RC9000-10/RC8000

SW8585 Compiler Collection

ALGOL8 User's Guide, Part 2

PN 99200389

RC Computer

Keywords:
RC9000-10, RC8000, Compiler, ALGOL, ALGOL8, User's Guide

Abstract:
This manual describes the standard procedures of the ALGOL language for RC9000-10 and RC8000 systems.

Date:
March 1989.

PN: 991 11280

Table Of Contents

1. Introduction 1

2. Procedure Description 3

2.1 abs 3
2.2 activate 4
2.3 activity 6
2.4 add 14
2.5 alarmcause 16
2.6 algol 17
2.7 and 19
2.8 arcsin 20
2.9 arctan 21
2.10 arg 22
2.11 array 23

2.12 blockproc 24
2.13 blocksout 25
2.14 blocksread 26
2.15 boolean 27
2.16 bracket 28
2.17 buflengthio 29

2.18 case 31
2.19 changekey6 33
2.20 changerec 34
2.21 changerecio 35
2.22 changerec6 37
2.23 changevar 39
2.24 character constant 43
2.25 check 44
2.26 checkvar 45
2.27 close 47
2.28 closeinout 49
2.29 closetrans 51
2.30 comment 52
2.31 comment string 53
2.32 context 54
2.33 continue 55
2.34 cos 56

2.35 deadsort 57
2.36 decimal point 59

2.37 disable 60
2.38 divide 61

2.39 endaction 62
2.40 entier 63
2.41 equal 64
2.42 equivalent 65
2.43 errorbits 66
2.44 exit 67
2.45 expellinout 68
2.46 exor 71
2.47 exp 73
2.48 exponentiation 74
2.49 extend 76
2.50 external 77
2.51 extract 78

2.52 f8000table 81
2.53 field 82
2.54 fpmode 83
2.55 fpproc 84

2.56 getalarm 86
2.57 getf8000tab 88
2.58 getposition 89
2.59 getshare 90
2.60 getshare6 91
2.61 getstate 93
2.62 getzone 94
2.63 getzone6 95
2.64 goto 102
2.65 greater than 103
2.66 greater than or equal 104

2.67 implication 105
2.68 in 106
2.69 increase 107
2.70 initkey 108
2.71 inoutrec 110
2.72 initsort 115
2.73 initzones 116
2.74 inrec 118
2.75 inrec6 119
2.76 intable 122
2.77 integer 125
2.78 integer divide 126
2.79 invar 127
2.80 isotable 132

2.81 ldink 133
2.82 ldunlink 137
2.83 len 139
2.86 lifesort 142
2.87 ln 143
2.88 lock 144
2.89 locked 147
2.90 logand 148

2.91 logor 150
2.92 long 151
2.93 long 152

2.94 message 153
2.95 minus 154
2.96 mod 156
2.97 monitor 157
2.98 movestring 184
2.99 multiply 185

2.100 newactivity 186
2.101 newsort 191
2.102 not 194

2.104 open 196
2.105 openinout 204
2.106 opentrans 206
2.107 openvirtual 208
2.108 or 210
2.109 out 211
2.110 outchar 212
2.111 outdate 213
2.112 outindex 214
2.113 outinteger 215
2.114 outrec 217
2.115 outrec6 218
2.116 outsort 220
2.117 outtable 223
2.118 outtext 224
2.119 outvar 226
2.120 overflows 228
2.121 own 229

2.122 passivate 230
2.123 plus 231
2.124 pos 233
2.125 priority 234
2.126 progmode 235
2.127 progsize 236

2.128 random 237
2.129 rc8000 238
2.130 read 239
2.131 readall 243
2.132 readchar 250
2.133 readfield 253
2.134 readstring 255
2.135 real 258
2.136 real 259
2.137 repeatchar 261
2.138 replacechar 262
2.139 resetzones 264
2.140 resume 265
2.141 round 266

2.142 setfpmode 267

2.143 setposition 268
2.144 setshare 272
2.145 setshare6 273
2.146 setstate 274
2.147 setzone 275
2.148 setzone6 276
2.149 sgn 277
2.150 shift 278
2.151 sign 279
2.152 sin 280
2.153 sinh 281
2.154 sortcomp 282
2.155 sqrt 283
2.156 startsort6 284
2.157 stderror 288
2.158 stopzone 289
2.159 string 290
2.160 string 291
2.161 swoprec 293
2.162 swoprec6 294
2.163 system 297
2.164 systime 308

2.165 tableindex 312
2.166 tofrom 313
2.167 tofromchar 315
2.168 trap 317
2.169 trapmode 321

2.170 underflows 322

2.171 value 323
2.172 virtual 324

2.173 wactivity 325
2.174 waittrans 327
2.175 write 331
2.176 writefield 339
2.177 writeint 342

2.178 zone 344

Appendix A. References 347

Appendix B. Survey of Format8000 Transactions 350

1. Introduction

This manual gives a detailed description of all the standard identifiers, standard procedures, operators, and a few other things supplied as a part of the compiler, and all the delimiters found in ALGOL8.

The standard identifiers and standard procedures may be used in RC FORTRAN programs as well as ALGOL programs.

The syntax of the operators and other delimiters is described rather informally. Take the description of 'divide' as an example.

```
Syntax          : <operand1>/<operand2>
Priority        : 3
Operand types   : integer, long, or real
Result type     : always real
```

This shows that 'divide' has two operands with type < > boolean. The result of applying 'divide' to these two operands is always of type real. The priority of operators is listed in the description of priority in this manual. The priority with value 1 is the highest.

The description of procedures follows a different sceme.

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). Note, that the evaluation of a string parameter will access the actual parameter repeatedly until the string end is supplied (cf. (15)).

The type string is unknown in FORTRAN. Literal hollerith strings of any length may be packed in arrays of any type located in COMMON blocks by DATA statements, and short hollerith strings may be assigned long variables.

The FORTRAN arithmetic types DOUBLE REAL and COMPLEX are not known to any of the standard procedures, but arrays of these types may be transferred as parameters to many procedures anyhow, in each case specified in the manual.

Throughout the manual, the names ALGOL5, ALGOL6, ALGOL 7 and ALGOL8 appear, so a short explanation is needed:

The names are not identical to any specific version or release of the compiler/library/runtime system, rather to a sequence of releases, introducing new concepts.

The name ALGOL1 goes all the way back to the DASK computer, the names ALGOL2, 3 and 4 were the compilers for the GIER computer, while ALGOL 5 became the name of the first compiler for RC4000, released in 1969. ALGOL 6, released in 1972, introduced the new types long and field, being it simple fields or array fields. A number of procedures were added to the library, some of which still have the letter '6' appended to the name.

ALGOL 7, released in 1976, introduced the concept of context blocks along with the language elements 'while' and 'repeat' statements. To the libary were added the new standard variables alarmcause, trapmode, RC8000, errorbits, blockout and progmode and the new procedures trap, lock, locked, initzones, openvirtual and resume. In 1975 RC8000 had been released and the compiler/library/runtime system migrated to that computer as well.

ALGOL 8, released in 1979, introduced the concept of activities, supported by a number of procedures in the library. In the library appeared the new standard variable endaction and the FORMAT8000 procedures, along with the procedures getalarm and virtual.

The release numbering started in 1978, with release 10.03 as the first one, containing ALGOL 7. The next release, 11.0, in 1979 contained ALGOL8 and this has been the name of the compiler/library/runtime system ever since.

The present release is named SW8500/2, release 4.0, 1989.02.01.

2. Procedure Descriptions

2.1 abs

This delimiter, which is a monadic arithemtic operator, yields the absolute value of the operand.

Syntax:

```
abs <operand>
```

Priority: 1

Operand type:

integer, long, or real

Result type:

The result is of the same type as <operand>

Semantic:

The absolute value of the argument is evaluated.

Example 1:

```
abs n
if abs sin(x) <= 0.5 then ...
j:= abs (0.5 + s/q)
```

2.2 activate

This long standard procedure activates (restarts) a non-empty activity at its restart point.

Call:

```
activate (actno)
```

```
activate  (return value, long). The value is composed by
          two integers
          activity_no shift 24 add cause
          describing the way in which activate returns
          (see below).

actno     (call value, integer). The identification of
          the activity to be activated. This activity
          must be non-empty. The value may be negative,
          in which case the activity identified by abs
          actno is "killed" rather than restarted (see 6,
          below).
```

Function:

The call:

```
result:= activate (actno)
```

proceeds as follows:

1) If the program is not in monitor mode or is not in activity mode the run is terminated with an alarm.

2) If actno = 0 or abs actno > max_no_of_activities the run is terminated with an alarm.

3) If the program is in monitor mode and actno designates an empty activity, the call returns with the result = activity_no shift 24 add (-1). If the program is in activity mode and act no designates an empty activity, the run is terminated with an alarm.

4) If the activity is virtual (see newactivity), and a shared virtual activity occupies the common stack area, waiting for an i/o transfer to be completed, the call returns with the result = shared_virtual shift 24 add (-2).

5) If the activity is virutal, and a shared virtual activity occupies the common stack area, not waiting for completion of an i/o transfer, the stack area is transferred to the virtual stora ge file in one output operation, and the virtual record of the new activity is transferred from the virtual storage file to the stack area in one input operation (swop).

6) If actno > 0, the designated activity is activted (restarted) at its restart point. If the program is in activity mode (i.e. activate is called from an activity), the activate call at the same time defines a restart point for the calling activity, and when the calling activity later on is activated, the cause returned will be 3.

   If actno < 0, the absolute value of actno designates the activity. In this case the restart is just the execution of an implicit run time alarm statement, with the text: "killed", and alarm cause = extend 0 add (-15). The alarm statement is executed, as if it was the first statement at the restart point.

Result values:

The return values activity_no and cause have the following meaning:

`activity_no`: The identification of the activity causing the return. In case of cause = -2, the activity waiting for an i/o operation to complete.

The value of cause indicates the return cause:

-3: activate returned because of an alarm in a started activity, which is now empty.

-2: activate returns, because the designated activity is virtual and shares storage with another activity waiting for an i/o operation to complete.

-1: activate returns because the designated activity is empty.

0: activate returns because a started activity terminated via its final end. This activity is now empty.

1: activate returns because a started activity is passivated by a programmed passivate statement.

2: activate returns because a started activity is passivated by an implicit passivate statement in the zone i/o system.

3: activate is called in activity mode, defining a restart point - and calling activity is resumed.

Example 1:

See Example 1 of activity.

Example 2:

See Example 2 of newactivity.

2.3 activity

This standard procedure creates a number of activity descriptors defining empty activities.

Call:

`activity (max_actno)`

`max_actno` (call value, integer). The number of activity descriptors to be created.

The program must be in neutral mode, which is the default mode of programs. The lexicographical enclosing block is now defined as the monitor block. When the program leaves this block, either by a goto statement or through its terminating end, the activity descriptors (and thereby the activities) are removed (including stop of pending zones), and the program mode returns to neutral mode.

An activity is an entity, which permits procedures to act as coroutines, and it is identified by a positive integer. Activities are operated by the following five procedures:

- activity (n); creates n activity descriptors, defining n empty activities, now identified:
  1, 2,...,n
  The mode is transferred from neutral mode to monitor mode.

- newactivity (actno, virt, p,...); initializes the empty activity actno, with the procedure p. The mode is transferred from mo nitor mode to activity mode.

- activate (actno); resumes the activity actno, at its restart point. The mode is transferred from monitor mode to activity mode, or is left unchanged in activity mode.

- passivate; deactivates executing activity, establishing its restart point. The mode is transferred back to monitor mode.

- w_activity (buf); waits for an event (message or answer) in the event queue (similar to the call: monitor (24,...)), supplying the identification of the responsible activity. The mode is left unchanged in monitor mode.

Example 1: A coroutine system

The following coroutine system runs controlled by the coroutine monitor described in ref. (21). The system reads input data from a number of FORMAT8000 terminals (see (17) and (18)), and spools the data on current output.

The system contains the following coroutines:

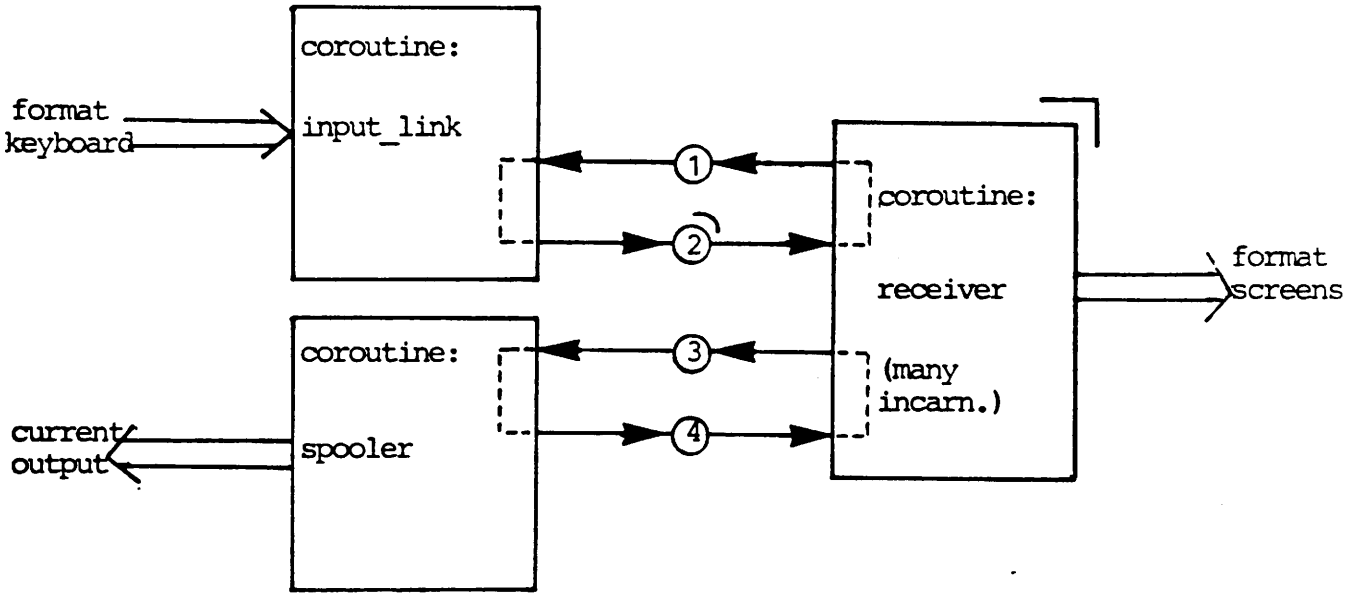
Procedure inputlink:
this coroutine has one semaphore (see ref. (21)) 'transinput' from which the next free buffer is awaited for. Into this buffer is transferred the transaction head from the next FORMAT8000 transaction from the zone 'input' (by wait trans). The contents of the buffer is signalled to a semaphore 'screen' depending on cu/dev (destination), the number of screen semaphores being the same as the number of screens.

Procedure receiver:
This coroutine runs in one incarnation per screen, each having its own screen semaphore. Each incarnation of the procedure
- receives a buffer from this screen semaphore,
- copies it to a buffer from the semaphore 'spool_pool',
- transfers the rest of the input transaction to this buffer, and
- signals it to the sempahore 'spool'.

When a receiver coroutine is started a screen mask is initialized (as shown in Example 3 of writefield).

Procedure spooler:
This coroutine has one semaphore, 'spool', from which buffers are received. The contents of the buffer are printed on current output as an exact copy of the screen picture (incl. the mask).

The system can be outlined in the following figure:



1: transinput semaphore
2: screen semaphores
3: spool semaphore
4: spool_pool semaphore

The coroutine monitor is described in details in ref. (21), here it is only sketched:

```
begin <*coroutine monitor*>
   integer array                 sem (1:no_of_sem), buf (2:buf_arr_length),
                                 corou1 (2:4*maxactivity + 2);
   integer array field           ready_q_first, ready_q_last, c_corout, data;
   integer field                 next, home_pool, operation, length,
                                 corout_no, incarn, bf;
   integer                       buf_base, buffer_addr, i, global_count,
   c_corout_no,                  sems_pr_incarn, buffer_head_length;
   real                          cputime, realtime;
   boolean                       finished;
   zone                          input (128,1, sense), output (128,1,
                                 stderror);

<*the contents of the following procedures are listed in ref. (21)*>

integer procedure sem_to_field (semno, incarnation);
integer procedure get_buf (bsize);
procedure init_buffer_pool (semno, incarnation, nb, bsize);
procedure init_corout_descr (actno, incarnation);
procedure empty_sem(s);
procedure signal (s,b);

integer procedure wait (s);
integer field          s;
begin
   <*passivates the coroutine until something arrives on the semaphore 's'*>
   integer array field i;

   if sem.s > 0 then
   begin <*the semaphore is open, i.e. a buffer is ready*>
      corout.c_corout.bf:= sem.s;
      i:= sem.s;
      sem.s:= buf.i.next;
   end open
   else

   begin    <*the semaphore is neutral or closed -queue the coroutine up on the
            semaphore*>
   corout.c_corout.next:= 0;
   if sem.s = 0 then
   sem.s:= -c_corout <*neutral*>
   else
   begin <*closed*>
      i:= -sem.s;
      while corout.i.next <> 0 do i:= corout.i.next;
      corout.i.next:= c_corout;
   end closed;

   passivate;          <*wait here*>

   end neutral or closed;
   wait:= corout.c_corout.bf;
end wait;

procedure sense_ready (z);
zone z;
```

```
begin
   getshare6 (z,ia,1);
   ia(4):= 2;<*sense*>
   setshare6 (z,ia,1);
   repeat monitor (16 <*send mess*>, z, 1, ia)
   until  monitor (18 <*wait answ*>, z, 1, ia) = 1
   and    ia(1) shift (-21)<*timer*> extract 1 = 0;
end sense_ready;

procedure sense (z,s,b); <***block proc for input***>
zone z; integer s,b;
begin
   if s extract 1 = 1 then stderror (z,s,b);
   if b = 0 then
   begin
      sense_ready (z);
      getshare6 (z, ia, 1);
      ia(4):= 3 shift 12; <*input*>
      setshare6 (z, ia, 1);
      monitor (16<*send mess*>, z, 1, ia);
      monitor (18<*wait answ*>, z, 1, ia);
      b:= ia (2);
   end;
end sense;

procedure claim(i); integer i;
begin integer array ii(1:i); end;

<*coroutine and variable declarations*>
   algol copy. coroutdecl;

<*initialize coroutine monitor*>
   . . . <*init field variables and clear arrays*>

   open (input,8,input_name, 1 shift 9 + 1 shift 1);
   monitor (8<*reserve process*>,z,0,ia);
   open (output,14,output_name,0);

activity (max_activity);

<*initialize coroutines and semaphores*>
   algol copy.initcor;

<*central logic of coroutine monitor*>
   finished:= false;
   repeat
      c_corout_no:= 0;
      if ready_q_first <> 0 then
      begin
         <*start the first coroutine in the ready queue*>
         - - -
      end
      else
      begin
         <*ready queue is empty, wait for an external event*>
         for i:= w_activity (buffer_addr) while i<0 do;
```

```
      if i=0 then
      begin <*message received, decode message*>
      - - -
      end
      else
      begin <*answer to a coroutine received*>
         c_corout_no:= i;
         c_corout   := 8*c_corout_no - 8 + 2;
      end answer;
   end ready queue empty;


   if c_corout_no > 0 then
   begin <*start the selected coroutine*>


      activate (c_corout_no);

      buffer_addr:= 0;
   end start coroutine;
until finished;
end program;
```

The coroutine declarations (in the file coroutdecl) appear as follows:

```
integer array screen_semaphore (1:max_incarn+1);

<******* coroutine *******>
procedure spooler (spool, spool_pool);
value                          spool, spool_pool;
integer                        spool, spool_pool; <*semaphores*>

begin
   integer array field bf;
   long    array field text;
   integer       field line;

   claim (400);        <*reserve stack space for the activity, see
                       newactivity*>

   repeat
      bf:= wait (spool);
      write (out
      _       <:<10><10> format::>,buf.bf.data(1),
      _       <: dest::>, buf.bf.data(2) shift(-12),
      _                   buf.bf.data(2) extract 12,
      _       <: aid::>,  buf.bf.data(3),
      _       <: cursor::>,buf.bf.data(4));

      line:= text:= data + 10;
      while buf.bf.line >= 0 do
      begin
         write (out,<:<10>line:>, <<dd>, buf.bf.line,
         _       <: :>, buf.bf.text);
```

```
      end lines;
      signal (spool_pool, bf);
      setposition (out,0,0);
   until false;
end spooler;

<******* coroutine *******>
procedure receiver (transinput, screen, spool, spool_pool);
value                          transinput, screen, spool, spool_pool);
integer                        transinput, screen, spool, spool_pool);
begin
   integer array field trans, bf;
   integer                        i, aid, dest;

   claim(400);

   trans:= wait (screen);
   dest:= buf.trans.data(2);
   set_mask (dest); <*set up a screen mask, see ex. 3 of writefield*>
   signal (transinput,trans);

   repeat
      bf := wait (spool_pool);
      trans:= wait (screen);
      aid := buf.trans.data(3);
      for i:= 1 step 1 until 4 do
         buf.bf.data(i):= buf.trans.data(i);
      if aid = 1 <*send*> then get_input_data (buf.bf.data);

      opentrans (output, 3, dest, 63 <*erase unprotected*>, 3);
      closetrans (output);
      signal (transinput, trans);
      signal (spool, bf);
   until aid = 17 <*clear*>;

   finished:= true;
end receiver;

<******* coroutine *******>
procedure input_link (transinput, screen);
value                             transinput;
integer                           transinput;
integer                                          array screen;
begin
   integer array field trans;
   integer                        screen_no;

   claim (300);
   sense_ready (input);

   repeat
      trans:= wait (transinput);
      waittrans (input,
      _                buf.trans.data(1), <*format*>
      _                buf.trans.data(2), <*dest *>
      _                buf.trans.data(3), <*aid *>
```

```
      _                buf.trans.data(4)); <*cursor*>
      screen_no:= buf.trans.data(2) extract 12 + 1;

      signal (screen(screen_no), trans);
   until false;
end input_link;

procedure set_mask (value dest; dest); <*see ex. 3 of writefield*> integer
dest;
begin
   integer line;
   opentrans (output,3,dest, 49, 3);
   for line:= 0 step 1 until 23 do
   begin
      writefield (output, 1, line*80);
      writefield (output, 2, 1 shift 5);
      write (output, <:line:>, <<dd>, line);
      writefield (output, 2, 0);
      if line=0 then writefield (output, 3, 0);
      writefield (output, 6, 'sp' shift 12 + line*80 + 79);
   end;
   closetrans (output);
end setmask;

procedure get_input_data (datapart);
integer array                          datapart);
begin
   integer field line;
   array       field text;
   integer               type, addr;

   line:= text:= 10;
   while read_field (input, type, addr) = 8 do
   begin
      datapart.line:= addr//80;
      readstring (input, datapart.text, 1);
      line:= text:= line+50;
   end;
   datapart.line:= -1;
end get_input_data;
```

The initialization of coroutines and semaphores (in the file initcor) is:

```
sems_pr_incarn:= 1;
for inc:= 0 step 1 until max_ incarn do
init_buffer_pool (1,  <*screen semaphore*>
                          inc, <*incarnation*>
                          0,0); <*no.of buffers, buf.size*>

inc:= max_incarn + 1;
init_buffer_pool (1,  <*transinput semaphore*>
                          inc, <*incarnation*>
                          1,4)+ <*no.of buffers, buf.size*>
```

```
init_buffer_pool (2, <*spool semaphore*>,
                          inc, <*incarnation*>
                          0,0); <*no.of buffers, bufsize*>

init_buffer_pool (3, <*spool pool semaphore*>
                          inc, <*incarnation*>
                          4,800); <*no.of buffers, bufsize*>

<*start the activities*>
for inc:= 0 step 1 until max_incarn do
begin
   init_corout_descr (inc+1, inc);

   new_activity (inc+1, 0, receiver,
   _                    sem_to_field (1, max_incarn+1),
   _                    sem_to_field (1, inc ),
   _                    sem_to_field (2, max_incarn+1),
   _                    sem_to_field (3, max_incarn+1));

   end;

   inc:= max_incarn+1;
   init_corout_descr (max_incarn+2, 0);

   new_activity (max_incarn+2, 0, spooler,
   _                    sem_to_field (2, inc),
   _                    sem_to_field (3, inc));

   for inc:= 0 step 1 until max_incarn do
      screen_semaphore (inc+1):= sem_to_field (1,inc);

   inc:= max_incarn+1;
   init_corout_descr (max_incarn+3,0);

   new_=activity (max_incarn+3, 0, input_link,
   _                     sem_to_field (1, inc),
   _                     screen_semahore);
```

2.4 add

This delimiter, which is a pattern operator, is used for packing of integer values into a real, long, integer, boolean, or string value.

Syntax:

```
<operand1> add <operand2>
```

Priority: 2

Operand types:

```
<operand1>: boolean, integer, long, real, or string.
<operand2>: integer, long, or real.
```

Result type:

The result is of the same type as <operand1>.

Semantic:

If <operand2> is real or long it is rounded to an integer. Now both operands are treated as binary patterns and the right hand integer is added to <operand1> 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.

Example 1:

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

```
b:= false add i 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;
```

or:

```
r:= real (extend i1 shift 24 add i2);
```

Note that the binary pattern of a negative number has zeroes in front of the 24 ordinary bits.

Example 3:

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

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

2.5 alarmcause

This long standard identifier contains the cause of a suppressed program break down.

The value of alarmcause is interpreted as two integers:

```
param:= alarmcause shift (-24)
cause:= alarmcause extract 24
```

where param is the integer output by the normal error message (cf. (15)) and cause is the errortype (e.g. the error message in dex 100 corresponds to param = 100 and cause = -2).

The value of cause means:

-16	index alarm (or zone index alarm) from ix instruction
-15	killed activity
-14	goto alarm
-13	trap alarm
-12	field alarm
-11	give up from stderror (algol check)
-10	end program (normal program termination)
-9	break
-8	param error
-7	real overflow
-6	integer overflow
-5	syntax error
-4	case alarm
-3	zone index error
-2	index alarm
-1	stack alarm
>=0	other alarms (e.g. sqrt, ln, etc.)

Example 1:

Alarmcause can be used to check why a traproutine is called (see Example 2 in trap).

Example 2:

See Example 1 of getalarm.

2.6 algol

This delimiter, which is a compiler directive, is used to call specific actions of the compiler.

Syntax:

```
algol [<modifier>]  [copy.<source>]  ;

<modifier>::= (list.on/list.off)

<source>  ::= (<name>    )
              (<integer>)
```

Only allowed after the delimiters begin or ;
(semicolon)

Semantic:

The modifiers list.on and list.off may be used to direct whether or not subsequent parts of the program should be listed during compilation. The ef fect of the list modifier may be overruled by use of the list modifiers of the FP command calling the compiler (cf. (15)).

The modifier copy.<source> makes it possible to copy the specified <source> into the main source text. The copying will take place when the delimiter is met.

Example 1:

```
algol list.on; <*change to listing.yes*>
```

Example 2:

```
algol list.off; <*change to listing.no*>
```

Example 3:

```
algol list.off copy.text;
<*copy text with listing.no. The listing modifier is only
valid for text*>
```

Example 4:

```
algol copy.t;<*copy t, listing as stated in the call of
              the compiler*>
```

Example 5:

```
algol copy.2;<*in case the call of the compiler specifies
               the parameter:
               copy.text2.text3 as the first 'copy'
               parameters, text3 is copied.
               A possible listing modifier in front of
               copy.2 will be blind (is overruled by the
               listmodifiers in the call of the
               compiler*>
```

Example 6:

See Example 1 of activity.

2.7 and (and &)

This delimiter, which is a logical operator, yields the logical and (logical product) of the two operands.

Syntax:

```
<operand1>  {and}  <operand2>
            { & }
```

Priority: 7

Operand types: boolean.

Result type: boolean.

Semantic:

The logical product of the operands is evaluated. This logical product is performed bit by bit in parallel on the twelve bits of the two operands. The truth value of the result pattern, when used in repetitive or conditional statements, is determined by the last (rightmost) bit in the pattern (0 = false, 1 = true).

Example 1:

Set the last 3 bits of a boolean.

```
a:= b and (false add 7);
```

Example 2:

```
if a and b then ...
else ...
```

2.8 arcsin

This real standard procedure performs the mathematical function arcsine (i.e. the inverse of the trigonometric function sine).

Call:

```
arcsin (r)
```

```
arcsin   (return value, real). Is the mathematical
         function arcsine of the argument r, in radians
         with - pi/2 <= arcsin <= pi/2
         pi = 3.14159 26536.

r        (call value, real, long, or integer) Must be in
         the interval: -1 <= r <= 1.
```

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

Example 1:

The sides of a triangle are a, b, c. The angle A (opposite a) can be calculated by:

```
pi := 2*arcsin(1);
s := (a+b+c)/2;
A := 2*arcsin(sqrt((s-b)*(s-c)/(b*c)));
if A <0 then A := A + pi;
```

If A is wanted in degrees then

```
A := A*180/pi;
```

2.9 arctan

This real standard procedure performs the mathematical function arctangent (i.e. the inverse of the trigonometric function tangent).

Call:

```
arctan (r)

arctan   (return value, real). Is the mathematical
         function arctangent of the argument r, in
         radians with -pi/2 <= arctan <= pi/2
r        (call value, real, long, or inte ger) in
         radians.
```

Accuracy:

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

When inserting into formulae containing a term like arctan (b/a) where a may be zero, the function arg (a,b) may be a better choice as it behaves sensible for a = 0.

2.10 arg

This real standard procedure performs the mathematical function of calculating the argument (also called angle) of a complex number from its cartesian coordinates.

Call:

```
arg (u, v)

arg     (return value, real). Is the argument in radians
        of the complex num ber u+i*v with -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 1:

The coordinates of the complex number a + i*b can be transformed from cartesian format (a,b) to polar format (modulus r, argument q) by:

```
r:= sqrt (a*a + b*b);
q:= arg (a,b);
```

Example 2:

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

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

2.11 array

This delimiter, which is a declarator, is used to declare arrays and to specify the kinds and types of formal parameters in procedures.

Syntax:

```
<type> array <array list>
<type> array <identifier list>
```

Semantic:

In the first description one or more array segments of the appropriate type (i.e. real, long, integer, or boolean) is declared. The next description specifies that the formal parameters mentioned in the identifier list are arrays of the appropriate type. <type> may be omitted, then real arrays are declared or specified.
If more than one array identifier appears in an array segment, the compiler will describe the locations (the base words) for the different arrays separately, but the description of the bounds (the dope vector) is shared by all the arrays of the array segment. The elements of an array are stored densely in lexicographical order.
A multidimensional array may be used as actual parameter where the corresponding formal is used as one dimensional. In this case the mapping used is given by the lexicographical ordering of the elements.

Example 1:

Declaration:

```
integer array A, B(1:100),x(0:2);
        array C(1:10), D(-15:17, 1:10);
```

Specification:

```
real    array K,L,M;
```

2.12 blockproc

This standard procedure performs the call of a blockprocedure as sociated 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 (cf. (15)).

Call:

```
blockproc (z, s, b)
```

```
z       (call and return value, zone). Specifies the
        procedure to be called. The procedure to be
        called is the blockprocedure connected to the
        zone z.

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 halfwords in a
        block transfer.
```

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

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

Example 1:

See Example 2 of swoprec6.

2.13 blocksout

This integer standard identifier is increased by one each time a segment of the virtual storage (context data) is transferred to the backing storage.

The initial setting is zero.

2.14 blocksread

This integer standard identifier 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 blocks read is printed at program end (cf. (15)).

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:

```
blockread:= 0;
The inner loop;
write(out,blockread);
```

The number printed then is the number of program segments transferred from disc to memory. If this explains the trouble, you could e.g.:

1) change the program so that fewer variables are declared, or

2) run the job in a greater core area, or

3) lock the segments of the inner loop in the memory.

In this example the integer printed after the end message is not the total number of segment transfers during the execution, 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 reser ve an array of maximum size (see system) and then decrease the length of the array whenever segments are transferred in the inner loop.

