(5)
          and l2 = extend bases(6) then scope := 3
          else
          if l1 = extend bases(7)
          and l2 = extend bases(8) then scope:= 4
          else
          if l1 <= extend bases(7)
          and l2 >= extend bases(8) then scope:= 5
          else scope:= 6;
        end

      end;
      write(out,<:the scope is: :>,case scope of(
            <:temp:>,<:login:>,<:user:>,
            <:project:>,<:system:>,<:***, i.e. undef:>));
```

9.41. Procedure open

Connects a document to a given zone in such a way that the zone may be used for input/output with the high level zone procedures.

Call: open(z,modekind,doc,giveup)

z (call and return value, zone). After return, z describes the document.

modekind (call value, integer). Mode shift 12 + kind. See below.

doc (call value, string). A text string specifying the name of the document as required by the monitor, i.e. a small letter followed by a maximum of 10 small letters or digits.

giveup (call value, integer). Used in connection with the checking of a transfer. See below.

Modekind

Specifies the kind of the document (typewriter, backing storage, magnetic tape, etc.) and the mode in which it should be operated (even parity, odd parity, etc).

The kind of the document tells the input/output procedures how error conditions are to be handled, how the device should be positioned, etc. This kind has nothing to do with the kind mentioned in ref. 1. As a rule, the procedures do not care for the actual physical kind of the document, but disagreements may give rise to bad answers from the document. If you, for example, open a backing storage area with a kind specifying printer, and later attempt to output via the zone, the backing storage area will reject the message because the document was initialised as required by a printer.

Mode and kind must be coded as shown in the table below. If you attempt a mode which does not fit into the table, the run is terminated.

kind:

0 internal process, mode = 0.
4 backing storage area, mode = 0.
8 typewriter, mode = 0.
10 paper tape reader, mode = 0 for odd parity, 2 for even parity (the normal ISO form), 4 for no parity, and 6 for conversion from flexowriter code to ISO.
12 paper tape punch, mode = 0 for odd parity, 2 for even parity (the normal ISO form), 4 for no parity, and 6 for conversion from ISO to flexowriter code.
14 line printer, mode = 0 for all printers, except centronics 101A via medium speed tmx where mode = 64.
16 card reader, see ref. 8 for full details.
18 magnetic tape (tapes of 6 or 8 bit physical characters). For RC 747 and RC 749:
  Mode = 0 or 4 means odd parity.
  Mode = 2 or 6 means even parity.
 For RC 4739 and RC 4775 modekind is defined to be:
 T shift 16 + Mode shift 12 + 18, where
  Mode = 0 means 1600 bpi, PE, odd parity.
  Mode = 2 means 1600 bpi, PE, even parity.
  Mode = 4 means 800 bpi, NRZ, odd parity.
  Mode = 6 means 800 bpi, NRZ, even parity.
 For output 0 <= T < 6 specifies that the last T physical characters in a block should not be output to the tape.
 For input T should be 0.
 If you use T <> 0 during output, you should set the word defect bit (1 shift 7) in your give up mask and after a check of bytes transferred simply ignore the bit in your block procedure.

Initialisation of a document

Open prepares the later use of the document according to kind:

Internal process, backing storage area, typewriter:
 Nothing is done. When a transfer is checked later, the necessary initialisation is performed.

Paper tape reader, card reader:
 First, open checks to see whether the reader is reserved by another process. If it is, the parent receives the message
  wait for <name of document>
 and open waits until the reader is free. Second, open initialises the reader and empties it. Third, open initialises the reader again (in order to start reading in lower case), sends a parent message asking for the reader to be loaded, and waits until the first character is available.

Paper tape punch, line printer:
 Open attempts to reserve the document for the job, but the result of the reservation is neglected.

Magnetic tape: If the tape is not mounted, a parent message is sent asking for mounting of the tape. The message is sent without wait indication (see ref. 7).

Some of these rules have been introduced to remedy a possible lack of an advanced operating system.

Giveup.

The parameter giveup is a mask of 24 bits which will be compared to the logical status word (see 6.3) each time a transfer is checked. If the logical status word contains a one in a bit where giveup has a one, the standard action for that error condition is skipped and the block procedure is called instead (the block procedure is also called if a hard error is detected during the checking).

Zone state.

The zone must be in state 4, after declaration. The state becomes positioned after open (ready for input/output) except for magnetic tapes, where setposition must be called prior to a call of an input/output procedure.

The entire buffer area of z is divided evenly among the shares and if the document is a backing storage area, the share length is made a multiple of 512 bytes. If this cannot be done without using a share length of 0, the run is terminated.

The logical position becomes just before the first element of block 0, file 0.

Example 1:

The normal usage of a tape reader named 'reader' goes like this:

```
begin zone z(25*2,2,stderror);
open(z,2 shift 12+10,<:reader:>,0);
read(z,...);...
close(z,true);
end;
```

If you replaced stderror with the procedure 'list':

```
procedure list(z,s,b),zone z; integer s,b;
write(out,<:<10:>,s,b);
```

and called open with 1 shift 1 instead of 0, the block procedure would be activated after each tape transfer and you would get a complete log of the actions of the reader. (The procedure 'list' should print in a better way to be really useful).

Example 2:

Assume you need two magnetic tapes in a job. Then the best communication with the operating system is obtained in this way:

```
open(z1,18,<:mt1706:>,0);
open(z2,18,<:mt1712:>,0);
setposition(z1,1,0); setposition(z2,1,0);
```

If none of the tapes are mounted, the operating system may get the messages:

```
mount mt1706 without wait indication (caused by open(z1,...))
mount mt1712 without wait indication (caused by open(z2,...))
mount mt1706 with wait indication (caused by setposition(z1...))
```

and the job is stopped by the operating system until the tape waited for has been mounted.

Example 3:

Nearly all document names will be supplied as data to the algol program and in many cases the kind and mode are given as data too. A convenient way of doing this is to use the following syntax of the data:

<kind and mode> <document name> <possibly a file number>

Kind and mode are represented as the mnemonic code of the fp-utility program 'set'.

The algol program may then look like this:

```
begin
boolean procedure openvar(z,giveup); zone z; integer giveup;
begin array text(1:3); integer i,j; .
  openvar:= true; j:= 0;
  readstring(in,text,1);
  for i:= 1 step 1 until 17 do
  if text(1) = real(case i of
  (<:ip:>,<:bs:>,<:tw:>,<:tro:>,<:tre:>,...)) then j:= i;

  i:= 1;
  if readstring(in,text,1) > 2 or j = 0 then openvar:= false
  else
  open(z,case j of(0,4,8,10,2 shift 12 + 10,...),
       string text(increase(i)),giveup),
  if j > 15 then
  begin read(in,i);
    setposition(z,i,0)
  end;
end openvar;

...
begin zone master,trans,new(256*2,2,stderror);
if -, (openvar(master,0) and openvar(trans,0)
and openvar(new,0)) then goto dataerror;

inrec6(master,m1); inrec6(trans,t1);...
```

9.42. Out

The standard identifier 'out' is a preopened zone variable for output on character level. The actual file connected to the zone is the current output file of the file processor. Out must be left in a state ready for output of characters when the run is terminated.

Example:

An FP source file containing

```
p = algol   ;
begin write(out,12,<:a:>) end
o f47       ; select f47 as current output, see ref 2 or ref 6.
p           ; execute
p           ; execute
```

will generate the following text in the file f47:

```
 12a
end      7
 12a
end      7
```

9.43. Procedure outchar

Prints one single character on a document.

call: outchar(z,i)

z	(call and return value, zone). Specifies the document, the buffering, and the position of the document.
i	(call value, integer). The last 8 bits of the integer are printed as a character.

Zone state as for write, 9.78.

Blocking as for write, 9.78.

Example:
See example 1 of readchar, section 9.53.

9.44. Procedure outinteger

The procedure prints an integer or a long with a specified number of the last digits preceded by a decimal point. The number may be preceded by a larger number of spaces than a usual layout. The procedure is specially designed to print amounts of currency.

Call: outinteger(z,psns,dec,amount)

z	(call and return value, zone). Specifies the document, the buffering, and the position of the document.
psns	(call value, integer). Specifies the total number of character positions to be printed. psns should be inside the range: abs(psns) ≤ 132.
dec	(call value, integer). Specifies the number of digits after the decimal point. Dec should be inside the range: 0 <= dec <= min(abs(psns)-3,15).
amount	(call value, integer or long). The integer or long to be printed.

The procedure prints an integer or a long with a specific number of characters as given by the absolute value of the parameter psns. If psns is negative and amount = 0 then a number of spaces equal to the absolute value of psns is printed. If psns is outside the allowed range, the procedure will output 132 characters.

Positive values of amount are printed without a sign whereas a negative amount is preceded by a minus sign. Character positions not occupied by digits and a possible sign and/or period are converted to spaces in front of the integer. An integer is always printed correctly even if the number of character positions is not adequate.

Zone state as for write, 9.78.

Blocking as for write, 9.78.

Example:
The program

```
begin long ll,ii;
  for ii:= 5, ll*ll while ll < 1000000 do
  begin ll:= ii; outinteger(out,8,2,ll); outchar(out,10) end;
end;
```

will print

```
    0.05
    0.25
    6.25
 3906.25
1525878906.25
```

9.45. Integer procedure outrec

This is the Algol 5 version of outrec6. A document may be filled sequentially by means of outrec, because the next call of outrec will create a record which is transferred to the next elements of the document.

Call: outrec(z,length)

outrec (return value, integer). The number of elements of 4 bytes each available for further calls of outrec before change of block takes place.

z (call and return value, zone). The name of a record. Determines further the document, the buffering, and the position of the document (see 6.1).

length (call value, integer, long or real). The number of elements of 4 bytes each in the new record. Length must be ≥ 0.

For further description see 9.46, outrec6.

Outrec may be used with advantage when the document is considered to contain reals.

Example, storing a matrix on backing store.
An n*n-matrix m may be output row by row to a backing storage area f13 in this way:

```
begin zone save((n+127)//128*128*2,2,stderror);
open(save,4,<:f13:>,0);

for i:= 1 step 1 until n do
begin
  outrec(save,n);
  for j:= 1 step 1 until n do save(j):= m(i,j);
end;
close(save,false)
end;
```

The zone declaration assures that the rows later may be read one by one and used directly.

9.46. Integer procedure outrec6

Creates a zone record which later will be transferred to a document. The contents of the record are initially undefined but the user is supposed to assign values to the record. The document may be filled sequentially by means of outrec6 because the next call of outrec6 will create a record which is transferred to the next bytes of the document.

Call: outrec6(z,length)

outrec6	(return value, integer). The number of bytes available for further calls of outrec6 before change of block takes place.
z	(call and return value, zone). The name of a record. Determines further the document, the buffering, and the position of the document (see 6.1).
length	(call value, integer, long or real). The number of bytes in the new record. Length must be $\geq$ 0. If length is odd, 1 is added.

Zone state

The zone z must be open and ready for record output(state 0 or 6; see 9.27, getzone), i.e. the zone may only have been used for record output since the latest call of open or setposition. To make sense, the document should be an internal process, a backing storage area, a typewriter, a line printer, a punch, a plotter, or a magnetic tape. In the latter case setposition(z,...) must have been called after open(z,...).

Blocking

Outrec6 may be thought of as transferring the record to the bytes just after the current logical pointer of the document and moving the logical pointer to just after the last byte of the record. The user is supposed to store information in the record before outrec6 is called again.

Because the output is blocked, the actual transfer to the document is delayed until the block is changed or until close or setposition is called.

The full record goes into the same block, so if the block cannot hold a record of the length attempted, the block is changed in this way:

1. Documents with fixed block length (backing storage): The remaining bytes of the share are filled with binary zeroes, and the total share is output as one block.
2. Documents with variable block length (all others): Only the part of the share actually used for records is output as a block.

The transfer is checked as described in 6.3. The record becomes the first bytes of the next share, but if the record still is too long, the run is terminated.

A record length of 0 is handled as for inrec6.

Example 1, records of variable length.

Records of variable length, with the length stored as the first word of the record, are output like this:

```
     open(z,...); setposition(z,...);
rep: ..... ; compute length
     outrec6(z,length);
     z.first_word:= length;
     if ... then goto rep;
     close(z,true);
```

Compare this with example 1 of changerec. The version here may be a little bit faster.

9.47. Procedure outtext

Prints a text stored as text portions in a real array or a zone record. The procedure prints a specific number of characters. If the string is shorter, it is supplemented with spaces, and if it is longer, it is cut.

Call: outtext(z,pos,ra,i)

- z (call and return value, zone). Specifies the document, the buffering, and the position of the document.
- pos (call value, integer). Specifies the total number of character positions to be printed. Pos should be inside the range: abs(pos) $\le$ 132, see below.
- ra (call value, real array). The text to be output is stored in ra(i), ra(i+1), and so on. For arrays of more dimensions the lexicographical ordering is used.
- i (call value, integer), see ra above.

The procedure prints a number of characters as given by the absolute value of the parameter pos. If pos is negative a NL character is output before the counting starts. If pos is outside the allowed range, the procedure will output 132 characters.

The characters to be printed are supplied from a string of text portions stored in a real array or a zone record. The characters are taken from the array until either the string has been exhausted or the number of characters as given by pos has been output.

If the text string is exhausted before the wanted number of characters are printed, spaces are printed as the following characters.

The string is considered exhausted when the last element of the array has been printed or when a null character is met.

Zone state as for write, 9.78.

Blocking as for write, 9.78.

9.48. Integer procedure outvar

This procedure is intended for output of records of variable length so that they may be read by means of invar. Outvar makes an output record ready and fills it from a real array (or a zone record). The first word of the element with lexicographical index 1 in the array must contain the length of the wanted record. The second word in the new record will contain a checksum.

Call: outvar(z,A)

outvar	(return value, integer). The number of bytes available for further calls of outvar before change of block takes place exactly as for outrec6.
z	(call and return value, zone). The name of the record. Determines further the document, the buffering, and the position of the document (see 6.1).
A	(call value, real array). An array to be copied into the zone record. The first word of the element with lexicographical index 1 contains the number of bytes to be copied. If the number is odd, 1 is added.

Zone state

The zone z must be open and ready for record output (state 0 or 6), i.e. the zone may only have been used by outvar or the like since the latest call of open or setposition. The free parameter (see 9.27) in the zone descriptor is used to count the number of records made by means of outvar. Usually only backing storage and magnetic tape documents make sense.

Blocking

Outvar may be thought of as transferring the data in the array to the bytes just after the current logical pointer of the document and moving the logical pointer to just after the transferred elements. The new record is placed in the same block, so if the present block cannot hold a record with the attempted length, outvar changes block exactly as outrec6, i.e. on backing store unused parts are filled with binary nulls, and on all other media only the used part is output.

Record format, checksum

The record consists of 2 words containing information on the record followed by an arbitrary number of words. The record length must not exceed the blocklength.

The 2 first words contain the record length measured in bytes in the first word and a checksum in the second word. The value of the checksum word is chosen so that the sum of all words in the record taken modulo 2**24 is equal to -3.

Note that the call outvar(z,z) produces one record identical to the last one.

9.49. Integer overflows

This standard identifier determines the action on floating point overflow:

overflows < 0 The run is terminated when overflow occurs.
overflows ≥ 0 The value of overflows is increased by one when overflow occurs. The result of the operation which caused the overflow is 0.

When the run starts, overflow is -1. A floating point overflow occurs when a real operation gives a result outside the range of real variables.

Due to an inconvenience in the machine structure an underflow caused by multiplication of 2 reals both in the interval $2**(-1024) <$ abs $r < 2**(-2048)$ will be classified as an overflow.

Example:
To check whether a real overflow occurred during the evaluation of an expression, proceed as follows:

```
overflows:= 0; Evaluate the expression;
if overflows > 0 then handle the overflow situation;
```

9.50. Real procedure random

Computes two pseudo-random numbers, a real and an integer.

Call: random(i)

random (return value, real). A pseudo-random number determined by i. $0 < \text{random} < 1$.

i (call and return value, integer). At call time the latest pseudo-random number generated (or a starting value for the generation). At return the next pseudo-random number. $0 < i < 8\ 388\ 587$.

Method:
Multiplicative generation with a period of 8 388 586. The starting value is not critical, because a result of 0 is prevented explicitly in the procedure.

9.51. Integer procedure read

Inputs a sequence of numbers given in character form on a document, converts them to algol values, and assigns them to variables.

Call: read(z,one or more destination parameters)

read (return value, integer). The absolute value of read gives number of destination variables to which numbers were input.

z (call and return value, zone). Specifies the document, the buffering, and the position of the document (see 6.1).

destination (return value; integer, long, real, integer array, long array, or real array). Read assigns numbers to the destination parameters in sequence from left to right. A simple parameter is used as one destination variable. An array is used as a sequence of destination variables, and read fills the entire array in lexicographical order (see 5.2).

Note that all the parameters are evaluated before the procedure is entered (except if the procedure is called as a formal procedure), so the call read(in,i,A(i)) will mean w:= i; i:= number; A(w):= number;

Syntax of numbers.

Read skips all blind characters (class 0, see 2.0.1). Among the remaining characters, 'read' accepts as a number any sequence of number constituents (class 2 to 5) terminated by some other character (class > 5). Leading characters of class > 5 are disregarded unless they contain the EM character (see below).

If the number constituents fulfill the rules for Algol 6 numbers, the number is assigned to a destination variable. If it is not an Algol 6 number or if it exceeds the range of the destination variable, the greatest positive number of the appropriate type is assigned instead.

Terminating reading:

Read scans the document and each time it meets a number (in the sense defined above) it stores it into the next destination variable. When the parameter list is exhausted, read returns. The reading stops immediately, however, if an EM character is met. In this situation the value of read is useful.

Zone state:

As for readchar, 9.53.

Blocking:

As for readchar, 9.53.

Example 1, reading and checking a matrix.

An n*n-matrix is punched on current input as n followed by the matrix elements. It may be read in this way with a simple check added:

```
if read(in,n) < 1 or n > 200 then goto dataerror;
begin array matrix(1:n,1:n);
  if read(in,matrix) < n**2 then goto dataerror;
```

The matrix might for instance be punched like this:

```
3
 1.507    -6.017     2.446
-6.017     3.852     0.025
 2.336     0.025    -8.170
```

It will be wise to check that a new line terminated the last number. That is done as follows:

```
repeatchar(in); readchar(in,i); if i <> 10 then goto dataerror;
```

Example 2:

The following character sequence represents 5 numbers as shown:

```
a- 1.7bcd-12345678 9'60 3+10ee&lt;EM&gt;
 1  2       3      4   5
```

If it is input by the call read(z,i,j,k,r,s,t), the variables will become:

i,j,k,(integers):	great,2,great(range exceeded)
r,s,t(reals):	9'60,great,unchanged(EM met)

Read itself has the value -5.

9.52. Integer procedure readall

Inputs a mixture of numbers in character form, text strings, and single characters. These items are stored in an array and their kind is stored as a code in a parallel array. The procedure is designed for fast input on character level with possibility for extensive checking of the input. Readall is often used in combination with intable.

Call: read all(z,val,kind,index)

read_all — (return value, integer). The number of elements in val to which items have been assigned. If read all terminates because val or kind is full, the value of read_all is minus number of elements.

z — (call and return value, zone). Specifies the document, the buffering, and the position of the document (see 6.1).

val — (return value, integer array, long array, or real array). The items are stored in val(index), val(index + 1), and so on. For arrays of more dimensions, the lexicographical ordering is used.

kind — (return value, integer array). The kind of the items is stored here, so that kind(i) describes the contents of val(i).

index — (call value, integer). See description of val above.

Syntax of items:

Readall divides an input string into items in this way:

1. All blind characters are skipped (class 0, see 2.0.1).
2. A delimiter character (class ≥ 7) is stored as a single character.
3. A character string starting with a letter (class 6), consisting of letters and number constituents (class 2 to 6), and terminated by a delimiter (class ≥ 7) is stored as a text string. The delimiter is not a part of the text string.
4. The remaining parts of the input string are stored as numbers in the way described under read.

In many cases the rules for text strings and numbers are inconvenient. It will then pay to use an alphabet (see intable, 9.32) defining most characters as delimiters of various classes and input one line of characters at a time. An example of the further handling of the characters is shown in example 2 of readchar.

Storing of items:

1. Blind characters are not stored.
2. A single character is stored in one element: val(i):= character value; kind(i):= character class.
3. A text string is packed as portions of 6 8-bit characters. The characters are packed from left to right. A portion is stored in 4 bytes, i.e. one real or long, or possibly two integers. The corresponding elements of 'kind' becomes 6. A null character is packed after the last character of the text string and the corresponding portion is filled up with null characters. A text packed in this way is easy to use as a string parameter.
4. A number is stored in one element: val(i):= converted number; kind:= 2 for a legal number, kind(i):= 1 for an illegal or syntactically wrong number.

Terminating reading:

Readall returns as soon as a terminator (class8) has been input and stored. If val or kind is filled up before that, readall returns with a negative value. In that situation, the last character read is not stored. You may get the character by means of repeatchar, but you cannot expect to continue reading as if nothing has happened, because readall may have terminated in the middle of a text string and the next character may be a digit or a delimiter.

Zone state:

As for readchar, 9.53.

Blocking:

As for readchar, 9.53.

Example 1:

A line input by read all with the standard alphabet to an integer array may be printed and 'reshaped' in this way:

```
n:= readall(z,ia,kind,1);
if n < 0 then write(out,<:illegal:>) else
for i:= 1 step 1 until n - 1 do
case kind(i) of
begin comment kind 1; write(out,<:illegal:>);
  comment kind 2; write(out,<<-dddddd>,ia(i));
  comment 3,4,5;;;;
  begin comment kind 6;
    write(out,<: :>,string(0.0 shift 24 add ia(increase(i))
                     shift 24 add ia(increase(i))));
    i:= i - 1
  end;
  comment kind 7, spaces are not printed;
  if ia(i) <> 32 then write(out,<: :>,false add ia(i),1)
end;
write(out,<:<10>:>);
```

Example 2:

The following character sequence represents 9 items if it is read with the standard alphabet:

```
ab : a1.2c ,    17.56    12345678 <NL>
 1 2   3  4 5     6   7     8        9
```

If it is input by readall as in example 1, the result becomes:

	1	2	3	4	5	6	7	8	9	10	11
ia	ab	0	58	al.	2c	44	32	18	32	great	10
kind	6	6	7	6	6	7	7	2	7	1	8

readall = 11

The print-out of example 1 will look like this:

```
ab : al.2c,          18illegal
```

Example 3, typical adp-input.

A list of employees is punched in this way:

<identification number><department number><status>,<surname>,<first names>
<identification number> ...

For example: 451 55z, bell, robert george

If you read this kind of input with readall and the standard alphabet, you would not get checked that the names are free of digits. Furthermore you would have troubles accepting the spaces in the names as name constituents. Instead, you may use an alphabet table of 2*128 entries (see intable). The first 128 entries describe the alphabet used during reading of the numbers. All letters are here described as shift characters which switch to the last 128 entries (class = 1, value = 128).

In this last part of the table, space and all letters are described as text constituents. All digits are delimiter symbols. In both parts of the alphabet table, new line is a terminator.

An input program which checks the syntax and outputs the list as a sequence of records may look like this:

```
       intable(alphabet);
       comment insert some pseudo values at the end of the kind table,
         so that the scanning below is terminated in all cases;
       kind(max+1):= kind(max+2):= 5;

  rep: tableindex:= 0; n:= readall(z,val,kind,1);
       if n > max or n < 1 then error;
       if n = 1 then
       begin if val(1) = 25 then goto terminate; goto rep end;

       comment check identification and department, dept points
         to department;
       if kind(1) <> 2 or kind(2) <> space then error;
       for i:= 2,i+1 while kind(i) = space do;
       if kind(i) <> 2 then error; dept:= i;
```

```
comment check status, transform it so some coded form;
if kind(i+1) <> 6 or kind(i+2) <> comma then error;
val(i+1):= transformed value;

comment check surname;
for i:= i + 2,i + 1 while kind(i) = 6 do;
if kind(i) <> comma or i = dept + 3 then error;

comment check first names, fnames points to first names;
fnames:= i + 1;
for i:= i + 1 while kind(i) = 6 do;
if kind(i) <> 8 or i = fnames then error;

outrec(empl,i-dept+1); comment now the line is accepted;
empl(1):= i-dept+1; empl(2):= val(1); empl(3):= val(dept);
empl(4):= val(dept+1); k:= 4;
for j:= dept+3 step 1 until i - 1 do
begin k:= k + 1; empl(k):= val(j); end;
goto rep;
```

9.53. Integer procedure readchar

Inputs one non-blind character from a document and supplies the character value and character class. Blind characters are skipped automatically.

Call: readchar(z,val)

readchar (return value, integer). The class of the character (see 2.0.1).

z (call and return value, zone). Specifies the document, the buffering, and the position of the document (see 6.1).

val (return value, integer). The value of the character (see 2.0.1).

Zone state:

The zone must be open and ready for character reading (state 0, 1, or 2; see 9.27, getzone6), i.e. since the latest call of open or setposition, the zone may only have been used for character reading. To make sense, the document should be an internal process, a backing storage area, a typewriter, a paper tape reader, a card reader, or a magnetic tape. In the latter case setposition(z,...) must have been called after open(z,...).

The first character read is normally the character just after the logical position of the document, but after a call of repeatchar it is character just before the logical position.

When readchar returns, the logical position is just after the last character read. The zone record is not available (it is of length 0).

Blocking:

Just after open or setposition or whenever a block of the document is exhausted, the next block is transferred and checked as described in 6.3. On a typewriter in online mode this means that an entire line must be typed before any of the characters in the line are available to readchar.

Example 1, copying.

A sequence of characters may be copied and counted in the following slow, but simple way. The copying stops when a termination character (class 8) is met.