```
n:= max;
repeat
   begin array ra (1:n);
   s:= blocksread;
   The inner loop;
   if blocksread > s then n:= n-128;
        <*make room for one more program segment*>
   end;
until sorted;
```

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

2.15 boolean

This delimiter, which is a declarator, is used in declarations and specifications of variables of type boolean.

Syntax:

```
boolean <namelist>
```

Semantic:

The variables in namelist will all be of type boolean and each comprises 12 bits in memory.

Example 1:

```
boolean bl;
boolean yes, no, r;

procedure truth (c);
boolean c;
```

2.16 bracket

These delimiters, which are brackets, are used to separate special items.

Syntax:

```
( <expression > )
```
is an expression or a subscript
```
( <parameter list > )
```
is a parameter part
```
( <expression list > )
```
is an expression or subscript

```
begin <statements> end
```
is a statement

```
<: <text not containing "<:" or ":>"> :>
```
is a string

```
'<character name>'
```
is an integer character constant
```
"<character name>"
```
is a boolean character constant

```
< <unsigned integer or integer character constant> >
```
is a character

```
<< <layout> >
```
is a layout string

```
<* <text not containing "<*" or "*>"> *>
```
is a comment string

Examples:

```
(a + c/(a+b))
(a,b, v+3)
(a+b, c, n+k/1)

begin a:= a+b; i:= i+1 end

<:bsarea:>
<:<10> data error:>
"nl"
'em'
<<-ddd.dd>
```

2.17 buflengthio

This integer standard procedure returns the value of number of doubleword elements to use in a zone array declaration to obtain a desired blocklength once using the procedure openinout to prepare for inoutrec.

Call:

`buflengthio (no_of_zones, no_of_shares, blocklength)`

`buflengthio` (`return value, integer`). The number of doubleword elements to use in a zone array declaration to obtain a desired blocklength when using openinout to prepare for inoutrec

`no_of_zones` (`call value, integer`). The number of zones in the zone array to be declared. Obviously, no_of_zones must be positive.

`no_of_shares` (`call value, integer`). The number of shares in the zones in the zone array to be declared. Obviously, no_of_shares must be positive.

`blocklength` (`call value, integer`). The desired blocklength, measured in halfwords, to be used in input/output by inoutrec.

Zone state:

Once the zone array is declared and the zones have been opened, one or more zones may become in a state containing the buflength error bit, 64 (64+0, 64+8). The reason is that the blocklength coming out of the call of open, where the buffer area of the zone is divided evenly among the shares, may not amount to a multiple of 512 halfwords, which is minimum for a zone connected to a backing storage area or a disc. When the buffer area is redistributed among the shares in openinout, the bit will disappear from the zone states, if the blocklength coming out of openinout is sufficient. After closeinout, the bit will reappear in the zone state. Using buflengthio to declare the zone array, and using a blocklength which is a multiple of 512 halfwords when just one of the zones is going to be connected to a backing storage area or a disc, the zone states will come out right after openinout.

See exampel 1 of inoutrec.

Example 1:

The declaration

```
zone array za (2,                    <*no of zones*>
        buflengthio (2,    <*no of zones*>
```

```
        3,   <*no of shares*>
        80), <*blocklength*>
  3 ,              <*no of shares*>
stderror         <*blockproc*>);
```

will declare a zone array of 2 zones, each having 3 shares, each of them with a blocklength of 80 half words after openinout, provided none of them are connected to a backing storage area or a disc.

2.18 case

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

Syntax:

```
case <operand> of
begin <statement list> end
        is a case-statement

case <operand> of (<expression list>)
        is a case-expression
```

A case-statement or a case-expression has the same syntactical position as an if-statement or an if-expression.

Operand types:

<operand> must be of type integer, long, or real.

The elements of a statement list are separated by semicolons and numbered 1, 2, 3,...

The elements of an expression list must either be all arithemtic, all boolean, all string, or all designational. The expressions are separated by commas and numbered 1, 2, 3, ...

Result type:

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.

When the elements of the expression list are of type boolean, string, or designational the resulting value will be of the same type.

Semantic:

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 ta ke the result as the value of the case-expression.

Example 1:

Initializing a table. An arrray 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.

2.19 changekey6

This integer standard procedure makes it possible to change the key code generated by startsort6.

The generated new key code must not be longer than the key code in the initial call of startsort6.

Call:

```
changekey6   (z, keydescr, noofkeys)

changekey6   (return value, integer). The number of
             records which reasonably may be placed in the
             zone.

keydescr     (call value, integer array). Holds
             information about types and relative loca
             tions of the key filds as described for
             startsort6.

noofkeys     (call value, integer). The number of
             significant rows in keydescr.
```

The note on the value of startsort6 is also valid for changekey6.

Zone state:

The zone must be initiated by startsort6 (state 9, in sort), and all records in the zone must be inactive.

The procedure must not be called in connection with initsort and initkey.

2.20 changerec

This integer standard procedure regrets the latest call of increc, outrec, or swoprec and makes a record of a new size available. The procedure is the ALGOL5 version of changerec6.

Call:

```
changerec (z, length)
```

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

For further details see changerec6.

2.21 changerecio

This integer standard procedure regrets the latest call of inoutrec and makes a record of a new size available.

Call:

```
changerecio (za, length)
```

changerecio	(return value, integer). The number of halfwords left in the pre sent block for further calls of inoutrec.
za	(call and return value, zone array). The names za(1),...za(n) are all equivalent names of the same zone record. Determines further the documents, the buffering and the positions of the documents.
length	(call value, integer, long or real). The number of halfwords in the new record. Length must be >= 0. If length is odd, one is added to the call value.

Zone state:

The zone must be in the state 32+9 i.e. after inoutrec (see getzone6), and it is left in the same state.

Blocking:

Changerecio can be used to regret a former call of the procedure inoutrec. This happens in the following way:

1) Check zone state = 32+9. Set the record length to 0 (zero) and the logical position just before the record base.

2) Start the record procedure with the same parameters as changerecio.

The terms zone state, record length, and record base are explained in getzone6.

If there is room in the current block for the new record size, a call of changerecio will not change block. In this case data in elements available both before and after the call are unchanged.

If there is not room enough in the current block for the entire new recordsize, a change of block takes place, cf. inoutrec.

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 changerecio is greater than that of the previous call of a record procedure.

Note that block change means block change in the output zones followed by blockchange in the input zone.

The blocking is explained in more detail in inoutrec.

Example 1:

See Example 1 of inoutrec.

2.22 changerec6

This integer standard procedure 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 halfwords 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 halfwords in the new re cord. 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), 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 change rec6. 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 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 in rec6, outrec6, and swoprec6.

Example 1:

Output of records with variable length. Records with variable length, where the length is stored in the first word (2 halfwords), may be output like this:

```
repeat
  outrec6(z,maxlength);
  ....; Fill the buffer and compute
  the actual length.
  z.firstword:= actuallength;
  changerec6(z,actuallength);
until...;
```

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

Example 2:

See Example 2 of invar.

Example 3:

In a zone with one share the last of a series of output records may be forced onto the document without using setposition or close:

```
begin zone z (buflength, 1, stderror);
  open (z, ...);
  ...
  repeat
    produce records be means of outrec6
    ...
    outrec6 (z, buflength * 4);
       <*this big dummy record will force the last
       record onto the document*>
    changerec6 (z,0);
       <*regret the dummy record*>
    getposition (z, file, block);
       <*file and block must be saved in some status
       area*>
   until ...
```

This can be useful in an online program for making an easy restart after a possible program break down.

2.23 changevar

This integer standard procedure is used in connection with outvar, as it replaces a record placed in the zone z by means of outvar with another, maybe of a new length. The call change var(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
           halfwords available for further calls of
           outvar before change of 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 halfwords. If it is odd, 1 is
           added.
```

Zone state:

The zone state must be after record output (state 6, see 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 treat ment, 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 outvar.

Record format, counting of records:

The record format is explained in outvar. The free zone parameter (see getzone6) is decreased by one if the new length is 0 (zero). Otherwise it is not changed.

Example 1:

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. Old record with same key as the previous is treated as a serious error. 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); ...
  <* the initialization of the type identifications
  'creation', 'removal', and 'changes' as well as the
  field variables 'key' and 'type' depend on the record
  format. The initialization 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
  are avoided*>

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

  <*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 that
  trans contains the lowest key, 4 means that old
  contains the lowest key. More than one key can be
  lowest, e.g. 6 means trans and old lowest*>

  action:= 0;
  while action <> 7 do
  begin

                                      <*old trans new*>
  action:= 7;                      <* 1 1 1 *>
  if old.key > trans.key or old.key > new key
    then action:= action - 4;     <* 0 1 1 *>
```

```
if trans.key > old.key or trans.key > new.key
  then action:= action - 2;     <* . 0 . *>
if new.key > old.key or new.key > trans.key
  then action:= action - 1;     <* . . 0 *>

case action of
begin

<*1. (001) new.key smallest, output the ready record*>
  begin
    outrec6 (new, guessed_len);
    new.key:= infinity; new.length:= guessed_len;
  end 1;

<*2. (010) trans.key smallest, the transaction should
be a creation *>
  begin
  if trans.type <> creation then error (1)
  else
  begin
    trans.type:= ...; <*perform necessary changes in
    trans*>
    changevar (new, trans);
  end;
  invar (trans);
end 2;

<*3. (011) trans.key and new.key smallest, the
transaction must be a removal or change*>

begin
  if trans.type = creation then error (2)
  else
  if trans.type = removal then new.key:= infinity
  else
  begin <*change*>
    ... <*perform changes in new, perhaps make:
          new.length:= new_len; changevar (new,new);*>
    ...
    checkvar (new);
  end;
  invar (trans);
end 3;

<*4. (100) old.key smallest, no transaction to this
record*>
  begin
    changevar (new, old); invar (old);
  end 4;

<*5. (101) old.key and new.key smallest, two old
records with the same key This is a serious error*>
  begin
    alarm (1);
  end 5;
```

```
    <*6. (110) old.key and trans.key smallest, let the
    transaction wait until we have been through the logic
    once more*>
      begin
        changevar (new, old); invar (old);
      end 6;

    <*7. (111) all keys are 'smallest', if all three keys
    are equal to infinity, we have finished, else serious
    error: two records with same key*>

      begin
        if old.key <> infinity then
        begin
          error(3);
          invar (old);
          action:= 0; <*avoid leaving the loop*>
        end;
      end 7;
    end case action;
  end merge loop, while action <> 7;

  <*At merge end put a correct end record into new, maybe
  check the end records of old trans, close the zones
  properly*>
```

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 getzone6), as outvar is never called.

2.24 character constant

This string may be used instead of an integer constant or a boolean constant.

Syntax:

```
'<character name>' is an integer constant
"<character name>" is a boolean constant
```

Semantic:

All ISO character values, listed in the character set table in ref. (14), may be written directly in the program text. <character name> is the mnemo nics shown in column G in the character table. The mnemonics corresponding to the ISO values 0:32 and 127 must be written with small letters.

The mnemonics are translated to the corresponding ISO values, as if these constants were written directly:

```
'<mnemonic>' <=> <digits in ISO value>
"<mnemonic>" <=> false add <digits in ISO value>
```

Example 1:

The following statements and expressions are identical two by two:

```
if char = 'ext' then ...
if char =  3    then ...

write (out, "sp", 3);
write (out, false add 32, 3);

<:a'y'x"nl"<'p'>:>
<:a'y'x"nl"<112>:>

alpha ('x'):= 'x' + 6 shift 12;
alpha (120):= 120 + 6 shift 12;
```

2.25 check

This standard 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 getzone6) is waited for and checked.

The algorithm is given in ref. (15), 2.4.4 wait transfer. Ref. (15) also describes the standard error actions.

2.26 checkvar

This integer standard procedure calculates the record checksum of a record with the format of a variable length record as generated by outvar. 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 the 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 getzone6). 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;
      setzone6 (z,descr);
      checkvar(z);
   end;
```

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

Example 2:

See Example 2 of invar.

Example 3:

See Example 2 of swoprec6.

2.27 close

This standard procedure 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. If the document is a magnetic tape which latest has been used for output (state 3 and 6, see getzone6), a tape mark is written.

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

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 (see (7)).
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.

2.28 closeinout

This standard procedure terminates the current inoutrec-use of a zone array and reinitializes the buffersizes and the sizes and locations of the shares of the zones in the array, bringing the zones back in the state as after open (and positioned).

Call:

```
closeinout (za).

za      (call and return value, zone array). The zones
        za(1),...,za(no of zones) determines the
        documents, the buffering and the positions of
        the documents, cf. (19).
```

Function:

Closeinout proceeds in 4 steps:
terminate current use, write tape mark, reinitialize zones, reopen zones.

Terminate Current Use:

Each of the zones are terminated, as described in setposition, after having its state set to 5 for the input zone and 6 for the output zones.

Write Tapemark:

If the document connected to any zone is a magnetic tape, which latest has been used for output, a tape mark is written in that zone.

Reinitialize Zones:

Each zone has its buffer and share descriptions reallocated and reinitialized as right after the zone array declaration.

Reopen:

Each zone has its zone and share descriptors reinitialized as right after the latest call of open.

Zone state:

The zone states must be one with the inoutrec bit, 32, and they are left as after open, i.e. 0, 8, 64+0, 64+8, being open, open on magtape, open with buflength error, open on magtape with buflength error, cf. getzone6.

Example 1:

See Example 1 in inoutrec.

Example 2:

See Example 1 in expellinout.

2.29 closetrans

This standard procedure terminates the current format8000 output transaction, i.e. writes an ETX character in the zone, and if the document is anything else than a backing storage area or a disc, the procedure outputs the current buffer to the document (see (19)).

Call:

```
closetrans (z)

z       (call and return value, zone). Specifies the
        document to which the transaction is
        transferred.
```

Zone state:

The zone must be in state 3 (after character writing). After the call the zone is in state 13 (re ady for opentrans).

If the document is an area process (4) or a disc process (6) and the zone is to be closed, the zone state must explicitly be set to "after character output" (3) before close is called, or the contents of the zone buffer will not be transferred to the document:

```
setstate (z, 3);
```

Example 1:

See Example 1 of opentrans, and Example 3 of writefield.

2.30 comment

This delimiter, which is a separator, is used to insert comments in the program text in order to increase the readability of the program. Comments may appear in 2 different forms:

Syntax:

```
; comment <text not containing ";"> ;
may replace any ; (semicolon)

begin comment <text not containing ";"> ;
may be replace any begin
```

Example 1:

```
...
begin comment h=height, l=length, w=width, v=volume;
   integer h, l, w, v;
   read (in, h, l, w);
   comment calculate the volume;
   v:= h*l*w;
   ...
```

2.31 comment string

This delimiter, which is a bracket, can be used everywhere a comment is needed.

Syntax:

```
<* <text not containing "<* or "*>" > *>
```

Semantic:

This comment string can replace a space everywhere in the program. The comment string is blind to the program.

Example 1:

```
procedure A (<*price*> p, <*item no*> item);

case i of (<*i=1*> a+b,
           <*i=2*> c/q+1)
```

2.32 context

This delimiter, which is a declarator, is used in the declaration of a context block.

Syntax:

```
context (incarnation, no_of_incarnations, context_mode)
```

`incarnation`	`(integer expression). Specifies the actual incarnation of the context block.`
`no_of_incarnations`	`(integer expression). Specifies the total number of incarna tions.`
`context_mode`	`(integer expression). Specifies a bit pattern, which defines the action at entry to and exit from a context block. The bits are used as follows:`

```
1 shift 0: read bit
1 shift 1: write bit
1 shift 2: save bit
1 shift 3: new block bit
1 shift 4: new incarnation bit
```

This declaration follows immediately the block begin, and must only appear once in the head of the block.

Semantic:

See the description in ref. (15), section 4.

Example 1:

See the Examples in ref. (15).

2.33 continue

This delimiter, which is a context operator, is used in a context block to jump to a context label, allowing context blocks to be considered coroutines in a number of incarnations (cf. (15)).

Syntax:

```
continue;
```

Semantic:

See the description in ref. (15).

Example 1:

See the examples in ref. (15).

Note:

The delimitor is only allowed in context blocks, and yet in inner block levels of context blocks.

2.34 cos

This real standard procedure performs the trigonometric funtion cosine.

Call:

```
cos (r)

cos     (return value, real). Is the trigonometrical
        function cosine of theargument r, in radians
        with -1 <= cos <= 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 1:

Let d be an angle in degrees.

```
pi:= 3.14159 26536;
cad:= cos (pi/180 * d);
```

Now cad contains cos of d.

2.35 deadsort

This standard procedure creates a zone record in memory which at a later stage is to take part in a sorting process. The contents of the record are initially undefined, but the user is supposed to assign values to the record variables before next call of any sorting procedure. The hereby defined record becomes an inactive record in the sorting process, i.e. it is not participating in the selection process of procedure outsort.

Call:

```
deadsort (z);

z       (call and return value, zone). The name of the
        record created.
```

Zone state:

As for newsort.

Example 1:

String generation by replacement - selection.
A large file on magnetic tape is read, and a string of sorted records as long as possible is generated from this input stream. The input takes place via zone x, and output is written via zone y. Record length is one double word, and the output string is sequenced on ascending values of this real. The memory is supposed to be able to hold about 1000 records.

```
begin
   zone z(2018, 1, sorterror);
   integer array key(1:1, 1:2);
   integer i, n;
   key(1,1):= 4; key(1,2):= 4; n:= 1000;
   startsort6(z, key,1,4);
   for i:= 1 step 1 until 1000 do
   begin
      inrec(x,1); newsort(z); z(1):= x(1)
   end;
   outsort(z);
   while true do
   begin
   outrec(y,1); y(1):= z(1); inrec(x,1);
   if sortcomp(z, x, y) < 0 then
   begin
      deadsort(z); n:= n -1
   end
   else
      newsort(z);
   z(1):= x(1);
   if n > 0 then
      outsort(z)
```

```
   else
   begin
      <*at this stage the zone z is filled up with inactive records which can
      be activated to form a new string*>
   end;
end while;
```

comment an end of file situation is not handled by this algorithm, i.e. the input file is assumed to be infinitely large;

2.36 decimal point (.)

This delimiter, which is a separator, is a part of a real number.

Syntax:

```
(+)
( ) (<unsigned integer>) .<unsigned integer>
(-)
```

Semantic:

The decimal point has the conventional meaning.

Example 1:

```
   0.7300
  +0.7614
-200.084
   -.083
   5.17
    .1
```

2.37 disable

This delimiter, which is a declarator, is used in connection with activities in order to let the following statement allocate stack space in the "open ended" stack of the monitor block (see (19) and activity). It can also be used to prevent an implicit passivate.

Syntax:

```
disable <statement>
```

Semantic:

During execution of a disable statement all stack ing/unstacking is done in the "open ended" stack of the monitor block, and the program is in disable mode. This means that a possible implicit pas sivate is not performed.

A disable statement must not be the body of a procedure declaration. Execution of activity procedures (activate, passivate) or goto statements is not allowed in disable mode.

Examples:

```
disable a:= proc (a, b, x);

disable
begin
   a:= a+1;
   ...
end;
```

2.38 divide (/)

This delimiter, which is an arithmetic operator, yields the quotient of the two operands.

Syntax:

```
<operand1> / <operand2>
```

Priority: 3

Operand types:

```
integer, long, or real.
```

Result type:

always real.

Semantic:

the operation may include a type conversion. Real values are represented with a relative precision of about 3'-11 (cf. (15)). This means that a real varible holding an integral value is represented exactly in the interval

```
-2**35 <= real <= 2**35-1.
```

As division of long values includes call of subroutines, and cannot be performed by built-in operations, a representation of certain long variables in real variables may be advantageous (cf. (15)).

Example

```
1: r:= a/b;
b:= a/b/c/d;
```

2.39 endaction

This integer standard identifier controls the way in which, the program will terminate:

endaction = 0:	At program termination, the FP end program will be called.
endaction = 1:	At program termination, the current output buffer is emptied (provided FP is present), a finis message with wait bit is sent to the parent, containing the alarm text and inte ger, the answer is waited for and the program will loop end lessly, waiting for the parent to take proper action.
endaction >0 or <1:	At program termination, the FP break action will be called.

Default value is:
-0 if the File Processor is present.
-1 if the File Processor is absent.

Just before program termination, the system assigns a value to endaction this way:

```
if fp.absent then
  endaction: = 1 <*finis message*>
else
if termination cause = break alarm then
  end action: = 'break action'; <*return to FP break*>
```

In case of end program or finis, the possible context data segments are transferred to the backing storage area (prog mode : = passive; reserve all core; and the stack with all its zones is released.

If fp is present, the current output buffer is emptied, and now the program branches to endaction.

2.40 entier

This delimiter, which is a monadic arithmetic operator, transfers an arithmetic expression of type real to the largest integer not greater than the real expression.

Syntax:

```
entier <operand>
```

Priority: 1

Operand type:

always real.

Result type:

always integer

Semantic:

The largest integer not greater than the real arithmetic expression is evaluated. The operation may cause integer overflow.

Example 1:

```
a:= 5.85;
b:= entier a;
<* now be has the value 5 *>
a:= -a;
c:= entier a;
<* now c has the value -6 *>
```

2.41 equal (=)

This delimiter, which is a relational operator, gives the value true or false.

Syntax:

```
<operand1> = <operand2>
```

Priority: 5

Operand types:

integer, long, or real.

Result type:

always boolean.

Semantic:

The relation takes on the value true whenever the bitpattern of the two operands are equal, otherwise false.

The relation is performed as a bit by bit comparison of the two operands (after a possible type conversion whenever the operands are of different types).

Example 1:

```
b:= 5=6; <*false*>
if a=c then ....
k:= 5=15/3; <*true*>
```

2.42 equivalent (= =)

This delimiter, which is a logical operator, yields the logical equivalence of the two operands.

Syntax:

```
<operand1> == <operand2>
```

Priority: 10

Operand types:

boolean.

Result type:

boolean.

Semantic:

The truth value is evaluated according to the following rule:

right / left	true	false
true	true	false
false	false	true

Example 1:

```
if a == b then ....;
<*is the same as
   if a and b or (-,a and -,b) then ...*>
```

2.43 errorbits

This integer standard identifier sets the values of the two FP modebits 'ok' and 'warning', when returning to the File Processor through the final end of the program.

The value of the errorbits is used as the "end program condition". Only the bits: errorbits extract 2 are used by the system. The two bits are interpreted as follows:

```
1 shift 0 : ok.no
0 shift 0 : ok.yes
1 shift 1 : warning.yes
0 shift 1 : warning.no
```

The default value of errorbits is 0.

If the program returns to the File Processor after an alarm in stead of leaving through the final end, the value of errorbits will not be used as "end program condition". (Instead a value >3 is used at "give up" alarm and the value 1 is used at any other alarm).

Example 1:

If you in an algol program count the number of errors recognized during the run,you may let the program decide whether to continue or stop the job by assigning errorbits before leaving the program:

```
  if errors > maxerrors then
  errorbits:= 1
  else
  if errors > 0 then
  errorbits:= 1 shift 1;
end program
```

The job may contain the following FP commands:

```
p       ; run the program
if ok.no
  (c = message too many errors. stop run finis)
if warning.yes
  c = message some errors.run continues

; execute the next program in the jobfile
...
```

2.44 exit

This delimiter, which is a context operator, is used in a context block to control jumps out of a context block in such a way that the same incarnation can restart6 in the next statement following the exit operator. The operator, together with exit, allows context block to be considered coroutines in a number of incarnations, cf (15).

Syntax:

```
exit (<designational expression>)
```

Semantic:

See description in (15).

Note:

The delimiter is only allowed in context blocks, at the context block level itself and outside repetitive statements.

Example

1: See the example in (15).

2.45 expellinout

This standard procedure expells one outputzone in a zone array prepared for inoutrec by the procedure openinout in such a way that no output will take place in that zone until closeinout reopens the zones of the array.

Call:

```
expellinout zone (za, index)
```

za (call and return value, zone array). The zones za (1),..., za (no of zones) determine the documents, the buffering and the positions of the documents. The zones must be prepared for inoutrec by a call of openinout, cf. zone states.

index (call value, integer, long, or real). The index in the zone array for the zone to be expelled. Index must be in the range 1,..., no of zones, and if za (index) is not an outputzone, the procedure is blind.

Zone state:

The state of the zone must be after openinout or after openinout on magtape (32+0, 32+8) and is left unchanged.

Partial word:

The value of the fixed partial word in the zone descriptor is changed to the address of the zone itself, cf. openinout.

Example 1:

Consider Example 1 in inoutrec, the section headed: <*handle end of tape in one of the outputzones*> and suppose that file (1) is connected to some backing storage area, which is to be transferred to the two tapes in parallel. If end of tape in an outputzone should lead to a change of tape to a new volume tape, transfer of some standard file to the tape and continued output in the two tapes in parallel, it could go like this:

```
if end_file (2)
or end_file (3) then
begin <*handle tape shift in one or both zones*>
   for i := 1, 2, 3 do
   begin <*stop all zones, position before tape mark*>
      stop_zone (file (i), endfile (i)); <*tape mark if endtape*>
      getposition (file (i), file_no (i), block_no (i));
   end;
```

```
   closeinout (file); <*check position operation and reallocate*>

   for i := 1, 2, 3 do
      if end_file (i) then
      begin <*change to next volume in this zone*>
         next_volume (file, i, file_no, block_no);

         expell (i) := end_file (i) := false;
      end <*change to next tape*> else
      begin <*after closeinout the zone states are 'unpositioned'*>
         setposition (file (i), file_no (i), block_no (i));
         expell (i) := true; <*set expell condition*>
      end;

   close (file (1), false); <*don't remove area process*>
   open (file (1), 4, <:std_file:>, 0);

   openinout (file, 1); <*prepare transfer standard file*>

   for i := 2, 3 do
      if expell (i) then
         expellinout (file, i);

for hwds := inoutrec (file, 0) while hwds > 2 do
   changerecio (file, hwds);

   for i := 2, 3 do
   begin <*stop zones, position after last block*>
      stopzone (file (i), true); <*tape mark*>
      getposition (file (i), file_no (i), block_no (i));
   end;

   closeinout (file);

   close (file (1), true);   <*remove area process:>
   open (file (1), 4, <:bs_area:>, 0); <*reconnect*>

   for i := 1, 2, 3 do       <*reposition*>
      setposition (file (i), file_no (i), block_no (i));

   openinout (file, 1);   <*reallocate for inoutrec*>
end <*handle tape shift in one or both zones*>;
```

It is assumed that the procedure next_volume is declared:

```
procedure next_volume (file, index, file_no, block_no);
value                        index                    ;
zone    array          file                           ;
integer                      index                    ;
integer array                      file_no, block_no ;
begin
   <*maybe some message that the tape ran out*>

   close (file (index), false add 1); <*release message*>
   open (file (index), 4 shift 12 + 18 <*mtll*>, <:next_volume:>, give_up);
   file_no (index) := 1;
   block_no (index) := 0;
```

```
    setposition (file (index), file_no (index), block_no (index));
    check       (file (index)); <*check position operation*>

end <*next volume*>;
```

The declarations:

```
integer array file_no, block_no (1:3);
boolean array expell (1:3);
```

are assumed, 100.

2.46 exor

This long standard procedure performs the logical function exclusive or on two 48 bit entities a and b. If the type length of a and/or b is 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 not,
       (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 cf. (14). 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 had been applied.

Boolean: A boolean is considered 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 value 2048 differs from the effect of a boolean with the value false add 2048.

Example 1:

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;
z.checkfield:= checkchar;
```

The data block including the checkcharacter will now be of even longitudinal parity.

2.47 exp

This real standard procedure performs the exponential function.

Call:

```
exp (r)

exp    (return value, real). The exponential function of
       the argument r, e**r. (e = 2.71828 18285).
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.

A value of r greater than 1000 will cause the run time alarm: exp 0.

Example 1:

```
e:= exp (1);
```

2.48 exponentiation (**)

This delimiter, which is an arithmetic operator, yields the involution of the left hand operand to the power indicated by the right hand operand.

Syntax:

```
<operand1> ** <operand2>
```

Priority: 2

Operand types:

integer, long, or real. When <operand2> is real, <operand1> must be positive (see below).

Result type:

Always real.

Semantic:

The operation denotes exponentiation, where <operand1> is the base, and <operand2> is the exponent. Thus for example

$$2**n**k \text{ means } {}^{n}(2\ )^{k}$$

while

$$(n\ )\ 2**(n**k) \text{ means } 2$$

Writing i for a number of type integer or long, r for a number of type real, and a for a number of type either integer, long, or real, the result is given by the following rules:

```
a**i:
i > 0:  a*a*...*a (i times)
i < 0:  1/(a*a*...*a) (abs(i) times),
        a must be <> 0
i = 0:  1

a**r:
a > 0:  exp (r * ln(a))
a < 0:  runtime alarm
a = 0:  runtime alarm (expect when a is of type integer
        or long and r > 0, in which case the result
        becomes 0.0).
```

Example 1:

```
r:= b * 10**a;
```

2.49 extend

This delimiter, which is a monadic arithmetic operator, converts an integer expression to a long.

Syntax:

```
extend <operand>
```

Priority: 1

Operand type:

integer.

Result type:

long.

Semantic:

The value of <operand> is converted to a 48 bits long by extension of the sign bit.

Example 1:

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
```

(whereas the expression: totals:= extend (pieces * price) will not work).

This is of course only relevant if totals reason ably may exceed 8 000 000 and is a long.

Example 2:

Two integers may be packed into one long variable in this way:

```
l:= extend i1 shift 24 add i2;
```

Example 3:

See Example 2 of long.

2.50 external

This delimiter, which is a compiler directive, 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.

Semantic:

A procedure translated in this way becomes a standard procedure, which means that other algol or fortran 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 or fortran program.

The name of a translated external procedure must not contain capital letters, because they are for bidden in names of backing storage areas.

Example 1:

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

```
tg=algol;   File processor commands, cf. (6).
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.

2.51 extract

This delimiter, which is a pattern operator, is used for unpacking of integer values from a real, long, integer, boolean, or string value.

Syntax:

```
<operand1> exract <operand2>
```

Priority: 2

Operand types:

<operand1>: boolean, integer, long, real, or string.

<operand2>: integer, long, or real.

Result type:

Always integer.

Semantic:

Extract treats <operand1> as a binary pattern (cf. (14), and <operand2> is rounded to an integer if it is of type long or real. Now a number of the rightmost bits are extracted from <operand1> as indicated by the value of <operand2>. 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, <operand2> has a value be low 0 or above 24.

Example 1: Simple splitting

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

```
i1:= b shift (-6)          extract 6
i2:= b                     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;
     i.e. 0 <= i+32 <= 63;
r:= r shift 6 add (i+32);
...
i:= r extract 6 - 32;
```

Example 3:

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);
   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 rf:= 4*n step -4 until 4 do
if ra.rf <> 0 then
```

```
begin
   bf:= rf;
   e:= ra.bf extract 12;
   if e >= 2048 then e:= e - 4096;
   e:= e - max;
   if e < -2048 then ra.rf:= 0
   else ra.bf:= false add e;
end;
```

2.52 f8000table

This standard procedure changes the input character alphabet (like the standard procedure intable) to the format8000 input table.

Call:

```
f8000table
```

The format8000 table consists of 256 entries and is stored in own core.

Note:

The procedures waittrans and readfield call this procedure, so it will not be necessary for the program to call f8000table when using the format8000 procedures.

2.53 field

This delimiter, which is a declarator, is used in declarations and specifications of field variables.

Syntax:

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

Semantic:

Field variables are used in field references or they may be used as integer variables. For a complete description cf. (15).

Example 1:

See (15).

Example 2:

See Example 1 of activity.

2.54 fpmode

This boolean standard procedure tests a bit in the FP modeword.

Call:

```
fpmode (modebit)

fpmode      (return value, boolean). True if the bit was
            set, otherwise false.

modebit     (call value, integer). 0 <= modebit <= 23.
```

Example 1:

The printing of testoutput from your program can be controlled by the modebits in the following way:

```
begin
   boolean testa, testb;
   .....
   testa:= fpmode (0);
   testb:= fpmode (1);
   .....
   if testa then write (out, .....);
   .....
   if testb then write (out, .....);
```

The job

```
mode 0.yes 1.no; set testa
p              ; execute program
```

will produce the testoutut controlled by testa, but not the testoutput controlled by testb.

2.55 fpproc

This standard procedure is used to execute a subset of the FP subroutines (cf. (3), Part Three), normally identified by their names.

Call:

```
fpproc (action, w1, w1, w2)

action (call value, integer).
```

The procedure simulates the call:

```
jb w3 fpbase + h-name (action)
```

in algol or fortran programs, i.e. calls the subroutine

```
h(concat)action
```

The subset accessible is described by the possible action set:

```
 7, fp end program
14, fp finis message to parent
22, fp inblock
23, fp outblock
24, fp wait and ready
25, fp inchar
26, fp outchar
27, fp connect input
28, fp connect output
29, fp stack zone
30, fp unstack zone
31, fp outtext
32, fp outinteger
33, fp outend
34, fp close up
35, fp parent message
48, fp wait and free
67, fp break message to parent
79, fp terminate zone
95, fp close up text output
```

```
w1, w2, w3  (call and return values, undefined). No check
            of kind and type of the actual parameters is
            performed at compilation time.
```

Certain checks of the w parameters are performed at execution time just before FP is entered, but still the procedure should be used with care.

The allowed kinds and types depend on the value of the action parameter, cf (22). Violation will terminate the program with the alarm:

```
param fpproc
called from ...
```

For details on the use of the procedure, cf. (22).

2.56 getalarm

This integer standard procedure can be used when runtime alarms are trapped, to supply the information which would have been printed by the File Processor if the run was terminated.

Call:

```
getalarm (arr)
```

getalarm	(return value, integer). If the latest runtime alarm was: give up (i.e. alarm cause extract 24 = -11), the value is the logical status word from the zone in question - otherwise it is undefined.
arr	(return value, array of any type). Must be declared to hold at least 8 words, the first word being at halfword index 1. Assume the declaration long array arr (1:4), then the contents of arr will be:
arr (1:2)	The alarmtext printed on current output.
arr (3:4)	If alarm_cause extract 24 = -11 (give up), the contents will be the document name of the zone in question -otherwise the empty string.

Example 1:

The following procedure should be called when an alarm has been trapped.

```
procedure writealarm (out);
zone   out ;
begin
   long array   text (1:4);
   long array field    docname;
   integer   status, cause, param, bit;
   real   comma;
   docname:= 8;
   status := getalarm (text);
   cause  := alarmcause extract 24;
   param  := alarmcause shift (-24);
   write (out, text, param);

   if cause = -11 <*give up*> then
   begin    <*output device information*>
    write (out, "sp", 1, text.docname);
    comma:= real <:: :>;
    for bit:= 1 step 1 until 24 do
      if status shift (bit-1) < 0 then
      begin
        write (out, string comma, case bit of (
```

```
<*1*><:intervention:>,
    <:parity error:>,
    <:timer:>,
    <:data overrun:>,
<*5*><:block length:>,
    <:end document:>,
    <:load point:>,
    <:att or tapemark:>,
    <:write enable:>,
<*10*><:mode error:>,
    <:read error:>,
    <:disk error/not connected:>,
    <:checksum:>,
    <:bit 15:>,
<*15*><:passivate:>,
    <:stopped:>,
    <:word defect:>,
    <:position error:>,
    <:non exist:>,

<*20*><:disconnected:>,
    <:unintelligible:>,
    <:rejected:>,
    <:normal:>,
    <:hard error:>));

        comma:= real <:, :>;
      end for, if;
    end device information;
  end write alarm;
```

2.57 getf8000tab

This standard procedure opens access to the format 8000 character input table.

Call:

```
getf8000tab (a, low, up)
```

a	(call and return value, integer array). The dope vector of the array is changed, so as to point directly to the format 8000 table. The array may be declared: integer array a(1:1) and must be a single declaration. Field arrays are not allowed.
low, up	(call values, integers). Define the wanted lower and upper limits for the array: a (low:up).

On exit there is direct access to the format8000 table, and care must be taken to avoid destruction of table values (0:31) and (127:255), which are used for special purposes by the format8000 procedures.

Example 1:

The following program part will make the user part of the character input table available for changes, but will prevent the special part of the table from being destroyed:

```
begin
  integer array table (1:1);
  getf8000tab (table, 32, 126);
  table (32):= ...; <*define space and other characters*>
  <*but a reference: table(31):= ...
  will give an index alarm*>
```

2.58 getposition

This standard procedure 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 (cf. (15)). 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 (cf. (15)). 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.

When the document is a backing storage area, the contents of the parameter block is the segment number within the area. If the share length of the zone is greater than one segment, the position will be the segment number of the first segment within the block.

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 to get the position within the block, or you may use the value of inrec or outrec6 to denote the position within the block (see Example 3 of getzone6).

2.59 getshare

This standard procedure moves the contents of a share descriptor into an integer array for further inspection. The procedure is the ALGOL5 version of getshare6.

Call:

```
getshare (z, ia, sh)
```

a	(call value, zone). Specifies the share together with sh.
ia	(return value,integer array,length >= 12 counted from lexicographical index 1). The elements ia(1), ..., ia(12) must exist.
sh	(call value, integer). The number of a share within z. The contents of the share descriptor are moved to ia(1,...,ia(12).

Works as getshare6 except that getshare computes first shared and last shared as a buffer index in stead of as a halfword index. The buffer index is equal to (halfword index+3)//4.

2.60 getshare6

This standard procedure 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 im plement 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 constists 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 (cf. (3)).

The share descriptor contains certain absolute addresses of half words within the zone buffer. The reason for this and the relation between the absolute address and the usual 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 counted from lexicographical index 1). The elements ia(1),...,ia(12) will become the below values.
sh	(call value, integer). The number of a share within z. The contents of the share descriptor are moved to ia(1),...,ia(12).
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 run ning child process. - 0 for a free share. See below. - 1 for a ready share. See below.
ia(2)	**First shared**. Halfword 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**. Halfword 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

            operation examples cf. (3)
              0 sense
              3 read
              5 write

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
            halfword 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 a 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, whereas 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 getzone6) 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 unde fined value):

```
ia(1)  ia(2)  ia(3)  ia(4)  ia(5)  ia(6)... ia(12)
state  f.sh.  l.sh.  opt    f.adr. l.adr.   top.tr.
```

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:

```
share 1      >0     1     400  output 30 000  30 350     X
used share    0    401    800    X    30 400  30 798     X
share 3       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
```

2.61 getstate

This standard procedure assigns the zonestate to an integer parameter.

Call:

```
getstate (z, state)

z          (call value, zone). The zone state of z is
           assigned to state.
state      (return value, integer).
```

Example 1:

See Example 1 of opentrans.

2.62 getzone

This standard procedure moves the contents of a zone descriptor into an integer array for further inspection. The procedure is the ALGOL5 version of getzone6 and works as getzone6 except that the record length is given in buffer elements instead of half words.

A buffer element consists of 4 halfwords.

Last halfword of a buffer element, the reference halfword, 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 ia(1),...,ia(20).
ia         (return value, integer array, length >= 20,
           counted from lexicographical index 1).
```

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 some of the procedures inrec6, outrec6, swoprec6, changerec6, invar, outvar, or changevar have been used.

For further description see getzone6.

2.63 getzone6

This standard procedure moves the contentst 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 half words, 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 halfword index in this way:

The reference halfword of a field in the zone has the absolute address:

```
base buffer area + halfword index.
```

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

Call:

```
getzone6 (z, ia)
```

```
z          (call value, zone). The contents of the zone
           descriptor are moved to ia(1),...,ia(20)
ia         (return value, integer array, length >= 20
           counted from lexicographical index 1). The
           elements ia(1),k...,ia(20) will become the
           following values:

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 magne tic tape handling. See explanation below.
ia(9)	Segment count. Only significant for hand ling of backing storage areas. See explanation below.
ia(10)	Give up mask. See (15).
ia(11)	Free parameter. Is used by the Fortran read/write system, by the var-procedures and by the in-out procedures. See explanation below.
ia(12)	Partial word. Used by the procedures for input-outputon character level to unpack or pack characters. See explanation be low.
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 halfword preceding the first halfword of the present record. During character in put 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 halfword. Absolute address of the last halfword 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 halfwords 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 of the backing storage area, (i.e. the segment number of the first segment in this block).

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 outvar, openinout, inoutrec, changerecio or closeinout. It is set to zero when the zone is declared. The var-procedures (changevar, invar, and outvar) use the free parameter as a counter of the logical records generated or read by the procedures, and as an indication of whether or not the record checksum should be checked by invar. The fortran read/write system uses the last bit of this parameter to signal if the latest call of read or write have used format or format0. A one in the last bit means that format0 was used and a zero means that format was used. See (9) for further details.

The procedure openinout will use the word as a pointer to the in put zone among the zones in the parameter zone array.

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 outp
ch2 shift 16 + ch3 shift 8 + 1
ch3 shift 16 + 1 shift 8                                  1 shift 8 +
1   shift 16                              1 shift 16 + ch1 shift 8 + ch
```

The procedure openinout will use the word for the zone's own index in the parameter zone array.

Zone state

The action of a high level zone procedure will in general depend on the latest operation upon the same zone. The following zone states are used:

zone state	
0	positioned after open.
1	after character reading.
2	after repeatchar.
3	after character writing.
4	after declaration or after close.
5	after record input (or fortran unformatted input).
6	after record output (or fortran unformatted output).
7	after record swop
8	after open on magtape (open and unpositioned)
9	in sorting

10	after waittrans
11	after waittrans
13	after closetrans
32	after openinout or setposition after open inout
32+8=40	after openinout on magtape
32+8=41	after inoutrec/changerecio
64+0=64	after open with buflength error
64+8=72	after open on magtape with buflength error
64+32+0=96	after openinout with buflength error
64+32+8=104	after openinout on mt with buflength error

Other zone states are used by procedures not described in this manual.

The high level zone procedures described in this manual use the zone state as shown in the following table:

procedure	zonestate before	zonestate after
<declaration>	-	4
initzones/resetzones	4	4
open	4	0,8,64+0,64+8
setposition	0,1,2,3,5,6,7,8	0
	32+0, 32+8, 32+9	32+0
	64+0, 64+8	64+0
read, readall, readchar, readstring fortran formatte read	0,1,2	1
repeatchar	1,2	2
write, writeint, outchar outdate, outinteger fortran formatted write	0,3	3
inrec, inrec6, invar fortran unformatted read	0,5	5
outrec, outrec6, outvar fortran unformatted write	0,6	6
swoprec, swoprec6	0,7	7
changerec, changerec6	5,6,7	unchanged
changevar	6	6
close	any but one with 32	4
changekey6, deadsort, initkey, liftsort, newsort, outsort	9	9
opentrans	0,13	3
closetrans	3	13
waittrans	0,1,2,10,11	10,11
readfield	1,2,10,11	1,2
openinout	0,8	32+0, 32+8
	64+0, 64+8	64+32+0, 64+32+8
expellinout	32+0, 32+8	unchanged
inoutrec	32+0, 32+9	32+9
changerecio	32+9	32+9
closeinout	32+0, 32+8, 32+9	0,8,64+0,64+8
	64+32+0, 64+32+8	

Example 1:

Let z be declared as z(2*128,2,stderror) and opened as the backing storage area <:sldata15:. Af ter 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),filecount	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 halfword	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	29 999
ia(20),buffer length	256

Example 2:

Character output to memory locations:

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.`

```
zone convert(10,1,stderror); integer array
ia(1:20);
open(convert,0,<:dummy:>,0);
repeat
  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); ...
  until ...;

comment the zone convert must not be closed;
```

Example 3: Improved setposition:

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

```
begin
   zone z(...), z1(...);
   integer array ia(1:20);
   integer       pos1, pos2, pos3, pos4, c;
   integer field partial_word, record_base,
                        used_share, no_of_shares,
                        base_buf_area, buf_length;
```

This example is made absolute by the development of write, making it able to write into arrays as well as zones, but for its illustrative qualitites concerning the concept of blockprocedures, it is repeated.

```
partial_word          := 12*2;
record_base           := 14*2;
used_share            := 17*2;
no_of_shares          := 18*2;
base_buf_area         := 19*2;
buf_length            := 20*2;

comment generalised getposition;
read (z, ...);
getposition (z, pos1, pos2);
getzone6 (z, ia);
pos3:= ia.record_base - ia.base_buf_area -
          ia.buf_length*4//ia.no_of_shares *
          (ia.used share-1);
    <*pos3 is the relative position within the used share*>
pos4:= ia.partial_word;

comment generalised setposition, perhaps with the device connected to another
zone;
setposition (z1, pos1, pos2);
readchar (z1, c);
    <* now the deviced is positioned and the first block is read into the
    first share *>
getzone6 (z1, ia);
ia.record_base:= pos3 + ia.base_buf_area;
ia.partial_word:= pos4;
setzone6 (z1, ia);
read (z1, ...);
```

2.64 goto

This delimiter, which is a sequential operator, can be used to change the program flow.

Syntax:

```
goto <designational expression>
```

Semantic:

A goto statement interrupts the normal sequence of operations. The next statement to be executed will be the one having this value as its label.

A jump (by a goto statement) out of an activity or a disable statement is not allowed. This will give the run time alarm "goto" (alarmcause = -14).

A jump (by a goto statement) into a for-, repeat-, or while-statement is also forbidden. This will give the error message "for label" during the compilation.

Example 1:

```
a:= q;
if a > p then goto lab;
....
lab:
b:= a + 17;
```

2.65 greater than (>)

This delimiter, which is a relational operator, yields the value true or false.

Syntax:

```
<operand1> > <operand2>
```

Priority: 5

Operand types:

integer, long, or real.

Result types:

Always boolean.

Semantic:

The relation takes on the value true whenever the corresponding relation is satisfied for the expressions involved, otherwise false.

In case one of the operands is different from type integer the relation is executed as a subtraction. Thus you must be prepared for overflow, underflow, or spill (see Example 3 of monitor).

Example 1:

```
a:= b > c;
if d > q then ... else ...;
while k > l do ...;
```

2.66 greater than or equal (> =)

Syntax:

```
<operand1> >= operand2>
```

Priority: 5

Operand types:

integer, long, or real.

Result types:

Always boolean.

Semantic:

The relation takes on the value true whenever the corresponding relation is satisfied for the expressions involved, otherwise false.

The relation is always executed as a subtraction. Thus, you must be prepared for overflow, under flow, or spill.

Example 1:

```
a: = a >= c;
if d >= k then ... else ...;
while x >= y do ...;
```

2.67 implication (= >)

This delimiter, which is a logical operator, yields the logical implication of the two operands.

Syntax:

```
<operand1> => <operand2>
```

Priority: 9

Operand types:

boolean

Result type:

boolean

Semantic:

The truth value is determined according to the following rule:

right left	true	false
true	true	false
false	true	true

2.68 in

This standard identifier 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 (cf. (15)).

The call

```
<program>
```

will let 'in' be connected to the current input file of the file processor.

The call

```
<program> <text file>
```

will let 'in' be connected to the file <text file>, if the program is not translated with the parameter connect.no.

The call

```
<program> <integer>
```

makes 'in' and the file processor unavailable (but frees some space in the job area), if the program is translated with the parameter fp.no.

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

Example 1:

```
read (in,a,b,c);
readchar (in,char);
```

Example 2:

```
prog= algol connect.no ...
prog param ; the program will not use param
           ; as input text file.
```

Example 3:

```
prog= algol fp.no ...
prog 7; in/out and fp are removed from the process.
```

2.69 increase

This integer standard procedure is 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 1:

This procedure could be used in a procedure call, where the formal parameter is a string, but the actual parameter is a real array, when the parameter is referenced only once.

The zonerecord in zone z contains a text from element 4. This text can be written out by:

```
i:= 4;
write (out, string z (increase(i)));
```

2.70 initkey

This standard procedure generates a piece of code for comparison of two records in a zone. Succeeding calls of the procedure new sort, outsort, and lifesort with the zone as parameter will cause this code to be used, provided the second parameter in the calls is omitted (see procedure newsort).

This procedure is the ALGOL 5 version of changekey6, and must not be used if the sorting was initiated by startsort6.

Call:

```
initkey (z, keydescr, n)

z          (call and return value, zone). The name of the
           zone used for sorting.
keydescr   (call value, array). A real array holding
           information about types and relative locations
           of the key field in a record (see below).
n          (call value, integer). Number of key fields
           (number of rows in keydescr).
```

Key descr:

The array keydscr must be declared as:

```
array keydscr(1:n, 1:2);
```

with the restriction:

```
1 <= n <= 4 * max. record length
```

Each row in the array holds a description of a key field in the records to be sorted. The priority of the key decreases with increasing row number of their description, so the highest priority key is described in the first row, and so on.

Column one holds information about key field type and rule of sequencing. For the key field type the following conventions hold:

```
value key field type
+/-1  12-bit integer
+/-2  24-bit integer
+/-4  48-bit real
```

A long field or a 12-bit unsigned integer cannot be specified as a key field.

The sign of the type descriptor indicates the rule of sequencing: plus for ascending, minus for descending.

Column two is the relative location of the described field within a record. The meaning of this location is related to the key field type as follows (r stands for record length):

key field type	possible relative locations
48-bit real	1,2,3,...,r-1,r
24-bit integer	1,1.5,2,2.5,...,r,r+0.5
12-bit integer	1,1.25,1.5,1.75,2,...,r,r+0.25, r+0.5,r+0.75

This means that the relative location of a key field is counted in fractions of reals.

Zone state:

The zone state must be 9, in sort, i.e. initsort must have been called. The state is not changed by the procedure.

Example 1:

A certain sort key consists of an integer, two halfwords and a real stored in this order in double words 2 and 3 of a record. The following sequence is wanted:

1. ascending on the real.
2. descending on the first halfword.
3. descending on the integer.
4. ascending on the second halfword.

The following program will generate a proper code for comparison of two records.

```
begin zone z(size, 1, error);
   real array crit(1:4, 1:2);
   crit(1,1):= 4; crit(1,2):= 3;
   crit(2,1):= -1; crit(2,2):= 2.5;
   crit(3,1):= -2; crit(3,2):= 2;
   crit(4,1):= 1; crit(4,2):= 2.75
   initsort(z, length);
   initkey(z, crit, 4);
   .....
end;
```

2.71 inoutrec

This integer standard procedure gets a sequence of halfwords from a document and makes them available as a zone record, which later will be transferred to one or more documents in parallel. The input document may be scanned sequentially by means of inoutrec because the next call gets the elements just after the elements got now. The output document(s) may be filled sequentially by means of inoutrec because the next call of inoutrec will create by input a record which is transferred to the next halfwords of the output document(s).

Call:

```
inoutrec (za length)
```

`inoutrec`	`(return value, integer). The number of the halfwords left available in the present block for further calls of inoutrec before a change of block takes place.`
`za`	`(call and return value, zone array). The names za(1, za(2, ... za(n, are all equivalent names of the same zone record. Determines further the documents, the buffering and the positions of the documents, cf. (15).`
`length`	`(call value, integer, long or real). The number of halfwords in the new record. If length is odd, 1 is added to the call value.`

Zone state:

The zones in the zone array must all be open and ready for inoutrec (state 32+0 or 32+9, see getzone), i.e. the zones must only have been used by inoutrec or changerecio since the latest call of openinout or setposition after openinout.

To make sense, the documents should be internal processes, backing storage areas, discs, terminals, printers, paper tape punches or magnetic tapes. In the latter case, setposition must have been called after openinout on the zone or zones connected to magnetic tapes.

Blocking:

Inoutrec may be thought of as transferring in parallel the contents of the actual zone record to the halfwords just after the current logical positions of the output documents and moving the logical positions to just after the last halfword of the record. Then transferring the halfwords just after the current logical position of the input document and changing the logical position to after the last halfword of the record. The user may modify the information in the record before in outrec is called again.

Concerning the output:

Because the input/output is blocked, the actual transfers to the documents are delayed until the block is changed or until closeinout 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 halfwords 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 (15). The record becomes the first halfwords of the next share, but if the record still is too long, an alarm occurs.

Concerning the input:

Since all halfwords of the record are taken from the same block, if the record cannot be taken from the current block, the block is changed as described in (15). Then the record becomes the first halfwords of that block, but if it still cannot hold the record the run is terminated (empty blocks are completely disregarded).

Blockchange caused by the record overloading the block is the same thing whether you look at it from the output zone(s) or the input zones point of wiev: the blocklength is the same in all zones and the record is common to all zones. The block change first happens in parallel in all output zones and then in the input zone.

Records of length 0 need a special explanation: if not even room for a single word is left in the block when a record of length 0 is requested, the block is changed and the logical position points to just before the first word of the new block.

At end of document in the input zone, the standard error action will simulate an "end block" containing one word of three end medium characters. So the call b:= inoutrec(z,0) will in this case give the result b=2, and the "end block" may be read by inoutrec(z,2).

Note that inoutrec, like inrec, changes the blocks in such a way that a portion at the end of a block in the input zone 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 magnetic tape file in triple buffered mode copying to two magnetic tape files in parallel also in triple buffered mode may be programmed in this way:

```
begin

   procedure end_of_file (z, s, b);
   value                    s    ;
   zone                  z       ;
   integer                  s, b ;
   begin

      integer array        zdescr (1:20), sdescr (1:12);
      integer              index, operation, file, block;
      long    array field docname;

      docname := 2; <*fields docname in zone*>

      if s             extract 1 = 1 <*hard error bit*>  and
          s shift (-22) extract 1 = 0 <*not parity    *> then
         std_error (z, s, b);         <*give up      *>

         getzone_6 (z, zdescr    );
         getshare_6 (z, sdescr, zdescr (17)); <*used share*>

         index     := zdescr (12);
         operation := sdescr ( 4) shift (-12);

         if s shift (-22) extract 1 = 1 then
         begin <*persistent parity error*>
         if operation <> 3 <*not input*> then
            std_error (z, s, b); <*give up*>

         getposition (z, file, block);

         write (out,
         "nl", 1, "*" , 3,
         <: persistent parity error in input from tape:>,
         "nl", 1, "sp", 4, true, 12, zdescr.docname,
         <: file, block no :>, file, <:, :> block);

         parity (index) := true;

         if b < 4 then
            b := 4; <*not filemark*>
      end <*persistent parity error*> else
      if s shift (-18) extract 1 = 1 <*end of tape*>
      or b                       > 0 <*tape mark  *> then
      begin <*end of file*>
         end_file (index) := true;

         if operation = 3 <*input*>           and
             b         =0 <*nothing xferred*> then
            b       := 2;
      end <*end of file*>;
```

```
end <*end file*>;

    zone    array     file (3, buflengthio (3, 3, 512),
                            3, end_of_file);
    boolean array     parity, end_file (1:3);
    integer           giveup, i, hwds, sumhwds;

    giveup := 1 shift 18 + 1 shift 16; <*eot, eof*>

    for i := 1, 2, 3 do
       open (file (i), 4 shift 12 + 18, <*mtll*>
         case i of (<:mt600300:>, <:mt600301:>, <:mt600302:>),
         giveup);

    openinout (file, 1); <*file (1) input zone*>

    for i := 1, 2, 3 do
       setposition (file (i), 1, 0); <*skip volume header label*>

    for i := 1, 2, 3 do
       end_file (i) := parity (i) := false;

    sumhwds := 0;

    for hwds := inoutrec (file, 0) while hwds > 2 do
    begin <*still not end of file in input zone*>

    if end_file (2)
    or end_file (3) then
    begin
      <*handle end of tape in one of the output zones*>
      .
      .
      .
      end_file (1 <*or 2*>) := false;
    end else
    if parity (1) then
    begin
      <*handle persistent parity error in input zone*>
      .
      .
      .
      parity (1) := false;
    end else

    begin
      changerecio (file, hwds); <*blockchange next inoutrec*>
      <*maybe modify contents of record*>

      sumhwds := sumhwds + hwds+
    end;

end <*for hwds*>;

closeinout (file);
```

```
for i := 1, 2, 3 do
  close (file (i), true);

  write (out,
  "nl", 1, <:hwds transferred : :>, sumhwds,
  "nl", 1, <:segs transferred : :>, (sumhwds + 511) // 512);
end;
```

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, among them persistent parity error. After the positioning (but before the first input operation) endfile 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 areas on the backing storage if the second and third parameter to open were changed.

2.72 initsort

This standard procedure initiates a sorting process in a zone, so that the procedures newsort, outsort, deadsort, and lifesort can be used with the zone as a parameter.

This procedure is the ALGOL 5 version of startsort6, and must not be used together with changekey6. A call of initsort followed by a call of initkey is analoguos to a call of startsort6.

Call:

```
initsort (z, length)

z          (call and return value, zone). The name of the
           zone used for sorting.
length     (call value, integer). Max. length in double
           words of the records to enter the sorting
           process.
```

Zone Declaration:

A sort zone capable of holding N records at the same time must be declared as follows:

```
zone z((N+9)*(length+1), 1, error)
```

where length is maximum record length. The declaration has the same form as a zone declaration for input/output use with buffer length = (N+9)*(length+1) and no. of shares = 1.

The error procedure must be supplied by the user, as described in startsort6.

Zone state:

The zone must be in state 4, after declaration. The state becomes 9, in sort.

2.73 initzones

This procedure changes the buffersize and number of shares of each zone in a zone array.

Call:

```
initzones (za, bufsize, shares)

za          (call and return value, zone array). The
            buffersize and number of shares are changed
            for all zones:
            za(1), za(2), ... za(no of zones). The zone
            state must be 4 (after declaration) for all
            za(i).
bufsize     (call value, integer array). Bufsize(i)
            specifies the number of elements of 4
            halfwords each in the bufferarea to be
            allocated to the zone za(i).
shares      (call value, integer array). Shares(i)
            specifies the number of shares to be as signed
            to za(i).
```

Note:

The sum of all bufsize(i), 1 < = i < = no_of_zones, must not exceed the original total buffer size for the zone array za, as determinated by the declaration. Obviously bufsize(i) and shares(i) must be positive integers.

Zone state:

All zones of the zone array must be in state 4, after declaration. The zone state is not changed by the procedure.

Example 1:

The procedure can be used in a program where a number of files are handled by means of a logical file number, but the files use different block lengths:

```
begin
   zone array za(n, total_bufsize//n, shares, stderror);
   integer array buf, sh(1:n);
   <* blocklength, no_of_shares*>
   buf(1):= 128;   sh(1):= 1;<*file1:    1     1     *>
   buf(2):= 128*2; sh(2):= 2;<*file2:    1     2     *>
   buf(3):= 128*4; sh(3):= 1;<*file3:    4     1     *>

   ...
   initzones (za, buf, sh);
   ...
```

```
open(z(i), ...);
invar(z(i)); <* read from file no. i *>
```

2.74 inrec

This integer standard procedure is the ALGOL 5 version of inrec6. Inrec gets a sequence of elements of 4 halfwords 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 halfwords 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.
length	(call value, integer, long, or real). The number of elements of 4 halfwords each in the new record. Length must be >= 0.

For further description see inrec6.

Inrec may be used with advantage, if the document is considered to contain reals.

Example 1:

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:

```
for remaining:= inrec(z,1) while z(1)<= 0 do
   inrec(z,remaining);
         comment unused elements are skipped;
   inrec(z, z(1) - 1);
```

Another solution is to call the block procedure after all normal answers and let it adjust or check the z(1). Note that the relation z(1) = 0 instead of z(1) <= 0 would not work because a backing storage area is filled up with binary zeroes.

2.75 inrec6

This integer standard procedure gets a sequence of halfwords from a document and makes them available as a zone record. The document may be scanned sequentilly 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 halfwords 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 (cf. (15)).`
`length`	`(call value, integer, long, or real). The number of halwords 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, i.e. the zone must 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 disc process, a backing storage area, a terminal, 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 must be thought of as transferring the halfwords just after the current logical position of the document and changing the logical position to after the last halfword of the record.

However, all halfwords of the record are taken from the same block, so if the record cannot be taken fromt the current block, the block is changed as described in (15). Then the record becomes the first halfwords 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 there is not room for even a single word in the block, the block is changed and the logical position points to just before the first word of the new block. At end of document, the standard error action will simulate an "end block" containing one word of the three end medium characters. So the call b:= in rec6(z,0) will in this case give the result b=2, and the "end block" may be read by inrec6(z,2). (This is not valid for inrec, as inrec cannot read less than 4 halfwords).

Note that inrec6 changes the blocks in such a way that a portion at the end of a block may be skip ped. So be careful to read a backing storage area with the same share length as that with which is 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 halfwords):

```
begin
   zone file(2*128,2,endfile);
   boolean in_file;
   integer i;

   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
      begin in_file:= false; b:=512; end;

   open(file,18,<:mt600304:>,
         1 shift 18 + 1 shift 16);
   setposition (file,1,0);
   comment skip the label in file 0,
   in_file:= true;
   for i:= inrec6 (file, 20) while in_file do
   begin
   ...
   end;
   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) endfile may be called with tape mark indication. In this case however, b = 0, while b > 0 after in put of a tape mark.

The same piece of code would work for an area on the backing storage if the file was generated with a share length of 128 elements of 4 halfwords and if the second and third parameter to open were changed.

Example 2:

Two files of records of 100 halfwords 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 inp(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(inp(1),...,1 shift 16); open (inp(2),...
setposition ... setposition ...
large:= (-1) shift (-1);
inrec6(inp(1),100); inrec6(inp(2),100);
for k:= if inp(1).keyf < inp(2).keyf then 1 else 2
while inp(k).keyf < large do
   begin
      outrec6(result,100);
      tofrom(result,inp(k),100);
      inrec6(inp(k),100);
   end;
   close(result, ...);
```

Example 3:

You may read a magnetic tape file or backing sto rage area block by block in this way:

```
for b:= inrec6(z,0) while b > 2 do
begin
   <* b is now the block length in halfwords,
   the standard actions simulate one word
   containing <:<25><25><25>:> at end of document*>
   inrec6(z,6);
   <* the block is now available as one record*>
... ;
end;

comment check that the last record is
            the simulated end block;
inrec6(z,2);
if z.firstword <>
   long <:<25><25><25>:> shift (-24) extract 24
then error;
```

2.76 intable

This standard procedure exchanges the current input alphabet used by all the read procedures on character level.

Call:

```
intable (alpha) or
intable (0)
```

```
alpha     (call value, integer array of one dimension).
          Contains the character class and the value of
          each character in the new input alphabet as
          described below.
0         (call value, integer). A zero signals that the
          standard alphabet is to be used.
```

intable (alpha):

The actual contents of alpha are used in all calls of read procedures, until another array or the standard 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+tableindex) = class shift 12 + value extract 12
```

Class in an integer 0 < = class < = 4095. Value is an integer, - 2048 < = value < = 2047. The character 'c' is an integer, 0 < = c < = 255. The ISO characters utilize only half of this interval. The standard integer 'tableindex' 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 tableindex 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 < = value < = 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.

The elements 0:127 of alpha may be initialized with the ISO alphabet by a call of the standard procedure isotable.

intable (0):

The standard alphabet given in (14) is used until a new alphabet is selected. The value of tableindex has no influence on the alphabet. When the program starts, the standard alphabet is selected automatically.

You should not hesitate to use a special phabet 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.

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.

Example 1: Number variants

Assume you want to read numbers coded in ISO but with space regarded as blind information and with out exponent part. You may then proceed like this:

```
isotable(table);
table('sp'):= 0; table('''):= 7 shift 12 + ''';
...
comment define space, apostrophe, and other
          special characters;
intable(table); tableindex:= 0;
read(z,...);
```

Example 2:

```
begin
      ....
      begin
            integer array table (0:127);
```

```
            <* an alphabet is initialized in table *>
            table ('sp'):= ...
            intable (table); tableindex:= 0;
            ....
            intable (0);
            <* if you forget returning to the standard alphabet before
            leaving the block where table is declared, an undefined alphabet
            will be used *>
        end block;
        ...
    end program
```

Example 3:

See Example 3 of readall.

2.77 integer

This delimiter, which is a declarator, is used in declarations and specifications of variables of type integer.

Syntax:

```
integer <namelist>
```

Semantic:

The variables in namelist will all be of type integer, and occupy 24 bits in the memory area.

The value of an integer is in the interval:

```
-8388608 <= value <= 8388607
```

Example 1:

```
integer       i1;
integer       i2, yes, no, price;

procedure pip (a);
integer      a ;
```

2.78 integer divide (//)

This delimiter, which is an arithmetic operator, performs an integer division.

Syntax:

```
<operand1> // <operand2>
```

Priority: 3

Operand types:

integer or long

Result type:

When both operands are of type integer the result is of type integer, otherwise the result is of type long.

Semantic:

The result of the operator is defined as follows:

a//b = sign (a/b)* entier(abs (a/b))

Example 1:

```
a:= 35//12; <* a has the value 2 *>
b:= -13//3; <* b has the value -4 *>
```

2.79 invar

This standard integer procedure together with outvar, changevar, and checkvar are intended for easy handling of records of variable length. Every record must contain its own length in halfwords 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 getzone6). This procedure may call the block procedure with the status 1 shift 11, checksum error, if the record length wanted is < 4 or > remaining halfwords 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
          halfwords 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.
```

Zone state:

The zone z must be open and ready for record input (state 0 or 5), i.e. the zone must only have been used by invar or the like since the latest call of open or setposition.

Free parameter:

The free parameter 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. The free parameter could be set in the program by:

```
getzone6(z,ia); ia(11):= 1 shift 23;
setzone6(z,ia);
```

See below for further details.

Blocking:

You may think of invar in the way that the procedure tests the value of the first word just after the current logical position of the document. Now invar exposes as many halfwords as the length word indicates, including the two halfwords of this word.

However all halfwords 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 outvar).

Length errors. Checksum:

If the length word is <> 0, it is expected to be even, >= 4 and <= the number of halfwords 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 (see Example 2 below). If check is wanted (see free parameter 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 record_length is odd then
   length_word + 1 else
   length_word;
```

Remaining means the number of halfwords remaining in the present block including the length word. The terms zonestate, free parameter, and record length are explained in getzone6.

s: The status word parameter has the value 1 shift 11.

b: The halfwords 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, (see Example 2).

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 then
      begin comment end of document; ...
      end else
      if 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 job. In such case it is better to give up once status errors occur. It must be recognized, however, that situations could occur where it would be essential to make as much sense out of a file as possible in one job and then try to pick up the defect records in a later job.

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 after_parity (z,s,b);
zone z; integer s, b;
begin
   own integer faults; integer field length;
   integer array ia (1:20);
   proc dure drop;
      write (out, <:record dropped, expected::>,
                  z.length,
                  <: dropped::>, b, <: halfwords<10>:>);
```

```
   length:= 2;
   if s shift (-18) extract 1 = 1 or
      s shift (-16) extract 1 = 1 then
   begin    comment end of document or tapemark, simulate a dummy record and
            switch off the sum check, in order to prevent invar from calling
            the block procedure again;
      z.length:= b:= min_length;
      end_medium:= true;
      getzone6(z, ia); ia(11):= ia(11) extract 23;
      setzone6(z, ia);
   end
   else

   if s = 1 shift 11 then
   begin
      faults:= faults + 1
      if faults > max then stderror (z,s,b);
      if b <> z.length then drop
      else
      begin comment maybe only checksum error;
         if b<min_length or b>max_length then drop
         else

         begin
            comment now check the contents of the possible record. If it does
                  not seem sensible then drop it, else set a mark that it may
                  be erroneous;

            ....
            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;
   end s = 1 shift 11
   else
   if logand (s, -1 - (1 shift 22 <*parity error*> +
                       1 shift 15 <*writing enabled*> +
                       1 shift 1 <*normal answer*> +
                       1 shift 0 <*hard error*> ))
   <> 0 then stderror (z,s,b);
   comment give up if hard error, except in connection with parity error, ring
         indication, and/or normal answer;
end after_parity;
```

When the zone with this block procedure is opened, the give up mask should contain the tapemark bit (or end document bit if reading from backing storage), as the standard action would simulate a dummy block of 2 halfwords, which invar would consider a length error. On the other hand the give up mask should not contain the parity error bit, as the standard action, 5 readings, is wanted for parity error. The bit for checksum wanted should be set in the free zone parameter (see getzone6):

```
open(z,18,<:...:>, 1 shift 18 + 1 shift 16);
setposition(z,1,0);
getzone6(z,ia); ia(11):= 1 shift 23;
setzone6(z,ia);
repeat
   invar(z);
   ..... handle the record, note the error-mark;
until end_medium;
```

Example 3:

See Example of changevar.

2.80 isotable

This standard procedure initializes an array with the ISO character table for use in intable or outtable.

Call:

```
isotable (alpha)
```

```
alpha     (return value, integer array of one dimension).
          The elements alpha (0:127) of the array are
          initialized with class shift 12 + value of the
          ISO characters, as defined in (14).
```

Example 1:

See Example 1 of intable.

2.81 ldink

This boolean procedure (LAN Device **Link**) creates **in** a **zone** an external process of a given name (and maybe device number) with a link to a given LAN (RC9000-10) or ADP (RC8000) controller main process and a devicehandler of a given type, cf. ref. [25].

Call :

`ldlin` `(z, devno, devname, devtype, cspname, reason)`

`ldlink` (return value, boolean). True if the creation succeeded, false otherwise. If false, see the parameter reason for explanation.

`z` (call and return value, zone).
At call z describes the LAN main process (the name in the zone) and the mode to be used (all users/one user). If the devtype specifies a 3270 output handler, the zone also contains the index of the devicehandler to which a link shall be made. If the devtype specifies a 3270 input handler, the zone will at return contain the index of the device handler to which a link was made (returned in the parameter reason, too).

`devno` (call and return value, integer).
The number of the external process to be used. If devno = -1, the first free external process of kind = 68 is used. In all cases, the number of the process used is returned in devno.

`devname` (call and maybe return value, (string or long expression, or array of any type). The parameter contains the name of the external process to be created, packed in the usual way. If the first word of the name is zero, a wrk-name is generated. An external process is created and if devname is an array or a zone record, the name used is returned in devname.

If create peripheral process fails, the link will be removed and the procedure returns false.

`devtype` (call value, integer). Specifies the type of the device on the ADP :

```
1 : CSP console handler
2 : IMC port handler
3 : mailbox handler
4 : 3270 input handler
5 : 3270 output handler
```

```
6 : lanstat handler
7 : floppy disk handler
8 : CSP printer handler
9 : streamer handler
```

cspname (call value, (string or long expression, or array of any type). The parameter is only used for CSP device handlers (type = 1 or 8). The parameter contains the name of the CSP device to be reached by the link, packed in the usual way.

reason (call and return value, long).
The call value is used as number of buffers to be allocated if devtype = 2, IMC port handler. If the value is not positive, the default value, which is the number of free message buffers of the calling process, is used instead.

If the procedure returns true, reason will be :

devtype = 2, IMC port handler :

```
reason =
maxsendsize shift 32 + buffers unused shift 24 +
0           shift 12 + device index
```

The value maxsendsize is the maximum number of characters to be sent or received in out- or input operations, incl. a possible header character. The value may be used to determine or check the size of zone buffers engaged in IMC port communication. Please note that the value is 16 bits wide.

The value buffers unused is the number of buffers unused after having allocated the specified number of buffers to the port. Please note that the value is 8 bits wide.

other devtypes :

```
reason =
0     shift 36 + 0       shift 24 +
0           shift 24 + device index
```

The value device index is the device index used. The value is only relevant for devtype = 4, 3270 input handler, where the device index should be reused for the corresponding 3270 output handler. The value is only given for your information, because when the zone is reused to link an outputhandler after having linked an inputhandler, the index will still be kept in the zone, and is automatically

```
reused for the output handler. If the
procedure returns false, reason has the coding
:

reason =
status shift 36 + result shift 24 +
0      shift 12 + 0

0 < status < 4095, result = 1,
```

status bit, normal answer :

status :

The create link result :

0 ok, link created
3 no monitor resources, i.e. no unused external process of kind = 68
4 no free device handlers, or another resource problem on the IFP/ADP
5 link already exists

```
status = 0 , result > 1,
```

dummy answer :

the main process (or the IMC port process) does not exist, the calling process is not a user of the main process, or could not reserve the IMC port process, etc.