```
i:= -1;
for i:= i + 1 while readchar(inz,c) <> 8 do
outchar(outz,c);
```

Blind characters may be copied too if another alphabet is selected (see intable 9.32).

Example 2, syntax check:

An octal signed integer may be read and checked by means of a state table. Each entry in the table gives the new state of the routine and the action to be performed when a character of that class is read in that state. The actions are shown as numbers, explained below.

input classes:	sign:	digit	other:
state-1, start	after sign, 1	after digit, 2	start, 3
state 2, after sign	after error, 4	after digit, 2	after error, 4
state 5, after digit	after error, 4	after digit, 2	start, 5
state 8, after error	after error, 3	after error, 3	start, 5

Action 1: set sign. Action 2: include digit in number.
Action 3: no action. Action 4: set error indication.
Action 5: complete number with sign.

This scheme is easiest to implement if a special alphabet is selected by means of intable. The digits 0 to 7 are given class 2, values 0 to 7. Plus and minus are given class 3, values 2 and 0. All other non-blind characters are given class 4.

The algorithm may then be written like this:

```
     state:= -1; sign:= 1; number:= 0; error:= false;

rep: class:= readchar(z,c) + state;
     action:= case class of
                    (1,2,3, 4,2,4, 4,2,5, 3,3,5);
     state:= case class of
                    (2,5,-1, 8,5,8, 8,5,-1, 8,8,-1);
     case action of
     begin comment 1, set sign; sign:= v - 1;
       comment 2, include digit;
       if number >= 1 shift 20 then error:= true else
       number:= number shift 3 + c - 48;
       comment 3, no action; ;
       comment 4, set error indication; error:= true;
       comment 5, terminate;
       begin number:= number*sign; goto terminate end;
     end;
     goto rep;
terminate:
```

A shorter solution might be found for this particular problem, but the main advantage of the method is that it applies to a lot of other input problems and the time spent per character will hardly depend on the complexity of the input syntax. The algorithm above will read about 2000 characters a second, but it may be speeded up to about 3000 characters a second if readall is used instead of readchar to input a big portion of characters. The character classes 2,3 , and 4 must then be replaced by 9, 10, 11 or the like.

A further increase in speed to about 4500 characters a second is possible if the input is performed blockwise by means of inrec6 and the characters are unpacked as shown in example 3 of extract (9.21).

9.54. Integer procedure readstring

Inputs a text string given as 8-bit characters on a document. The text string is packed in a way which makes it easy to use a string parameter.

Call: readstring(z,arr,i)

readstring (return value, integer). The number of elements in arr to which a text portion has been assigned. If readstring terminates because arr is full, the value of readstring is negative.

z (call and return value, zone). Specifies the document, the buffering, and the position of the document (see 6.1).

arr (return value, long array, or real array). The text is stored in arr(i), arr(i+1), and so on. For arrays of more dimensions the lexicographical ordering is used.

i (call value,integer). See arr above.

Syntax of a text string.

Readstring skips all blind characters (class 0, see 2.0.1). Among the remaining characters, readstring accepts as a text string any sequence of text constituents (class 2 to 6) terminated by a delimiter (class > 6).

Leading characters of class > 6 are disregarded unless they contain the EM character (see below).

The text constituents, omitting all blind characters, are packed into arr with 6 8-bit characters to an element. The characters are packed from left to right. The character values packed are given by the values in the alphabet selected for the moment (see intable, 9.32). When the standard ISO alphabet is used, the values are shown in 2.0.1. A null character is packed after the last character of the text string and the corresponding element of arr is filled up with null characters.

Terminating reading

Normally, readstring returns when the text and the terminator have been read. The reading stops immediately, however, if an EM character is met or if arr is filled. In the latter case, the value of readstring is negative, the text string is not terminated by a null character, and the last character is read, but not packed.

Zone state.

As for readchar, 9.53.

Blocking.

As for readchar, 9.53.

Example 1: input and output of text.

A text (for instance a heading) may be input and later printed in this way:

```
begin long array text(1:n);
intable(ia); comment define space etc. as text constituents;
if readstring(in,text,1) = n
and text(n) extract 8 <> 0 then goto dataerror;
...
i:= 1; write(out,string text(increase(i)));
comment see 9.70, string;
```

Example 2:

See example 3 of open.

9.55. Real

This monadic operator changes the type of a string expression to type real. In this way, strings may be stored and analysed. Note that this use of the delimiter real is totally different from its use in a declaration or specification.

Syntax: real <string> is of type real.
real <long> is of type real
Priority higher than **.

The value of real <string> has the same binary pattern as the value of <string>. The value of real <long> has the same binary pattern as the value of <long>. The binary pattern of a string is described in 3.6.

Example 1:

Let s be a formal string parameter which actually is a text string. The statement

```
r:= real(case i of(<:abs:>,<:long text:>,s));
```

will assign a text to r. Depending on the value of i, r will hold a packed text, a string point, or the string value of the formal parameter s.

In the first two cases and in the third case with s describing a literal text string, the text may be printed in this way:

```
write(out,string r);
```

Example 2, computing a layout.

Assume you want to print numbers with a layout depending on the relative accuracy, eps, of the numbers. If the layout is to be used many times, it is wise to hold it in a real variable like this:

```
d:= -ln(eps)/ln(10) + 0.5;
if d < 3 then d:= 3 else
if d > 6 then d:= 6;
r:= real(case d -2 of
(<<-d.dd'-dd>,<<d.ddd'-dd>,<<-d.dddd'-dd>,<<-d.ddddd'-dd>));
...
write(out,string r,x,y,...);
```

9.56. Procedure repeatchar

Makes the latest character read from the zone specified available for reading once more.

Call: repeatchar(z)

z (call and return value, zone). Specifies the document, the position of the document, and the latest operation on z.

After a call of repeatchar(z), the next character read from z is the character just before the logical position of the zone, i.e. the latest character read. Note that the logical position is unchanged.

If repeatchar is to have any effect, the zone should be in the state 'after character input' (state 1), i.e. one of the read procedures must have been called since the latest call of open or setposition working on that zone. In all other states repeatchar is blind.

The definition of repeatchar implies that several calls of repeatchar have the same effect as one call.

Example:

See example of read, 9.51.

9.57. Round

This monadic operator rounds the value of a real expression to the nearest integer value or cuts the value of a long expression to an integer. The operation may cause integer overflow.

Syntax: round <real> is of type integer
round <long> is of type integer
Priority higher than **.

Example:

Two reals with absolute values below 2**23 may be integer divided in this way:

```
round r1//round r2
```

9.58. Boolean procedure setposition

Terminates the current use of a zone and positions the document to a given file and block on devices where this makes sense. The positioning will only involve time-consuming operations on the document if this is a magnetic tape.

Call: setposition(z,File,Block)

setposition (return value, boolean). True if a magnetic tape positioning has been started, false otherwise.

z (call and return value, zone). Specifies the document, the position of the document, and the latest operation on z.

File (call value, integer). Irrelevant for documents other than magnetic tape. Specifies the file number of the wanted position (see 6.1). Files are counted 0, 1, 2, ...
File 0 will normally contain the tape volume label, so that file 1 is the first file available for data. File = -1 specifies that the tape is to be unwound.

Block (call value, integer). Irrelevant for documents other than magnetic tape and backing storage. Specifies the block number of the wanted position (see 6.1). Blocks are counted 0, 1, 2, ...

Setposition proceeds in 3 steps: Terminate the current use, write tape mark, and start positioning.

Terminate current use.

If the zone latest has been used for output (state 3, 6, and 7; see getzone6 9.27), the used part of the last block is sent to the document. A block sent to a backing storage area is not filled with zeroes, contrary to outrec6, or outvar. If the zone latest has been used for character output, the termination may involve output of one or two Nulls in order to fill the last word of the buffer.

Next, all the transfers involving z are completed, the input transfers are just waited for, and the output transfers and other operations are checked as usually.

The physical position of a magnetic tape used for input is sh - 1 blocks ahead of the logical position where sh is the number of shares. If some of these sh - 1 blocks are tape marks, the positioning strategy is affected, as explained below.

Write tape mark.

If the document is a magnetic tape which latest has been used for output, a tape mark is written. The document is then in a position after that tape mark, which influences the positioning strategy (see below).

Start positioning.

Setposition assigns the value of Block to the zone descriptor variable 'segment count' and returns then for all devices other than magnetic tape.

If the document does not exist or if the job is not a user of the device, setposition sends a parent message asking for stop of the job until the tape is ready.

If the name of the document is zero (<::>), the tape requested is a work tape, and setposition accepts as the future tape name the name returned by the parent (which means that setposition changes the document name in the zone accordingly).

Setposition starts the first operation involved in the tape positioning. The remaining operations are executed the first time the zone is used for input or output, or the first time setposition(z,...) is called again. That may be used for simultaneous positioning of more tapes (see example 3).

The positioning is accomplished by means of the operations position tape, rewind, backspace file, upspace file, backspace block, upspace block, and unwind tape. The positioning is complete as soon as File and Block match the monitors count of the tape position for that device. Checking against tape labels is not performed.

Positioning strategy in system 2.

If the actual physical file number differs from File, the tape is first positioned to block 0 of that file. Setposition chooses between rewind and backspace file in this way:

```
if actual file number//2 >= File then rewind else backspace file;
```

This tends to minimise the number of tape operations.

During positioning within a file, setposition chooses between backspace file (rewind for File = 0) and backspace block in this way:

```
if actual block number//2 >= Block then backspace or rewind file
                                 else backspace block;
```

If the tape is not mounted when setposition is called, the normal mount-tape-action is performed (see 6.1) before the positioning starts.

In system 3 a position tape operation is sent.

Zone state.

The zone must be open when setposition is called (state 0, 1, 2, 3, 5, 6, 7, or 8). Setposition changes the zone state to opened and positioned.

The logical position of a magnetic tape or a backing storage area becomes just before the first element of the block specified by File and Block. The logical position is unchanged for other devices.

Example 1, online Conversation.

When you alternatively type out something on a terminal and read from it, you must make sure that the output really is sent to the terminal and does not stay in the buffer. Assume that you run with Boss as parent and that you have online yes as job parameter, assume further that your program is started with <program name> term. Such a conversation may then be programmed like this:

```
rep: write(out,<:Type yes or no:>);
     setposition(out,0,0);
     readstring(in,ra,1); ...
     goto rep;
```

Example 2, random access to backing storage.

Let the backing storage area bs25 contain records of 80 bytes originally output in shares of 128 elements (= 1 segment, 512 bytes). You may get record j in this way:

```
begin zone z(128,1,stderror);
comment double buffering will not pay in this case;
open(z,4,<:bs25:>,0);

i:= ...; setposition(z,0,j//6);
comment 6 records are stored on one segment;
for i:= j//6*6 step 1 until j do inrec6(z,80);
```

Example 3, simultaneous tape positioning.

Let z1 and z2 be two zones which describe magnetic tapes positioned at file 2 or 3. If you start reading from file 1 in this way:

```
setposition(z1,1,0); inrec6(z1,p);
setposition(z2,1,0); inrec6(z2,p);
```

then the call of setposition(z1,...) will start rewinding z1. Inrec6(z1,p) will wait for the rewind, upspace file 1 (file 0 is usually short), and read the first block. First at that moment, the rewind of file z2 will be started.

The following solution will rewind the two tapes simultaneously:

```
setposition(z1,1,0); setposition(z2,1,0);
inrec6(z1,p); inrec6(z2,p);
```

If file 0 should be long, it is better to upspace the tapes simultaneously too.

```
setposition(z1,0,0); setposition(z2,0,0);
setposition(z1,1,0); setposition(z2,1,0);
inrec6(z1,p); inrec6(z2,p);
```

Example 4, output of tape mark and empty file

Two tape marks in sequence may be output in this way:

```
outrec6(z,...); getposition(z,f,b); setposition(z,f+1,0);
outrec6(z,0); setposition(z,f+2,0);
```

A call of outrec6(z,0) is also useful when you generate a magnetic tape file which may happen to be empty. If you omit outrec6(z,0), the tape mark may be omitted.

9.59. Procedure setshare

This procedure is the 'reverse' of getshare, 9.24, in the sense that it assigns values to a share descriptor. The procedure is the Algol 5 equivalent of setshare6.

Call: setshare(z,ia,sh)

- z (call and return value, zone). Specifies the share together with sh.
- ia (call value, integer array, length > 12). The contents of ia have the meaning explained in 9.24, getshare. The contents of the first 12 elements of ia are transferred to the share descriptor, provided that the restrictions below are fulfilled.
- sh (call value, integer). The number of the share within z.

Restrictions

The following explanation assumes that ia has been declared as ia(1:12).

- ia(1) Share state. As for setshare6;
- ia(2) First shared. Must be a buffer index.
- ia(3) Last shared. Must be a buffer index.
- ia(4) Operation. As for setshare6.
- ia(12) Top transferred. As for setshare6.

9.60. Procedure setshare6

This procedure is the 'reverse' of getshare6, 9.25, in the sense that it assigns values to a share descriptor.

Call: setshare6(z,ia,sh)

- z — (call and return value, zone). Specifies the share together with sh.
- ia — (call value, integer array, length ≥ 12). The contents of ia have the meaning explained in 9.25, getshare6. The contents of the first 12 elements of ia are transferred to the share descriptor, provided that the restrictions shown below are fulfilled.
- sh — (call value, integer). The number of the share within z.

Restrictions

The following explanation assumes that ia has been declared as ia(1:12).

- ia(1) Share state. If the state of the share descriptor is 0 or 1 at call time, ia(1) will be transferred. In this case ia(1) must be 0 or 1.
- ia(2) First shared. Must be a byte index in the buffer.
- ia(3) Last shared. Must be a byte index in the buffer.
- ia(4) Operation shift 12 + mode. If operation is odd, ia(5) and ia(6) are restricted to absolute addresses within the zone buffer.
- ia(12) Top transferred. Must be an absolute address corresponding to a block within the zone buffer.

If the restrictions are violated, the run is terminated. The restrictions are natural, in the sense that the following always is allowed (provided that sh is a share number and ia has at least 12 elements):