```
status = 4095, result > 1,
```

create peripheral process failed :

result = result of create peripheral process

Zone state

Zone state of the zone must be 0 = after open and is not changed by the procedure.

The zone must have been opened with modekind = mode shift 12 + kind, where

mode = 0 means only the calling process will be user of the external process
mode = 1 means all users of the IFP/ADP mainprocess will be users of the external process

kind = 0, internal process

Function

After parameter check, in case of devtype = 4 or 5, 3270 input handler or 3270 output handler, the procedure establishes a device index : for 3270 input handler it becomes the non specific value 255, for 3270 output handler it is taken from the zone (free parameter) where it is supposed to have been left over from the last link creation for an input handler. In case of devtype = 1 or 8, CSP console or CSP printer, the cspname is inserted in used share of the zone in mess + 8, ... mess + 14. The other parameters (devtype, device index and device number) are inserted in mess + 2, ... mess + 6, and the create link operation is sent and waited for. If status > 0, the procedure returns false, if status = 0, the actually used device number is returned in devno, and in case of devtype = 4, 3270 input handler, the actually used device index is left in 'free param' of the zone. If the first word of devname is zero, a wrk-name is generated for devname, and an external process is created with the given name and the actually used device number. If 'create peripheral process' fails, the link is removed again, and the procedure returns false with the result in reason. In case of devtype = 2, IMC port handler, the external process (IMC port process) is reserved, and as many buffers are allocated as the value of 'reason' at call, if it is positive, else the number of free message buffers of the calling process. The procedure returns the value of maxsendsize and number of unused buffers if <devtype> is 2, IMC port handler, in any case it returns the value of the device index used on the device handler and returns true.

2.82 ldunlink

This boolean procedure (**LAN Devide Unlink**) removes a link between an external process of a given name or with a given device number and a device handler on a LAN (RC9000-10) or ADP (RC8000) controller main process given in a zone, cf. ref. [25].

Call :

```
ldunlink   (z, devno, devname, reason)

ldunlink   (return value, boolean). True if the link was
           removed, false otherwise.

z          (call value, zone). At call z describes the
           LAN main process (the name in the zone).

devno      (call and return value, integer). The number
           of the external process to be used (must be >=
           0). If first word of devname is not zero, and
           an external process of that name is found, its
           device number overrules the device number
           given, else the device number given is used.
           In all cases, the number of the process used
           is returned in devno.

devname    (call value, string or long expression, or
           array of any type). The parameter contains the
           name of the external process to be removed,
           packed in the usual way. If the first word of
           the name is zero, the device number given in
           devno is used.

reason     (return value, long).

           Irrelevant if the procedure returns true
           (reason = 0 shift 36 + 1 shift 24 + 0).

           If the procedure returns false, reason has the
           coding :

           reason =
           0  shift 36 + result  shift 24 +
           0  shift 12 + 0
           result = 3, the device number does not
           identify an external process with a link to a
           device handler on the LAN main process given
           in the zone.
```

Zone state

Zone state of the zone must be 0 = after open, and is not changed by the procedure.

Function

After parameter check, the external process device number is identified. If the name in devname identifies an external process, its device number is used, else the device number given in devno is used. The number used is returned in devno, and a removelink operation is sent, waited for and checked.

2.83 len

This integer standard procedure returns the number for non-zero characters found in an array.

Call:

```
len ( arr )
```

len	(return value, integer). The number of characters found in the parameter array from character positions no. 1 and until the last non-zero character or until the last character position in the array.
arr	(call value, array of any type or zone record). Character position no. 1 is the 8 most significant bits in the word with halfword index 1, the last character position is the 8 least significant bits in the last word of the array. For boolean arrays, the lower bound and the upper bound must suit the word boundaries, i.e. lower bound must be odd (lower field index must be even), and the array must contain an even number of elements.

2.84 less than (<)

This delimiter, which is a relational operator, yields the value true or false.

Syntax:

<operand1> < <operand2>

Priority: 5

Operand types:

integer, long, or real.

Result type:

Always boolean

Semantic:

The relation takes on the value true whenever the corresponding relation is satisfied for the expressions involved, otherwise false.

In case one of the operands is different from type integer the relation is executed as a subtraction. Thus, you may be prepared for overflow, underflow, or spill.

Example 1:

```
a:= b < c;
if d < q then .... else ...;
while k < l do ...;
```

2.85 less than or equal (< =)

This delimiter, which is a relational operator, yields the value true or false.

Syntax:

```
<operand1> <= <operand2>
```

Priority: 5

Operand types:

integer, long, or real.

Result type:

Always boolean

Semantic:

The relation takes on the value true whenever the corresponding relation is satisfied for the expressions involved, otherwise false.

The relation is always executed as a subraction. Thus, you may be prepared for overflow, underflow, or spill (see Example 3 of monitor).

Example 1:

```
q:= a <= c;
if d <= k then .... else ...;
while x <= y do ...;
```

2.86 lifesort

This standard procedure makes available that zone record which is to be the next record in a sorted string of records from a zone. The winning record is selected among the active and inactive records in the zone, and at the same time all inactive records are activated.

The user is supposed to move the record away from the zone before next call of any sorting procedure.

Call:

```
lifesort (z, key) or liftsort (z)

z       (call and return value, zone). The name of the
        selected record.
key     (call value, integer procedure or empty). The
        name of a procedure for comparison of two
        records (see procedure newsort).
```

The key parameter is omitted if standard sequencing is used (see procedure startsort6).

Zone state:

The zone state must be 9, in sort, i.e. startsort6 (or initsort) must have been called. The state is not changed by the procedure.

Example 1:

See Example 1 of newsort for a user specified key procedure.

Example 2:

See Example 2 of deadsort.

2.87 ln

This real standard procedure performs the mathematical function 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.

Example 1:

A procedure which gives the decimal logarithm may look like this:

```
real procedure log(x);
value x; real x;
log:= ln(x) / ln(10);
```

2.88 lock

This integer standard procedure immediately transfers a number of program or context/virtual activity data segments to memory and locks tem: i.e. they will not be released at a later time caused by some request for space for other segments.

Call:

```
lock ( one or more parameter pairs );

lock  (return value, integer). Number of segments locked
      as a result of this call of lock. May include
      lock's own segment although not specified.
pair  (call value, integer, label or procedure). Each
      parameter pair specifies a number of segments to be
      locked.
```

Lock scans the parameters from left to right and evaluates each parameter pair according to its type:

- a procedure identifier indicates the entry segment of the procedure.

  (Notice that the name of a type procedure without parameters is a function designator i.e. an arithmetic expression evaluated at call time, and cannot be used here).

- a label expression defines a point in the program indicating a segment.

- an integer expression defines a segment number in the program, numbered consecutively 0,1,2, ... when it is first parameter or when it is second in a pair of integer expressions. When it is second in a pair, the first one being a procedure identifier or a label expression, it indicates the segment numbered:
  segment indicated by first parameter + integer.

If the segment number indicated by the second parameter exceeds the last program or context/virtual activity segment, it is reduced properly.

Now lock transfers the segments in the interval indicated by the pair including the segments indicated, excluding those already locked, to memory and locks them i.e. all the segments in the interval are locked.

Segments already in memory but unlocked will be transferred to memory and locked.

The segment of lock itself may be locked although not specified, if its place in memory is 'overhauled' by the segments transfer red to memory.

If progmode < = 0 at call, the number of segments thus specified (abs progmode) will be released prior to the locking of new segments (cf. 2.126), and at return progmode = 1, i.e. locking passive.

Segment numbers of all segments of the program, including external procedure segments and the first virtual data segment, may be obtained from a compilation (algol or fortran) with the option survey.yes (cf. (15)). Segments of the non resident runtime system may be locked, too, specified by segment number. The runtime system takes up the first 15 segments of the object program:

No.	:	functions	Name:
0	:	resident part of runtime system	no name
.	:	.	.
.	:	.	.
.	:	.	.
7	:	will not be locked by lock	-
8	:	alarm segm. 0, only active at runtime alarm,	<:alarm segm0:>
9	:	alarm segm. 1, only active at runtime alarm,	<:alarm segm1:>
10	:	declaration of zones, init data/zone common,	<:zone declar:>
11	:	long integer operations, call user blockproc, stderror	<algolcheck:>
12	:	i/o: inblock, outblock, check	<check :>
13	:	i/o: standard error actions, parent,00	<:check spec:>
14	:	power function : a**x	<:power func.:>
15	:	extent area, parent message, label alarm	
16	:	first main program segment	line interval
.	:	.	.
.	:	.	.
.	:	.	.
n	:	last main program segmt	-
n+1	:	first line interval external procedure segment name external	
.	:	.	-
.	:	.	-
.	:	.	-
n+m	:	last external procedure segm last external	-
n+m+1	:	first virtual data segment	no name

Error Messages:

The procedure may terminate the program with the runtime alarm:

```
param <kind> lock
called from ...
```

where kind is coded:

```
kind = paramno * 10 - explanation
```

Explanation = 1,2,3:	The parameter is an integer, label or procedure identifier, but the segment numbers in the pair (first, last) relate: first > last
Explanation = 4:	The second parameter of the pair is missing
Explanation = 5:	The parameter is neither a procedure identifier, a label expression, or an integer expression.

Example 1:

If you want to lock in memory segment 11 (long division and multiplication), two segments containing procedure readin, and one or more segments surrounding label seglock1 to label seglock2, this could be done by the call:

```
lock(11,11, readin, 2, seglock1, seglock2);
```

Example 2:

All program segments excluding context/virtual activity data segments are locked:

```
lock (0, progsize);
```

Note: segments 0-7 are not locked, they are the resident runtime system.

Example 3:

All context/virtual activity data segments existing so far are locked:

```
lock (progsize, 8388607);
```

2.89 locked

This integer standard procedure transfers the segment numbers (0,1,...) of the segments, which are locked for the moment, to an integer array parameter.

Call:

```
locked (ia)

locked   (return value, integer). Number of segments
         locked at the moment.
ia       (return value, integer array). ia(1), ia(2),...
         ia(locked) is the list of segment numbers for
         the segments locked at the moment.
```

Example 1:

If the call

```
locked (ia);
```

is used after the call shown in Example 1 under lock, the results could be:

```
ia(1) = 11
ia(2) = 24
ia(3) = 25
ia(4) = 32
ia(5) = 33
ia(6) = 34
```

2.90 logand

This long standard procedure 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 (15)). 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 had been applied. Boolean: A boolean is considered 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 1:

See Example 2 of invar.

Example 2: Information retrieval

Each record in a certain file has an element inf that contains binary information in each binary position (sex, salary on hour basis or not, ...). A statement that writes out the identification for each record that fulfil all the criteria in a search element read, could look like this:

```
for i:= inrec6 (z, length)
while z.ident <> endident
do
   if logand (searchcrit, z.inf) = searchcrit then
      write (out, "nl", 1, z.ident);
```

2.91 logor

This long standard procedure performs 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.
```

Example 1:

About the same as Example 2 for logand, but you are now searching for records that are not in conflict with the searc criteria i.e. each found re cord should only contain all, some or none of the information found in the search criteria. This could be done by replacing

```
logand (searchrit, z.inf) = searchrit
```

by:

```
logor (searchrit, z.inf) = z.inf
```

2.92 long

This delimiter, which is a declarator, is used in declarations and specifications of variables of type long.

Syntax:

```
long <namelist>
```

Semantic:

The variables in namelist will all be of type long, and occupy 48 bits in the storage area.

The value of a long is in the interval:

```
-140 737 488 355 328 <= value <= 140 737 488 355 327.
```

All values can be assigned to the variable by use of shift and the like, but if you assign by constants or character reading procedures you are confined to the range:

```
-140 737 488 355 327 <= value <= 140 737 488 355 327
```

Example 1:

```
long l1;
long l2, yes, no, price;

procedure pip(a);
long          a ;
```

Example 2:

The greatest possible positive and negative long values can be assigned by the statements:

```
max_pos:= extend (-1) shift (-1);
max_neg:= extend   1  shift 47;
```

2.93 long

This delimiter, which is a transfer operator, changes the type string and real to type long.

Syntax:

```
long <operand>
```

Priority: 1

Operand type:

real or string.

Result type:

long.

Semantic:

Changes the type of a string or a real to type long. The binary pattern of the operand is unchanged. The binary pattern of a string and a real is described in (14).

Note: This use of the delimiter long is totally different from its use in a declaration or specification.

Example 1:

```
1:= long <:abcde:> add 'f';
```

Example 2:

See the examples in real and replace real with long where real is used as a transfer operator.

Example 3:

See Example 1 of swoprec6.

2.94 message

This delimiter, which is a compiler directive, may print a message during the translation of a program. Messages follow the same rules as comment.

Syntax:

```
; message <text not containing ";"> ;
          may replace any ; (semicolon)

begin message <text not containing ";"> ;
              may replace any begin
```

Semantic:

The text between message and semicolon is printed on current output if the program is translated with the parameter message.yes.

Example 1:

You can spare 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 default value). The messages are then printed with their line numbers attached and you can easily find any other line given its line number.

2.95 minus (-)

This delimiter, which is an arithmetic operator, can be used both as a dyadic and as a monadic operator.

1. Dyadic:

Syntax:

```
<operand1> - <operand2>
```

Priority: 4

Operand types:

integer, long or real.

Result type:

```
<integer> - <integer> is of type integer
<integer> - <long>    is of type long
<integer> - <real>    is of type real
<long>    - <integer> is of type long
<long>    - <long>    is of type long
<long>    - <real>    is of type real
<real>    - <integer> is of type real
<real>    - <long>    is of type real
<real>    - <real>    is of type real
```

Semantic:

This operator yields the normal arithmetic difference of the expressions involved.

2. Monadic:

Syntax:

```
- <operand>
```

Priority: 4

Operand type:

integer, long, or real.

Result type:

- <integer>	is of type integer
- <long>	is of type long
- <real>	is of type real

Semantic:

This monadic operator yields the "opposite" value of the operand.

Example 1:

```
5-7
a-q
-5-a
s shift (-11)
(-1) shift (-1)
```

2.96 mod

This delimiter, which is an arithmetic operator, yields the remainder corresponding to an integer division.

Syntax:

```
<operand1> mod <operand2>
```

Priority: 3

Operand types:

integer or long.

Result type:

When both operands are of type integer, the result is of type integer, otherwise the result is of type long.

Semantic:

The value of i mod j is defined as

i - i//j*j

The sign of i mod j is the same as the sign of i.

Example 1: 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;
```

2.97 monitor

This integer standard procedure is the algol equivalent of the monitor procedures, i.e. the algol interface to the monitor. 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 Monitor manuals (1) and (2) for the details and the ideas behind each entry.

Be aware that the monitor tables have halfword addresses.

If the requirements stated below are not fulfilled, or if the situation termed 'parameter error' in (2) occurs, the run will be terminated with the alarm: `entry`.

Values of fnc not mentioned below will terminate the program with the same alarm.

Call:

```
monitor (fnc, z, i, ia)
```

```
monitor  (result value, integer). In most cases the
         result of the corresponding call of a monitor
         procedure, the meaning of which is found in the
         manual, (2).
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,
         counted from lexicographical index 1, are
         required in the various cases.
         Whenever the array is used, it is used from
         lexicographical index 1 and the required length
         is stated in each case, measured in words.
```

In most cases only some of the last 3 parameters are actually used by the procedure. The value of fnc determines the function as follows:

fnc = 4, process description:

```
monitor  result, i.e. process description address, 0 if
         the process does not exist.
z        (call value). Contains the process name.
```

```
i       dummy
ia      dummy
```

fnc = 6, initialize process:

```
monitor  result, i.e. 0 means process initialized, 1,2,3
         means not initialized.
z        (call value). Contains the process name.
i        dummy
ia       dummy
```

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.
i        dummy
ia       dummy
```

fnc = 10, release process:

```
monitor  result dummy
z        (call value). Contains the process name.
i        dummy
ia       dummy
```

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). Deviced number.
ia       dummy
```

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.
ia       dummy
```

The format of messages and answers used in the following 6 functions, and in 70, 82, 84, 124, can be found in (2) and (3) and in the description of getshare6.

fnc = 16, send message:

```
monitor  buffer address, 0 if the buffer claim is
         exceeded.
z        (call value). Contains the process name of the
         receiving process.
i        (call value). The number of a share with in 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 getshare6).
         Note that you may change the message in the
         share by means of the procedure setshare6.
ia       dummy
```

The value of "current activity no" is used as message identification. This is stored in the message buffer (buffer address - 2), which later (e.g. in wait event) may supply the number of the sending activity (see (19)), concurrent i/o transfers).

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 with in z.
         The share state must be the buffer address at
         call time, at return time it is 0.
ia       (return value, length >= 8). The answer is
         stored here.
```

If the program is in activity mode, and the give up mask in the zone includes 1 shift 9, this function will execute an implicit passivate statement just before call of the monitor procedure.

fnc = 20, wait message:

```
monitor  result, i.e. process description address of the
         sender, positive for a normal message, negative
         for a message from a removed process, or 0 if
         the buffer claim is exceeded.
z        (return value). The process name is stored here.
i        (return value). Buffer address.
ia       (return value, length >= 8). The message is
         stored here.
```

fnc = 22, send answer:

```
monitor  result dummy
z        dummy
i        (call value). Buffer address.
ia       (call value, length >= 9). The first 8 elements
         contain the answer, the 9th element contains the
         result, which is 1 for a normal answer, 2,3,4,5,
         for a dummy answer.
```

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 >= 8). If a message is
         received, it is stored here.
         If an answer is received, the message
         identification (in buffer address -2) is stored
         in ia(1).
         In this case the buffer address returned in i
         corresponds to a message buffer address in the
         share state of some zone (see getshare6).
         The message identification has the following
         meaning:

         ia(1)>0:   the activity number of the activity,
                    which sent the message
         ia(1)=0:   the message was sent in monitor or
                    neutral mode
         ia(1)<0:   the message was sent in disable mode.
```

(cf. fnc = 16, send message).

Note that concerning the name of the sending process supplied in z for messages, and the message identification supplied in ia(1) for answers, the ALGOL equivalent of wait_event differs from the monitor procedure wait_event and equals the monitor procedure test_event.

fnc = 26, get event:

```
monitor  result dummy
z        dummy
i        (call value). Buffer address pointing to a
         message. An answer must not be released in this
         way - use wait answer (or procedure check)
         instead.
ia       dummy
```

fnc = 28, test users/protectors/reserver:

```
monitor  result, 0 means process tested 3, 4, not tested.
         If result=0, further information is returned in
         i.
z        (call value zone). Contains the name of the
         external process.
i        (return value, integer) Contains further
         information:
bit 23=1: internal process   is   user     of     external
bit 22=1: internal process   is   reserver of     external
bit 21=1: other    processes are  users    of     external
bit 20=1: another  process   is   reserver of     external
bit 19=1: internal process   has  write protected external
bit 18=1: other    processes have write protected external

ia       <call value, length >=4). contains the name of
         the internal process. If the first word of the
         name is zero, it means calling process.
```

fnc = 30, set writeprotect

```
monitor  result, i.e. 0 means area process write
         protected, 1,2,3 means not writeprotected.
z        (call value). Contains the area process name.
i        dummy
ia       dummy
```

fnc = 32, remove writeprotect:

```
monitor  result, i.e. 0 means writeprotection removed, 3
         means not removed
z        (call value). Contains the area process name.
i        dummy
ia       dummy
```

fnc = 34, set number of active processors:

```
monitor  result, i.e. 0 means function executed, 1, 2
         mean function not executed
z        dummy
i        (call value). The number of processors set
         active
ia       dummy
```

The format of the catalog entry tails used in the following 3 functions is described in (6).

fnc = 40, create entry:

```
monitor  result, i.e. 0 means entry created, 2,3,4,5,6,7
         means entry not created.
```

```
z       (call value). Contains the entry name.
i       dummy
ia      (call value, length >= 10). Contains the tail of
        the entry.
```

fnc = 42, lookup entry:

```
monitor result, i.e. 0 means entry looked up, 2,3,6,7
        means not looked up.
z       (call value). Contains the entry name.
i       dummy
ia      (return value, length >= 10). The tail of the
        entry is stored here.
```

fnc = 44, change entry:

```
monitor result, i.e. 0 means changed, 2,3,4,5,6,7 means
        entry not changed.
z       (call value). Contains the entry name.
i       dummy
ia      (call value, length >= 10). Contains the new
        tail of the entry.
```

fnc = 46, rename entry:

```
monitor result, i.e. 0 means entry renamed, 2,3,4,5,6,7
        means entry not renamed.
z       (call value). Contains the present entry name.
i       dummy
ia      (call value, length >= 4). Contains the new
        entry name.
```

fnc = 48, remove entry:

```
monitor result, i.e. 0 means entry removed, 2,3,4,5,6,7
        means entry did not exist or entry is not
        removed.
z       (call value). Contains the entry name.
i       dummy
ia      dummy
```

fnc = 50, permanent entry:

```
monitor result, i.e. 0 means entry made permanent,
        2,3,4,5,6,7 means entry not permanent.
z       (call value). Contains the entry name.
i       (call value). Catalog key.
ia      dummy
```

fnc = 52, create area process:

```
monitor  result, i.e. 0 means area process created,
         1,2,3,4,5,6 means process not created.
z        (call value). Contains the process name.
i        dummy
ia       dummy
```

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        dummy
ia       dummy
```

fnc = 56, create internal process:

```
monitor  result, i.e. 0 means process created, 1,2,3,6
         means process not created.
z        (call value). Contains the process. The process
         will be created in the buffer area of z.
i        dummy
ia       (call value, length >= 9). Contains the
         parameters in this way:
         ia(1) buffer index for start of process
         ia(2) buffer index for last of process
         ia(3) buffer claim shift 12 + area claim
         ia(4) internal claim shift 12 + function mask
         ia(5) mode
         ia(6) lower limit of max base
         ia(7) upper limit of max base
         ia(8) lower limit of std base
         ia(9) 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.
         (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.
ia       dummy
```

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.
ia        dummy
```

The value of 'current activity no' is used as message identification. It is stored in the message buffer (address-2), which later (e.g. in wait event) may be obtained to identify the sending activity.

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.
i         dummy
ia        (call value, length >= 6). Contains the modified
          registers (w0-w3, ex, ic).
```

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.
i         dummy
ia        dummy
```

fnc = 66, test event:

```
monitor   result
          0 next event is a message,
          1 next event is an answer,
          -1 the event queue is empty or
          next event is a message but claims exceeded

z         (return value). If result = 0, the name of the
          sending process is stored here, else z is
          unchanged.
i         (call and return value).
          At call the previous message buffer address. At
          return the next message buffer address.
          If result = -1, next message buffer address = 0
          If result = -1 i is unchanged.
ia        (return value, length >= 8).
          If result = 0, the message is stored in ia
          (1:8).
          If result = 1, the message identification (from
          the message flag) is stored on ia (1).
          In this case, the message buffer address
          returned in i corresponds to a message buffer
          address in the share state of some zone, cf. the
          procedure getshare6. The message identification
          then has the following meaning:
          ia (1) > 0 :    the activity number of the
                          activity which has sent the
```

```
                    message
        ia (1) = 0 :  the message was sent in monitor
                      mode or in neutral mode
        ia (1) < 0 :  the message was sent in disable
                      mode
```

cf. the procedure send message, fnc = 16.

fnc = 68, generate name:

```
monitor  result, i.e. 0 means name generated, 2 means
         name not generated.
z        (return value). The generated name is stored
         here.
i        dummy
ia       dummy
```

fnc = 70, copy memory 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 halfword.
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 (see (3)):
         ia(1) then should be set by the user
         ia(2) if result <> 0 then 0 else halfwords
         copied
         ia(3) if result <> 0 then 0 else characters
         copied
         ia(9) if result = 3 then 3 else 1.
```

fnc = 72, set catalog base:

```
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 first-word-null-name, meaning own
         process.
i        dummy
ia       (call value, length >= 2). Contains the base to
         be set.
         ia(1) lower limit of the base
         ia(2) upper limit of the base
```

fnc = 74, set entry base:

```
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.
i        dummy
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:

```
monitor  result, 0 means entry looked up, 2,3,6,7 means
         entry not look up.
z        (call value). Contains the entry name.
i        dummy
ia       (return value, length >= 17). The head and tail
         of the entry is stored here. (The format is
         described in (2).
```

fnc = 78, set backing storage claims:

```
monitor  result, 0 means claims set, 1,2,3,6 means claims
         not set.
z        (call value). Contains the name of a child
         process.
i        dummy
ia       (call value, length >= 4 + 2*no of keys). The
         first 4 words contain the name of the backing
         storage document. The next:
         ia (5) entry claim, key 0
         ia (6) segment claim, key 0
         .......
         ia (5+2*max key) entry claim, max key
         ia (6+2*max key) segment claim, max key
```

fnc = 80, create pseudo process:

```
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.
i        dummy
ia       dummy
```

fnc = 82, regret message:

```
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.
ia       dummy
```

fnc = 84, general copy

```
monitor  result, 0 means area copied, 2 and 3 mean area
         not copied.
z        (call value). Contains the memory area to or
         from which the copying will take place. The
         limits of the copying will be
         z.first_field_index and z.last_field_index.
i        (call value). The message buffer address of the
         input or output message defining sender's copy
         area.
ia       (call and return value, length >= 9). At call ia
         contains information about the copying wanted,
         at return it contains information about the
         copying performed, almost ready to be used by
         send answer:
```

call:	return:
ia (1) function	should be set by user
ia (2) first field index	if result <> 0 then 0 else halfwords transferred
ia (3) last field index	undefined
ia (4) start relative	unchanged
ia (9) dummy	if result = 3 then 3 else 1

fnc = 86, lookup aux entry

```
monitor  result, 0 means entry looked up, 2, 3, 6, 7 mean
         entry not looked up.
z        (call value). Contains the entry name.
i        dummy.
ia       (call and return value, length >= 21). Contains
         the document name in word 18, ..., 21. At
         return, the tail of the entry is stored in word
         8, ..., 17.
```

fnc = 88, clear statistics in aux entry

```
monitor  result, 0 means statistics initialized, 2, 3, 6,
         7 mean statistics not initialized.
z        (call value). Contains the entry name.
i        dummy.
ia       (call value, length >= 21). Contains the
         document name in word 18, ..., 21.
```

fnc = 90, permanent entry in auxiliary catalog:

```
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 backing storage document.
```

fnc = 92, create entry lock process

```
monitor  result, 0 means entry lock process created, 1,
         2, 3, 6, 7 mean entry lock process not created.
z        (call value). Contains the entry name.
i        dummy.
ia       dummy.
```

fnc = 94, set priority

```
monitor  result, 0 means priority changed, 3, 6 mean
         priority not changed.
z        (call value). Contains the name of the child
         process.
i        (call value). Priority level offset.
ia       dummy.
```

fnc = 96, relocate process

```
monitor  result, 0 means that the child process is ready
         to start in the new memory area starting in
         z.field_index, 3, 6 mean that it is not.
z        (call value). Contains the name of the child
         process.
i        (call value). Field index in zone record.
ia       dummy.
```

fnc = 98, change address base

```
monitor  result, 0 means that the logical address base of
         the child process is changed, 3, 6 mean that it
         is not.
z        (call value). Contains the name of the child
         process.
i        (call value). Address displacement.
ia       dummy.
```

fnc = 102, prepare backing storage:

```
monitor  result, i.e. 0 means chaintable allocated,
         1,2,3,4,5,6,7 means chaintable not allocated.
z        (call value). The zone record holds the chain
         head.
```

```
i       dummy
ia      dummy
```

fnc = 104, insert entry:

```
monitor result, i.e. 0 means entry inserted in
        maincatalog, 1,2,3,4,5,6,7 means entry not
        inserted in maincatalog.
z       (call value). The zone record holds the chain
        head.
i       dummy
ia      (call value, length >= 17). Contains head and
        tail of the entry.
```

fnc = 106, insert backing storage document:

```
monitor result, i.e. 0 means document inserted 1,2,4,6
        means document not included.
z       dummy
i       dummy
ia      (call value, length >= 21). Contains the
        document name in word 18, ..., 21.
```

fnc = 108, delete backing storage document:

```
monitor result, i.e. 0 means document removed, 1,2,4,5,6
        means document not removed.
z       dummy
i       dummy
ia      (call value, length >= 21). Contains the
        document name in word 18, ..., 21.
```

fnc = 110, delete entries:

```
monitor result, i.e. 0 means entries deleted, 1,2,3,4,6
        means not deleted.
z       dummy
i       dummy
ia      (call value, length >= 21). Contains the
        document name in word 18, ..., 21.
```

fnc = 112, connect main catalog:

```
monitor result, i.e. 0 means catalog connected,
        1,2,3,4,5,6,7 means catalog not connected.
z       (call value). The zone record holds the chain
        head.
i       dummy
ia      (call value, length >= 4). Contains the catalog
        name.
```

fnc = 114, remove main catalog:

```
monitor  result, i.e. 0 means catalog removed, 7 means
         main catalog not removed.
z        dummy
i        dummy
ia       dummy
```

fnc = 118, lookup backing storage claims

```
monitor  result, 0 means claims looked up, 2, 3, 6 means
         claims not looked up.
z        (call value). Contains the name of an internal
         process. If the first word of the name is zero
         it means calling process.
i        dummy
ia       (call and return value,
         length >= 4+2 * no_of_keys.


ia (1)   call value, name of the backing storage document
.        -
.        -
ia (4)   -
ia (5)   return value, entry   claim, key = 0
ia (6)   return value, segment claim, key = 0
.
.
.
ia (5 + 2 * maxkey), entry   claim, key = maxkey
ia (6 + 2 * maxkey), segment claim, key = maxkey
```

fnc = 120, create aux entry and area process:

```
monitor  result, i.e. 0 means entry and area process
         created, 1,2,3,4,5,6 means entry and area
         process not created.
z        (call value). Contains the area process name.
i        dummy
ia       (call value, length >= 21). Contains head and
         tail in word 1, ..., 17, and the document name
         in word 18, ..., 21.
```

fnc = 122, remove aux entry:

```
monitor  result, i.e. 0 means entry removed, 1,2,3,6
         means entry not removed.
z        dummy
i        dummy
ia       (call value, length >= 21). Contains head and
         tail in word 1, ..., 17 and the document name in
         word 18, ..., 21.
```

fnc = 124, send pseudo message

```
monitor  buffer address, 0 if the buffer claim is
         exceeded.
z        (call value). Contains the name of the receiving
         process.
i        (call value). The number of a share within z.
         The share state at call time must be 0 or 1, at
         return it is the buffer address. The message
         sent is given in the share descriptor, cf. the
         procedure getshare6.
         Note that you may change the message in the
         share by means of the procedure setshare6.
ia       (call value, length >= 2). The pseudo process
         description address is contained in ia (1). ia
         (2) is for the time being not used.
```

The procedure works as the procedure send message (fnc = 16) with the following two exceptions:

1) the receiver process must be an internal process

2) the pseudo process is inserted as sender of the message buffer

The value of "current activity number" is passed on in the message flag as for send message.

fnc = 126, set common protected area

```
monitor  result, 0 means cpa set, 3 means cpa not set.
z        (call value). Contains the process name.
i        (call value). Contains the cpa limit.
ia       dummy.
```

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);
integer i;
open(z,4,<:sldata3:>,0);
<*  The zone contains now the document name. The document is not initalized in
    case of kind=4*>
tail(1):= s;
tail(2):= 3; <*on the disk with the most permanent resources*>
for i:= 3 step 1 until 10 do tail(i) := 0;
tail(6):= systime (7,0, 0.0); <*shortclock*>
if  monitor(40<*create_entry*>,z,0,tail) > 0 then error(1);
outrec(z,...);
```

Example 2: Scope user of an area:

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 (2)), the entry base may be set to the user base of the process.
Let the zone z be connected to the area to be scoped.

```
system(11)bases:(i,ia);
ia(1):= ia(5); ia(2):= ia(6); <* fetch the user base;*>
if monitor(50)permanent entry:(z,3,ia) <> 0 then
error(1)
else
if monitor(74)set_base:(z,0,ia) <> 0 then
error(2)
else ...
```

Example 3: Find scope of an entry:

As the catalog base of an internal process and of a catalog entry may cover almost the full integer range they must be handled as longs when relations of type < = or > = between them are calculated, in order to prevent overflow.

```
system(11)bases:(i,bases);
monitor(76 <*head_and_tail*>, z,0,entry) <> 0
     then goto error;
case entry(1) extract 3 + 1 of
begin

   <*key 0, maybe temp *>
   if entry(2) = bases(3) and
      entry(3) = bases(4)
   then scope:= 1  <*temp*>
   else scope:= 6; <*undef*>

   <*key 1 *>
   scope:= 6; <*undef*>

   <*key 2, maybe login *>
   if entry(2) = bases(3) and
      entry(3) = bases(4)
   then scope:= 2  <*login*>
   else scope:= 6; <*undef*>

    <*key 3, user, project, or system *>
   begin
        l1:= entry(2); l2:= entry(3);
     if l1 = bases(5) and
        l2 = bases(6) then scope:= 3 <*user*>
     else
     if l1 = bases(7) and
        l2 = bases(8) then scope:= 4 <*project*>
     else

     if l1 <= extend bases(7) and
        l2 >= extend bases(8) then scope:= 5
<*system*>
```

```
     else scope:= 6; <*undef*>
   end

end case;
write(out,<:the scope is: :>,case scope of(
    <:temp:>,<:login:>,<:user:>,
    <:project:>,<:system:>,<:***, i.e. undef:>));
```

Example 4: Get the claims of a process on a given bs-device

```
boolean

procedure claimproc (keyno,bsno,bsname,entries,segm,slicelength);
value keyno;
integer keyno,bsno,entries,segm,slicelength;
long array bsname;
<*
claimproc          (return, boolean). True if bsno>=-1 and bsno <= max_bsno
                   and keyno is legal else false. If claimproc is false then
                   all return parameters are zero.
keyno              (call, integer) 0=temp
                                   2=login
                                   3=user/project
bsno               (call/return, integer). If call value is -1 then return
                   value is main backing storage device number, else bsno is
                   unchanged
bsname             (return, long array 1:2). Name of called device
entries            (return, integer). No. of entries of key=keyno on device
segm               (return, integer). No. of segm. of key=keyno on given
                   device
                   slicelength (return, integer). Slicelength on given device
*>

begin
   integer bsdevices,firstbs,ownadr,mainbs,i;
   long array field name;
   integer array core(1:18);

   system(5,92,core);
   bsdevices:=(core(3)-core(1))//2;
   firstbs:=core(1);
   mainbs:= core(4);
   ownadr:=system(6,i,bsname);
   if bsno<=1 or bsno>=bsdevices or
   keyno<>0 and key<>2 and keyno<>3 then
   begin
      claimproc:=false; goto exitclaim
      end;
      claimproc:=true;
      begin integer array nametable(1:bsdevices);
      name:=18;
      system(5,firstbs,nametable);
      if bsno= -1
      then <*find main device number*>
      for bsno:= bsno + 1 while nametable(bsno+1) <> mainbs do;
      system(5,nametable(bsno+1)-36,core); <*get chaintable*>
```

```
      if core(10)=0 then goto exitclaim;
      bsname(1):=core.name(1); bsname(2):=core.name(2);
      slicelength:=core(15);
      system(5,ownadr+core(1),core); <*get process description*>
      entries:=core(keyno+1) shift (-12);
      segm:=core(keyno+1) extract 12 * slicelength;
   end;
   if false then
   begin
      exitclaim:
            entries:=segm:=slicelength:=0;
            bsname(1):=bsname(2):=0;
   end;
end claimproc;
```

Claims on a specific device are found as follows:

```
bsno:=-1;
for bsno:=bsno+1 while
      claimproc(keyno,bsno,bsname,entries,segm,slicelength)
      and
      not (searchname(1)=bsname(1) and searchname(2)=bsname(2)) do;
```

Max claims are found as follows:

```
maxentr:=max:=maxslice:=0;

maxbs(1):=maxbs(2):=0;
bsno:=-1;
for bsno:=bsno+1 while
      claimproc(keyno,bsno,bsname,entries,segm,slicelength) do
if entries>0 and segm>max then
begin
   maxentr:=entries; max:=segm; maxslice:=slicelength;
   maxbs(1):=bsname(1); maxbs(2):=bsname(2);
end;
```

Example 5: Renaming an entry

The procedure renames a catalog entry. The procedure works just like the utility program rename, with the extension that a working name may be returned.

```
integer
procedure renameproc (oldname, newname);
long array oldname, newname;
<*
   renameproc (return integer) 0 ok
                               1 new name exists already
                               2 cat i/o error
                                 document not mounted or
                                 document not ready
                               3 oldname not found
```

```
                                4 name protected
                                5 name in use
                                6 name format illegal
                                7 catalog inconsistent

   oldname (call long array) contains old name

   newname (call long array) contains new name or <::>
                             if newname(1)=long<::> then
                                newname is a return parameter,
                                containing a wrk-name.
>*

begin
  integer i; boolean wrk;
  long array field laf;
  zone zhelp(1, 1, stderror);
  integer array ia(1:20);
  wrk:=newname(1)=long<::>;
  if wrk then
  begin
generate_next:
    monitor(68<*generate*>, zhelp, 0, ia);
    getzone6(zhelp, ia);
    laf:=2;
    newname(1):=ia.laf(1); newname(2):=ia.laf(2);
  end;

  laf:=0;
  ia.laf(1):=newname(1);
  ia.laf(2):=if newname(1) extract 8 = 0 then long<::>
             else newname(2);
  open(zhelp, 0, oldname, 0);
  renameproc:= i:= monitor(46<*rename*>, zhelp, 0, ia);
  if i=3 then
  begin <*oldname not found, or newname already exists*>
    i:=monitor(42<*lookup*>, zhelp, 0, ia);
    if i=0 then
    begin
      if wrk then goto generate_next else renameproc:=1
    end;
  end
end renameproc;
```

Example 6: Listing of an entry tail

Monitor 42, lookup entry, will place the catalog tail in an array, which can be listed by the following procedure:

```
procedure list_tail (zout, tail);
zone zout; integer array tail;

<* the procedure lists the contents of array tail as a catalog entry

   zout   (return, zone)    zone for output
```

```
   tail   (call, integer array)  contains entry tail
*>

begin
  integer n,i;
  long array field docname;
  real r;

  docname:=2;
  n:=tail(1);
  if n>=0 then write(zout,<<z>,n) else

    write(zout,<<z>,n shift (-12) extract 12,
              <:..:>,n extract 12);

  n:=tail(2);
  if n=0 or n=1 then write(zout,n) else
    write(zout,<: :>,tail.docname);
  n:=tail(9) shift (-12);
  i:=6;
  if not (n=4 or n>=32) and tail(6)<>0 then
    begin
      write(out,<:d.:>,<<zddddd.dddd>,
        systime(6, tail(6), r)+r/1000000);
      i:= 7;
    end;

  while i<11 do
  begin
    n:=tail(i);
    if n>=0 and n<4096 then write(zout,<<z>,n) else

      write(zout,<<z>,n shift (-12) extract 12,
                <:.:>,<<z>,n extract 12);

    i:= i + 1;
  end;
end list_tail;
```

Example 7: Creating a new entry

The procedure creates a new catalog entry with scope temp or changes an alreay existing entry (with scope temp) according to the parameters. The procedure works just like the utility program set, with the extension that a working name may be returned.

```
integer procedure setproc (name, tail);
long array name; integer array tail;
<*
    setproc (return, integer) 0 ok
                              1 change kind impossible
                              2 bs device unknown
                              3 change bs device impossible
                              4 no resources
                              5 in use
```

```
                                6 name format illegal
                                7 catalog inconsistent

    name (call, long array) contains the entry name.
                            if name (1) = long <::> a working name
                            is created and name is return parameter.
    tail (call, integer array) contains the entry tail.
                                1    size or modekind
                                2:5  docname
                                6    shortclock, in case
                                     shortclock is wanted in
                                     the entry
                                7:10 remaining tail
*>

begin
  integer i;
  lon array field laf;
  zone zhelp(1,1,stderror);
  integer array ia(1:20);


  open(zhelp, 0, name ,0);
  for i:= 1 step 1 until 10 do ia(i):= tail(i);
    <*tail could possibly be a fielded array. As
      fielding once did not work in monitor the
      contents of tail is moved to the non-fielded
      array ia*>
    setproc:= i:= monitor(40<*create*>,zhelp,0,ia);

    if name(1)=long<::> then
    begin <*get wrkname*>
      getzone6(zhelp,ia);
      laf:=2;
      name(1):=ia.laf(1);
      name(2):=ia.laf(2);
    end;
    if i=3 then
    begin <*entry exists*>
      i:=monitor(42<*lookup*>,zhelp,0,ia);
      if i<>0 then
      begin
        setproc:=7; goto exit_setproc
      end;
      if tail(1)<0 or ia(1)<0 then
      begin
        if tail(1)>=0 or ia(1)>=0 then
        begin
          setproc:=1; goto exit_setproc
        end;
        goto change
      end;
      if tail(2)=0 or tail(2)=1 then goto change;
      if tail(3) extract 8=0 then tail(4):=tail(5):=0;
      if tail(2)<>ia(2) or tail(3)<>ia(3) or
         tail(4)<>ia(4) or tail(5)<>ia(5) then
      begin
```

```
      setproc:=3; goto exit_setproc
    end;

change:
    for i:= 1 step 1 until 10 do ia(i):= tail(i);
    i:=monitor(44<*change*>,zhelp,0,ia);
    if i=6 then i:=4;
    setproc:=i;
  end entry exists;

exit_setproc:
end setproc;
```

In case an array containing head_and_tail exists, it may be used as follows:

```
iaf:= 14; setproc(name,head_and_tail.iaf);
```

Example 8: Changing an entry

The procedure changes the specified entry. The procedure works just like the utility program changeentry.

```
integer procedure changeentryproc (name, tail);
long array name;
integer array tail;
<*
     changeentryproc (return, integer)
                     0 ok
                     1 change kind impossible
                     2 cat i/o error, doc. not mounted or not ready
                     3 name not found
                     4 name protected
                     5 name in use
                     6 name format illegal
                     7 catalog inconsistent
                     8 change bs device impossible
                     9 claims exceeded

    name             (call, long array)
                     contains the entry name

    tail             (call,integer array)
                      contains new entry tail

*>
    begin
      integer i;
      integer array ia(1:10);
      zone zhelp(1,1,stderror);

      open(zhelp,0,name,0);
      i:= monitor(42<*lookup*>,zhelp,0,ia);
      if i<>0 then
      begin
```

```
      changeentryproc:=i; goto exit_changeentryproc
    end;

    if tail(1)<0 or ia(1)<0 then
    begin
      if tail(1)>=0 or ia(1)>=0 then
      begin
        changeentryproc:= 1; goto exit_changeentryproc
      end;
      goto change;
    end;

    if tail(2)=0 or tail(2)=1 then goto change;
    if tail(3) extract 8=0 then tail(4):=tail(5):=0;
    if tail(2)<>ia(2) or tail(3)<>ia(3) or
       tail(4)<>ia(4) or tail(5)<>ia(5) then
    begin
      changeentryproc:=8; goto exit_changeentryproc
    end;
  change:
    <*tail could be a fielded array, cf. example 7*>
    for i:=1 step 1 until 10 do
    ia(i):= tail (i);
    i:=monitor(44<*change*>,zhelp,0,ia);
    if i=6 then i:=9;
    changeentryproc:=i;

  exit_changeentryproc:
  end changeentryproc;
```

In case the programmer only wants to change shortclock and maybe size, it may be done as follows:

```
begin
  integer i;
  zone z(1,1,stderror);
  integer array ia(1:10);


  open(z,0,<:pip:>,0);
  close (z, true);
...
...
  i:=monitor(42<*lookup*>,z,0,ia);
  if i<>0 then
  begin <* alarm, see lookupproc (ex. 9)*> end;
  ia(1):=36; <*new size*>
  ia(6):= systime (7, 0, 0.0); <*shortclock*>
  i:=monitor(44<*change*>,z,0,ia);
  if i<>0 then
  begin <* alarm, as changeentryproc, except 6=claims
           exceeded *>
  end;
...
```

Example 9: Lookup of an entry

The procedure performs a lookup of the specified name.

```
integer procedure lookupproc (scope, name, tail);
long array scope, name;
integer array tail;
<*
     lookupproc (return, integer)
                 0 found
                 1 The call param scope does not contain a legal
                   scope name
                 2 cat i/o error
                 3 not found
                 6 name format illegal

     scope       (call, long array)
                 contains the name of a scope or <::>.
                 if scope(1)=long <::> then scope will
                 be a return parameter which may be <:***:>

     name        (call, long array)
                 contains the name of the entry

     tail        (return, integer array)
                 contains tail of the entry:
                 1    size or modekind
                 2:5  docname
                 6    shortclock, in case shortclock is found in
                      the entry
                 7:10 remaining tail
*>

     begin
       integer scopeno, permkey, i;
       long    l1, l2;
       zone    zhelp(1, 1, stderror);
       integer array bases(1:8), ba(1:2), head_and_tail(1:17);
       real array field raf;

       procedure return (val);
       value val; integer val;
       begin
         lookupproc:= val;
         for i:= 1 step 1 until 10 do tail(i):= 0;
         if scopeno <> 6 then reset_base;
         goto exit_lookupproc;
      end;

      procedure reset_base;
      begin
        close(zhelp,false);
        open(zhelp,0,<::>,0);
        monitor (72<*set bases*>, zhelp, 0, bases);
      end;
```

```
lookupproc:= 0;
l1:= scope(1) shift (-8) shift 8; <*first 5 chars*>
scopeno:= 6;
for i:= 0 step 1 until 5 do
  if l1 = long (case i+1 of (
          <::>,     <:temp:>, <:login:>,
          <:user:>, <:proje:> <:syste:>))
  then scopeno:= i;
if scope = 6 then return(1); <*illegal scope specified*>

system (11<*catalog bases*>, 0, bases);
i:= if scopeno < 3 then 3 else
    if scopeno = 3 then 5 else 7;
ba(1):= bases(i); ba(2):= bases(i+1);
open(zhelp,0,<::>,0);
monitor (72<*set cat base*>, zhelp,0,ba);
close (zhelp, true);
open (zhelp,0,name,0);
i:= monitor (76<*head_and_tail*>, zhelp,0,head_and_tail);
if i<>0 then return(i);

if scopeno>0 and scopeno<5 and
  (head_and_tail(2) <> ba(1) or
   head_and_tail(3) <> ba(2))
then return(3); <*entry not found*>

permkey:= head_and_tail(1) extract 3;
if scopeno=1 and permkey<>0
   or scopeno=2 and permkey<>2
   or scopeno>2 and permkey<>3
then return(3); <*entry no found*>

if scopeno=5 then
begin
  if not (head_and_tail(2) < ba(1) or
          head_and_tail(3) > ba(2))
  then return(3); <*entry not found*>
end;

<*now the entry has been found, find the scope
  if this was not specified*>
if scopeno = 0 then
begin
  l1:= head_and_tail(2);
  l2:= head_and_tail(3);
  case permkey+1 of

  begin
<*0*> if l1 = bases(3) and
         l2 = bases(4)
     then scopeno:= 1 <*temp*>
     else scopeno:= 6;<*undef*>

<*1*> scopeno:= 6;<*undef*>

<*2*> if l1 = bases(3) and
         l2 = bases(4)
```

```
        then scopeno:= 2 <*login*>
        else scopeno:= 6;<*undef*>

   <*3*> if l1 = bases(5) and
            l2 = bases(6) then scopeno:= 3 <*user*>
         else
         if l1 = bases(7) and
            l2 = bases(8) then scopeno:= 4 <*project*>
         else
         if l1 <= extend bases(7) and
            l2 >= extend bases(8) then scopeno:= 5 <*system*>
         else scopeno:= 6;<*undef*>
       end case;

       scope(2):= 0;
       movestring (scope, 1, case scopeno of (
                  <:temp:>, <:login:>, <:user:>,
                  <:project:>,<:system:>,<:***:>));
     end scopeno = 0;

     <*assign the catalog tail to the return parameter*>
     for i:= 1 step 1 until 10 do
       tail(i):= head_and_tail(i+7);
     reset_base;

   exit_lookupproc:
 end lookupproc;
```

If the entry of smallest scope should be looked up, the previous procedure is "too much", as this can be done by the following procedure or similar statements:

```
integer procedure lookup (name, tail);
long array name; integer array tail;
begin
  zone zhelp(1,1,stderror);
  integer i;
  open(zhelp,0, name ,0);
  lookup:= i:= monitor(42<*lookup*>,zhelp,0,tail);
  if i<>0 then
    for i:=1 step 1 until 10 do tail(i):=0;
end lookup
```

where lookup will contain the value corresponding to lookupproc with the exception of the value 1.

Example 10: Wait

The procedure waits as many seconds as specified by the parameter.

```
procedure pause (time);
integer          time ;
begin
```

```
  integer array ia(1:20);
  zone          clock(1,1, stderror);

  open (clock, 2, <:clock:>, 1 shift 9);
  <*implicitly passivated if called in activity*>
  getshare6(clock, ia, 1);
  ia(4):= 0;
  ia(5):= time;
  setshare6(clock, ia, 1);
  if monitor (16<*send message*), clock, 1, ia) = 0 then
    system(9,6,<:<10>break:>); <*buffer claim exceeded*>
  monitor (18<* wait answer*>, clock, 1, ia);
end pause;
```

Example 11: See Example 4 of system.

2.98 movestring

This integer standard procedure is used for moving textstrings of various kinds.

Call:

```
movestring (ra, i, st);
```

`movestring`	(return value, integer). The number of elements in ra to which a string portion has been assigned. It is negative if the string was too big for the array.
`ra`	(return value, boolean, integer, long, real, double or complex array or zone record). The string 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.
`st`	(call value, string). Layout or text string terminated by a NULL character.

Moves either a layout or a whole textstring until a null character is encountered or the array is filled. The various string expressions are defined in (15).

Example 1:

A textstring can be inserted in a long array:

```
begin
   long array la(1:10);
   ...
   movestring (la, 1, <:some text:>);
```

2.99 multiply (*)

This delimiter, which is an arithmetic operator, yields the product of the two operands.

Syntax:

```
<operand1> * <operand2>
```

Priority: 3

Operand types:

integer, long, or real.

Result type:

```
<integer> * <integer> is of type integer
<integer> * <long>    is of type long
<integer> * <real>    is of type real
<long>    * <integer> is of type long
<long>    * <long>    is of type long
<long>    * <real>    is of type real
<real>    * <integer> is of type real
<real>    * <long>    is of type real
<real>    * <real>    is of type real
```

Semantic:

The operation may include a type conversion. Real values are represented with a relative precision of about 3'-11 (cf. (15)). This means that real variables holding an integral value is represented with full precision in the interval -2**35 < = real < = 2**35-1.
As multiplication of long values includes call of subroutines (cf. (15)), and cannot be performed by built in operations, a representation of certain long variables in real variables may be advantageous (cf. (15)).

Example:

```
i:= 5*a;
j:= i*(b+4);
r:= k * z.leaf(4) * arr(7*c)
```

2.100 newactivity

This long standard procedure initializes an empty activity with a procedure (a coroutine), and starts the activity.

Call:

```
new_activity (actno, virt, proc, params);
```

new_activity	(return value, long). The value is composed of two integers: activityno shift 24 add cause describing the way in which newactivity returns (see below).
actno	(call value, integer). The identification of the activity to be initialized. The activity must be empty.
virt	(call value, integer). Determines whether or not the designated activity will share stack area with another activity: virt = 0: The designated activity will not be virtual, i.e. it will have its own stack area. virt > 0: must be a legal activity number. The activity actno, will be virtual, i.e. share stack area with the activity numbered virt. If virt <> actno, virt must define a non-empty virtual activity.
proc	(call value, to type procedure). The activity actno is initialized with this procedure, which is called with the actual parameters stated in the parameter list params. The procedure must not be a type procedure.
params	(call values). A (possibly empty) list of parameters to be used in the call of proc.

Storage resources for variables etc. are allocated in the stack, the top of which is defined by the runtime system variable, last_used.

The stack administration is for activity purposes supplied with another runtime system variable, max_last_used, defining the upper limit for last used. Stack overflow is detected by comparison of these two variables.

To each activity is allocated a maximum contigous stack area, defined by the pair: (initial_last_used, max_last_used) for the activity. The very first call of new_activity for each activity, determines the values of the corresponding initial_last_used, max_last_used). When an activity is entered (by activate), the runtime system uses the stack allocated to the activity. When an activity returns to the monitor block (by passivate), the runtime system uses the stack allocated to the monitor block.

Buffer areas and share descriptors belonging to zones declared in activities will be allocated in the stack area for the activity, and never in the high end partition of memory.

Function:

new_activity must be called at the monitor block level (cf. procedure activity).

The execution of:

result := new_activity (actno, virt, proc, params);

now proceeds as follows:

0) If the procedure is a type procedure (function), new_activity terminates with an alarm.

1) If the block level of the call is not the monitor block level, new_activity terminates with an alarm.

2) If the program is not in monitor mode, the run is terminated with an alarm.

3) If actno < 1 or actno > max_no_of_activities or virt < 0 or virt > max_no_of_activities, the run is terminated with an alarm.

4) If the activity actno, is not empty, new_activity returns with the result = actno shift 24 add (-1).

5) If the stack area is not allocated, the actual value of last_used (monitor last used) defines the stack and initial_last_used for the activity.

6) If the activity is to be virtual, and a shared virtual occupies the stack area, the stack area is transferred to the virtual storage file in one output operation. If, however, the shared virtual waits for an i/o operation to complete, newactivity returns with the result:
shared_virtual shift 24 add (-2).

7) The procedure, proc, is called with params as actual parameters, and the program is now in activity mode. It is not checked, that this parameter list corresponds to the formal specifications in the procedure declaration. The procedure returns:
   - executing a passivate statement or
   - leaving through its final end or
   - because of a runtime alarm.

   defining the value of result. At return, the program is back in monitor mode.

8) If a stack area was not allocated,, when newactivity was called, the following is done at return from newactivity: The maximum last_used appearing during the execution in activity mode is selected to represent max_last_used of the activity, and at the same time last_used of the monitor block.

9) If the activity is to be virtual, max_last_used of the activity is increased to extend the stack area to the nearest multiple of 512 halfwords and a record in the virtual storage file is reserved to hold the stack area of the activity.
The transfers between stack and virtual storage file (swops) are initiated by newactivity or activate, when an activity, not at present occupying the stack area, is to be activated (restarted).
The virtual storage file is the same one as used for context blocks.

If the designated activity is empty, but has been initialized before in a previous call of new_activity, the stack area is a priori fixed by limits defined in the very first call of new_activity for the designated activity. A non-empty activity becomes empty, if it:

- leaves throuth its final end or
- terminates with a runtime alarm.

Result values:

The return values activityno shift 24 add cause have the following meaning:

activity_no: The identification of the activity causing the return. In case of cause = -2, however, the activity waiting for an i/o operation to complete.

cause indicate the return cause:

cause = -3: new_activity returned because of an alarm in a started activity, which is now empty.

cause = -2: new_activity returns, because the designated activity is virtual and shares storage with another activity waiting for an i/o operation to complete (see the parameter virt).

cause = -1: new_activity returns because the designated activity is not empty.

cause = 0: new_activity returns because a started activity terminated via its final end. The activity is now empty.

cause = 1: new_activity returns because a started activity is passivated by a programmed passivate statement.

cause = 2: new_activity returns because a started activity is passivated by an implicit passivate statement in the zone i/o system.

Example 1:

If a minimum space is needed in the stack for an activity (for later procedure calls, arrays in inner blocks etc.) this may be assured by

calling the following procedure from an activity, during its very first execution (cf. 8 in the description above) and before its first passivation;

```
procedure claim (n); value n; integer n;
begin array a (1:n); end;
```

Example 1 of activity shows the use of the procedure.

Example 2:

```
begin integer i, j; <*monitor block*>

procedure p (n, m);
integer n, m;
begin integer s, t;
...<*activity mode*>
   s:=2; t:=3;
   passivate;
   ...
end;

procedure q(x, y, z);
value x, y, z; real x, y, z;
begin integer k, l;
...<*activity mode*>
   k:=4; l:=5;
   passivate;
   ...
end;

<*neutral mode*>
activity (5);
<*monitor mode*>
for i:=1, 2 do
new_activity (i, 0, p, i, j);

for i:= 3, 4, 5, do
new_activity (i, 3, q, 0, 0, 0);
...
```

In this example, 5 activities are created and initialized. Activity no. 1 and 2 are both started with the procedure p. The virtual activities no. 3, 4, and 5 are all started with the procedure q. In the figure above is shown the contents of stack and virtual storage after initialization of activity no. 5.

Note that activity no. 1 and 2 via name parameter work on common variables. Note also, that the virtual record of activity no. 5 is empty, because the stack is still in memory.

2.101 newsort

This standard procedure creates a zone record in memory which is to take part in a sorting process. The contents of the record is initially undefined, but the user is supposed to assign values to the record variables before next call of any sorting procedure. The so defined record becomes an active record in the sorting process (see procedure outsort).

Call:

```
newsort (z, key) or newsort (z)

z       (call and return value, zone). The name of the
        record created.
key     (call value, integer procedure or empty). The
        name of a procedure for comparison of two
        records (see below).
        The key parameter is omitted if standard
        sequencing is used (see procedure startsort6).
```

Zone state:

The zone state must be 9, in sort, i.e. startsort6 (or initsort) must have been called. The state is not changed by the procedure.

Sequencing:

The integer procedure key is supplied by the user, when a non-standard sequencing is wanted. It determines which of two records is to appear first in a sorted string of records:

Call:

```
key (record1, record2)

record1, record2,    (call values, arrays). The name of
                     two records to be compared.
key                  (return value, integer). The result
                     of comparison of record1 and
                     record2. This return value must
                     adhere to the following rules.
                     record1 before record2: key < 0
                     record1 after  record2 : key > 0
                     sequence unimportant : key = 0
```

Example 1:

This is an example of a key procedure which can be used to sort a number of records of length 10 double words on ascending number of identical double words within a single record.

```
integer procdure mostequals(v, w);
real array                  v, w ;
begin
   integer procedure equals(x);
   real array               x ;
   begin
     integer count, i, j, k;
     real z;
     count:= 0;
     for i:= 1, i+1 while i<= 10 and count < i do
     begin
        z:= x(i); k:= 1;
        for j:= i + 1 step 1 until 10 do
           if z = x(j) then k:= k + 1;
        if k > count then count := k;
     end;
     equals:= count
  end equals;

  mostequals:= equals(v) - equals(w)
end mostequals;
```

Note that the structure of a record (for instance the record length) is supposed to be known to the procedure as global information (for instance by certain values of certain fields in the record).

Example 2:

100 records of length 10 double words per record are stored in array B(1:100,1:10). These records are to be sorted on ascending number of equal double words within a single record. Therefore, the key procedure of Example 1 is used.

```
begin
   zone z(1199,1,sorterror);
   <*buffer length as required by initsort*>
   integer i, j;
   initsort(z, 10);
   for i:= 1 step 1 until 100 do
   begin
      newsort(z, mostequals);
      for j:= 1 step 1 until 10 do z(j):= B(i,j)
   end;
   for i:= 1 step 1 until 100 do
   begin
      outsort(z, mostequals);
      for j:= 1 step 1 until 10 do B(i, j):= z(j)
   end
end;
```

Example 3:

Same as Example 2, but now the sorting is on ascending double word one followed by descending double word five (standard sequencing).

```
begin
   zone z(1022,1,sorterror);
   <*buffer length as required by startsort6*>
   integer i, j;
   integer array keys(1:2, 1:2);
   keys(1,1):= 4; keys(1,2):= 4;
   keys(2,1):=-4; keys(2,2):=20;
      <* definition of sequencing, see startsort6*>
   startsort6(z, keys, 2, 40);
   for i:= 1 step 1 until 100 do
   begin
      newsort(z);
      for j:= 1 step 1 until 10 do z(j):= B(i, j)
   end;
   for i:= 1 step 1 until 100 do
   begin
      outsort(z);
      for j:= 1 step 1 until 10 do B(i,j):= z(j)
   end
end;
```

2.102 not (-,)

This delimiter, which is a monadic logical operator, yields the logical negation of the operand.

Syntax:

```
not <operand> or -, <operand>
```

Priority: 6

Operand type:

boolean

Result type:

boolean

Semantic:

The negation of a boolean gives the result false if the boolean is true and the result true if the boolean is false.

Example

```
1: not b
not (a=2)
not (not p)   is equivalent to p
not (p and q) is equivalent to not p or not q
not (p or q)  is equivalent to not p and not q
not p or q    is equivalent to p => q
```

2.103 not equal (< >)

This delimiter, which is a relational operator, gives the value true or false.

Syntax:

<operand1> <> <operand2>

Priority: 5

Operand types:

integer, long, or real.

Result type:

always boolean.

Semantic:

The relation takes on the value true whenever the bitpattern of the two operands are equal, otherwise false.

The relation is performed as a bit by bit comparison of the two operands (after a possible type conversion whenever the operands are of different types).

Example 1:

```
c:= d <> q;
if m <> idle then ... else ...
while empty <> full do ...
```

2.104 open

This standard procedure 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 or any type array i.e.
          boolean, integer, real, long, double real,
          complex). A text string or an array containing
          a text, 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.
```

Doc:

The call value, doc, may be either
- a string variable or a string expression
- a long variable or a long expression
- any type array (boolean, integer, long, real, zone record).

In case of an array, it is supposed to contain text, packed as text portions, i.e. 6 characters to a double word, 3 to a word.
In case of a string or a long variable or expression, it is supposed to contain either a string point or a text portion.
The name is inserted as the process name of the zone.

Note: the name will not cleared by a call of close, only by another call of open or setzone (set zone6).

Modekind:

Specifies the kind of the document (terminal, 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 (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 initialized as required by printer.
If kind is not one of the numbers 0,2,4,...,20 an alarm occurs.
Mode may be anything as far as the procedure open is concerned.
Below table gives a brief overview of the systems present interpretation of mode and kind combinations. For further details, you should consult the manual belonging to the proper external process.

Kind: Name of process and interpretation of mode:

0 **internal process**
any mode

2 **interval clock process**
mode = 0: time interval is specified in seconds

mode = 2: time interval is specified in 0,1 millisecs.
mode = 4: time interval in seconds specifies a clock value
mode = 6: time interval in 0.1 millisecs. specifies a clock value

4 **backing storage area**

IDA801 RC834X Storage Modules:
output mode:
mode = 1 : read after write to correct correctable errors

input mode:
none

Other Disc Storage Modules:
mode = 0, specifies full error recovery.
mode = 2, specifies suppression of automatic error recovery. Will override mode = 4.
mode = 4, specifies limited error recovery on parity errors occuring at an input operation.

6 **disc process**

IDA801 - RC834X Disc Storage Modules: output mode:
mode = 1: read after write to correct correctable errors

input mode:
none

Other Disc Storage Modules:
mode = 0: i/o operations concern the data part of segments, full error recovery.
mode = 1: i/o operation concern the address part of segments, full err or recovery.
mode = 2: suppression of automatic error recovery, override mode =4
mode = 3: mode 1 + mode 2

mode = 4: limited error recovery on parity error at input
mode = 5: mode 1 + mode 4

8 **terminals**
CSP terminals, see ref. [25], other terminals: input modes:
mode = 0, intended for conversational operation by means of keyboard cf. (3)
mode = 2, paper tape input from teleterminal, ISO characters in even parity.
mode = 4, paper tape input from teleterminal, no parity.
mode = 8, as mode = 0, except the input is invisible i.e. the characters are rubbed out on the screen.

output modes:
mode = 0, normal text mode i.e. 7 bit ISO characters, with or without parity..
mode = 2, transparent text mode i.e. 7 bit ISO characters, even parity.
mode = 4, transparent output mode i.e. 8 bit characters, no parity.

10 **paper tape reader**
mode = 0, odd parity.
mode = 2, even parity.
mode = 4, no parity.
mode = 6, flexowriter to ISO conversion.

12 **paper tape punch**
mode = 0, odd parity
mode = 2, even parity
mode = 4, no parity
mode = 6, ISO to flexowriter conversion.
mode = 8, even parity (<10> implies <13> <10> <127>).

14 **line printer**
CSP printers, see ref. [25], other printers:
mode = 0, the answer is returned when the transmitted block has been written.
mode = 2, the answer is returned when the transmitted block has been received.
mode = 4, the block is supposed to contain formatted data, cf. (3).

16 **card reader**
mode = 0, punched binary.
mode = 10, punched decimal with conversion.
mode = 64, mark sense binary.
mode = 74, mark sense decimal with conversion.
mode = 256, basic cards.

See [3] for further information.

18 **magnetic tape**
modekind is:
mode shift 12 + 18

where

mode =
bgl shift 9 + speed shift 7 + trail shift 4 + density shift 2 + parity shift 1

```
bgl     = block gap length, values 0, 1      , 0 : std, 1 : long
speed                     , -      0, 1      , 0 : low, 1 : high
trail     cf. ref. [13]   , -      0, ..., 7
density   and recording   , -      RC9000-10 : 0, ..., 3, cf. below
                            -      RC8000    : 0, 1       cf. below
parity                      -      0, 1      , 0 : odd, 1 : even
```

RC9255 9 Track Streamer Unit

```
block gap length          : unused
speed                     : unused, automatic selection
                          between 25/75 ips
trail                     : only output, specifies that the
                          last n (<7) characters from the
                          output area are not to be output
density                   : 0 : 6250 bpi GCR
                            2 : 3200 bpi DDPE
parity                    : unused, always odd
```

RC9250 9 Track Streamer Unit

```
block gap length          : unused
speed                     : unused, automatic
                            selection between 50/100 ips
trail                     : as for RC9255
density                   : 0 : 6250 bpi GCR
                            1 : 1600 bpi PE
                            2 : 3200 bpi DDPE
parity                    : unused, always odd
```

RC8344 9 Track Streamer Unit

```
block gap length          : the values of std and long depend on
                            the speed, the density and streaming or
                            start/stop
speed                     : low  : 25 ips
                            high : 75 ips
trail                     : as for RC9255
density                   : 0 : 6250 bpi GCR
                            1 : 1600 bpi PE
parity                    : unused, always odd
```

RC8343 9 Track Streamer Unit

```
block gap length          : the values of std and long depend on
                            the speed, the density and streaming or
                            start/stop
```

```
speed                   : low  : 12.5 ips low dens. /
                                 25 ips high dens.
                          high : 100 ips low dens. /
                                 50 ips high dens.
trail                   : as for RC9255
density                 : 0 : 3200 bpi DDPE
                          1 : 1600 bpi PE
parity                  : unused, always odd
```

RC3715 9 Track Tape Unit

```
block gap length        : unused
speed                   : unused, always 45 ips
trail                   : as for RC9255 when controlled by IDA801,
                          else n < 5, the last n characters from
                          the last double word are not output
density                 : 0 : 1600 bpi PE
                          1 :  800 bpi NRZ
parity                  : unused, always odd
```

RC Streaming Casette Tape

mode 0 simulated MT
mode 1 fixed blocklength of 512 characters

For further information see [3].

RC3625/26 Casette Tape Unit
mode =
SLEN shift 20 + ECMA shift 19 + T shift 16 + dens shift 2
parity shift 1

ECMA = 1 for ECMA version 1 (read only),
= 0 otherwise.
SLEW = 0, normal
= 1, slew mode

T see description of 9-track tape.

dens = 1, = 4, 800 bpi NRZ
parity = 0, odd

Common for kind 18

If you use T <> 0 during output you should set the word defect bit (1 shift 7) and the stopped bit (1 shift 8) in your give up mask and after a check of halfwords transferred simply ignore the bit in your block procedure.

Initialization of a document

Open prepares the later use of the document according to kind:

Internal process, interval clock process, backing storage area, disc process, terminal:

Nothing is done. When a transfer is checked later, the necessary initialisation is performed.

Paper tape, 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 requesting the tape to be mounted. The message is sent without wait indication (see (7)).

Some of these rules have been introduced to remedy a possible absence of an advanced operating system, like BOSS.

Giveup:

The parameter giveup is a mask of 24 bits which will be compared to the logical status word (15) 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 is skipped and the block procedure is called instead (the block procedure is also called if a hard error is detected during the checking).
The bit 1 shift 9 in giveup has a special meaning:
If the zone is opened with this bit set to one, and the program is in activity mode, then an implicit passivate statement is executed in the run time systems check routine and in monitor (18 ...) making concurrent i/o transfers possible in a program containing corotines. For further informa tion cf. (19).

The bit 1 shift 10 in give up has a special meaning, too: If the zone is opened with the bit set to one, and the zone is in any other state than 4 (after declaration, after close) or 9 (in sorting) when the block declaring the zone is left again, then the block procedure is called.