```
getshare6(z,ia,sh); setshare6(z,ia,sh);
```

9.61. Procedure setzone

This procedure is the 'reverse' of getzone, 9.26, in the sense that it assigns values to a zone descriptor. The procedure is the Algol 5 equivalent of setzone6.

Call: setzone(z,ia)

- z — (call and return value, zone). The descriptor of z is changed.
- ia — (call value, integer array, length ≥ 17). The contents of ia have the meaning explained in 9.26, getzone. The contents of the first 17 elements of ia are transferred to the zone descriptor, provided that the restrictions below are fulfilled.

Restrictions

The following explanation assumes that ia has been declared as ia(1:17).

ia(1) Mode shift 12 + kind. As for setzone6.
ia(14) Record base. As for setzone6.
ia(15) Last byte. As for setzone6.
ia(16) Record length. Measured in elements of 4 bytes each otherwise as for setzone6.
ia(17) Used share. As for setzone6.

9.62. Procedure setzone6

This procedure is the 'reverse' of getzone6, 9.27, in the sense that it assigns values to a zone descriptor.

Call: setzone6(z,ia)

z (call and return value, zone). The descriptor of z is changed.

ia (call value, integer array, length $\geq$ 17). The contents of ia have the meaning explained in 9.27, getzone6. The contents of the first 17 elements of ia are transferred to the zone descriptor, provided that the restrictions shown below are fulfilled.

Restrictions

The following explanation assumes that ia has been declared as ia(1:17).

ia(1) Mode shift 12 + kind. The range of the kind is $\leq$ kind $\leq$ 18. The kind must be even.
ia(14) Record base. Must be an absolute address corresponding to a record within the zone buffer. Record base must be odd.
ia(15) Last byte. Must be an absolute address within the zone buffer.
ia(16) Record length in bytes. Must correspond to a record within the zone buffer.
ia(17) Used share. Must be the number of a share within z.

If the restrictions are violated, the run is terminated. The restrictions are natural, in the sense that the following always is allowed (provided that ia has at least 20 elements):

```
getzone6(z,ia); setzone6(z,ia);
```

Example:

See example 2 of getzone6, 9.27.

9.63. Integer procedure sgn

Yields -1 or 1 according to the sign of the parameter.

Call: sgn(r)

sgn (return value, integer). Sgn is 1 for r $\geq$ 0, -1 for r < 0.

r (call value, integer, long, or real).

9.64. Shift

This dyadic operator is used for packing and unpacking of reals, longs, integers, and booleans.

Syntax: <real> shift <primary> is of type real.
<long> shift <primary> is of type long.
<integer> shift <primary> is of type integer.
<boolean> shift <primary> is of type boolean.
Priority as **.

Shift treats the left hand operand as a binary pattern (see 3.1). The right hand operand is rounded to an integer if it is long or real. This value is then used to indicate the number of bits the left hand operator is to be shifted. The shift is to the left if the possible rounded value is positive and the right if the value is negative. The shift is a logical shift, which means that zeroes are shifted in to the right or left.

Examples:
See 9.2, add, and 9.21, extract.

9.65. Integer procedure sign

Call:	sign(r)	
	sign	(return value, integer). Sign is 1 for r > 0, 0 for r = 0, and -1 for r < 0.
	r	(call value, real, long, or integer).

9.66. Real procedure sin

Call:	sin(r)	
	sin	(return value, real). The mathematical function sine of the argument r.
	r	(call value, real, long, or integer). The argument in radians.

Accuracy:
See cos, 9.15.

9.67. Real procedure sinh

Call:	sinh(r)	
	sinh	(return value, real). The mathematical function sinh of the argument r.
		(call value, real, long, or integer). -1000 < r < 1000.

Accuracy:
r = 0 gives sinh = 0
abs(r) < ln(2)/2 gives a relative error below 1.0'-10.
(n-0.5)*ln(2) ≤ abs(r) < (n+0.5)*ln(2) gives a relative error below 1.2'-10 + n*7'- 11.

Alarm
If abs(r) ≥ 1000, the run is terminated.

9.68. Real procedure sqrt

Call: sqrt(r)

sqrt	(return value, real). The square root of r.
r	(call value, real, long, or integer). $r \geq 0$.

Accuracy:

r = 0	gives sqrt = 0.
r > 0	gives a relative error below 6.4'-11.

Alarm

The run is terminated if r < 0.

9.69. Procedure stderror

Terminates the run with an error message specifying an error condition on a peripheral device. It is used as the block procedure of zones where you don't care for device errors.

Call: stderror(z,s,b)

z	(call value, zone). Specifies the name of the document.
s	(call value, integer). The logical status word after a device transfer.
b	(call value, integer). The number of bytes transferred.

The run is terminated with the alarm message:

```
bytes <value of b> ...
called from ...
```

The file processor prints an interpretation of the logical status word 's' after the alarm message from the algol program.

Example:

See example 2 of inrec6, where stderror is used in two ways.

9.70. String

This monadic operator changes the type of a real or long expression into type string. The operator is required when a string stored in real or long variables is used as a parameter of type string. Note that this use of the delimiter string is totally different from the string specification.

Syntax: string <real> is of type string.
string <long> is of type string.
Priority higher than **.

The value of string <real> or string <long> has the same binary pattern as the value of the operand. The binary pattern of a string is described in 3.6. Depending on the value of the operand, the resulting pattern may mean a layout, a complete text string, or a text portion.

Example 1, layout:
See example 2 of real, 9.55.

Example 2, a long string:
Let the real array ra(1:n) hold a sequence of text portions terminated by a null character. Such contents of ra may for instance have been obtained by readstring.

This variable text may be used as a string parameter in this way, for instance:

```
i:= 1; write(out,string ra(increase(i)));
```

Write will reference the second parameter, which in turn calls increase(i) and yields the value of ra(1). At the same time i becomes 2. Write will print the text portion held in ra(1) and if it does not contain a null character, write will reference the second parameter again, and so on until the null character signals the end of the text.

9.71. Integer procedure swoprec

This procedure is the Algol 5 version of swoprec6. It gives you direct access to a sequence of elements of 4 bytes each of a document so that they may be updated directly.

Call: swoprec(z,length)

swoprec	(return value, integer). The number of elements of 4 bytes each left in the present block for further calls of swoprec.
z	(call and return value, zone). The name of the record. Specifies further the document, the buffering, and the position of the document (see 6.1).
length	(call value, integer or real). The number of elements of 4 bytes each in the record. Length must be ≥ 0.

Except that the record length is measured in elements of 4 bytes each, swoprec works as swoprec6.

9.72. Integer procedure swoprec6

This procedure gives you direct access to a sequence of bytes of a document. The bytes become available as a zone record, and you may modify them directly without changing the surrounding elements of the document. This makes sense for a backing storage area, only.

The procedure works as a combination of inrec6 and outrec6 in the sense that it gets a sequence of bytes from a document and later transfers them back to the same place of the document. The document may be scanned and modified sequentially by means of swoprec6.

Call: swoprec6(z,length)

swoprec6 (return value, integer). The number of bytes left in the present block for further calls of swoprec.

z (call and return value, zone). The name of the record. Specifies further the document, the buffering, and the position of the document (see 6.1).

length (call value, integer, long, or real). The number of bytes in the record. Length must be ≥ 0. If it is odd, 1 is added.

Zone state.

The zone z must be open and ready for record swop (state 0 or 7, see 9.27 get zone), i.e. the zone may only have been used for record swop since the latest call of open or setposition. To make sense, the document must be a backing storage area.

Blocking

Swoprec6 may be thought of as transferring the bytes just after the current logical pointer of the document and moving the logical pointer to the last byte of the record.

Because the records are blocked, the actual transfer back to the device is delayed until the block is full or until close or setposition is called.

All bytes of the record are taken from the same block and when the block cannot supply a record requested, the block is transferred back to the document and the next block is read. The checking of all transfers takes place as described in 6.1. If the block still cannot supply the record, the run is terminated. A record length of 0 is handled as for inrec6.

If the zone contains 3 shares, one of them is used for input, while another is used for output, and the last holds the current record. This ensures maximum overlapping of computation and input-output.

Be careful to use the same share length as that with which the backing storage area was written, because the unused parts of the blocks otherwise might be treated as significant data.

Example, direct updating.

Each word of the backing storage area ma28 may be added to the corresponding word of the area ma30 in this way:

```
          begin zone ad(512*2,2,endarea),res(512*3,3,endarea);
          comment this block length is the most economical with respect
            to utilising the speed of a drum;
          procedure endarea(z,s,b); zone z; integer s,b;
          if extract 1 = 0 then goto endscan else
          stderror(z,s,b);

          open(ad,4,<:ma28:>,0); open(res,4,<:ma30:>,0);

     rep: inrec6(ad,2048); swoprec6(res,2048);
          for i:= 1 step 1 until 512 do res(i):= real(long res(i) +
                                                  long ad(i));
          comment only if we are sure that overflow will not occur;
          goto rep;
 endscan: close(ad,true); close(res,true);
```

9.73. Integer procedure system

This procedure gives access to various system and job parameters. Some of the functions of system require knowledge of the job organisation (see ref. 2) and the multiprogramming system (see ref. 1).

Call: system(fnc,i,arr) or
system(fnc,i,s)

- system — (return value, integer). Meaning depends on fnc.
- fnc — (call value, integer). Specifies the function of system.
- i — (call or return value, integer). Meaning depends on fnc.
- arr — (call or return value, array of various types). Meaning depends on fnc.
- s — (call value, string). Meaning depends on fnc.

The value of fnc is restricted to $1 \leq fnc \leq 11$, with the following meanings:

System(1,i,arr), floating point precision

- system — 0 if floating point precision was 36 bits mantissa, 1 if the precision was 33 bits mantissa.
- i — (call value, integer). Specifies the new floating point precision to 36 bits for i = 0, 33 bits for i = 1. The run starts with a precision of 36 bits.
- arr — Not used.

System(2,i,arr), free core, program name

- system — The number of bytes available in the job process for reservation of further variables. Section 8.3 gives the rules for computing the number of bytes occupied by a set of variables.
- i — (return value, integer). Gets the same value as system.
- arr — (return value, long array or real array, length > 2). The name of the document which holds the program file. The document is always a backing storage area.

System(3,i,arr), array bounds

- system — The lower index bound for arr.
- i — (return value, integer). The upper index bound for arr.
- arr — (call value, integer array, long array, real array, or boolean array). If the array is of more dimensions, the lexicographical index as defined in 5.2 is used as the value of system and i.

System(4,i,arr), file processor parameter

This call does not make sense if the program was called with the fp-command <program><integer>

- system — The separator and length for item i in the call of the program. The coding of separator and length is given in ref. 2 and ref. 6, part 1, section 2.4. System is 0 if i specifies an illegal number.

i	(call value, integer). The number of an item in the file processor command which called the program. The items are counted from 0 and up.
arr	(return value, real array, length ≥ 2). The value of item i is converted to a real and assigned to the first element of arr for an integer item, to the first and second element for a text item.

An item is a name or an integer together with the preceding separator. The following two examples show the numbering of items:

```
s source a. b          r=pip a b c
0   1    2 3           0  1  2 3 4
```

System(5,i,arr), move core area

system	1 if the moving was ok, 0 otherwise.
i	(call value, integer). The absolute address of a cell in the core store (see ref. 4).
arr	(return value, integer array, long array, or real array). System attempts to copy the core area from absolute address i and on into the first element of arr and on.

The copying stops when either arr is filled or when the word referenced is outside the core store. In the latter case system becomes 0. The copying takes place word by word, so that for instance core(i) and core(i+2) go into the first element of arr if arr is real or long.

It is necessary to move core areas in connection with some of the entries in procedure monitor, 9.40, but you may also use system(5,...) for investigation of tables in the monitor (see ref. 1) and in that way find the set of peripheral devices on the actual computer.

System(6,i,arr), own process, any message

system	The process description address for the job process, i.e. the process which executes the program (see ref. 1).
i	(return value, integer). If the message queue of the job process is empty, i becomes 0. Otherwise i becomes the buffer address for the first message in the queue.
arr	(return value, long array or real array, length ≥ 2). The name of the job process.

System(7,i,arr), primary output

system	The process description address for primary output (see ref. 1).
i	(return value, integer). The kind of the primary output process.
arr	(return value, long array or real array, length ≥ 2). The name of the console.

System(8,i,arr), parent description

system	The process description address for the parent of your job.
i	(return value, integer). The kind of the parent process (always 0).
arr	(return value, long array or real array, length ≥ 2). The name of the parent process.

System(9,i,s), run time alarm

i (call value, integer). The value to be printed following the alarm cause.

s (call value, string). The text to be printed as the alarm cause. The text should be a new line character followed by at most 8 non-blind characters. SEE EX 3 PAGE 6.17 PROC. LABELCHECK

This entry terminates the run with an alarm message similar to the standard alarms. It is intended for use in library procedures, where it may terminate the users program if he supplies wrong parameter values.

System(10,i,s) or
System(10,i,arr), parent message

system The result of the answer from the parent or 0 meaning that the message has not been sent as the message claim is exceeded. The normal result is 1 (see ref. 1).

i (call value, integer). Only significant if the third parameter is a string. If so, the value 1 will indicate a request to the parent to stop the process until the answer arrives.

s (call value, string). A text of up to 21 non-blind characters will be sent as a print message to the parent. If the text is shorter than 21 characters, the text will be supplemented with null characters. If it is longer, it will be cut to 21 characters.

arr (call and return value, integer array, long array, or real array, length >= 8 words). The contents of the 8 words will uncritically be sent as a message to the parent. If the wait indication is set in the first word (the last bit is 1, see ref. 7), the answer is awaited in the array, otherwise it is awaited in an anonymous location, and the contents of the array is unchanged.

The parent messages defined for the moment are described in ref. 7, section 10.6.

System(11,i,arr), catalog bases

This entry can only be used in system 3.

system Always = 0 (null).

i (integer) not used.

arr (return value, integer array, length >= 8). Contains the catalog bases associated with the job process.

1st and 2nd element	the catalog base
3rd and 4th element	the standard base
5th and 6th element	the user base
7th and 8th element	the max base

The user base is only defined when FP is present in the job process (cf. 8.3.2).

When Boss is the parent, the standard base gives the temp scope or the login scope, the user base gives the user scope, and the max base gives the project base. The catalog base is the base used for the moment, usually it is the standard base.

Example 1, reserving a maximum array

The following program reserves the greatest array possible at that point of the algorithm. However, the program in the inner block will probably run very slowly because of frequent transfers of program segments from the backing storage.