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 halfwords. If this cannot be done without using a share length of 0, the zone state becomes "after open with buflength error" (64) or

"open on magtape with buflength error" (64+8).
If any input/output procedure is called with the zone in that state, a run time alarm, zone state, terminates the program.
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(4*21*512*2,2,stderror);
   open(z,18,<:mtdp0001:>,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 tape. (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,<:mtdp1706:>,0);
open(z2,18,<:mtdp1712:>,0);
setposition(z1,1,0); setposition(z2,1,0);
```

If none of the tapes are mounted, the operating system may get the messages:

```
message <proc> mount mtdp1706
(caused by open(z1,...))
message <proc> mount mtdp1707
(caused by open(z2,...))
pause <proc> mount mtdp1706
(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 fileno>
```

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;

      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,...),
             text, giveup);
      if j > 15 then <*magnetic tape*>
      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 dataerror
   else
   begin
     inrec6(master,m1); inrec6(trans,t1);...
```

2.105 openinout

This standard procedure changes the buffer size and the sizes and locations of the shares of the zones in a zone array, making it ready for record input/output by the procedure inoutrec.

Call:

```
openinout (za, inputzone)
```

za	(call and return value, zone array). The buffersize and the sizes and locations of the shares in the zone array are changed for all the zones in the zone array, making it ready for record in/output by the procedure inoutrec. The number of zones in the array must exceed 1, of course, and the number of shares in each zone must be the same for all zones.
inputzone	(call value, integer, long or real). The zone za (inputzone) is selected as input zone, leaving all the others as outputzones. The value of input zone must be in the interval 1,2,...no of zones.

Zone state:

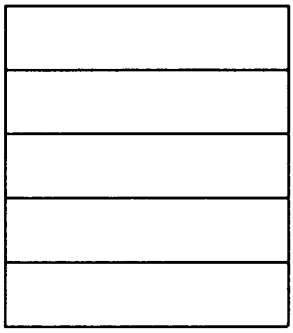
The zone states of all zones in the array must be 0 or 8, opened or opened on magtape, or 64+0, 64+8, meaning opened with buflength error or opened on magtape with buflength error, cf. getzone6. The zone state of each zone will become 32, or 32+8 if magtape and not positioned, i.e. ready for inoutrec, maybe after setposition. In each zone, then, only setposition may have been called since the lastest call of open. For magnetic tape, no positionoperation may still be pending (cf. setposition) or a share state alarm oc curs. A possible pending position operation may be claimed by the call check (za(index)).

The entire buffer area of the zone array is collected as one contigous storage space, locating all share descriptors in the top of the area. The entire buffer area is divided in no of shares * 2+1 buffer areas of length equal to the zone buffer area of the first zone in the array, the desired blocklength, (cf. buflengthio). If the document connected to any zone is a backing storage area or a disk, the length is made a multiple of 512 halfwords. If this cannot be done without using a buffer area length of zero, the buflength error bit, 64, is added to the zone state of all zones.

Blocking:

The entire buffer area of all the zones in the array is collected as one contigous storage space, moving all share descriptors to the top of the area.
The entire buffer area is divided in no of shares x 2+1 buffer areas of

the desired blocklength, cf. above.
The buffer area described in each zone becomes the entire buffer area, while the record description is unchanged (no record) and the logical position of the document stay unchanged.
the first share of all zones becomes used share and will describe the first buffer area.
The next shares in the input zone will describe the next buffer areas in sequence, and common to all the output zones the next shares will describe the buffer areas succeeding the input buffer areas, also in sequence.
Assume that each zone has 3 shares, one input zone, two output zones:



```
input zone: outputzone 1, 2

share(1)    share(1) share(1)

share(2)

share(3)

                share(2) share(2)

                share(3) share(3)
```

Free param, partial word:

The procedure, together with inoutrec/changerecio, expellinout and closeinout, makes special use of the fields "free param" and "partial word" in each zone descriptor of the zone array. The procedure openinout will set "free param" of all zones to point to the input zone, and "partial word" in each zone will contain its own index in the zone array. The procedure expellinout will change the value in partial word to the address of the zone itself, and the procedure closeinout will reinitialize "free param" and "partial word" in all zones with the values 0 and 1, resp.

Example 1:

See Example 1 of inoutrec.

2.106 opentrans

This standard procedure outputs a transaction head of a format 8000 transaction, and initiates a zone for character writing. Opentrans may be regarded as the reverse operation of waittrans. Note, that no waiting takes place.

Call:

```
opentrans (z, format, destination, aux1, aux2)
```

z	(call and return value, zone). Specifies the document to which transactions are transferred.
format	(call value, integer). Defines the format of the transaction to be sent: 1: read modified format 2: short read format 3: write format 4: read buffer format 5: read status format 6: connect format
destination	(call value, integer). Designates the receiver of the transaction, i.e. display terminal, computer, or RC8000 application.
aux1	(call value, integer). Depending on the format of the transaction, aux1 specifies the attention type, write command code, or is not used.
aux2	(call value, integer). Depending on the format of the transaction, aux2 specifies the cursorposition, write control character, or is not used.

Note that in communication with display terminals, only format 3 (write format) makes sense, while the other formats may be sent to computers or other RC8000 applications.

For further details about format, aux1, and aux2 see waittrans.

Zone state:

The zone must be open and ready for opentrans (state 0 or 13), i.e. since the latest call of open, setposition, or closetrans. The state becomes 3, i.e. the zone is ready for character output by means of the procedures writefield, write, outtext, etc.

See Example 1 of inoutrec.

2.107 openvirtual

This standard procedure closes the present virtual storage connected the program, writing back present values of owns, common blocks (in FORTRAN programs), context blocks and stack pictures of virtual activities before it connects a backing storage area to the calling program as a virtual storage according to the description in (15).

Call:

```
openvirtual (filename)

filename  (call value, string). A text string specifying
          the name of the backing storage area to be used
          as virtual storage.
          The program file itself is used as virtual
          storage in case of the empty string.
```

The procedure must not be called within a context block. For further information, [15].

Note 1:

The structure of own variables and context blocks in the calling program must be exactly the same as in the program file specified by filename. Remember in this connection also the own variables included into the program by external procedures (see Example 1). The following algol standard procedures and those with the following names as document names are using own variables:

activity
closetrans
fpproc
read
write

Note 2:

If in an ALGOL or FORTRAN program containing DATA statements or zone common blocks in the main program or in any subroutine, openvirtual is called at a time when the present virtual storage is the program file itself, it must be retranslated before another startup.
Explanation: At program startup, DATA and ZONE initialization of COMMON variables is activated as code allocated in the COMMON block area of the program file. If openvirtual has written back the values from the common blocks, this code is destroyed. In this case the procedure 'virtual' should be used.

Example 1:

When two programs are using the same virtual storage, the following procedure could be declared in the outermost block of each program, to avoid confusion in the structure of own variables. Notice, that only one of the procedures sharing the same document name, and thus the same owns, need be included in the list.

```
procedure insert_owns;
if false then
begin integer i;
   <*the following list must contain a call of all external procedures using
      own variables. The procedure is never called*>.
   read(in,i);
   write(out,i);
   ...
end insert_owns;
```

2.108 or (and !)

This delimiter, which is a logical operator, yields the logical or (logical sum) of the two operands.

Syntax:

```
                (or)
<operand1>      (   ) <operand2>
                (!  )
```

Priority: 8

Operand types:

boolean

Result type:

boolean

Semantic:

The logical sum of the two operands is evaluated. The logical sum is performed bit by bit in parallel on the twelve bits of the two operands. The truth value of the result pattern, when used in conditional or repetitive statements, is determined by the last (rightmost) bit in the pattern (0=false, 1=true).

Example 1:

Input checking of number range:

```
if a < 10 or a > 100 then error;
```

Example 2:

```
repeat
......
until found or i>10;
```

2.109 out

This standard identifier 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 1:

An FP source file containing

```
p = algol          ;
begin write(out,12,<:a:>) end
o f47              ; select f47 as current output,
                   ; see ref. 6.
p                  ; execute
p                  ; execute
o c                ; terminate the use of f47
```

will generate the following text in the file f47:

```
12a
end                7
12a
end                7
```

2.110 outchar

This standard procedure 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.

Blocking:

As for write.

Example 1:

```
outchar (z, 12 <*ff*>);
outchar (z, 'nl');
outchar (z, a); <*the character with the ISO
value of the variable a*>
outchar (z, 'a'); <*the character a*>
```

Example 2:

See Example 1 of readchar.

2.111 outdate

This standard procedure prints a date on a document.

Call:

```
outdate (z, i)
```

```
z       (call and return value, zone). Specifies the
        document, the buffering, and the position of the
        document.
i       (call value, integer). The value of the integer
        specifies day, month, and year according to the
        formula day*100 00 + month*100 + year.
```

The procedure prints the date in the following fixed format

```
<day>.<month>.<year>
```

where the three fields consist each of 2 digits. No check is performed to assure that the date makes sense.

The current value of date is easily determined by the standard procedure systime.

Zone state:

As for write.

Example 1:

The procedure can, of cause, also print an ISO date in the format

```
<year>.<month>.<day>
```

e.g., the call

```
outdate (z, round systime (5, 0, 0.0));
```

will print the current date.

2.112 outindex

This integer standard identifier is used by all character printing procedures, when a non-standard alphabet is selected. See outtable.

The default value of outindex is 0.

2.113 outinteger

This standard 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 especially designed to print amounts of currency.

Call:

```
outinteger (z, pos, dec, amount)

z          (call and return value, zone). Specifies the
           document, the buffering, and the position of
           the document.
pos        (call value, integer). The absolute value of
           pos specifies the total number of character
           positions to be printed. pos should be inside
           the range:
           abs (pos) <= 132.
dec        (call value, integer). Specifies the number of
           digits after the decimal point. dec should be
           inside the range:
           0 <= dec <= min (abs (pos)-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 prameter pos. If pos is nega tive and amount = 0 then a number of spaces equal to the absolute value of pos is printed. If pos 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 decimal point 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.

Blocking:

As for write.

Example 1:

The program

```
begin
   long a;
   for a:= 123456789,
               1111100,
               0,
               12345678901234 do
   begin
      outchar (out, ':');
      outinteger (out, -12, 2, a);
      outchar (out, ':');
      outchar (out, 'nl');
   end;
end
```

will print the following text on current output:

```
:  1234567.89:
:    11111.00:
:           :
:123456789012.34:
```

2.114 outrec

This integer standard procedure 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 halfwords each, available for further calls of outrec before change of block takes places.
z	(call and return value, zone). The name of a record. Determines further the document, the buffering, and the position of the document (15).
length	(call value, integer, long or real). The number of elements, of 4 halfwords each in the new record. Length must be >= 0.

For further description see outrec6.

Outrec, instead of outrec6, may be used with advantage when the document is considered to contain reals.

Example 1: Storing a matrix on backing storage

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.

2.115 outrec6

This integer standard procedure 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 halfwords of the document.

Call:

```
outrec6 (z, length)
```

outrec6	(return value, integer). The number of halfwords 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 (15).
length	(call value, integer, long or real). The number of halfwords in the new record. Length must be >=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 getzone), i.e. the zone must 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 disc process. a backing storage area, a terminal, 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 halfwords just after the current logical position of the document and moving the logical position to just after the last halfword 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 at tempted, the block is changed in this way:

1. Documents with fixed block length (backing storage): The remaining halfwords of the share are filled with binary zeroes, and the total share is output . s 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 (15). The record becomes the first halfwords of the next share, but if the record still is too long, an alarm occurs.
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,...);
repeat
   ..... ;<*compute length*>
   outrec6(z,length);
   z.first_word:= length;
until.....;
close(z,true);
```

Compare this with Example 1 of changerec6. The version here may be a little bit faster.

2.116 outsort

This standard procedure makes available a zone record which is the winner of the active records in the zone. The user is supposed to move this record away from the zone before next call of any sorting procedure.

Call:

```
outsort (z, key)
or
outsort(z)
```

```
z       (call and return value, zone). The name of the
        selected record.
key     (call value, integer procedure or empty). The
        name of a procedure for comparison of two
        records (see procedure newsort).
```

The key parameter is omitted if standard sequencing is used (see procedure startsort6).

Record selection:

The zone record made available by outsort is selected among the active records in the zone z.
The record selected is the one which, according to the sequencing mechanism (procedure key or standard sequencing), is the first record in a string of sorted active records in the zone (the winner).
The status of the selected record is changed from active to nonexisting.

Zone state:

As for newsort.

Example 1:

See Example 1 of newsort for a user specified key procedure.

Example 2:

See Example 2 of newsort.

Example 3:

See Example 3 of newsort.

Example 4:

Merging of k sorted input files into one output file. The input files are stored on k magnetic tapes. The record length is one double word, and end of file is signalled by reading end of file mark. Input is read via zone array x (k, buflength, shares, endpr), and output is written via zone y(buflength, shares, endpr). Sequencing is on ascending longs.

```
begin
  zone z((k+1)*((6+2)//4)+3+6, 1, sorterror);
       <*sorterror: see startsort6*>
  zone array x(k, buflength, shares, endpr);
  zone       y(buflength, shares, endpr);
  long field lf; integer field no;
  integer n, i;
  integer array key(1:1, 1:2);

  procedure endpr(z, s, b); zone z; integer s, b;
  begin
    if s shift (-16) extract 1 = 1 then
    begin
      n:= n - 1; goto loop
    end
    else stderror(z, s, b);
  end;

  key(1,1):= 3; key(1,2):= 4;
  startsort6 (z, key, 1, 6);
  for i:= 1 step 1 until k do
    open(x(i), 18, string tape(i), 1 shift 16);
  open(y, 18, string tape(k+1), 0);
  for i:= 1 step 1 until k do
    setposition (x(i), 1, 0);
  setposition (y, 1, 0);
  ln:= 4; no:= 6;
  n:= k;

  <*read the first record from each input file.
    The block procedure assumes at least one record in each file*>
  for i:= 1 step until k do
  begin
    inrec6(x(i), 4); newsort(z);
    z.lf:= x(i).lf;
    z.no:= i;
  end;

loop:
  if n <> 0 then
  begin
    outsort(z); outrec6(y,4);
    y.lf:= z.lf;
    i:= z.no;
    inrec6(x(i),4); newsort(z);
    z.lf:= x(i).lf;
    z.no:= i;
    goto loop;
  end;
```

```
stop:
close (y,true);
for i:= 1 step 1 until k do close (x(i),true);
end;
```

2.117 outtable

This standard procedure exchanges the current output alphabet used by all the write procedures on character level (write, outchar, outtext etc).

Call:

```
outtable (alpha)
or
outtable (0)

alpha    (call value, integer array of one dimension).
         Contains the character value of each character
         in the new output alphabet, as described below.
0        (call value, integer). A zero signals that the
         standard alphabet is to be used.
```

outtable (alpha):
The actual contents of alpha are used in all calls of character printing procedures, until another array or the standard alphabet is selected. This means that any change in the contents of alpha may have effects on the character printing. If a charater printing procedure is called at a place where alpha is undeclared, an undefined alphabet is used.

To each character 'c' in the interval lower index to upper index of alpha the output character value is defined in this way:

```
converted_char := alpha (c + outindex) extract 12
character_class:= alpha (c + outindex) shift (-12)
```

If the character_class equals 1, i.e the converted character is a 'shift character' the value of converted_char is assigned to outindex and the converted character is looked up again in the alphabet to determine class and value.

If converted_char > 255 or c + outindex is not a legal index in alpha the character will not appear in the output. The standard integer outindex is normally 0, but you may use it to modify the alphabet.

The character conversion defined by outtable is performed after a possible conversion of special characters on account of replace char.

The array may be initialized with the ISO alphabet by a call of isotable.

2.118 outtext

This standard procedure 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). The absolute value of pos
        specifies the total number of character
        positions to be printed. Pos should be inside
        the range:
        abs (pos) <= 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 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, ending spaces (see replacechar) 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.

Blocking:

As for write.

Example 1:

The statements

```
movestring (ra, 1, <:this is a text:>);
outchar (out, '*');
outtext (out, 20, ra, 1);
outchar (out, '*');
```

will print the following line on current output:

*this is a text *

2.119 outvar

This integer standard procedure is intended for output of variable length records so that they may be read by means of invar. Outvar makes an output record ready and fills it from an array of double word type (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 halfwords
         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 (15)
a        (call value, real, long, double real or complex
         array or zone record). An array to be copied
         into the zone record. The first word of the
         element with lexicographical index 1 contains
         the number of halfwords 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 must only have been used by outvar or the like since the latest call of open or setposition. The free parameter (see getzone6) 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 halfwords just after the current logical position of the document and moving the logical position to just after the transferred elements. The full 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 storage unused parts are filled with binary nulls, and on all other documents 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 halfwords in the first word an 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.

2.120 overflows

This integer standard identifier determines the action on floating point overflow:

overflows < 0	The execution 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 execution starts, overflow is -1. A floating point over flow occurs when a real operation gives a result outside the range of real variables.

Example 1:

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;
```

2.121 own

This delimiter, which is a declarator, is used to declare own variables of any type.

Syntax:

```
own long    <namelist>; declares own long    variables
own real    <namelist>; declares own real    variables
own integer <namelist>; declares own integer variables
own boolean <namelist>; declares own boolean variables
```

Semantic:

Own variables have the same scope as other variables declared in the same block. Upon reentry into the block, they will, however, have values which are unchanged from their value at exit from the block. Upon the first entry into a block, the own variables of that block will have the binary pattern zero, meaning that own booleans are initially false, own integers and own longs are initially 0, and own reals contain a pattern equivalent to real <::>.

If a variable is declared own in the body of a recursively called procedure or in an activity procedure, the same location is used in all dynamic incarnations of that procedure.

Note:

In a program using the context procedure openvirtual the number of own variables in the program and in a non empty data file connected to the program by openvirtual/virtual must be exactly the same.

Example 1:

If you want to abort a program after 100 calls of an error procedure it could be done in this way:

```
procedure error (...);
  ...
  begin
    own integer no_of_err;
    no_of_err:= no_of_err + 1;
    if no_of_err > 100 then
    <*abort action*>
    else
    ...
```

2.122 passivate

This standard procedure deactivates the executing activity, establishing its restart point (waiting point).

Call:

```
passivate
```

If the program is not in activity mode, the run is terminated with an alarm. The restart point is defined as the statement following the passivate statement. Passivate returns to the call of activate (or newactivity), which entered the current activity, defining the cause = 1 (return values of activate and newactivity). The program is now back in monitor mode.

Note:

An implicit passivate statement is found in the standard i/o system in check and monitor (18,...) (cf. description of entry 18 of monitor) This implicit passivate statement supplies the return value cause = 2 (for activate and newactivity). Note also that activity termination caused by runtime alarms and by execution of the final end of an activity, are regarded as implicit passivate statements, defining return causes -3 and 0 of the corresponding newactivity and activate statements. The passivate statement will always return to the corresponding newactivity/activate statement in monitor mode.

Example 1:

See Example 1 of activity.

2.123 plus (+)

This delimiter which is an arithmetic operator can be used both as a dyadic and a monadic operator.

1. Dyadic:

Syntax:

```
<operand1> + <operand2>
```

Priority: 4

Operand types:

integer, long, or real.

Result type:

```
<integer> + <integer> is of type integer
<integer> + <long>    is of type long
<integer> + <real>    is of type real
<long>    + <integer> is of type long
<long>    + <long>    is of type long
<long>    + <real>    is of type real
<real>    + <integer> is of type real
<real>    + <long>    is of type real
<real>    + <real>    is of type real
```

Semantic:

This operator yields the normal arithmetic sum of the expressions involved.

2. Monadic:

Syntax:

```
+ <operand>
```

Priority: 4

Operand types:

integer, long, or real.

Result type:

+ <integer> is of type integer
+ <long> is of type long
+ <real> is of type real

Semantic:

This monatic operator yields the value of the operand.

Example 1:

```
18+9
+ 18+c
c+b
```

2.124 pos

This integer standard procedure searches among characters in an array for a given substring and returns its first character position, or zero if the substring is not found.

Call:

```
pos (sub, arr)
or
pos (sub, arr, startpos)
```

pos	(return value, integer). If the character substring given in sub is found in arr, pos is the position of its first character in its first appearance within arr. If it is not found, pos is zero.
sub	(call value, array of any type or zone record). The character sequence from character position no. 1 to the last non zero character or to the last character position in sub is searched in arr.
arr	(call value, array of any type or zone record). The characters in arr from character position no. 1, or from the character position given in startpos, until the last non zero character position or until the last character position are searched for the substring given in sub. Concerning boolean arrays, cf, 2.xx, len.
startpos	(call value, integer). The parameter is optional. If given, the search in arr starts in character position no. startpos.

2.125 priority

Aritmetic and boolean expressions are in principle evaluated from left to right with addition of the rules of priority given in the sections 3.1.1. and 3.4.1. (15).
The priorities of the operators listed from high to low priority are

```
1:  abs entier extend long real round string
2:  ** add extract shift
3:  *   /    //      mod
4:  +   -
5:  <   <=   =   >=     >  <>
6:  not  -,
7:  and &
8:  or !
9:  =>
10: :=
```

An expression included in parentheses is evaluated by itself before its value is used in subsequent calculations.

Example 1:

The two boolean expressions:

```
1: entier b ** abs i/15 > 17 and p <= 4
2: ((((entier b) ** (abs i))/15) > 17) and (p<=4)
```

are equivalent.

Example 2:

The two arithmetic expressions:

```
1: p/b*c/d/e - a * b/c
2: ((((p/b)*c)/d)/e) - ((a*b)/c)
```

are equivalent.

2.126 progmode

This integer standard identifier controls the paging algorithm of the run time system, in such a way that program (and context data) segments may be locked in memory. The value of progmode is used only when the paging algorithm is activated, i.e. when a segment requested (called current segment) is not in memory. The value specifies whether the current segment should be locked or not. A locked segment will never be a victim, when the paging algorithm needs space in memory for another segment, i.e. a locked segment will remain in memory. The value of progmode determines the action as follows:

progmode >0 _ (bitpattern):

1 shift 0:	Lockmode is passive, i.e. the current segment is not locked.
1 shift 1:	The current segment (program or context data) is locked.
1 shift 2:	If the current segment is a program segment, it is locked.
1 shift 3:	If the current segment is a context data segment, it is locked.

progmode = 0:

All segments locked are released again, i.e. these segments may be victims again.

progmode < 0:

The latest: abs (progmode) locked segments are released. This number may be calculated by means of the standard variable: blocksread.

Another (program structure dependent) locking method is described in procedure lock.
The default value of progmode is 1.

2.127 progsize

This integer standard identifier holds at any time in the lifetime of an ALGOL or FORTRAN program the size of the program in segments, i.e. program segments excluding virtual storage segments (owns/context data/virtual activity stack pictures/COMMON data blocks/zone COMMON data blocks), i.e. the size of the program right after translation minus one segment.

Example 1:

See Example 2 of lock.

Example 2:

See Example 3 of lock.

2.128 random

This real standard procedure 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 < random < 1.
i        (call and return value, integer). At call time
         the latest pseudo-random number generated (or a
         starting value for the ge neration). At return
         the next pesudo-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.

Example:

A random integer 0 < = p < 99 can be found in this way:

```
p: = entier (random (i) * 99);
```

2.129 rc8000

This boolean standard identifier is true if the program is executed on RC8000, otherwise false. Because the value is evaluated at run time, a binary program may be moved from RC8000 to RC9000-10 (and vice versa) still maintaining the correct definition of RC8000.

2.130 read

This integer standard procedure inputs a sequence of numbers given in character form on a document or in an array, 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. If one or more illegal
            numbers are read the value of read is negative.
z           (call and return value, zone or call value,
            array of type boolean, integer, long, real,
            double real or complex). In case of a zone it
            specifies the document, the buffering, and the
            position of the document cf. (15).
            In case of an array, it contains the characters
            packed 3 to a word starting in the word with
            halfword index 1.
destination (return value, integer, long, real, or arrays
            of the stated types, zone record or call value,
            boolean).
            Each parameter is handled according to its type
            as follows:

boolean     A boolean must be followed by an integer
            parameter. The integer value is used as "max
            charcount" (cf. below) for the next destination
            parameter, i.e. a single destination variable or
            the destination variables of an array or a zone
            record.
            The default value of "max charcount" is
            "infinity". If the integer parameter is non
            positive, the parameter pair is ignored.
other       Read assigns numbers to the destination
allowed     parameters from left to right. A simple
types       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.
            Finishing each destination parameter, read
            resets "max charcount" to its default.
```

Syntax of numbers

Read skips all blind characters (class 0, cf. (15). 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) or by "max charcount" characters having been read, whichever occurs first. Leading characters of class > 5 are disregarded unless they contain

the EM character (see below). If the number constituents fulfil the rules for algol numbers cf. (14), the number is assigned to a destination variable. If it is not an algol number or if it exceeds the range of the destination variable, the greatest positive number of the appropriate type is assigned instead, and the return value of read becomes negative.

However, the "greatest" negative value of an integer or long variable cannot be read by the procedure, because a negative number is read as a positive value and the sign is changed afterwards.

For the same reason, rounding of a negative real value to accomodate a destination variable of type long or integer will result in the rounded positive value with a negative sign, i.e. -1.5 becomes - 2.

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 or, in case of reading from an array, when the first character after the upper index element is to be read, whichever occurs first. The following destination variables are unchanged. In this situation the value of read is useful.

Zone state:

As for readchar.

Blocking:

As for readchar.

Example 1: reading and checking a matrix

An n*n-matrix is recorded 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 real array matrix(1:n,1:n);
   if read(in,matrix) < n*n then goto dataerror;
```

The matrix might for instance be recorded 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 <> 'nl' then
   goto dataerror;
```

Example 2:

The following character sequence represents 5 numbers as shown:

```
 a- 1.7bcd-12345678 9'60 3+10ee<EM>
```

```
  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.

Example 3:

The following character sequence appears on current input:

```
123456789012345
```

When read by the call:

```
read (in, true, 3, ia)
```

Where ia is declared integer array ia (1:5), the variables will become:

```
123, 456, 789, 12 and 345
```

2.131 readall

The integer standard procedure inputs a mixture of numbers in character form, text strings, and single characters stored on a document or in an array. 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 or call value,
          array of type boolean, integer, long, real,
          double real or complex). In case of a zone, it
          specifies the document, the buffering, and the
          position of the document cf. (15).
          In case of an array, it contains the
          characters, packed 3 to a word, starting in the
          word with halfword index 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 kinds of the
          items are 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).
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) 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 halfwords, 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 filed up with null cha racters. 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 (i):= 2 for a legal number,
   kind (i):= 1 for an illegal or syntactically wrong number.

Terminating reading:

Readall returns as soon as a terminator (class 8) has been input and stored, or in case of reading from an array when the first character after the upper index element is to be read, whichever occurs first. 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.

Blocking:

As for readchar.

Example 1:

A line input by read_all with the standard alphabet to an iteger array may be printed and 'reshaped' in this way:

```
begin
  zone z(...);
  integer array ia, kind (1:...);
  long array field laf;
  integer i, j, n;

  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
  <*kind 1*> write (out, <:illegal:>);
  <*kind 2*> write (out, <<-dddddd>, ia(i));
  <*kind 3,4,5*> ;;;
  <*kind 6*> begin
               laf:= (i-1)*2;
               j:= (write (out,<: :>,ia.laf) +5)// 6*2;
               i:= i + j -1;
             end;

  <*kind 7*> if ia(i) <> 'sp' then
               write (out, <: :>, false add ia(i), 1)
end case;
write (out, "nl", 1);
```

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  45  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   0    11
ia   ab  0   58 a1. 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 : a1.2c ,    18illegal
```

Example 3: Typical adp-input

A list of employees is recorded in this way:

```
<identification number><department number><status>,
   <surname>,<first names><NL>
<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 have 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 a text constituents. All digits are delimiter symbols. In both parts of the alphabet table, new line and end medium are terminators.

An input program which checks the syntax and outputs the list as a sequence of records may look like this (the procedure error contains besides alarm message also setting of a boolean error found):

```
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;

<*field assignments*>
lgth:= 2; ident:= 4; dept:= 6; stat:= 8; ename:= 8;
table index:= 0; error_found:= false;
for n:= readall (z, val, kind, 1) while val(1) <> 'em' do
begin
  if val(1) = 'nl' then <*nothing*>
  else
  if n > max or n<1 then error(1, <:line length:>)
  else
  if kind(1)<>2 or val(2)<>'sp' then error(2,
  <:identification:>)

  else
  begin
    <*check department*>
    i:= 2;
    repeat i:= i+1; <*skip spaces*> until val(i) <> 'sp';
    if kind(i) <> 2 then error(3, <:department:>)
    else
    begin
      dept_inx:= i; <*save index for department*>
      <*check status*>
      if kind(i+1)<>6 or val(i+2)<>',' then
      error(4, <:status:>)
      else
      begin
        val(i+1):= ...; <*transform status*>
        <*check surname*>
        for i:= i+2, i+1 while kind(i)= 6 do;
        <*skip surname*>
        if val(i)<>',' or i = dept_inx+3 then
        error(5, <:surname:>)
      else
      begin
      <*check first names*>
        f_names_inx:= i+1; save index for first names*>
        for i:= i+1 while kind(i) = 6 do; <*skip first names*>
        if kind(i)<>8 or i=f_names_inx then
          error(6, <:first names:>)

          else
          begin
          <*line accepted*>
            recl:= (i-dept_inx-1)*2; <*record length*>
            outrec6(empl, recl);
            empl.lgth := recl;
            empl.ident:= val(1);
            empl.dept := val(dept_inx);
            empl.stat := val(dept_inx+1);
            vname:= (dept_inx+2)*2; <*namefield in array val*>
            tofrom(empl.ename, val.vname, recl-8);
```

```
            end line accepted;
          end check first names;
        end check surname;
      end check status;
    end check department, identification, linelength, newline;

    if val(n)= 'em' or error_found then
    repeatchar(z);
    tableindex:= 0; error_found:= false;
  end for n:= readall;
```

2.132 readchar

This integer standard procedure 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 cf. (14).
z         (call and return value, zone). Specifies the
          document, the buffering, and the position of
          the document cf. (15).
val       (return value, integer). The value of the
          character cf. (14).
```

Zone state:

The zone must be open and ready for character reading (state 0, 1, or 2, see getzone6), i.e. since the latest call of open or setposition, the zone must only have been used for character reading. To make sense, the document should be an internal process, a disc process, a backing storage area, a terminal, 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 the 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 (15). On a terminal 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);
outchar (outz, 'em');
```

Blind characters may be copied, too, if another alphabet is selected (see intable).

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:    2 sign:         3 digit:        4 other:
       state:
-1,  start       after sign,  1  after digit, 2  start       3
 2,  after sign  after error, 4  after digit, 2  after error, 4
 5,  after digit after error, 4  after digit, 2  start,       5
 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 3. Plus and minus are given class 2. All other non-blind characters are given class 4. Class 1 cannot be used because of its shift action.

The algorithm may then be written like this:

```
<*assign the alphabet*>
for c:= 'nul' step 1 until 'del' do
  alpha(c):= (if c='+' or  c='-' then 2 else
     if c>'/' and c<'8' then 3 else
     if c='em'          then 8 else 4)
       shift 12 + c;
inable (alpha);
...
state:= -1;
<*The possible states are:

start=-1, after sign=2, after digit=5, after error=8*>

sign:= 1; number:= 0; error:= false;

repeat
  index:= readchar (z, c) + state;

<*find the index for the 4 rows state table,
  seen as a one dimensional array*>
  action:= case index of
```

```
  (1,2,3, 4,2,4, 4,2,5, 3,3,5);
  state:= case index of
  (2,5,-1, 8,5,8, 8,5,-1, 8,8,-1);

  state action of
  begin
    <*1, set sign*>
      sign:= 44 - c; <* +=43, -=45 *>
    <*2, include digit*>
      if number >= 1 shift 20 then
        begin error:= true; state:= 8 end
      else
        number:= number shift 3 + c - '0';
    <*3, no action*>
      ;
    <*4, set error indication*>
      error:= true;
    <*5, terminate number*>
      number:= number * sign;
  end case;
until action = 5;
```

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 may be speeded up 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 is possible if the input is performed blockwise by means of inrec6 and the characters are unpacked as shown in Example 3 of extract.

2.133 readfield

This integer standard procedure inputs the next field designator of a format 8000 transaction.

Call:

```
readfield (z, fieldtype, aux)
```

readfield	(return value, integer). Defines the type of the first character of the field according to the fieldtype table shown below. If the field does contain data, this value is 8, otherwise it is the fieldtype of the next field (or ETX or EM).
z	(call and return value, zone). Specifies the document from which current transaction is input.
fieldtype	(return value, integer). The type of the field designator, according to the fieldtype table shown below.
aux	(return value, integer). The interpretation depends on fieldtype see table be low).

On exit the zone is positioned at the first character of the data. If there is no data, the zone is positioned at the next field designator, or ETX (end transaction), or EM.

The connection between fieldtype and aux is as follows:

field-type	Command	ISO char	aux
1	SBA: Set buffer address	17	char position
2	SF: Start field	29	attribute char
3	IC: Insert cursor	19	undefined
4	EUA: Erase unprotected to addr	18	char position
5	PT: Program tab	9	undefined
6	RA: Repeat to addr	20	char shift 12 + charposition
7	EXT: End of text	3	undefined
8	All others (i.e. data)		undefined
9	EM:	25	undefined
10	USM	31	undefined

Zone state:

The zone must be in state 1,2,10, or 11, i.e. after character reading or after waittrans. The new state becomes 1 or 2, ready for character reading.

If the zonestate is after waittrans, readfield returns the fieldtype corresponding to the first character after the transaction head, i.e. fieldtype = 8 if there is no field designator.

On the other hand, if the zone state is after character reading, readfield always returns the fieldtype of the next field designator, causing a skip of possible unread data fields.

Example 1:

See Example 1 of activity (the procedure get_input_data).

2.134 readstring

This integer standard procedure inputs a text string given as 8-bit characters on a document or in an array. The text string is packed in a way which makes it easy to use as a string parameter.

Call:

```
readstring (z, arr, i)
or
readstring (z, arr, i, max_charcount)
```

```
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 or call value,
              array of type boolean, integer, long, real,
              double real or complex). In case of a zone,
              it specifies the document, the buffering,
              and the position of the document cf. (15).
              In case of an array, it contains the
              characters, packed 3 to a word, starting in
              the word with halfword index 1.
arr           (return value, integer, long, real, double
              real, complex array or zone record). 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.
max_charcount (call value, integer). The reading
              terminates when "max charcount" characters
              have been read, cf. below about terminating
              reading.
              The parameter is optional, the default being
              "infinity".
```

Syntax of a test string:

Readstring skips all blind characters (class 0, cf. (14)). Among the remaining characters, read string 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 3 8-bit characters to a word. 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). The values given by the standard ISO alphabet are shown in (14).

A null character is packed after the last character of the string and the corresponding element (at most the double word) of arr is filled up with null characters. The terminator is read, but not packed into arr. If read in the zone z, it can be examined by means of:

```
repeatchar (z); readchar (z,c);
```

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, if arr is filled, if "max charcount" characters have been read, or reading from an array, when the first character after the upper index element is to be read, whichever occurs first.

When the array is filled, the value of readstring is negative, and the textstring is not terminated by a null character. The last character read is the last one packed in arr.

When an EM character is met the reaction depends on its position. If the EM character appears as terminator of the text the reading is terminated as usual. When the EM character appears as a leading delimiter, however, the value of readstring becomes zero, and the first element of arr is undefined.

Zone state:

As for readchar, 2.120.

Blocking:

As for readchar, 2.120.

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) < 0 then dataerror;
  ...
  write(out, text);
```

Example 2:

See Example 3 of open.