```
begin integer i; array arr(1:2);
begin array ra(1:system(2)free_core:(i,arr)//4);
  length:= i//4; ...
```

If you instead programmed like this

```
begin integer i; array arr(1:2);
system(2)free_core:(i,arr);
begin array ra(1:i//4-p);
```

you would have to subtract some value p corresponding to the further locations occupied by variables of the inner block.

Example 2, array bounds

An array of arbitrary dimensions might be cleared by means of the following procedure:

```
procedure clear(ra); array ra;
begin integer low,up;
  for low:= system(3)bounds:(up,ra) step 1 until up do
  ra(low):= 0
end;
```

Example 3, message buffers available

A program may find the number of message buffers it may use for communication with other processes:

```
begin integer array descr(0:34); integer i,bufs;
long array la(1:2);
comment first the process description address of the job is
  found, next the description is copied to descr;
system(5)move_core:(system(6)own_process:(i,la),descr);
bufs:= descr(13) shift (-12) extract 12;
comment the description format is given in ref. 5;
```

The program should now restrict itself to using bufs - 1 double buffered zones simultaneously.

Example 4, on-line interaction (only system 3).

Assume you want to modify the central loop of an online program. As soon you send the text 'test' to the job, the job should produce auxiliary output. As soon as you send something else to the job, the job returns to the normal mode.

This can be done by inserting the following code into suitable places of the inner loops:

```
system(6)any_message:(message,arr);
if message >¯0 then operator;
```

Here, the procedure operator sets the boolean 'testmode' which is used in the inner loops to determine whether auxiliary output is printed. Operator looks like this:

```
procedure operator;
begin zone term(10,1,stderror); long array la(1:1);
  integer array buff(1:9); integer i;
  monitor(20)wait_message:(term,i,buff);
  buff(9):= 1; monitor(22)send_answer:(z,i,buff);
  open(term,8,<:terminal:>,0);
  readstring(term,la,1);
  testmode:= la(1) = long<:test:>;
  close(term,false);
end;
```

The job must run with the job parameters 'online yes' and 'attention yes'.

Example 5, opening to a 'hidden' area

This example is only relevant in system 3.

Suppose you want to connect a zone to an area with scope project, but the area is 'hidden' behind an area with the same name on scope user. The following procedure may do the job.

```
procedure openproject(z,doc,giveup);
zone z; string doc; integer giveup;
begin zone myself(1,1,stderror);
  integer array catbase(1:8);
  open(myself,0,<::>,0);
  system(11)bases:(0,catbase);
  comment now set the catalog base to max base;
  catbase(1):= catbase(7); catbase(2):= catbase(8);
  monitor(72)set_cat_base:(myself,0,catbase);
  comment now open, create area process, establish the name
    table address and leave the zone as just opened;
  open(z,4,doc,giveup);
  inrec6(z,0);
  setposition(z,0,0);
  comment at last set the catalog base to standard;
  catbase(1):= catbase(3); catbase(2):= catbase(4);
  monitor(72)set_cat_base:(myself,0,catbase);
end;
```

Example 6, find scope of an entry

See example 3 of monitor, 9.40.

9.74. Real procedure systime

Systime gives access to the real time clock in the monitor and to the CPU time used by the job. Further, it may convert elapsed time into date and clock.

Call: systime(fnc,time,r)

systime	(return value, real). Meaning depends on fnc.
fnc	(call value, integer).
time	(call value, real). Is a time expressed in elapsed seconds since midnight 31 December 1967.
r	(return value, real). Meaning depends on fnc.

The value of fnc is restricted to $1 \leq fnc \leq 4$ and determines the meaning of systime as follows:

fnc = 1, time measuring

systime	The CPU time used by the job. The time is given in seconds with an accuracy given by the length of a time slice (usually 25.6 milliseconds).
time	Base for real time measurement.
r	Real time given as the number of seconds elapsed since the moment given by 'time'. Real time is given with an accuracy of 0.1 milliseconds, but the limited accuracy of r may cause a somewhat greater error.

fnc = 2, date and clock (ddmmyy)

systime	becomes day*100 00 + month*100 + year corresponding to time. The year is taken modulo 100.
time	The time to be converted to date and clock.
r	becomes hour*100 00 + minute*100 + second. Fractions of a second are cut off.

fnc = 3, set clock

This function is usually forbidden in a job process. If this is the case, the run is terminated.

systime	Undefined
time	The real time clock is initialised with the value of time.
r	Not changed.

fnc = 4, ISO date and clock(yymmdd)

systime	becomes year*100 00 + month*100 + day corresponding to time. The year is taken modulo 100.
time	The time to be converted to date and clock.
r	Becomes hour*100 00 + minute*100 + second. Fractions of a second are cut off.

Example 1, timing a loop

The following program prints the CPU time and real time used by a part of the program as seconds with 2 decimals:

```
cpu:= systime(1,0,time);
The program part to be timed;
cpu:= systime(1,time,time) - cpu;
comment complete timing before printing;
write(out,<<dddd.dd>,cpu,time);
```

If the time measured is short, you should compensate for the time spent by calling systime. The cpu time will depend somewhat on the activities of other processes. The real time used is highly dependent on other processes.

The real time measuring shown above will be inaccurate with about 1 millisecond for each year that has passed since 1967. This is due to the limited accuracy of the real numbers. An accuracy of 0.1 millisecond may be obtained by measuring relative to a base, like this:

```
systime(1,0,base);
cpu:= systime(1,base,time);
The program part to be timed;
cpu:= systime(1,base,t) - cpu; time:= t - time;
```

Example 2, print date and clock

```
systime(1,0,time);
write(out,<< dd dd dd>,systime(4,time,r),r);
```

will produce output like this:

```
74 05 28 22 53 37
```

9.75. Integer tableindex

This standard identifier is used by all the character reading procedures when a non-standard alphabet is selected. See intable, 9.32.

9.76. procedure tofrom

The procedure is intended for copying large sets of data to one array field from the other.

Call: tofrom(to_field,from_field,size)

to_field (return value, boolean array, integer array, long array, real array, or zone record). The contents of from_field (see below) is copied into to_field. The copying starts with the byte with index 1 and ends with the byte with index size.

from_field (call value, boolean array, integer array, long array, real array or zone record). The contents is copied into to_field. The copying starts with the byte with index 1 and ends with the byte with index size.

size (call value, integer). The number of bytes to be copied. Size must be >= 0.

The reference byte of both to_field and from_field must be a right hand byte. I.e. odd valued field variables should not be used to indicate the array parameters.

The procedure performs an action equivalent to

```
begin long field lf; integer field intf;
  boolean field bf;
  check size...;
  for lf:= 4 step 4 until size do
    to_field.lf:= from_field.lf;
  intf:= size - 1;
  bf:= size;
  if size > 1 then
  to_field.intf:= from_field.intf;
  if size > 0 then
  to_field.bf:= from_field.bf;
end;
```

The parameters are only evaluated once.

Example 1, clearing an array.

A large array can be cleared (each element is set to the binary value zero) by setting the first double word to zero and then let tofrom do the rest. Suppose that the array arr is declared

```
real array arr(low:up)
```

and that raf and raf1 are real array fields, then

```
raf:= 4*low; raf1:= raf - 4;
arr.raf1(1):= real <::>;
tofrom(arr.raf,arr.raf1,(up-low)*4);
```

may do the job.

Example 2:

See example 2 of inrec6, 9.31.

9.77. Integer underflows

This standard identifier determines the action on floating point underflow:

underflows < 0	The run is terminated when underflow occurs.
underflows ≥ 0	The value of underflows is increased by one when underflow occurs. The result of the operation which caused the underflow is 0.

When the run starts, underflows is 0. A floating-point underflow occurs when a result gets closer to zero than 1.6'-617 without being zero exactly. Because of an inconvenience in the machine structure, multiplication of 2 reals both in the range 2**(-1024) < abs r < 2**(-2048) will be classified as overflow.

Example:

See overflows, 9.49.

9.78. Integer procedure write

Prints text, numbers, and single characters on a document. Any number of such items in any sequence may be output by one call of write.

Call: write(z,one or more source parameters)

write — (return value, integer). The absolute value of write gives the number of characters printed. Write is negative if a parameter error has been encountered, otherwise write is positive.

z — (call and return value, zone). Specifies the document, the buffering, and the position of the document (see 6.1).

source — (call value, string, integer, long, real, or boolean). The source parameters specify what is to be printed.

If write is not called as a formal procedure, all parameters, which are not string expressions have been evaluated before write was entered. Now, write scans the source parameters from left to right. Each parameter is evaluated if it was not evaluated before write was entered, and then it is handled according to its type as follows:

string: A text string is printed as the corresponding sequence of characters. The null character which terminates the string is not printed. A layout string is stored and used for printing of succeeding numbers in the parameter list. Layouts are described below.

real,long,integer: The number is printed as a sequence of ISO characters according to the latest layout in the list. If no layout has appeared in the present parameter list, the standard layout << -dd.dddd> is used to print a real, and the standard layout <<_d> is used to print an integer or a long.

A real number is printed with a relative accuracy of about 6'-11, provided that the layout has a sufficient number of significant digit positions.

boolean: A boolean parameter must be followed by an integer parameter. The last 8 bits of the boolean pattern (see 3.1) are printed as a character as many times as specified by the integer parameter. If the integer is ≤ 0, nothing is printed.

If a source parameter cannot be classified as above, write will print the alarm text <:<10>***write: param<10>:>, drop the parameter and continue interpretation of its parameter list.

Zone state.

The zone must be open and ready for character printing (state 0 or 3, see 9.27, getzone6), i.e. since the latest call of open or setposition, only character output may have been made on that zone. To make sense, the document should be an internal process, a backing storage area, a typewriter, a tape punch, a line printer, a plotter, or a magnetic tape. In the latter case setposition(z,...) must have been called after open(z,...).

The first character is printed just after the logical position of the document.

When write returns, the logical position of the zone points to just after the last character printed. The zone record is not available (it is of length 0).

Blocking.

Whenever a share of the zone is filled with characters, the share is output as one block to the document and later checked as described in 6.3. This way of changing the block implies that one character more always may be stored in the block, and empty blocks may thus exist during the normal use of write.

Layouts.

The symbols of a layout give a symbolic representation of the digits, spaces, and other 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 below).

The general form of a layout is a sequence of layout characters enclosed in << >. The sequence of layout characters is composed like this:

<spaces><sign><number part><exponent part>

The number part is composed of a sequence of digit positions like this:

<first letter><d's><zeroes>

where one point representing the position of the decimal point may be inserted between two of the digit positions. A space or _ may be inserted between any two digit positions which then are separated by a space in the finally printed number.

Layout constituents:

<spaces> : consist of a (possible empty) sequence of spaces or _'s. They will appear as that many spaces in the printed number.

<sign> is empty : A positive number is printed without a position for the sign. A negative number is printed with an alarm layout (see below).

<sign> is - : The sign of the number is printed as space for a positive number, - for a negative number.

<sign> is + : The sign of the number is printed as + for a positive, - for a negative number.

<first letter> is z: Digit positions preceding the first non-zero digit are printed as zeroes. A possible sign is printed in front of the first digit position.

<first letter> is d : Digit positions preceding the first non-zero digit are printed as spaces if they are in front of the first digit position before the point, and as zeroes otherwise. The sign is printed just before the first digit printed.

<first letter> is f : Digits are printed as for <first letter> = d. The sign is printed in front of the first digit position.

<first letter> is b : Exactly as for <first letter> = d, except if all digits are 0. Then all the layout positions are printed as spaces.

<d's> : Consist of a (possibly empty) sequence of the letter d. The length of this sequence + 1 (for the first letter) gives the maximum number of printed significant digits. All numbers will be correctly rounded to the number of significant digits printed.

<zeroes> : Consist of a (possibly empty) sequence of zeroes. If a non-empty exponent part is specified, the significant digits of the number are allowed to move to the right, using the digit positions given by <zeroes>. This is done in such a way that the decimal point is kept in the position specified and the exponent part is made divisible by m + 1, where m is the number of zeroes in the layout.
Unused digit positions to the right of the point are printed as spaces.

<exponent part> is empty: No exponent part is printed as the digit positions must be able to hold the digits of the number. Otherwise an alarm layout is used.

<exponent part> is '<sign><first letter><d's>:
The exponent part is printed as the symbol ' followed by a tens exponent printed as an integer with the layout <sign><first letter><d's>. <first letter> cannot be b in this case. If <first letter> is d or f and the tens exponent is 0, the entire exponent part is printed as spaces.

Limitations:

Write refuses to print real numbers with more than 12 significant digits. If more are attempted only the first 12 are used.

The number of digit positions in front of the decimal point may not exceed 15. The number of digit positions after the decimal point may not exceed 15.

The number of digit positions in an exponent part may not exceed 3. The number of leading spaces plus the number of digit positions in front of the last space may not exceed 22.

Alarm printing:

If a negative number is printed without a sign position, a minus is inserted consuming one extra position.

If an integer is printed with a layout containing too few d's but no zeroes, no decimal point, and no exponent part, the necessary number of d's is inserted.

If a number in other cases is too large to be printed with the layout given, an exponent part is inserted with the necessary number of digit positions. An existing exponent part is just extended with one or two d's.

A number which is too small to be printed with the specified number of significant digits is printed with fewer significant digits.

Example 1:

```
write(out,<:<10>:>,false add 97,4, -12,<<_+ddd.dd>,<: and:>,13)
```

will produce this line of output:

```
aaaa -12 and +13.00
```

Example 2:
The call write(out,s,<:,:>,r,<:,:>) where s is a layout string and r is a real will print as shown below with various layouts:

```
d.dd dd        -zddd          _+fdd00          -bd.000'-d
,0.00 12,      , 0000,        , + 1230,        , -1.2      ,
,0.12 35,      ,-0012,        , -     1,       ,           ,
,-0.12 35,     , 1235,        , +12300,        , -0.012' 4,
,1.23 45'1,    ,-1235'12,     , +12300'3,      , 12.    '-4,
                                               , 12.    ' 12,
```

Example 3, tabulation:

```
write(out,false add 32,100-write(out,<:<10>:>,string text),string text2)
```

will print text2 in column 100 and on, except if text is longer than 100 characters or contains new line characters.

9.79. Zone

This delimiter occurs in declarations of zones and zone arrays and in specifications of zones and zone arrays. The formal definition is given in 5.5 and 5.6. Details about input/output are given in 6.

Zone declaration:

zone<list of zone segments>; declares one or more zones.
One zone segment is composed in this way:
<list of zone identifiers>(buf,sh,blproc)

buf	(integer). The number of elements of 4 bytes each in the entire buffer area. See below.
sh	(integer). The number of shares. See below.
blproc	(procedure with 3 parameters: a zone and two integers). The block procedure. It may be called by blockproc (see 9.6) or when an operation on a document is checked by a high level zone procedure (see 6.1).

Zone array declaration:

zone array<list of zone array declarations>;
Declares one or more zone arrays. One zone array declaration is composed in this way:
<zone array identifier>(n,buf,sh,blproc)

n	(integer). The number of zones in the zone array.
buf	(integer). The number of buffer elements of 4 bytes each in each of the n zones.
sh	(integer). The number of shares in each of the n zones.
blproc	(procedure with 3 parameters: a zone and two integers). The block procedure associated with the n zones.

Zone and zone array specification:

zone<list of zone identifiers>;
Specifies one or more formal parameters as zones.
zone array<list of zone array identifiers>;
Specifies one or more formal parameters as zone arrays.

Buffer length.

The buffer area may be divided in any way among the sh shares. The procedure 'open' will divide the buffer area evenly among the sh shares.

Shares.

Each of the sh shares may be used for one uncompleted operation on a document or for one running child process (see 5.5).

In high level zone procedures, sh specifies the number of buffers used for input/output to the document connected to the zone. In these cases sh will usually be 1, 2, or 3. Section 6.3.1 contains hints on when to use 1, 2, or 3.

Zone state.

Just after the declaration of a zone, no document is connected to the zone. The zone record describes the entire buffer area, which has an undefined content. All the shares are free and each of them describes the entire buffer area.

Example 1:

The following block head declares 3 zones. Two references to the record of 'new' are also shown. The standard zone 'out' is not accessible inside the block, because it is redeclared.

```
begin zone new,old(2*512,2,stderror),out(25,1,stderror);
  new(1):= new(1024):= 0;
```

Example 2:

Two zone arrays must be declared as shown below, because zone array za1,za2(...) is forbidden. One reference to the record of za1(1) and one to the record of za1(3) are shown. The use of a subscripted zone as a parameter is shown too.

```
begin zone array za1(3,2*512,2,stderror),za2(3,2*512,2,stderror);
real field rf;
za1(1,1024):= za1(3).rf:= 0;
open(za2(3),4 shift 12+18,<:mt123456:>,0);
```

APPENDIX A, EXECUTION TIMES IN MICROSECONDS

The times given below represent the total physical times for execution of algorithmic constituents. The total time to execute a program part is the sum of the times for the constituents. The times are only valid under the following assumptions:

1. The time for transfer of program segments from the backing storage is negligible (see 9.7 and 8.3).
2. The program is not waiting for peripheral devices (see 6.1).
3. The time slice interval is 25.6 milliseconds or more (see ref. 1).
4. The program is the only internal process running in the computer.

When the computer is time shared, assumption 4 is not fulfilled, but then the times represent the CPU time used by the program.

A.1. Operand references

Reference to local identifiers and constants	0
Reference to non-local identifiers (variable, zone, or array)	0-4

An array parameter is referenced as if it was declared locally in the outermost block of the procedure.

If a sequence of identifiers from the same non-local block are referenced without intervening references to other non-local blocks, the first reference costs 4 microseconds and the later one usually 0.

Reference to name parameter, actual is simple	9
Reference to name parameter, actual is composite	150
Reference to own variable	0-4

A.2. Constant subexpressions

Operations are performed during the translation and thus do not contribute to the execution time in the following cases:

+-*/shift extend working on constant operands.
conversion of an integer constant to a real constant, or vice versa.
real string long working on all operands.

The result of an operation performed during translation is again treated as a constant. Examples:

A(-2+6/5)	is reduced to A(-1)
1+0.5-0.25	is reduced to 1.25
p+1/2-1/4	is only reduced to p+0.5-0.25 because p+0.5 must be evaluated first.

A.3. Saving intermediate results

By the term 'composite expression' we shall mean any expression involving operations to be executed at run time. Examples:

A(2)	b+1	a shift 8	pr(i,i,<:ab:>)	are composite
11.5	real<:ab:>	5 shift 20		are not composite (see A.2)

During evaluation of expressions, one intermediate result is saved in the following cases:

+, *, and, or, all relations, shift, extract	when working on 2 composite expressions.
-, /, //, mod	when the right hand expression is composite.
add	when both operands are composite or when the left hand operand is a composite real.

The saving of one intermediate result takes

integer or boolean value saved	9
real value saved	14

Examples:

A(i)+B(i)+C(i)	uses 2 savings (+,+)
a<b and b<d	uses 1 saving (and)
a<b+c and t	uses 0 savings
a+b+2-e	uses 0 savings
a-f*(g+h)	uses 1 saving (-)

A.4. Operators

integer+integer, integer-integer	4
long+long, long-long	6
real+real, real-real	10
and, or	4
integer*integer, spill.no	16
integer*integer, spill.yes	23
integer//integer, integer mod integer	18
real*real, real/real	28
long*long	230
long//long, long mod long	425
p extract<constant>	4
p extract i	12+i/2
real add i, long add i, string add i	5
integer add i, boolean add i	4
real shift i, integer shift i	5+abs(i)/2
boolean shift i	8+abs(i)/2
entier real	25
round real	17
round long	7
extend integer	6
abs real	17
abs integer	5
abs long	17
subscripted variable with check against bounds, one subscript	21
subscripted variable without check against bounds, one subscript	14
subscripted variable for each extra subscript add	18
integer:=integer	9
integer:=long, spill.yes	16
integer:=long, spill.no	9
integer:=real	20
long:=integer	18
long:=long	14
long:=real	82
real:=integer	20
real:=long	90
real:=real	14
goto local label	7
for i:=1 step <constant> until n do, each loop	17-35
if i<j then else , all relations among integers	12
if r<q then else , all relations among reals	20
i<j, other connections	11
r<q, other connections	19
if b then, other connections	12
case i of	25
call of procedure with empty body, no parameters	150
parameter, for each parameter add	20

A.5. Execution times for certain standard procedures

Procedure	Time
arcsin	740
arctan	570
arg	740
cos	610
exor	200
exp	600
increase	165
intable	170
ln	485
logand	200
logor	200
random	235
sgn, sign	225
sin	575
sinh	715
sqrt	500
tofrom	318+13.5*double words moved

Example:

We show the computation of the time for the following loop:

```
for i:=n step -1 until j+1 do          21-35 (for do)
                                           4 (j+1)
if ia(i)=3 and                            21 (ia(i))
                                          11 (=3)
                                          13 (save, and)
   ra(i+1)>1 then                         25 (ra(i+1),+)
                                          19 (>1)
                                          12 (if then)
                                     -------
                                     126-140
p:=p + ra(i+1);                           25 (ra(i+1),+)
                                          10 (real +)
                                          14 (p:=)
                                     -------
                                     175-189
```

The result is that the loop takes from 175 to 189 microseconds when the last statement is executed, from 126 to 140 otherwise.

APPENDIX B, FILE PROCESSOR COMMANDS

The general rules for the File Processor are given in ref. 2 and ref. 6.

B.1. Call of compiler

B.1.1. Syntax

```
                     { <s> <source>   }∞
<bs file> = algol    { <s> <modifier> }0

<source> ::= <text file>

                ( { index   }             )
                ( { spill   }   { yes }   )
                ( { list    } . { no  }   )
                ( { message }             )
                ( { survey  }             )
                (                         )
                (       { yes         }   )
<modifier> ::=  ( stop. { no          }   )
                (       { <last pass> }   )
                (                         )
                (       { no                }                                          1 1 )
                ( xref. { { yes           } } { .<intervals> { .<sortarea> }       }     )
                (       { { <connections> } }                                 0     0  )
                (                         )
                (          { yes                                                    } )
                (          { no                                                     } )
                ( details. { <first pass>.<last pass>                               } )
                (          { <first pass>.<last pass>.<first line>.<last line>      } )

                  { all     }∞
                  { declare }
<connections> ::= { assign  }
                  { use     }1

<intervals>::= <first line>.<last line>{.<first name line>.<last name line>}1
                                                                            0

{ <first line>      }
{ <last line>       }
{ <first name line> } ::= <integer>
{ <last name line>  }
{ <first pass>      }
{ <last pass>       }

<sortarea> ::= <name>
```

B.1.2. Semantics

<bs file> A file descriptor describing a backing storage area. It is used as working area for the compilation, and the object code ends here and is described in the file descriptor. If <bs file> does not exist an area is created, preferably on drum. If the job has no drum resources, the area is created on the disc where the job has maximum temporary resources. After a possible creation, the area is made as large as possible leaving 1 slice on the device. An existing area is never cut, however.

In system 2 a creation of an area is only made if <bs file> is an empty note. In this case a working area of 100 segments is created. If the translation is successful, <bs file> will contain a complete object program or an external procedure (a standard procedure).

In system 3, and in system 2 with translation into an empty note, the area is cut to the segments necessary.

<source> The list of sources specifies the input files to the compiler (see 2.0.3). If no source is specified, the compiler reads the source from current input.

<modifier> The list of modifiers is scanned from left to right. Each modifier changes the variables that controls the compilation. When the scan starts, the variables are initialised to the value explained below.

index.no Code for dynamic check of subscripts against bounds is omitted. The initial setting is index.yes.

spill.yes Dynamic check of integer overflow is performed. Even if the external procedures referenced were translated with spill.no, a partial check of integer overflow is performed when they are executed (see 8.2).
The initial setting is spill.no.

list.yes The entire source text is listed on current output with line numbers in front of each line.
The initial setting is list.no.

message.no Normally, the text preceding the first begin and all comments denoted by message in the source text are listed with line numbers. With 'message.no' this listing is omitted.
The initial setting is message.yes.

survey.yes A summary is printed on current output after the completion of each pass of the translation. The meaning of the summary is explained in ref. 10.
The initial setting is survey.no.

stop.<last pass> The translation is terminated after the pass specified. Stop.yes terminates the translation after the last pass. The translation is regarded as unsuccessful.
The initial setting is stop.no.

xref.yes A crossreference list (xref-list) is printed on current output after a possible listing. The xref-list is a listing of the identifiers used in the program. The list contains an occurrence list for each identifier. The occurrence list is 3 lists of line numbers each preceded by a letter giving the kind of the list. The kind letters may be:
D(eclaration), A(ssignment), or U(sed), see further B.1.2.1
The xref-list is made with no regard to the block structure of the program. The identifier names are sorted according to the collating sequence
abcdefghijklmnopqrstuvwxyzæøå
ABCDEFGHIJKLMNOPQRSTUVWXYZÆØÅ
0123456789
Further details are found in B.1.2.1.
The initial setting is xref.no.

details.yes Intermediate output from all the passes of the compiler is printed on current output. The output may be restricted to an interval of pass numbers and to an interval of line numbers. The output from pass 8 (for instance caused by 'details.8.8') consists of a list of those line numbers which correspond to a segment boundary in the object program.
The initial setting is details.no.

B.1.2.1. Details on xref-list and -modifications

The occurrence kind is one of the three

D: meaning the identifier is found in a declaration or specification. A label is considered declared in the line where it is defined.
A: meaning the identifier occurred in front of := . A switch declaration is indicated with a D.
U: meaning all other occurrences.

The possible connections are

declare
assign
use The occurrence-lists will only be those with the specified occurrence-kinds, D, A, U respectively.
all This connection is equivalent to the connections declare.assign.use.
xref.yes is equivalent to xref.all.

The intervals may be a line interval specifying a limitation of the line numbers appearing in the occurrence lists. This line interval may be followed by a name line interval specifying that only the names that appear in this line interval will appear in the set of identifier names.

<first name line>.<last name line>
Only those identifier names, that appear in the specified part of the program, are listed in the xref-list. This parameter restricts the set of identifier names in the xref-list. If not specified, the name line interval will include the entire program.
<first line>.<last line>
The occurrence-lists will only contain line numbers belonging to the specified interval. If not specified, the line interval will include the entire program.

The sortarea is usually created by the compiler. This area is used for sorting the occurrences of the identifiers. In system 3 a part of the area used for compilation of the program is taken. In system 2 a sort area of 100 segments is created. 1 segment can hold approximately 100 occurrences. For very large programs it may be necessary to create a specific sort area. The name of this area may then be specified at the end of the list of modifications to the xref-request.

B.1.3. Examples

```
o lp
sl1=algol list.yes s sl3
```

The final program is stored in sl1. The source is taken from the file described in s followed by the file sl3. The entire source text and all error messages appear on lp.