2.135 real

This delimiter, which is a declarator, is used in declarations and specifications of variables of type real.

Syntax:

```
real <identifier list>
```

Semantic:

The variables in the identifier list will be of type real, and occupy 48 bits in the memory.

The value of a real is in the interval:

```
-1.6'616 <= value <= 1.6'616
```

The real must not have more than 14 significant digits or 14 decimals.

Example 1:

```
real r1;
real r2,yes,no,price;

procedure pip(a);
real a;
```

2.136 real

This delimiter, which is a transfer operator, changes the type string and long to type real.

Syntax:

```
real <operand>
```

Priority: 1

Operand type:

long or string.

Result type:

real.

Semantic:

Changes the type of a string expression or a long primary to type real. The binary pattern of the operand is unchanged. The binary pattern of a string and a long is described in (14).

Note that this use of the delimiter real is totally different from its use in a declaration or specification.

Example 1:

Let s be a formal string parameter which actually is a text string. In the procedure 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 (i.e. a long text string reference), 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.ddd'-dd>,<<-d.ddddd'-dd>));
...
write(out, string r, x, y,...);
```

Example 3:

See Example 1 of swoprec6.

2.137 repeatchar

This standard procedure 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 one, i.e. the latest character read. Note that the logical position is unchanged.

Zone state:

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. The zonestate becomes 2, 'after repeatchar'.

In all other states repeatchar is blind, and the state is unchanged.

The definition of repeatchar implies that several calls of repeatchar have the same effect as one call, i.e. only one character can be repeated.

Example 1:

See Example of read.

2.138 replacechar

This integer standard procedure changes the special characters printed by write, writeint, outtext, and outinteger to user defined characters. The special characters are: leading space, space in number, +, -, decimal point, exponent mark, * (alarm printing), and ending space (fill character).

Call:

```
replace_char (special, newchar)
```

replace_char	(return value, integer). The character value of the present special character, which may be used for restore purposes.
special	(call value, integer). Defines which special character is to be replaced: 0: leading space 1: space in number 2: + (positive sign) 3: - (negative sign) 4: . (decimal point) 5: ' (exponent mark) 6: ending space (fill character) 7: * (termination star). 8: fill character used to fill up current word when terminating writing into array
newchar	(call value, integer). The character value to replace the present special character. If newchar > 255 or newchar < 0 the character will not appear in the output.

Note!

If the termination star is replaced by a character value > 255 or < 0, the character last output in the last position will not be destroyed.

Example 1:

The statements

```
c1:= replace_char (1<*space in number*>, '.');
c4:= replace_char (4<*decimal point*>, ',');
write (out, <<ddd ddd ddd.dd>, 133542876.25);
```

will print:

```
133.542.876,25
```

The old values of the special characters may be restored by

```
replace_char (1, c1);
replace_char (4, c4);
```

Example 2:

The statements

```
for i:= 1 step 1 until 8 do
   replace_char (i-1, case i of (
      '*', '.', 'p', 'm', 'd', 'E', '-', -1));
write (out, << dd dd.d+    >, 12345.678, -11);
```

will print:

```
**12.34d6E1p-*****11d0m---
```

2.139 resetzones

This standard procedure resets the buffersize and number of shares of each zone in a zone array to the size and number they had right after declaration.

Call:

```
resetzones (za)
```

```
za      (call and return value, zone array). The
        buffersize and no of shares are reset for all
        zones za (1),...,za (no of zones) to the values
        given in the declaration. The procedure is the
        "undo" of the procedure initzones.
```

Zone state:

The state of each zone must be 4, after declaration, and is not changed by the procedure.

2.140 resume

This standard procedure works as the context operator continue, except that the context label is cleared after execution, cf. (15).

Call:

```
resume
```

2.141 round

This delimiter, which is a monadic arithmetic 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 <operand>
```

Priority: 1

Operand type:

long or real.

Result type:

integer.

Example 1:

Two reals with absolute value below 2**23 may be integer divided in this way:

```
round i1 // round i2
```

Example 2:

In the following statement you will get integer overflow if r is outside integer range:

```
writeint (z, round r);
```

2.142 setfpmode

This standard procedure sets or removes a userbit from fp's mode word.

Call:

```
setfpmode (modebit, bool)

modebit  (call value, integer). 0<=modebit<=11.
bool     (call value, boolean).
         True => the bit is set,
         false => the bit is removed.
```

The standard FP modebits are numbered beyound 11, and cannot be changed by setfpmode. The ok-bit and the warning-bit, however, can be changed by the standard identifier errorbits.

Example 1:

If you in an algol program p execute the statement:

```
set_fp_mode(5, true);
```

then in the jobfile:

```
40 ...
50 p data
60 if 5.no
70 ...
80 ...
```

line 70 will be skipped.

2.143 setposition

This boolean standard procedure 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 timeconsuming 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 cf. (15). 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. A negative value of file will unload the tape.
block	(call value, integer). Irrelevant for documents other than magnetic tape and backing storage. Specifies the block number (segment number when backing storage) of the wanted position cf. (15). Blocks are counted 0, 1, 2, ... A negative value of block will lead to param alarm.

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), 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 used 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 usual.

Write tape mark:

If the document is a magnetic tape which latest has been used for output, a tape mark is written.

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 name the name returned by the parent (which means that setposition changes the document name in the zone accordingly).

Setposition starts the tape positioning, by sending a setmode operation, the result of which is ignored and a move operation to the external process. The positioning is not waited for until 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).

If the value of file is negative, an unload message will be sent. Only if it is checked later on (input, output, setposition, close), will the move operation be sent, resulting in a mount tape request, as the tape was unloaded.

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.

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 (0).

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 alternately write 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:

```
repeat
   write(out,<:type yes or no:>);
   setposition(out,0,0);
   readstring(in,ra,1);
until...;
```

Example 2: Random access to backing storage (relative organized file)

Let the backing storage area bs25 contain records of 80 halfwords originally output in shares of 128 elements (= 1 segment, 512 halfwords). You may get record j (the records being numbered 0,1,2...) in this way:

```
begin zone z(128,1,stderror);
   <*double buffering will not pay in this case*>
   open(z,4,<:bs25:>,0);

   j:= ...;
      setposition(z,0,j//6);
      <* 6 records are stored on one segment*>;
   for i:= j//6*6 step 1 until j do inrec6 (z,80);
```

Example 3: Simultaneously 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, and 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);
```

Example 4: Output of tape mark and empty file

Two tape marks in sequence i.e. an empty file 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.

2.144 setshare

This standard procedure is the 'reverse' of getshare, 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 counted
        from lexicographical index 1). The contents of
        ia have the meaning explained in getshare. The
        contents of ia (1), ..., ia (12) 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 restrictions apply to the values of ia:

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.

2.145 setshare6

This standard procedure is the 'reverse' of getshare6, 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,
        counted from lexicographical index 1). The
        contents of ia have the meaning explained in
        getshare6. The contents of ia (1), ..., ia (12)
        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 restrictions apply to the values of ia:

ia(1)	Share state. Only if the state of the share descriptor is 0 or 1 at call time, will ia (1) be transferred. In this case ia (1) must be 0 or 1.
ia(2)	First shared. Must be a halfword index in the zone buffer.
ia(3)	Last shared. Must be a halfword index in the zone 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 ad dress corresponding to a block within the zone buffer.

If the restrictions are violated, an alarm occurs. 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 the elements ia(1), ..., ia(12):

```
getshare6(z,ia,sh);
setshare6(z,ia,sh);
```

Example 1:

See Example 10 of monitor.

Example 2:

See Example 1 of activity (the procedures sense and senseready).

2.146 setstate

This standard procedure changes the zone state.

Call:

```
setstate (z, state)

z       (call and return value, zone). The zone state of
        z is changed to the value of state.
state   (call value, integer). The new zone state.
```

2.147 setzone

This standard procedure is the 'reverse' of getzone, in the sense that it assigns value to a zone descriptor. The procedure is the ALGOL 5 equivalent of setzone6.

Call:

```
setzone (z, ia)
```

`z`	`(call and return value, zone). The zone descriptor of z is changed.`
`ia`	`(call value, integer array, length >= 17 counted from lexicographical index 1). The contents of ia have the meaning explained in getzone. The contents of the ia (1), ..., ia (17) are transferred to the zone descriptor, provided that the restrictions below are fulfilled.`

Restrictions:

The following restrictions apply to the values of ia:

ia(1)	Mode shift 12 + kind. As for setzone6.
ia(14)	Record base. As for setzone6.
ia(15)	Last byte. As for setzone 6.
ia(16)	Record length. Measured in ele ments of 4 halfwords each, other wise as for setzone6.
ia(17)	Used share. As for setzone6.

2.148 setzone6

This standard procedure is the 'reverse' of getzone6 in the sense that it assigns values to a zone descriptor.

Call:

```
setzone (z, ia)

z       (call and return value, zone). The zone
        descriptor of z is changed.
ia      (call value, integer array, length >= 17,
        counted from lexicographical index 1). The
        conents of ia have the meaning explained in
        getzone6. The contents of ia (1), ..., ia (17)
        are transferred to the zone descriptor, provided
        that the restrictions below are fulfilled.
```

Restrictions:

The following restrictions apply to the values of ia:

ia(1)	Mode shift 12 + kind. The range of the kind is 0 < = kind < = 20. 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 halfword. Must be an absolute address within the zone buffer.
ia(16)	Record length in halfwords. 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(1), ..., ia(20) exist):

```
getzone6(z,ia);
setzone6(z,ia);
```

Example 1:

See Example 2 and Example 3 of getzone6.

2.149 sgn

This integer standard procedure yields -1 or 1 according to the sign of the parameter.

Call:

```
sgn (r)

sgn     (return value, integer). sgn is 1 for r>= 0, -1
        for r < 0.
r       (call value, integer, long, or real).
```

Example 1:

```
a:= sgn (r)*r;  <*
```

is equivalent with

```
a:= abs (r)*>
```

2.150 shift

This delimier, which is a pattern operator, is used for packing and unpacking of reals, longs, integers, booleans, and strings.

Syntax:

```
<operand1> shift <operand2>
```

Priority: 2

Operand types:

<operand1>: boolean, integer, long, real or string.

<operand2>: integer, long or real.

Result type:

The result is of the same type as <operand1>.

Semantic:

Shift treats <operand1> as a binary pattern cf. (14). <operand2> is rounded to an integer if it is long or real. This value is then used to indicate the number of bits <operand1> is to be shifted.
The shift is to the left if the (possibly rounded) value of <operand2> is positive and to 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.

Example 1:

See Examples of add, extend, extract, and long.

2.151 sign

This integer standard procedure yields 1, 0, or -1 according to the sign of the parameter.

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).
```

Example 1:

```
a:= sign (r)*r; <*
```

equivalent with

```
a:=abs (r)*>
```

Example 2:

```
case sign (master.ident - trans.ident) + 2 of begin
  <* -1: master < trans*>
     ...
  <* 0: master = trans*>
     ...
  <* 1: master > trans*>
     ...
end case;
```

2.152 sin

This real standard procedure performs the trigonometric function sine.

Call:

```
sin (r)

sin     (return value, real). Is the trigonome tric
        function sine of the argument r, in radians with
        -1 <= sin <= 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 sin is completely undefined for abs (r) > 3'10, and then the result is always 0.

Example 1:

```
sin (pi/6); <*is 0.5*>
```

2.153 sinh

This real standard procedure performs the mathematical function sinh.

Call:

```
sinh (r)

sinh    (return value, real). Is the mathematical
        function sinh of the argument r.
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, a runtime alarm occurs (Sinh 0).

2.154 sortcomp

This integer standard procedure compares two records by using the comparison code generated by procedure startsort6, changekey6, or initkey.

Call:

```
sortcomp (z, a, b)

sortcomp  (return value, integer). The result of the
          comparison, as follows:
          sortcomp < 0: a < b
          sortcomp = 0: a = b
          sortcomp > 0: a > b
where     a < b (or a > b) means that the record a would
          precede (or succed) the record b if the records
          were sorted. The equal sign means, that the
          order is immaterial.
z         (call value, zone). The zone associated with
          the comparison code (see procedure startsort6
          or initkey).
a         (call value, array). The name of an array with
          lower index = 1 (or zone) holding the first
          record.
b         (call value, array). The name of an array with
          lower index = 1 (or zone) holding the second
          record.
```

Example 1:

See Example 1 of deadsort.

Example 2:

A comparison of the records in zone no. i and zone no. j of zone array x. The key types, relative addresses, and sequencing rule are the same as in Example 1 of initkey. The statement

```
diff:= sortcomp (z, x(i), x(j));
```

will result in diff having the symbolic sign value of x(i) - x(j), when x(i) and x(j) are considered to be real numbers to be compared.

2.155 sqrt

This real standard procedure yields the square root of the parameter.

Call:

```
sqrt (r)

sqrt    (return value, real). The square root of r.
r       (call value, real, long, or integer). r >= 0.
```

Accuracy:

r = 0	gives sqrt = 0.
r > 0	gives a relative error below 6.4'-11.

Alarm:

If r < 0, a runtime alarm occurs (sqrt 0).

Example 1:

```
c:= sqrt(a**2 + b**2);
```

2.156 startsort6

This integer standard procedure initiates a sorting process in a zone, and generates a piece of comparison code, so that the procedures newsort, outsort, deadsort, lifesort, and sortcomp can be used with that zone as parameter.

Call:

```
startsort6 (z, keydescr, noofkeys, reclength)
```

`startsort6`	`(return value, integer). The maximum number of records which can be kept in the zone.`
`z`	`(call value, zone). The name of the zone used for sorting.`
`keydescr`	`(call value, integer array). The array holds information about types and relative locations (halfword numbers) of key fields in a record.`
`noofkeys`	`(call value, integer). Number of key fields (rows) in keydescr.`
`reclength`	`(call value, integer). Maximum number of halfwords in the records which are to enter the sorting process.`

Zone declaration and the length of the record:

The length of the key code generated by startsort6 varies according to the value of noofkeys. A zone capable of holding N records at one time + the key code must be declared as follows:

```
zone z((N+1)*((reclength + 2)//4) + 3*noofkeys + 6, 1,
error)
```

where reclength is the maximum record length measured in halfwords. The meaning of noofkeys is described below. The value of reclength must be even and positive.

The error procedure must be supplied by the user. It has the same form as an ordinary block procedure i.e. error (z, s, b), where z is a zone, and s and b are integers (call values only). The error condition is indicated by the values of s and b; s is the error type, for which the following conventions hold:

s = 1:	attempt to input more than N records to the zone z. b = no. of inactive records in the zone.
s = 2:	attempt to output a record from the zone z holding no active records. b = no. of inactive records in the zone.
s = 3:	attempt to call changekey6 (or initkey) with active records present in the zone. b = no. of active records in the zone.

Zone state:

The zone must be in state 4, after declaration. The state becomes 9, in sort.

Description of keys and the number of keys:

The integer array keydescr must be declared at least

integer array keydescr(1:noofkeys, 1:2);

The value of noofkeys must fulfil

0 < noofkeys < = min(reclength, 170)

Each row in the array holds a description of a key field in the record to be sorted.

Column one in the array holds information about the key field type and the rule of sequencing. For the key field type the following conventions hold:

Value	Key Field Type	Sequencing
+1	12 bit signed integer	ascending
-1	12 bit signed integer	descending
+2	24 bit integer	ascending
-2	24 bit integer	descending
+3	48 bit long	ascending
-3	48 bit long	descending
+4	48 bit real	ascending
-4	48 bit real	descending
+5	12 bit unsigned integer	ascending
-5	12 bit unsigned integer	descending

When sorting alphabetically, three ISO characters packed in one 24-bit integer or six characters packed in one 48-bit long, can be treated as one key field.

Column two in the array is the relative position of the keyfield, within the record, specified by the number of the last halfword in the field, as for a field variable. The relative position must be < = min (reclength, 2046).

Note on the value of startsort6:

The value returned by startsort6 is the number of records which in the normal situation may be placed in the zone. In one situation, however, one more record may be placed in the zone by means of newsort. This is the case after the call of startsort but before a call of outsort or life sort. Therefore startsort6 may yield values from -1 and upwards. The meaning of the value -1 is that no record can possibly be placed in the zone, but the zone may of course be used for sortcomp. The value 0 means that a call of newsort will yield space for one record. This space may be used for temporary storage of data.

Example 1:

See Example 3 of newsort, where a zone to hold 100 records of length 10 double words per record is declared.

Example 2:

See Example 4 of outsort, where a zone for k-way merging of two word records is shown.

Example 3:

A simple error procedure would be the following:

```
procedure sorterror (z, s, b);
zone z; integer s, b;
begin
   case s of
   begin
      write (out, <:input error in sort:>);
      write (out, <:output error in sort:);
      begin
         write (out, <:initializing error in sort.:>);
         stderror (z, s, 4*b)
      end
   end;
   write(out, <: no of inactives =:>, <<dddd>, b);
   goto errorhandling
end;
```

The call of stderror will cause the number of active halfwords to be printed and the execution is terminated.

2.157 stderror

This standard procedure terminates the program with a runtime alarm (the program gives up) specifying an error condition on a peripheral device. It is used as the block procedure of zones where the program wants to give up in case of device errors or in blockprocedures to give up explicitly in case of 'give up' bit (hard errorbit) in the status word.

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 halfwords
        transferred.
```

The program is terminated with the run time alarm:

```
giveup <value of b> algolcheck
called from ...
```

The file processor prints an inerpretation of the logical status word 's' after the alarm message from the algol program.

However, if the error situation is trapped by a traplabel (see trap), the file processor is not activated. In this case the logical status word can be made available to the program by the standard procedure getalarm.

Example 1:

See Example 2 of inrec6, where stderror is used in two ways: in a zone declaration, and as a procedure call in a user specified block procedure.

2.158 stopzone

This standard boolean procedure terminates the current use of a zone and optionally, if the document connected to the zone is a magnetic tape which latest has been used for output, it may write a tape mark.

Call:

```
stopzone (z, mark)

stopzone   (return value, boolean). True, if input/output
           may continue without repositioning, false if
           it takes a positioning to continue
           input/output (cf. be low).
z          (call and return value, zone). Specifies the
           document, the position of the document, and
           the latest operation on z.
mark       (call value, boolean). If the document is a
           magnetic tape and z latest has been used for
           output and mark is true, a tape mark is
           written to terminate the file, else no mark is
           output.
```

Stopzone proceeds in two steps: terminate the current use and may be write a tape mark.
The steps are exactly as described in setposition

Zone state:

The zone must be open when stopzone is called (state 0,1,2,3,5,6,7 or 8). Stopzone changes the zone state to "opened" and positioned" (0) and stopzone returns true, except for magnetic tape, which has latest been used for input, when the zone state becomes "opened but unpositioned" (8) and stopzone returnes false.

Logical position:

The logical position for a backing storage area (or a disc) becomes just after the last block output or just after the last block input, which also is the physical position. If the input is multishared, this position is (no of shares -1) shares ahead of the latest available input. If input (or output) is continued without a reposition, then, that many shares are skipped. The logical position for a magnetic tape becomes just after the last block output, which also is the physical position, so output may continue without a reposition. In case of input, the logical position becomes just after the last block available as input, but in case of multishared input, the physical position will be (no of shares - 1) shares ahead, so a reposition will have to take place before continued input/output, whether it should continue from the latest block available as input or a number of blocks really should be skipped.

2.159 string

This delimiter, which is a specificator, is used in specifications of variables of type string.

Syntax:

```
string <namelist>
```

Semantic:

The variables in the namelist will all be of type string.

Example 1:

```
procedure pip(text);
string text;
begin
   .
   .
   .
   c:= long text;
```

2.160 string

This delimiter, which is a transfer operator, changes the type real or long to type string. The operator is required when a string stored in real or long variables is used as a parameter of type string.

Syntax:

```
string <operand>
```

Priority: 1

Operand type:

long or real

Result type:

string.

Semantic:

The value of string <operand> has the same binary pattern as the value of the operand. The binary pattern of a string is described in (14). Depending on the value of the operand, the resulting pattern may mean a layout, a complete text string, or a text portion. Note that this use of the delimiter string is totally different from the string specification.

Example 1: Layout

See Example 2 of real.

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 a 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.

An easier way to output a text stored in an array is

```
write (out, ra);
```

Example 3:

See Example 1 of real (operator).

2.161 swoprec

This standard procedure is the ALGOL5 version of swoprec6. It gives you direct access to a sequence of elements of 4 halfwords 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 halfwords 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 cf. (15).
length	(call value, integer, long, or real). The number of elements of 4 halfwords each in the record. Length must be >= 0.

Except that the record length is measured in elements of 4 halfwords each, swoprec works as swoprec6.

2.162 swoprec6

This standard procedure gives you direct access to a sequence of halfwords of a document. The halfwords 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 halfwords 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 halfwords left in the present block for further calls of swoprec6.
z	(call and return value, zone). The name of the record. Specifies further the document, the buffering and the position of the document cf. (15).
length	(call value, integer, long, or real). The number of halfwords 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 getzone), i.e. the zone must 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 halfwords just after the current logical position of the document and moving the logical position to after the last halfword 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 halfwords 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 (15). 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 communication 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 1: direct updating

Each double word of the backing storage area ma28 may be added to the corresponding double word of the area ma30 in this way:

```
begin zone ad(512*2,2,endarea),res(512*3,3,endarea);
  boolean endfile; integer i;
  procedure endarea(z,s,b);
  zone z; integer s,b;
  if s extract 1 = 0 then
   begin endfile:= true; b:= 2048; end
    else
  stderror(z,s,b);
  open(ad,4,<:ma28:>,1 shift 18);
  open(res,4,<:ma30:>,1 shift 18);
  endfile:= false;

  for i:= inrec6(ad,2048) while -,endfile do
  begin
    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:
  end;
  close(ad,true); close(res,true);
```

Example 2: swopvar

The following procedure performs sequential updating of variable length records by means of swoprec6. Note that the checksum of a possibly changed record must be calculated by a call of checkvar before the call of swopvar.

```
integer procedure swopvar (z);
zone                       z;
<*swopvar (return value, integer). The number of
          halfwords left in the present block for
          further calls of swopvar/swoprec6.
z         (call and return value, zone). The name
          of the record. Specifies further the
          document, buffering and the position of
          the document.
*>
begin
  integer b;
  integer field lngf, sumf;
```

```
  lngf:= 2; sumf:= 4;
  for b:= swoprec6 (z, 4) while z.lngf = 0 do
  swoprec6 (z, b);
  b:= z.lngf;
  changerec6(z, 0);
  swopvar:= swoprec6 (z, b);
  if checkvar (z) <> z.sumf then
    blockproc (z, 1 shift 11, b);
end;
```

2.163 system

This integer standard procedure gives access to various system and job parameters. Some of the functions of system require knowledge of the job organisation cf. (6) and the multiprogramming system cf. (1) and (2). Be aware that monitor tables are with halfword addresses.

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.
        Notice that whenever real array is allowed, zone
        record is, too.
s       (call value, string). Meaning depends on fnc.
```

The value of fnc is restricted to 1 <= fnc <= 15, with the following meaning:

System(1,i,arr), floating point precision

No function.

System(2,i,arr), free memory, program name

```
system  The number of halfwords available in the low end
        partition of the job process for reservation of
        further variables leaving only 1024 halfwords for
        program segments. When called in an activity, no
        1024 halfwords are left for segments.
        [15] gives the rules for computing the number of
        halfwords occupied by a set of variables.
i       (return value, integer). Gets the same value as
        system.
arr     (return value, integer, long, real, double, or
        complex array, length >= 2 doublewords).
        The name of the document which holds the program
        file is assigned to the first 2 doublewords of
        arr. 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, array of any type, incl. boolean,
        double precision and complex). If the array is of
        more dimensions, the lexicographical index is
        used as the value of system and i. If called from
        fortran program and the array is of more
        dimensions, the successor function is used for
        the value of system and i.
```

System (4,i,arr), file processor parameter

This call does not make sense if the program was translated with the parameter fp.no, and called with the fp-command

```
<program> <integer>
```

The presence of the file processor may be tested by system (13, ...

```
system  The preceding separator and the length of item i
        in the call of the program, packed as
        separator shift 12 + length
        system is 0 if i specifies a non-existing item.
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, integer, long, real, double real,
        or complex array, length >= 2 double words).
        For an integer item the value is converted to a
        real, in case of long array extended to a long,
        and assigned to the first two words of arr.
        For a text item, the value is assigned to the
        first and second double word of arr.
        In case of more text, it is moved to arr,
        doubleword by doubleword, until the text ends or
        arr is full.
```

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
```

The exact format of separator and length is given in [6]. The values relevant in a program call are listed below:

```
separator 0: (    )
                  ) can only occur when i= 0
          2: <NL> )
          4: <SP>
          6: -
          8: .

length    2: the next separator follows
          4: integer item
```

```
        10: text item
    8*n+10: generalized text item (n=1,2,...,7)
```

System(5,i,arr), copy memory area

```
system  1 if the moving was ok, 0 otherwise.
i       (call value, integer). The absolute address of a
        memory location cf. (4) or (5).
arr     (return value, integer, long, real, double or
        complex array). System attempts to copy the
        memory area from absolute address i and on into
        the first word of arr and on.
```

The copying stops when either arr is full or when the word referenced is outside the visible memory. In the latter case system becomes 0. The copying takes place word by word, so that for instance memory (i) and memory (i+2) go into the first element of arr if arr is a multi word type array.

It is necessary to copy areas in connection with some of the entries in procedure monitor, but you may also use system (5,...) for investigations of tables in the monitor cf. (2) and in that way, e.g., find the set of external processes in the actual job host system.

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 cf. (1).
i       (return value, integer). If the message queue of
        the job process is empty, i becomes 0.
        If the buffer claim of the job process is
        exceeded by finding a message in the queue, i
        becomes -1.
        Otherwise i becomes the buffer address for the
        first message in the queue.
arr     (return value, integer, long, real, double or
        complex array length >= 2 double words). The name
        of the job process is assigned to the first 2
        double words of arr.
```

System (7,i,arr), primary output

```
system  The "process description address" for the primary
        output process cf. (2) and (6). The "process
        description address" is not to be trusted, since
        it either is no real process description address,
        or the process description has moved to another
        location. The name and kind is to be trusted,
        though.
i       (return value, integer). The kind of the primary
        output process.
arr     (return value, integer, long, real, double or
        complex array, length >= 2 double words).
```

The name of the primary output process is assigned to the first 2 double words of arr.

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, integer, long, real, double or complex array, length >= 2 double words). The name of the parent process, is as signed to the first 2 elements of arr.

System (9,i,s), run time alarm

```
i       (call value, integer). The value to be printed
        following the alarm cause. Layout is <<-ddddd>.
s       (call value, string). The text to be printed as
        the alarm cause. At most 9 characters will be
        printed. Usually the text should start with an NL
        char.
```

This entry terminates the program execution with a runtime alarm 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. When a program is terminated in this way the ok bit becomes false.

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
        cf. (2).
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, long, real,
        double or complex array, length >= 8 words). The
        contents of the first 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, cf. (20), 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 (20).

System (11,i,arr), catalog bases

```
system  always = 0 (null).
i       dummy
arr     (return value, integer, long, real, double or
        complex array, length >= 8 words). Contains the
        catalog bases associated with the job process.
        1st and 2nd word: the catalog base
        3rd and 4th word: the standard base
```

```
5th and 6th word: the user base
7th and 8th word: the max base
```

The user base is only defined when FP is present in the job process.

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 scope. The catalog base is the base used for the moment, usually it is the standard base.

System (12,i,arr), internal activity description

This entry is intended for debugging purposes.

system	The number of activities declared by the procedure activity. 0 means called in neutral mode, in this case the values of arr are undefined.
i	(call value, integer). Defines the number of the activity in question. If i=0 only the value of system is determined.
arr	(return value, integer, long, real, double or complex array, length>= 2 double words). The internal activity description for the specified activity is stored in arr (1) and on for as many words as arr contains. If declared integer array arr (1:13), the contents of arr will be:
arr(1):	If the activity is waiting in an implicit passivate statement, arr (1) is the message buffer address, otherwise arr (1) = 0.
arr(2):	The number of the activity at present using the allocated memory space for the stack. This number is only interesting for virtual activities sharing the same storage area.
arr(3):	The current "mode" of the program when system (12,...) is called: >0: activity mode (system called from activity no: arr(3)). <0: disable mode (system called from activity no: -arr(3)). =0: monitor mode.
arr(4):	First addr: The absolute address of first word in the activity stack.
arr(5):	Stack bottom: The absolute address of last addr + 1 in the activity stack.
arr(6):	Last used: The absolute address defining the current stack top for the activity.
arr(7):	Virtual. If this value is 0, the given activity uses a memory resident stack of its own. If the value is >0, it defines a word no in the virtual storage, in which case the given activity shares memory space with at least one other activity; here the value of arr (2) may

be of interest.

arr(8): State of the activity:
- -0: The activity is empty, i.e. after declaration, after execution of final end, or after runtime alarm termination.
- -1: The activity is waiting in a passivate statement, written in the procedure.
- -2: The activity is waiting in a passivate statement in the zone i/o system.
- -3: The activity is waiting in an activate statement, i.e. it has activated another activity.

If the activity is executing (i.e. system (12) is called from the activity itself), arr (8) shows how the activity latest was passivated.

arr(9): Youngest zone. Head of the zone chain of the activity. If arr (9) = 1 shift 22, this chain is empty for the moment.

arr(10): CSR. The context block stackref of the given activity.

arr(11): CZA. The context descriptor chain for the given activity.

arr(12): Trap chain. The chain of trap labels for the given activity.

arr(13): Limit last used. The absolute address of the highest stack top so far for the activity. Last used oscillates between stack bottom and first addr, updating limit last used each time it is exceeded. The difference between limit last used and first addr designates the so far not used part of the activity stack.

System (13, i, arr), fp present, compiler version and release, rts segments

system 0 means fp is present in the job process
1 means that it is not

i (return value, integer). The version number of the compiler, which translated the program.

arr (return value integer, long, real, double or complex array,
length > = 2 double words).
Release and rts segment information is stored in the first word of arr and on.
If arr is declared integer array arr (1:4), the values will be:
- arr(1) release no shift 12 + subrelease no
- arr(2) release year shift 12 + month*100+day
- arr(3) no of resident rts segments. Since the segments of the program are numbered 0,1,..., progsize, arr (3) will be the number of the first non-resident rts-segment.
- arr(4) no of rts segments, resident and non resident. May also be interpreted as

the number of the first program segment.

System (14, i, ia), latest answer

system	Undefined.
i	(dummy)
ia	(return value, integer, long, real, double or complex array, length >= 16 halfwords). If declared integer array ia (1:11) the array contains:
	ia (1) - ia (8): latest answer received in the i/o system of the runtime system, i.e. the answer to the latest i/o operation caused by a blockchange in the call of any character or block input or output procedure throughout the entire program.
	ia (9) - ia (11): dummy.

System (15,i,arr), free memory, virtual data file name

system	The number of halfwords available in the low end partition of the job process for reservation of further variables leaving only 1024 halfwords for program segments. When called in an activity, no 1024 halfwords are left for segments. Ref. [15] gives the rules for computing the number of halfwords occupied by a set of variables.
i	(return value, integer). The number of halfwords available in the upper end partition of the job process for reservation of further zone buffers and share descriptors, leaving only 1024 halfwords for program segments. The value is not different when called in activity mode, only activities will never allocate zone buffers in the high end partition.
arr	(return value, integer, long, real, double, or complex array, length >= 2 doublewords). The name of the document which holds the virtual data file is assigned to the first 2 doublewords of arr. The document is always a backing storage area.

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 (unless it has sufficient storage space beyound the address 1M for paging). Furthermore, the program will not be able to call a procedure, as this again is a block needing reservation.

```
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,i,arr); <* free core *>
begin array ra(1:i//4-p);
```

you would have to subtract some value p corresponding to the further locations occupied by the block declaration of the inner block (see ref. [15]).

Example 2: Array bounds

An array of arbitrary dimensions might be zero set by means of the following procedure:

```
procedure clear(ra); array ra;
begin integer low,up;
  for low:= system(3,up,ra) <* bounds *>
        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(1:14); 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(14) shift (-12) extract 12;
   comment the description format is given in (2).
```

The program should now restrict itself to using bufs - 1 double buffered zones simultaneously.

Example 4: Opening to a 'hidden' area

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);
  <*the name must be empty in the open call used
  by setcatbase*>
  open(myself,0,<::>,0);
  system(11,0,catbase); <*bases*>
  <*now set the catalog base to max base*>
  catbase(1):= catbase(7);
  catbase(2):= catbase(8);
  monitor(72,myself,0,catbase); <*set catbase*>
  <*now open, crete area process, establish the
  name table address and leave the zone just
  opened*>
  open(z,4,doc,giveup);
  inrec6(z,0);
  setposition(z,0,0);
  <*in case of output to the area the process
  must be reserved*>
  monitor (8, z, 0, catbase); <*reserve process*>
  <*at last set the catalog base to standard*>
  catbase(1):= catbase(3);
  catbase(2):= catbase(4);
  monitor(72,myself,0,catbase); <*set catbase*>
end;
```

Example 5: Conversion of files

The procedure converts the specified file on the specified printer and paper, when the program is executed as part of a Boss job. The procedure works exactly as the utility program convert.