```
sl1=algol list.yes stop.1
```

The source text is read from current input and listed on current output. The translation stops after pass 1, i.e. just after the listing.

The following examples show the calls of the compiler with xref-modification, and the corresponding output. The underlined lines are the commands.

```
algol text list.yes xref.yes
begin integer i, j;
    2   procedure pip(a,b);
    3     value a;
    4     real a; integer array b;
    5       b(a):=a;
    6
    6   switch b:=A;
    7   integer array ia(1:2);
    8
    8     goto b(increase(1));
    9 A:  pip(i, ia);
   10
   10 end
   < a FF character is printed here >

a          D:  4
           U:  2   5
b          D:  4   6
           A:  5
           U:  2   8
i          D:  1
           U:  9
ia         D:  7
           U:  9
increase   U:  8
j          D:  1
pip        D:  2
           U:  9
A          D:  9
           U:  6
algol end 11
```

algol text xref.assign

```
begin
< a FF character is printed here >
a
b          A:   5
i
ia
increase
j
pip
A
algol end 11
```

algol text xref.all.2.10.1.1

```
begin
< a FF character is printed here >
i          U:   9
j
algol end 11
```

algol text xref.all.4.7.6.6

```
begin
< a FF character is printed here >
b          D:   4   6
           A:   5
A          U:   6
algol end 11
```

B.2. Call of object program

B.2.1 Syntax

```
                          (<empty>                  )
{<name> =}1 <bs file>     (<s> <source> <anything>  )
          0               (<s> <integer>            )
                          (<s> <param> <anything>   )

<param> ::= { <integer> } . { <integer> }
            { <name>    }   { <name>    }
```

B.2.2. Semantics

<name> = Has no direct significance. However, <name> may be accessed from the running program by means of 'system'.

<bs file> A file descriptor describing a backing storage area which contains an object program from an algol translation.

<empty> The program is called with 'in' as current input.

<source> Specifies a text file to be used as 'in'. Current input is not touched in this case.

<integer> The program cannot use 'in' and 'out' and it cannot print error messages. When the program terminates, it sends a parent message corresponding to a 'break' and specifying the cause of the termination. On the other hand, 3000-4000 bytes more are available in this way. This possibility is mainly intended for operating systems, which 'never' are terminated, never use 'in' and 'out', and work satisfactorily in a very short core area.

<param> Works as <empty>. The command parameters <param> and <anything> may be accessed from the running program by means of 'system' and interpreted in any way.

<anything> See <param>.

B.2.3. Examples

```
s=algol sl2
s sl3
s
```

Translates the source program in sl2. Executes it once with input from sl3 and once with input from current input.

APPENDIX C, ERROR MESSAGES

C.1. Messages from the compiler

Four formats of error messages exist:

1. <pass number> line <line number>.<operand number> <text>
   (e.g. 6. line 12.6 type)
2. <pass number> <text> (e.g. 8. program too big)
3. <pass 9> <name> <text> (e.g. 9. write program too big)
4. ***algol <text> (e.g. ***algol param)

Below, the error messages are sorted according to <text>. The messages are classified as:

(alarm) The translation is terminated immediately as an unsuccessful execution. The program cannot be executed.

(warning) The message has no effect. The erroneous construction is skipped.

Nothing The message allows the translation to continue and the program to be executed until the erroneous construction is met or until an undeclared or doubly declared identifier is used.

C.1.1. Line and operand numbers

The lines of the program are counted 1, 2, 3, ... where line 1 contains the first 'begin' or 'external'. Only lines containing visible (printing) symbols are counted.

The operands within a line are counted 1 2, 3, ... An operand is an identifier, a constant, or a string.

The point of the program where an error of form 1 is detected, is specified by the line number and the number of operands passed within the line, for example:

```
source line 12:      if a<=1.5 then b(i):= real<:cd:>; else
operand numbers:        1  2       3 4        5
error message:       6. line 12.5 termination
```

C.1.2. Alphabetic list of error texts

algol end <i> This is not an error message. The algol program has been translated. The object code occupies <i> segments. The ok-bit (see ref. 2 and ref. 6) is set to yes. The warning -bit is set to no if no error messages have occurred, otherwise it is set to yes.

algol sorry <i> An alarm has occurred. The ok-bit is set to no (see ref. 2 and ref. 6). The integer i shows the number of segments the compiler has attempted to make.

block proc (pass 6). The block procedure of a zone is declared wrongly.

blocks (alarm, pass 5). More than 62 nested blocks.

call (pass 6). A procedure call has a wrong number of parameters.

catalog (alarm, pass 2). Trouble with reading the backing storage catalog.
(alarm, pass 9). Trouble with catalog lookup, for instance because a standard identifier has disappeared. The result of the lookup is printed.

char or illegal (warning, pass 6). Illegal character or wrong use of a graphic.

comment (pass 6). Comment or message not after begin or semicolon.

constant (pass 6). Syntactical error in a constant number.

+declaration (pass 6). Identifier declared twice or more times in the same block. The message appears at each place of declaration.

delimiter (pass 6). Impossible sequence of delimiters.

-delimiter (pass 6). Two operands follow each other.

entry (alarm, pass 9). A standard identifier has been changed in the catalog during pass 9.

error at source no: (alarm, pass 1). Trouble with input from the source file specified. Either because of hard errors, because characters > 127 are read, or because the file could not be connected.

ext param (alarm, pass 5). More than 7 parameters in an external procedure.

external (pass 6). External-end does not surround a procedure declaration.

for label (pass 6). Label which labels a statement inside a for statement is used outside.

head (pass 6). Impossible procedure head. The line number points to the first symbol of the procedure body.

kind (alarm, pass 9). A standard identifier has been changed in the catalog since the translation started. This is most likely to happen in connection with an external procedure which was translated assuming a certain standard identifier, but now this identifier has been changed in the catalog.

layout (pass 6). Impossible layout.

local (pass 6). Local variable used in array or zone declaration.

not text (pass 1). A source text contains a character > 127.

object area (alarm, ***algol). The file specified for the object code does not exist, cannot be used, or cannot be created (empty note, see B.1.2).

operand (pass 6). Operand appears in wrong context or is missing.

-operand (pass 6). Operand missing at end of construction.

overflow (pass 7). Integer or real overflow during evaluation of a constant expression.

param (warning, ***algol). Illegal parameter in the FP-command. The parameter is ignored.

pass trouble (alarm, pass 1-12). The job area is too small to load the next pass or the next pass has been destroyed.

program err 1 pass 7 (alarm, pass 7). An undetected error in the algol compiler.

program too big (alarm, pass 1-12). The backing storage area specified cannot hold the object code.

relative (alarm, pass 9). An un-debugged code procedure is assembled. The procedure contains a relative reference outside the interval 0 <= r <= 510.

right par improper (pass 6). The construction) <letter string> is not followed by :(.

sorry <i> (alarm, ***algol). The translation is unsuccessful, because of an alarm or because the FP-parameter 'stop' was used. See also 'algol sorry <i>'.

source exhausted (pass 1). The source text is exhausted before the program was complete. A clue to the missing termination is printed.

sort area (alarm, pass 12). Cross references could not be made because the sort area could not be created or connected.

stack (alarm, pass 2-12). The job area is too short for the translation tables (see 8.1.2).

subscripts (pass 6). A subscripted variable has a wrong number of subscripts.

termination (pass 6). Parantheses or bracket like structures do not match.

text (warning, pass 6). Illegal constituent of text string, usually <: or digits in < >, evt. <*, or *>

type (pass 6). The declaration or type of an operand is not in accordance with its use.

undeclared (pass 6). The identifier is not declared. Later occurrences of the identifier in the same block will not print a message.

variables (alarm, pass 5). More than 1951 bytes of simple variables and simple zones in one block, or more than 2047 bytes of owns in entire source text, or more than 2047 labels and procedures in entire source text.

works (alarm, pass 7). More than 96 bytes of working locations in one block.

xref too big (alarm, pass 12). The area used for sorting is not large enough.

zone (pass 6). Wrong number of subscripts after zone or zone array.

zone declaration (pass 6). Wrong number of commas in zone array declaration.

C.2. Messages from the running program

C.2.0. Initial alarm

Before the first begin of the program is entered, the alarm

    ***<program name> call

may appear. It is due to either: the program is not on backing storage, the source is not a text, or the job process is too short.

C.2.1 Normal form

When the program is called with <program> <integer>, a run time alarm appears as a parent message (see B.2.2).

In the normal case, a run time alarm terminates the program with a message of the form:

    <cause> <alarm address>
    called from <alarm address>
    called from ...

A list of the possible alarm causes is given in C.2.3. The program is terminated unsuccessfully except after the message 'end'.

An alarm address shows where the error occurred. If this is a procedure or a name parameter, a line specifying the call address or the point where the name parameter was referenced is printed too. The process is repeated if several calls or references were active at the time of the alarm. If more than 10 calls or references are active, the process stops after having printed the last 'called from', but before the last alarm address is printed.

An alarm address may take 3 forms:

1. name of a standard procedure or a set of standard procedures
2. line <first line> - <last line>
3. ext <first line> - <last line>

Form 2 specifies a line interval in the source text of the main program. Form 3 specifies a line interval in an external algol procedure. The accuracy of a line interval corresponds to about 16 instructions of generated code. The first line number may sometimes be 1 too great if the line is not terminated with a delimiter. The line number of a procedure call points to the end of the paranthesis.

The following alarm addresses from standard procedures are used:

char input	(read, readall, readchar, readstring, repeatchar, intable)
check	(All high level zone procedures use the check procedure)
checkspec	(The standard error actions in the check procedure)
ch/outvar	(changevar, checkvar, outvar)
invar	(invar)
long/check	(The subprocedure in the check procedure calling the user's block procedure. May also be the code performing certain operations on long)
monitor	(monitor)
open	(open)
outchar	(write, outchar, outtext, outinteger)
position	(close, getposition, setposition)
recprocs	(changerec, inrec, outrec, swoprec)
recprocs6	(changerec6, inrec6, outrec6, swoprec6) GIVEUP 0 ALGOL CHECK, POSSIBLY TOO FEW AREAS
stand.fct.1	(exp, ln, sinh)
stand.fct.2	(arctan, arg, sin, cos)
stand.fct.3	(arcsin, sqrt)
stderror	(The code giving up the run of the algol program)
system	(system, increase)
system10	(system, entries 10 and 11)
systime	(systime, logand, logor, exor)
tofrom	(tofrom)
zone declar	(The code that declares zones and zone arrays)
zone share	(getzone, getshare, setzone, setshare)
zone share6	(getzone6, getshare6, setzone6, setshare6)

C.2.2. Undetected errors

If all parts of a program have been translated with index.yes and spill.yes, the following errors may still pass undetected:

1. Parameters in the call of a procedure which is a formal parameter do not match the declaration of the corresponding actual procedure. Any reaction may result.
2. Number of subscripts of a formal array do not match the number of subscripts of the actual array. Wrong results may be produced, but the control of the program remains intact.

3. A subscript may exceed the bounds in an array declaration with more dimensions as long as the lexicographical index is inside its bounds. The control of the program remains intact.
4. The program may write into the backing storage area occupied by the program itself. Any reaction may result.
5. Undebugged standard procedures in machine language may cause any reaction.

The monitor and the operating system will usually limit the consequences of errors in such a way that no other job or process in the computer can be harmed (see ref. 1).

C.2.3. Alphabetic list of alarm causes

The error messages below cover only the standard procedures described in this manual. The set of messages is expected to grow in step with the growth of the standard procedure library.

arcsin 0	Illegal argument to arcsin.
block <i>	Too long record or record with a negative length in call of changerec6, inrec6, outrec6, or swoprec6. The block length is shown.
break <i>	An internal interrupt is detected. <i> is the cause of the interrupt, usually meaning: 0 index error in program translated with index.no 6 too many message buffers used (see 8.3.3) 8 program breaked by the parent, often because it is looping endlessly. In this case, the alarm address should be taken with some reservation. The break alarm will often be called as a result of the undetected errors described in C.2.2.
bytes <i>	Printed by stderror. The number of bytes transferred is shown. The File Processor prints the name of the document and the logical status word.
case <i>	Case index outside range. The index attempted is shown. The line number points to 'of'.
end <i>	The program has passed the final end. The integer printed after end shows the value of blocksread (see 9.7) as the program terminated. This is not an error message.
entry <i>	Illegal function code or entry conditions in a call of monitor, system, or systime. The function code attempted is shown.
exp 0	Illegal argument to exp.
field <i>	Field reference outside bounds. The illegal byte address is shown.
index <i>	Subscript outside bounds. The lexicographical index is shown. This message occurs also for subscripted zones or record variables. The character input procedures call the index alarm if they cannot assign a single result to their return parameters or if a character outside the current alphabet is met. The procedure 'check' calls the index alarm if a block procedure specifies a too long block. In this case, the value of the parameter 'b' is shown.
integer	Integer overflow.
length <i>	Illegal record length in call of inrec6, outrec6, or swoprec6. The attempted length is shown.
ln 0	Argument to ln is <= 0.

modekind <i>	Illegal modekind in call of open. The kind is shown.
movesize <i>	Tofrom is called with the number of bytes to be moved < 0. The attempted size is shown.
movefld <i>	Tofrom is called with an array where the byte numbered 1 or the byte numbered size does not exist.
oddfield <i>	Tofrom was called with an array where the word boundaries are not between an even numbered byte and its odd numbered successor. The parameter number (1 or 2) is shown.
param	Wrong type or kind of a parameter.
reclen <i>	Changevar or outvar was called with a length word < 0 or 0 < length word < 4.
real	Floating point overflow or underflow.
segment	A text seems to be a long string but could not be found as a text constant.
share <i>	An illegal share number is specified. The number attempted is shown.
sh.state <i>	A share in an illegal state is specified. The share state is shown.
sinh 0	Illegal argument to sinh.
sqrt 0	Argument to sqrt is < 0.
stack <i>	The number of variables exceeds the capacity of the job area, or an array or a zone is declared with a nonpositive number of elements. The number of bytes attempted in the reservation of storage is shown. OR TOO SMALL 'SIZE'
syntax	The program is terminated at a point where an error was detected during the translation.
value <i>	The contents of ia(i) in setzone(z, ia) or setshare(z, ia, sh) is illegal. The value is shown.
z.kind	Swoprec is not used on a backing storage area.
z.length <i>	The buffer length is too short. The actual buffer length is shown.
z.state <i>	A high level zone procedure is called in an illegal zone state. The actual state is shown.

<=> ... 3.4.7
** ... 3.3.4

Abs ... 9.1
absolute address ... 9.18
accuracy ... 3.1.6, 3.3.6, 9.7.3
actual parameter, see parameter
add ... 9.2
alarm, see error messages
algol 6 numbers ... 2.5.5, 2.5.6
algorithms for i/o ... 6.3.2
alphabet ... 2.0.1, 9.32
and ... 2.3
any message ... 9.73
arcsin ... 9.3
arctan ... 9.4
area, see backing storage
area process ... 8.1.2
arg ... 9.5
arithmetic ... 3.1.7, 3.3.6, 3.4.7
arithmetic expression ... 3.3
array ... 4.7.5, 9.73
array declaration ... 5.2
array field ... 2.8, 3.1, 4.7.3, 5.4.5, 5.7.5, 9.22.2, 9.22.3
assembly ... 8.2

Backing storage 6.1.1, 6.3.3, 9.31, 9.40
base buffer area ... 9.27
binary pattern ... 3.1.6, 3.6.5
blank, see space
block, (i/o) ... 6.3.1
block, reading of ... 9.31
block exit ... 5.5.5
block gap ... 6.1.7
block length ... 6.1.7, 6.3.3
block number ... 6.1, 6.1.7, 9.27
blockproc ... 9.6
block procedure ... 5.5.3, 5.5.4, 6.3.4
blocksread ... 9.7, C.2.3
boolean ... 2.3, 3.1.6
boolean expression ... 3.4
bound byte .. 4.7.3, 5.7, 9.22.2, 9.22.3
bounds ... 5.2.4, 9.73
bpi ... 6.1.7
branch test ... B.1.2
buffer area ... 5.5.3, 9.27, 9.79
buffer index ... 5.5.3, 9.27
buffering ... 6.6.3, 9.41
byte ... 8.3.2

CAN ... 2.0.1
card reader ... 6.1.6, 6.3.3
case ... 9.8
catalog ... 5
changerec ... 9.10
changevar ... 9.11
character classes ... 2.0.1, 9.32
character handling ... 9.21, 9.28, 9.32, 9.52, 9.53
character set ... 2.0.1
characters, 6-bit ... 6.1.7
check ... 9.12
checkvar ... 9.13
child process ... 5.5.5, 6.4.3
clock ... 9.74
clock process ... 6.1.9
close ... 9.14
code, see machine code
coding of characters ... 2.0.1
compilation ... 8.1, B.1
 - , speed of ... 8.1.3
compound symbols ... 2.3
console (see also typewriter) ... 9.73
constants ... 2.5, 4.7.5.2
control characters ... 2.0.1
controlled variable ... 4.6
conversation ... 9.58, 9.73
cos ... 9.15

Date ... 9.74
DEL ... 2.0.1
delimiters ... 2.3
density (see also mag tape) ... 6.3.3
designational expression ... 3.5
device, see documents
disc file, see backing storage
disconnected ... 6.3.3
documents (see also (i/o) ... 6.1, 9.41
driver ... 6.4.2
drum, see backing storage

EM ... 2.0.1, 2.0.3
end of document ... 6.3.3
entier ... 9.16
error messages, compiler ... C.1
 - , program ... 9.73, C.2
error reactions, i/o ... 6.3.3
event ... 6.4.3
execution ... 8.3, B.2.3
execution, speed of ... 8.3.1, 9.7, A
exit from block ... 5.5.5
exor ... 9.17
exp ... 9.18
exponentation ... 3.3.4
expression, boolean ... 3.4
 - , designational ... 3.5
 - , integer, real ... 3.3
extend ... 9.19
external ... 9.20
external procedure ... 5, 9.20
extract ... 9.21

False 2.2.1, 3.1.6
fat comma 4.7.1
FF 2.0.1
field 2.8, 3.1, 5.7, 9.22
field base 3.1, 5.2.6.2, 9.22.2
field reference 3.1, 9.22
field variable . 2.7, 3.1, 4.7.3, 5.4.5, 5.7, 9.22
file 6.1.7
- , source 2.0.3
file mark, see tape mark
file number 6.1, 6.1.7
file processor .... 2.0.3, 9.73, B
flexowriter 2.0.1, 9.32
for-statement 4.6
free core 9.73
functions 3.2, 3.3.5

Getposition 9.23, 9.27
getshare 9.25
getzone 9.27
give mask 6.3.2, 9.41
goto 2.3, 4.6.6, 5.5.5

Hard error 6.3.3
high density 6.3.3
high level zone procedures 6.2

Identifier 2.4
if-then-else 3.3.2
in 9.28
increase 9.29
index check 8.2, B.1.2
initial values 5
input, see i/o
inrec 9.31
intable 9.32
integer 3.1.6, 3.1.7, 3.3.6
internal process 6.1.8, 6.3.3
intervention 6.3.3
invar 9.33
i/o 6, 6.2
- , algorithms 6.3.2
- , check of 6.3.1, 6.3.3
- , driver for 6.4.2
- , errors 6.3.3
- , high level 6.2
- , primitive level 6.4.1
- , speed of 6.1.1, 6.3.1
- , termination of 9.58
ISO-CODE 2.0.1

Label, magnetic tape 6.3.4
labels 3.5.1, 4.6.6
layout 3.6.5, 9.55, 9.78
lexicographical ordering 5.2.6
line printer 6.1.5, 6.3.3
listing of program 9.38, B.1.2

literals, see constants
ln 9.34
load point 6.1.7, 6.3.3
logand 9.35
logical position, see position
logical status 6.3.3
logor 9.36
long 3.1.6, 9.37

Machine code 5
magnetic tape 6.1.7, 6.3.1, 6.3.3, 9.14, 9.41, 9.58
mathematical functions 3.2.4
merging 9.11, 9.31
message (comment) 9.38
message-answer 6.4.2, 9.73
message buffers 6.3.1, 8.1.2, 8.3.3
mod 9.39
mode-kind of document 9.41
monitor 6.4.1, 9.40
mount-tape-message 6.3.3, 9.41
move core 9.73

New line 2.0.4, 2.4
NL 2.0.1
normal answer 6.3.3
null character 2.0.1, 9.58
numbers 2.5
numerical functions 3.2.4

Object program, see execution
of 9.8
on-line interaction 9.73
open 9.41
operating system 6.43
or 2.3
out 6, 9.42
outchar 9.43
outinteger 9.44
output, see i/o
outrec 9.46
outtext 9.47
outvar 9.48
overflow (see also spill) .. 3.3.6, 9.49
overflows 9.49
overrun 6.3.3
own 4.7.9, 5.2.1

Packing 9.2
paper tape punch, see tape punch
paper tape reader, see tape reader
parameter, actual-formal .. 4.7.1, 5.4.5
parameter, file processor 9.73
parameter comma 4.7.1
parent 9.73
parent message 6.3.3
parity error 6.3.3
partial word 9.27

passes of compiler 8.1.1
pattern, see binary pattern
peripheral device, see documents
physical position, see position
position (logical or physical) 6.1, 6.3.1, 9.27
position error 6.3.3
positioning of magnetic tapes 9.14,9.58
precedence of operators ... 3.3.5, 3.4.6
precision, see accuracy
primitive level zone procedures 6.4
printer, see line printer
printing, see write
procedure call 4.7
procedure declaration 5.4
program 2.0.2, 4.1.1
program execution, see execution
punch, see tape punch

Random 9.50
range of values 3.1.6, 3.3.6
read 9.51
readall 9.52
readchar 9.53
reader, see tape reader
readstring 9.54
real 3.1.6.4, 3.1.7, 3.3.5, 3.3.6
real (operator) 9.55
record 5.5.3, 9.79
record, i/o of 6.2
record variable 3.1.3, 5.5.3
recursive procedures 4.7.9
reference byte 9.22.2
rejected 6.3.3
relational operators 3.4.7
release 9.14
repeatchar 9.56
representation of variables 3.1.6,3.6.5
requirements of compiler 8.1.2
requirements of program 8.3.2
round 9.57
run time, see execution

Scope 5, 5.5.5, 5.6.5
scope of area 9.40, 9.73
segmentation 8.3.1, 9.7
segments 6.1.1
setposition 9.27, 9.58
setshare 9.60
setzone 9.62
sgn 9.63
share descriptor 5.5.3, 6.4.1, 9.25
shared area 5.5.3, 6.3.1, 9.41
shares, no. of 6.3.1
shift 9.64
side-effect 3.1.3, 3.3.5, 4.7.3
sign 9.65
sin 9.66
sinh 9.67
source program 2.0.2
SP 2.0.1
space 2.0.4, 2.3, 2.4
specifications 4.7.5, 5.4.5
spill 3.3.6, 4.2.3, 8.2, B.1.2
splitting 9.21
sqrt 9.68
standard error (i/o) 6.3.3
standard identifiers 3.2.4, 5
standard procedures 9
state table 9.53
statements 4
status, see logical status
stderror 9.69
stopped 6.3.3
storage requirements 8.1.2, 8.3.2
string (operator) 9.70
string expressions 3.6, 4.7.1
string, short, long 3.6.3
string variable 9.55, 9.70
strings (constants) 2.6
subscript check 8.2, B.1.2
subscripts, no. of 4.7, 5.3
suspend 9.14
switch 3.5.6
swoprec 6.3.1, 9.72
syntax check of data 9.52, 9.53
syntax of chapter 9 9
system 9.73
system control 7
systime 9.74

Table, initialisation of 9.8
tableindex 9.32, 9.75
tape, see magnetic tape
tape mark 6.1.7, 6.3.3, 9.14, 9.30, 9.58
tape punch 6.1.4, 6.3.3
tape reader 6.1.3, 6.3.3
terminate i/o 9.58
text portion 3.6.5
time measuring 9.74
timer 6.3.3
tofrom 9.76, A.5
transfer functions 3.2.5
translation, see compilation
true 2.2.1, 3.1.6
type 3.3.4, 5.4.5
type length 3.1.6, 9.22.1
type transfer, see transfer functions
typewriter 6.1.2, 6.3.3

Underflow 3.3.6
underflows 9.77
underlining 2.0.4, 2.3
unintelligible 6.3.3
unpacking 9.21

Value 4.7.5, 5.4.5
values 3.1.6, 3.1.7, 3.6.5, 3.7.3
variable field 2.8, 3.1, 5.7.4, 9.22.2, 9.22.3
variable length of record ... 9.31, 9.46
variable string 9.55, 9.70
variables 3.1

Word 8.3.2
word boundaries .. 5.2.8, 9.22.1, 9.22.3
word defect 6.3.3
write 9.78
write enable 6.3.3

Zero, real 3.1.6
zone 3.7, 5.5, 7.79
zone array 3.7, 4.7, 5.6
zone buffer, see buffer area
zone declarations 5.5.3, 5.6, 9.79
zone descriptor 5.5.3, 9.27
zone expressions 3.7, 4.7
zone record, see record
zone state 9.27

HEADQUARTERS: FALKONER ALLÉ 1 · DK-2000 COPENHAGEN F · DENMARK
TELEPHONE: (01) 10 53 66 · TELEX: 16282 RCHQ DK · CABLES: REGNECENTRALEN

AUSTRIA
DENMARK
ENGLAND
FINLAND
GERMANY
HOLLAND
NORWAY
SWEDEN

Printed in Denmark by R. Roussell

ISBN 87 7557 018 1 (AUG/74)