```
integer procedure convertproc(name,printer,paper):
long array name;
long printer; integer paper;
<*

convertproc (return, integer)

0   ok
1   cfbuf exceeded
2   name not found
3   login scope
4   temp resources exceeded
5   name in use
6   name is not area
7   name is not a text file
10  attention status at remote batch term.
```

```
20  device unknown
21  device not printer
22  parent device disconnected

name     (call, long array contains the name of the
         file
printer  (call, long) contains the name of the
         printer:

         <::>output on remote printer if any is
         present
         <:std:>output on standard printer, local
         to the host
         <:printername:> output on the remote
         printer with the specified name(max 5
         chars)
*>

begin integer array m(1:8);
long field lf;
  m(1):= 30 shift 12 + 1 shift 9 + 1;
  lf:= 6;
  m.lf:= if printer = long<::>
         then long<:conv:> else printer;
  m(4):= paper;
  lf:= 12;
  m.lf:= name(1);
  lf:= 16;
  m.lf:= if name(1) extract 8 = 0
         then 0 else name(2);
  system(10<*parent message*>,1,m);
  convertproc:= m(1);
end convertproc;
```

Example 6:

Further examples on the use of system can be found in the Examples 2, 3, 4, and 9 of monitor.

Example 7:

See Example 1 of w_activity.

2.164 systime

This real standard procedure 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 or integer). 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 <= fnc <= 7 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 of: RC9000-10: 20
          milliseconds, RC8000: 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 :
          RC9000-10: 20 milliseconds, RC8000: 0.1
          milliseconds, but the limited accuracy of reals
          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 so the execution 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.
```

fnc = 5, time measuring (ISO date and clock)

```
systime   Becomes year*100 00 + month*100 + day.
          The year is taken modulo 100.
time      Base for real time measurement.
r         Becomes hour*100 00 + minute*100 + second.
          Fractions of a second are cut off.
```

This function works as systime (1,...); systime (4,..);

fnc = 6, transforms shortclock to decimal time

```
systime   Becomes year*100 00 + month*100 + day.
          The year is taken modulo 100.
time      Shortclock.
r         Becomes hour*100 00 + minute*100 + se cond.
          Time is given with an accuracy of 2 minutes.
```

fnc = 7, get shortclock

```
systime   Becomes shortclock.
time      Base for real time measurement.
r         Becomes shortclock.
```

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,base);
The program part to be timed;
cpu:= systime(1,base,time) - cpu;
comment complete timing before printing;
write(out,<<dddd.dd>,cpu,time);
```

If the time measured is short compared with the accuracy, you should compensate for the time spent by calling systime, and make a loop that executes the statement(s) several thousand times (here you must compensate for the loop). The cpu time will depend some what on the activities of other processes. The real time used is highly dependent on other processes.

In RC8000, 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 for the real numbers. An accuracy of 0.1 millisecond may be obtained by measuring relatively 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;
```

In RC9000-10, you may compensate for the limited accuracy by timing a loop over several thousand repetitions.

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:

```
80 05 28 22 53 37
```

The same can be obtained in this shorter version:

```
write (out,<< dd dd dd>, systime(5, 0, r),r);
```

Example 3: Print shortclock

```
write (out,<<dddddd.dddd>,
      systime(6,tail(6),r) + r/1 000 000);
```

will output the shortclock stored in tail(6).

Example 4: Get shortclock

```
tail(6): = systime(7,0,short);
```

2.165 tableindex

This integer standard identifier is used by all the character reading procedures when a non standard alphabet is selected. See intable.

The default value of tableindex is 0.

Example 1:

See Example 3 of readall.

2.166 tofrom

This standard procedure is intended for copying sets of data to one array field from another.

Call:

```
tofrom (to_field, from_field, size)
```

to_field	(return value, boolean, integer, long, real, double, or complex aray or zone record). The contents of from_field (see below) are copied into to_field. The copying starts with the halfword with index 1 and ends with the halfword with index size.
from_field	(call value, boolean, integer, long, real, double or complex array or zone record). The contents are copied into to_field. The copying starts with the halfword with index 1 and ends with the halfword with index size.
size	(call value, integer). The number of halfwords to be copied. Size must be >= 0.

The reference halfword of both to_field and from_field must be a right hand halfword, i.e. odd valued field variables should not be used to indicate the array parameter.

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 mod 4 > 1 then
    to_field.intf:= from_field.intf;
  if size mod 2 > 0 then
    to_field.bf:= from_field.bf;
  end;
```

The parameters are only evaluated once. For one dimensional arrays the copying by means of tofrom is faster than a repetitive loop when 8 or more double words are moved.

When executed on a CPU capable of doing the "move halfword" instruction, the copying is performed by this instruction, unless the two array fields overlap and data is moved towards higher addresses (cf. Example 1). Copying by means of the "move halfwords" instruction is even faster than any repetitive loop.

Example 1: Zerosetting an array

On account of the above shown equivalence a large array can be cleared (each element is set to the binary value zero) by setting the first double word to binary 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.

2.167 tofromchar

The procedure copies a number of characters from one array field, starting in a given character position, to another array field, starting in a given character position.

Call:

```
tofromchar    (to_field, to_pos, from_field, from_pos,
              chars);

to_field      (return value, array of any type, zone
              record). The target array field to which
              characters are copied.

to_pos        (call and return value, integer). At call
              the character position in the target array
              field where the first character should go,
              at return the character position after the
              last character copied.

from_field    (call value, array of any type or zone
              record). The source array field from where
              the characters are copied.

from_pos      (call and return value, integer). At call
              the character position in the source array
              field from where the first character should
              be taken, at return the character position
              after the last character copied
              (frompos+chars).

chars         (call value, integer). The number of
              characters to be copied. The number must be
              >= 0.
```

Character position

The character positions in an array field are numbered 1, 2, 3, ... Position no. 1 is the 8 most significant bits of the word with halfword index no. 1. If one of the start positions, to_pos and from_pos, or one of the end positions, to_pos + chars and from_pos + chars, exceeds the bounds of its array field, a field alarm occurs.

Copying

The reference halfword of both the to_field and the from_field must be a right hand halfword, i.e. odd field values should not be used to indicate the array fields (or an 'oddfield alarm occurs).

The procedure performs the copying by doing as many consecutive double word assignments as possible, in case the source and target

relative character positions within the word boundaries are the same, or double word load, single word store with one or two character shifts in between to outbalance the position differences.

The copying will go towards high or towards low addresses to prevent overwriting already copied characters.

The parameters are only evaluated once.

2.168 trap

This standard procedure assigns a value to the traplabel of an algol block, so that the program itself may control run time alarms.

Call:

```
trap (tlabel) or
trap (no)
```

```
tlabel    (call value, label). The value of tlabel, which
          defines a program point, is evaluated and
          assigned to the traplabel of the block, where
          the procedure is called. tlabel must be local
          to this block.
no        (call value, integer). When no = 0 the
          traplabel of the block, where the procedure
          call took place, is cleared. When no<>0 an
          alarm is provoked.
```

To each algol block (even an incarnation of a context block) is associated an anonymous label variable, called the traplabel of the block. At block entry this variable is cleared (in context blocks initialized as other context variables). Assignment of a label value to the traplabel is done by a call of trap(tlabel).
When an alarm occurs, the run time system proceeds as follows:

- Depending on the value of the standard variable trapmode, the ususal alarm message is printed as described in (15).

- Instead of terminating the program, the traplabel of the block where the alarm occured, is examined. If the value of the traplabel is not zero, a jump (goto) is made to the program point defined by that label.
  If the value of the traplabel is zero, the dynamically enclosing block is entered, and the traplabel of this block is examined as described above. If no non-zero label value is found in any active block, the program terminates as described in (15).

- When the program point defined by the traplabel is entered caused by an alarm, the traplabel is cleared - to prevent endless loops in traproutines. Finally the type of alarm is signalled in the standard variable: alarm_cause.

The traplabel of a block can be cleared by the call: trap(0).

When trap(no) is called, where no< >0, a run time alarm is provoked, with the text

```
trap <no>
```

If this alarm is trapped by a non-zero traplabel, the value of alarmcause is:

```
alarmcause = <no> shift 24 add (-13).
```

This call of trap is intended for use as termination of a traproutine in give up situations. The explicit value of alarm cause makes it possible for traproutines in dynamicly enclosing blocks to detect give up situations from traproutines in dynamicly inner blocks.

Example 1: tracing program flow

When debugging programs it may be convenient to trace the program flow at some points in the program. A procedure facilitating this might be:

```
procedure trace(n); integer n;
begin
   trap(finis); <*traplabel := finis*>
   trap(n);     <*provoke alarm*>
   finis:
end;
```

If the procedure is called somewhere in the program the error message will be

```
trap <n>     zone/trap
called from  line <*call point in procedure trace*>
called from  line <*call point in program*>
called from  ...
called from  ...
```

After this message the program will continue at the traplabel of the procedure trace, i.e. from the code following the label finis, and just return to the program.

Example 2: Time out of endless loops

If a program is executed as part of a Boss job (which will be the case in many online applications) the trapsystem supplies a possibility for writing pieces of code controlling endless loops of the program itself:

```
begin integer array mess(0:7);
...
  trap(timeout);
  mess(0) := 28 shift 12 + 3 shift 5;
  mess(6) := 30;
  mess(7) := 1000;
  system(10,0,mess);
    <*send timer message to Boss, expecting an     interrupt (break 8) after 30
seconds*>
  <*now the critical code is executed*>
  ...
  timeout: <*traproutine:*>
    cause := alarmcause extract 24;
    type  := alarmcause shift (-24);
```

```
        if cause = -9 and type = 8 then
          begin
            <*a parent break has occured, i.e. timer        break from Boss*>
            ...
          end else
          begin <*some other alarm occured*>
            trapmode := 1 shift 13;
              <*ignore trap message*>
            trap(1); <*give up*>
          end;
end;
```

Example 3: Traproutines in two levels

The following sketch illustrates the use in two block levels:

```
begin                                   alarms occuring here will cause a program
   <*declarations*>                     termination
   ...
   trap(alarm_1);
```

```
   ...                                  alarms occuring here will cause the program
   begin                                to continue at alarm_1
      <declarations>
      ...
      trap(alarm_2);
```

```
                                        alarms occuring here will cause the program
      ...                               to continue at alarm_2
```

```
      if false then                     alarms occuring here will cause the program
      begin                             to continue at alarm_1
alarm_2:
   <*traproutine2*>
      end
      ...
   end;
   ...
```

```
   if false then                        alarms occurring here will cause a program
   begin                                termination
alarm_1:
   <*traproutine1*>
   ...
end;
```

2.169 trapmode

This integer standard identifier is used in connection with suppression of errormessages.

The value of trapmode is a bitpattern, which may cause the printing of an error message to be ignored. To each value of alarm cause extract 24 corresponds a bit in trapmode, which is tested by the run time system, when an error message has to be printed, according to the following algorithm:

```
cause:= alarm_cause extract 24;
if cause >= 0 then cause := 0;
if trapmode shift cause extract 1 = 0 then
output error message;
```

The default value of trapmode is 0.

Example 1:

The statement:

```
trapmode := 1 shift 7 + 1 shift 10;
```

causes the printing of the standard error message for real overflow to be ignored. After normal program termination the message;

```
end <no of segments transferred>
```

will not appear in the output.

By means of trapmode it is possible to program your own alarm output routine.

Example 2:

See Example 2 in trap.

2.170 underflows

This integer standard identifier determines the action on floating point underflow:

```
underflows < 0
```

The execution 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 execution starts, underflows is 0.
A floating-point underflow occurs when a result gets closer to zero than 1.6'-617 without being zero exactly.

Example 1:

To check whether a real underflow occured during the evaluation of an expression, proceed as follows:

```
underflows:= 0;
Evaluate the expression;
if underflows > 0 then handle the underflow situation;
```

2.171 value

This delimiter, which is a specificator, is used in specifications.

Syntax:

```
value <identifier list>
```

Semantic:

The value specificator specifies, that the formal parameters mentioned in the identifier list are called by value cf. (14).
When a formal parameter of arithmetic type (integer, long, or real) is value specified, the actual parameter may be of any arithmetic type.

Example 1:

```
procedure p (a, b, c);
value       a, b, c ;
integer     a;
real           b;
long              c;
begin ... end;

<*all of the following procedure calls are legal*>
 p (1, 0.5, extend 2);
 p (0.5, extend 2, 1);
 p (extend 2, 1, 0.5);
```

2.172 virtual

This standard procedure connects a backing storage area to the calling program as a virtual storage like openvirtual, but with out writing back anything into the virtual storage area being disconnected (cf. openvirtual).

Call:

```
virtual (filename)

filename   (call value, string). A text string specifying
           the name of the backing storage area to be
           connected as virtual storage.
           In case of the empty string, the program file
           itself is connected as virtual storage.
```

Function:

The procedure works as openvirtual, except that the contents of own core and internal context descriptions are not transferred to the virtual storage area being disconnected. The procedure is useful when

- you don't have writeaccess to the program file
- the program file is a file of compressed programs
- the program is a FORTRAN program with DATA initialization of COMMON variables and/or ZONE COMMONS, and it should start over and over again without having to be recompiled.

On the other hand, it is not possible to restart the program connecting to the program file after a break down without a recompilation.

2.173 wactivity

This integer standard procedure waits for an event (message or answer) in the event queue, supplying the identification of the responsible activity.

Call:

```
w_activity (buf)
```

`w_activity`	`(return value, integer). The value designates the kind of the event received from the event queue (see below).`
`buf`	`(call and return value, integer). If the call value is 0, the event queue is scanned from the beginning, otherwise the value must be the address of a message buffer belonging to the calling process. In the latter case, the event queue is scanned from the event arriving after the given buffer. (cf. monitor 24, wait event). The return address is the address of the next buffer in the queue.`

Function:

The procedure waits for an event (message or answer) in the event queue, similar to the call monitor (24,...).

The procedure must be called in monitor mode, otherwise the run is terminated with an alarm.

The intended use of w_activity is scheduling activities concurrently executing i/o transfers.

See also monitor (66,...), test event.

Result value:

The return value of w_activity designates the kind of the event and the identification of the responsible activity:

-1: The event is an answer, and there is no activity at present waiting for that answer.

0: The event is a message, sent by a process (by send_message)

>0: The event is an answer, and the value of w_activity is an activity number. The designated activity is waiting for the answer just received, i.e. the activity is waiting in an implicit passivate statement.

Example 1:

The following procedure, using monitor (24,...) and system (12,...) is equivalent to w_activity:

```
integer procedure wait_activity (buf);
integer buf;
begin
  integer result, actno, buf;
  zone z(1, 1, stderror);
  integer array ia(1:13);

  result:= system (12,0,ia); <*activity descr*>
  if result = 0 then system (9,0, <:<10>mode:>);
  result:= monitor(24,z,buf,ia);
  actno:= ia(1);
  if result = 1 <*answer*> then
  begin

  if actno > 0 then
  begin
    system (12, actno, ia);
      if ia(1)= buf then result:= actno
      else result:= -1;
    end
    else result:= -1;
  end;
  wait_activity:= result;
end wait_activity;
```

Example 2:

See Example 1 of activity.

2.174 waittrans

This integer standard procedure awaits the arrival of a format 8000 transaction. On arrival the transaction head is input and converted to integers, which on return are assigned to return parameters of the procedure.

Call:

```
waittrans (z, format, destination, aux1, aux2)
```

wait_trans	(return value integer). The field_type of the first field of the transaction (see readfield).
z	(call and return value, zone). Specifies the document from which transactions are arriving.
format	(return value, integer). Defines the format of the transaction: 0: unknown format 1: read modified format 2: short read format 3: write format 4: read buffer format 5: read status format 6: connect format In communication with display terminals, the transaction format will always be 1, 2, or 4. The formats 0, 3, 5 and 6 may be actual in communication with other computers or other RC9000/RC8000 applications.
destination	(return value, integer). Designates the originator (sender) of the transaction, i.e. display terminal, computer, or RC9000/RC8000 application (see below).
aux1	(return value, integer). Depending on the format of the transaction, aux1, the AID code specifies the attention type, write command code, or is undefined (see below).
aux2	(return value, integer). Depending on the format of the transaction, aux2 specifies the cursorposition, write control character, status byte or is undefined (see below).

Function:

The orginator of the transaction is returned in destination as a pair of numbers (cu, dev);

```
cu shift 12 + device
```

where cu is the communication unit, and device is a display terminal, computer, or an RC9000/RC8000 application. cu and device are ISO

characters converted to integers in the range 0,1,2,....., 127. The routing of transactions and connection between integers and process names is done by the environment.

The interpretation of aux1 and aux2 depends on the transaction format as follows (cf.(7), (18) and appendix B):

Format 1,2 and 4:
aux1 is the attention key identification (AID) as shown in the table below.
aux2 is the cursor position defined as an integer in the range 0,1,..., 1919. The cursor position is a conversion of the 2-character ISO character representation of the cursor address. (aux2 is undefined when format = 2).

Format 3:
aux1 is the write command code (WCODE) and aux2 is the write control character (WCC).
WCODE defines the type of write operation as shown in the table below.
WCC is a bitpattern defining the modes of display terminal output operations by the following bits:
1 shift 0: Reset "Modified Data Tag" (MDT) bits
1 shift 1: Keyboard Restore
1 shift 2: Sound alarm

Format 5:
aux1 is the command byte STATUS and aux2 the status byte.

Format 6:
aux1 is the command byte CONNECT and aux2 is undefined.

Format 0:
aux1 and aux2 are undefined.

Format	aus1	aux2
No: Name:	Code: Name:	Value: Name:
0 illegal	- undefined	- undefined
1 read modified	1 SEND (129) 2 PF1 (130) 3 PF2 (131) . . . . 13 PF12 (141) 19 PF13 (147) 20 PF14 (148) 21 CURSEL (149) 42 MSR (170) 43 3278MSR (171) 45 PF15 (173) 54 PF24 (182)	0-1919 cursor pos.
2 short read	14 PA1 (142) 15 PA2 (143) 16 PA3 (144) 17 CLEAR (145) 22 PA4 (150) 23 PA5 (151) 24 USM (152) 27 RESET (155) 34 PA6 (162) 38 PA10 (166)	- undefined
3 write	49 WRT (177) 53 EWRT (181) 55 CPY (183) 63 EAJ (191) 50 RB (178) 54 RM (182) 62 RAA (190)	128+(0-63) WCC 128+(0-63) WCC 128+(0-63) CCC - undefined - undefined - undefined - undefined
4 read buffer	*18 PF13 (146) 32 NO AT (160) 33 NO AP (161)	0-1919 cursor pos.
5 read status	28 STATUS (156)	0-295 status byte
6 connect	29 CONNECT (157)	- undefined

* in special systems only

Zone state:

The zone state must be ready for character reading, or ready for waittrans (state 0, 1, 2, 10, or 11 i.e. since the latest call of open, only setposition, character reading or waittrans). On exit from waittrans, the character reading states 1 or 2 are changed to 9 + 1 or 9 + 2, ie., zone state = "after waittrans", to ensure that the next call is readfield or a new call of waittrans.

If the application using waittrans is a Boss job, the zone must be opened with mode kind = 10: open (z, 10, <name of input>, <give up>);

Alphabet:

The character input table used by the algol character reading procedures is changed, so that later calls of these procedures translate the characters: SBA, SF, IC, EUA, PT, RA and ETX to class = 8 and value = 25, causing an EM-reaction, when character reading procedures attempt exceeding a transaction field.
However, the procedure readchar may continue reading beyond this pseudo EM, in which case a field designator may be lost.

Example 1:

See example 1 of activity (the procedure input_link).

2.175 write

This integer standard procedure prints text, numbers, and single characters on a document or into an array. 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 as been
         encountered, otherwise write is positive.
z        (call and return value, zone or return value,
         array of type boolean, integer, long, real,
         double or complex).
         In case of a zone, it specifies the document,
         the buffering, and the position of the document
         cf. (15).
         In case of an array, the characters will be
         packed into the array, 3 to a word, starting in
         the word with lexicographical index 1.
source   (call value, string, integer, long, real,
         boolean, or array of type integer, long, real,
         double or complex or zone record).
         The source parameter specify what is to be
         printed.
```

Function:

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 described below.

If the parameter z is an array, the writing stops when the array is full and the remaining parameters are abandoned.

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 printed with an integer or long layout, i.e. no decimal point and no exponent part, will be rounded to the correct value, but notice that negative values are printed as a sign followed by the possibly rounded positive value.

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. If the boolean contains the bit pattern "true" cf. (14) the parameter pair is used as a fill parameter (see below). Otherwise the last 8 bits of the boolean pattern cf. (15) are printed as a character as many times as specified by the integer parameter. If the integer is <= 0, nothing is printed.

array:
An array of type integer, long, real, double or complex or a zone record is considered to contain a text stored as text portions, in an integer array possibly ending with half a text portion cf. (15).
Starting with the element with lexicographical index 1, cf. (14), the text portions are printed word by word, in lexicographical order until either a null character is met, or the upper index is reached. If the lexicographical lower index is greater than 1, the array parameter is considered to be an erroneous parameter - see below.

unknown type or type violation:
If a source parameter cannot be classified as above or rules of parameter types are violated, write will print one of the runtime warnings:

```
param <integer> write
called from...
```

or

```
index <integer> write
called from ...
```

or

```
string <integer> write
called from...
```

on current output zone. The <integer> will be

```
<parameter no>*100 + kind
```

kind is one of the kinds in (10), page 39.

Next, write will print the alarm text

```
<:<10>***write: param<10>:>
```

in the zone or array used as target for writing, drop the parameter and continue interpretation of its parameter list.

Writing into a zone:

Zone state:
The zone used must be open and ready for character printing (state 0 or 3, see 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 disc, a terminal, a tape punch, a line printer or a magnetic tape. In the latter case setposition(z,...) must have been called after open(z,...).

First character:
The first character is printed just after the logical position of the document.

Last character:
When write returns, current word is filled up with NULL characters (if not already filled up), its value will tell how many characters were written, disregarding the possible terminating fill characters, and the logical position of the zone points to just after the last character of current word.

The zone record is not available (it is of length 0).

Writing into an array:

Zone state:
Is irrelevant.

First character:
The first character is printed in the first position of the word with lexicographical index 1.

Last character:
When write returns, current word word is filled up with 'DEL' character or any other character set by replacechar (8, character) (if not already filled up) and its value will tell how many characters were written disregarding the possible terminating fill characters, but apart from that there is no logical position preserved.

Blocking:

When writing into an array, the array is the block, and when it is filled with characters, write returns abandoning further source parameters.

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 numer 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><exp.part><sign><fill>
```

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 sepa rated by a space in the finally printed number.

Layout constituents:

<spaces>:
Concist 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.

<sign> can be placed either in front of the <number part>, or in the end of the layout (before a possible <fill>). In the first case the sign of the number is printed as a leading sign in front of the first digit or first digit position depending on <first letter> of the <number part>.

When <sign> is placed in the end of the layout, the sign of the number is printed after the last digit in a position defined by <fill> and <first letter> of the <number part>.

<first letter> is z:
Digit positions preceding the first non-zero digit are printed as zeroes. A possible leading sign is printed in front of the first digit position, while a possible sign in the end is printed just after the last 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 zero es otherwise. A possible leading sign is printed just before the first digit printed, while a possible sign in the end is printed just after the last digit position.

<first letter> is f:
Digits are printed as for <first letter> = d. A possible leading sign is printed in front of the first digit position, while a possible sign in the end is printed just before the last fillposition.

<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) give the maximum number of printed significant digits. All numbers will be correctly rounded to the number of significant digits printed but notice, that negative values are printed as the sign followed by the possibly rounded positive value.

<zeroes>:
Consist of a (possibly empty) sequence of zeroes. If a non-empty exponent 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.

<exp.part> is empty:
No exponent is printed as the digit positions must be able to hold the digits of the number. Otherwise an alarm layout is used.

<exp.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, and the exponent <sign> cannot be placed in the end. If <first letter> is d or f and the tens exponent is 0, the entire exponent part is printed as spaces.

<fill>:

Consist of a (possible empty) sequence of spaces or _'s. They will appear as that may spaces in the end of the printed number, and the number is printed with a fixed number of printing positions, even in case of alarm printing (see below).

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 must not exceed 15. The number of digit position after the decimal must not exceed 15.

The number of digit positions in an exponent part must not exceed 3. The number of leading spaces plus the number of digit positions in front of the last space must not exceed 16.

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 are 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.

When the layout contains a non-empty <fill> of m positions, the first m-1 positions may be used for the alarm layout. If this is not enough, the number is cut off, and a termination star (*) is printed in the fill position.

Fill parameter:

The parameter pair (true, integer) has a special meaning: it defines the number of printing positions for the next parameter to be printed, depending on the type of the parameter:

If the next parameter is a string, the defined number of printing positions will be printed. If the string value is too long (i.e. there is not room for at least one ending space), it will be truncated - the last character being a star (*).

If it is shorter than the defined number of printing positions, it will be extended by ending spaces.

If the next parameter is a pair (boolean, integer), the pair (true, integer) will be ignored.

If the next parameter defines a number value, and the most recent layout contains ending spaces (see above), the pair (true, integer) is ignored.

If the next parameter defines a number value, and the most recent layout does not contain ending spaces (that includes the standard layouts), the pair (true, n) will print the number value using n printing positions. If more positions are needed, for the number and one ending space, the value is truncated, the last character being a star, (*) otherwise it is extended by ending spaces.

Replacing characters:

All special characters used in a layout or by a fill parameter may be changed by means of the procedure replacechar.

All characters output by write (and any other character printing procedure) may be changed by selecting an output character table (see outtable.)

Example 1:

```
write (out,<:<10>:>, "a", 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,
```

Example 3: Tabulation

```
write(out,true,100,string text, string text2);
```

will print text2 in position 100 and on, except if text is longer than 99 characters or contains new line characters. See also Example 1 of outtext.

Example 4:

Printing with ending sign and fill positions in layout:

The statement

```
for l:=1, 12, 123, 1234, 123456,
      -1, -12, -123, -1234, -123456 do
  write out, <:::>, string layout,l,<::<10>:>);
```

will print as shown below for various layouts:

```
<<ddd-   > <<zdd+   > <<fdd+   > <<ddd.d-   > <<ddd.000+   >
 :  1    :  :001+   :  :  1  + :  :  1.0    :  :  1.00 +   :
 : 12    :  :012+   :  : 12  + :  : 12.0    :  : 12.0  +   :
 :123    :  :123+   :  :123  + :  :123.0    :  :123.0  +   :
 :1234   :  :1234+  :  :1234 + :  :123.4'1   :  :  0.123'4+ :
 :123456 :  :123456*:  :123456*:  :123.5'3   :  : 12.3  '4+ :
 :  1-   :  :001-   :  :  1  - :  :  1.0-   :  :  1.0  -   :
 : 12-   :  :012-   :  : 12  - :  : 12.0-   :  : 12.0  -   :
 :123-   :  :123-   :  :123  - :  :123.0-   :  :123.   -   :
```

```
:1234-  :  :1234-  :  :1234 - :  :123.4'1- :  :  0.123'4- :
:123456*:  :123456*:  :123456*:  :123.5'3- :  : 12.3  '4- :
```

Example 5: Printing with fill parameter:

```
write (out,
  "nl", 1, true, 10, <:example:>   ,
  "nl", 1, true,  7, <:example:>   ,
  "nl", 1, true,  5, 12345         ,
  "nl", 1, true,  6, <<d>      , 26 ,
  "nl", 1, true,  2, "a"       , 7  ,
  "nl", 1, true,  7, "b"       , 2  ,
  "nl", 1, true,  2, <<d.d   >, 3.4,
  "nl", 1                          );
```

will print the following text:

```
:example    :
:exampl*:
: 123*:
:26    :
:a*:
:bb:
:3.4   :
```

2.176 writefield

This standard procedure outputs a field designator in a format 8000 transaction.

Call:

```
writefield (z, fieldtype, aux)
```

z	(call and return value, zone). Specifies the document to which the current format 8000 transaction is output.
fieldtype	(call value, integer). The type of the field designator to be output according to the table shown below.
aux	(call value, integer). The interpretation depends on fieldtype (see table below).

Zone state:

The zone must be in state 3 (ready for character output), and is left unchanged.

The connection between fieldtype and ux is as follows:

field type	command	ISO char	aux
1	SBA: Set buffer address	17	char position
2	SF : Start field	29	attribute char
3	IC : Insert cursor	19	not used
4	EUA: Erase unprotected to addr	18	char position
5	PT : Program tab	9	not used
6	RA : Repeat to addr	20	char shift 12 + charposition
7	ETX: End of text	3	not used
10	USM: Unsolicited Message	31	not used

Character position is an integer in the range 0,1,..., 1919, which is a conversion of the 2-character ISO character representation of the cursor address.

An attribute character, which contains a bitpattern, defines the start of a field and the characteristics for all character locations of the field. (The field ends at the nex attribute charac ter).

bit no		characteristic	
2	x shift 5	x= 0: unprotected,	1: protec ed
3	x shift 4	x= 0: alphanumeric,	1: numeric
4,5	x shift 2	x= 0: norm bright	
		x= 2: high bright	

```
                    x= 3: non-display
  7     x shift 0           Modified Data Tag (MDT);
                            identifies modified fields during
                            read modified generation:
                    x= 0: not modified,  1: modified.
```

For further information see (17), (18), and appendix B.

Example 1: Move cursor

The procedure moves the cursor to a specified line (0-23) and position (0-79):

```
procedure cursor (z, line, pos);
zone              z;
integer              line, pos ;
   writefield (z, 1, line*80 + pos);
```

Example 2:

The following procedure erases all fields, unprotected as well as protected, in the lines specified by the parameters, by writing spaces.

```
procedure erase (z, line1, line2);
zone             z;
integer             line1, line2 ;
begin
   writefield (z, 1, line1*80 + 0);
   <* the same as cursor (z, line1, 0)*>
   writefield (z, 6, 'sp' shift 12 + line2*80 + 79);
   writefield (z, 1, line1*80 + 0);
   <*now the lines are ready for building new fields*>
end;
```

Example 3:

The following procedure creates a screen picture, in which each line consists of a protected field with the text line <lineno> and an open alphanumeric field:

```
procedure setmask (z, dest);
zone z; integer dest;
begin
   opentrans (z, 3<*write*>, dest, 49<*write*>, 3);
   for line:= 0 step 1 until 23 do
   begin
      writefield (z, 1, line*80);   <*set buf.adr*>
      writefield (z, 2, 1 shift 5); <*set field prot.*>
      write (z, <:line:>, << dd>, line);
      writefield (z, 2, 0); <*set open field*>
      if line=0 then
         writefield (z, 3, 0); <*insert cursor*>
      writefield (z, 6, 'sp' shift 12 + line*80 + 79);
```

```
        <*clear rest of the line with sp-char*>
    end;
    closetrans (z);
end;
```

2.177 writeint

This integer standard procedure is primarily intended for printing entities which are held in integer or long variables, but which should be printed as decimal fractions e.g. amount of currency. The procedure works almost as write except that writeint refuses to print real variables and that longs and integers may be printed with a decimal point placed according to a layout.

Call:

```
writeint (z, one or more source parameters)
```

`writeint`	`(return value, integer). The absolute value of writeint gives the number of characters printed. Writeint is negative if a parameter error has been encountered, otherwise writeint is positive.`
`z`	`(call and return value, or return value, zone array of type boolean, integer, long, real, double or complex). In case of a zone, it specifies the document, the buffering, and the position of the document (see ref. 15). In case of an array, the characters will be packed into the array, 3 to a word, starting in the word with lexicographical index 1.`
`source`	`(call value, string, integer, long, boolean, or array of type integer, long, real, double or complex or a zone record). Specifies what is to be printed.`

Writeint evaluates its parameters as write. The parameters are handled as for write, except that the appearance of a real will be considered to be a parameter error, and integers and longs may be printed with a decimal point inserted, see below.

Zone state:

as for write.

Blocking:

as for write.

Layouts:

Layouts are written as described for write. If a layout with a decimal point appears in the parameter list, subsequent numbers (i.e. integers and longs) will be printed with a decimal point inserted. Consider a layout with "dec" d's or zeroes ap pearing to the right of the decimal

point. The subsequent numbers will then be printed as if they were devided by 10**dec.

Zeroes in layouts to writeint will be handled as d's. A possible exponent part will be printed as spaces.

Alarm printing:

If a negative number is printed without a sign position, a minus is inserted consuming one extra position.

If the layout contains too few d's, the necessary number of d's is inserted to the right. The decimal point is moved the same number of places.

Example 1:

In order to illustrate alarm printing, consider the following program:

```
begin long number, layout; integer k,l;
number := 0;
for l := 0 step 1 until 9 do
  begin
    number := -sgn(number) * (abs number * 10 + 1);
         k : = 18;
    for layout := long << zd dd.dd>,
        long <<+b ddd ddd.dd>,
        long <<-fd.ddd>,
        long <<-d.dd dd> do
    k:= k-18+writeint(out, string layout, "sp",
        18-k, <:,:>, number, <:,:>);
    write (out, "nl",1)
  end
end
```

which will produce the following output:

```
, 00 00.00,      ,            ,  , 0.000,       , 0.00.00,
, -00 00.01,     ,       -0.01,  ,-0.001,       ,-0.00 01,
, 00 00.12,      ,       +0.01,  , 0.012,       , 0.00 12,
, -00 01.23,     ,       -1.23,  ,-0.123,       ,-0.01 23,
, 00 12.34,      ,      +12.34,  , 1.234,       , 0.12 34,
, -01 23.45      ,      -123.45, ,-12.345,      ,-1.23 45
, 12 34.56,      ,    +1 234.56, , 123.456,     , 12.3 456,
, -12 345.67,    ,   -12 345.67, ,-1234.567,    ,-123 .4567,
, 12 3456.78,    ,  +123 456.78, , 12345.678,   , 123 4.5678,
, -12 34567.89,  ,-1 234 567.89, ,-123456.789,  ,-123 45.6789,
```

2.178 zone

This delimiter, which is a declarator, is used in declarations and specifications of zones and zone arrays. Details about input/output are given in [15].

Zone declaration:

```
zone <list of zone segments>;
```

A zone declaration 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 halfwords 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 or when an operation on a document is checked by a high level zone procedure cf. (15).

Zone array declaration:

```
Zone array <list of zone array declarations);
```

A zone array declaration declares one or more zone arrays. One zone array declaration is composed in this way:

n	(integer). The number of zones in the zone array.
buf	(integer). The number of buffer elements of 4 halfwords 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 each of the n zones.

Zone and zonearray 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.

Memory Allocation

If the job process has sufficient memory space beyond the address 1 M Halfwords, the high end partition, the entire buffer area and all the share descriptors of the zone, the list zones or the zones in the zonearray will be allocated there.

If there is not enough memory space in high end partition, then the active area will be allocated in the normal variable stack in the low end partition. (cf. the procedure system, entry 15).

Zone or zone arrays declared in activities, but outside disabled blocks, will always be allocated in the normal variable stack in low end partition.

Buffer length:

The buffer area of a zone may be divided in any way among the sh shares. The procedure 'open' will divide the buffer area evenly among the shares.

The buffer area of a zone array is divided evenly among the n zones. This distribution can be changed by a call of procedure initzones and reset by a call of the procedure resetzones.

The buffer area of a zone array is divided in another way by the procedure openinout and reset by the procedure closeinout.

Shares:

Each of the sh shares may be used for one uncompleted operation on a document or for one running process cf. (15).

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 ((15) contains hints about when to use 1, 2, or 3).

Zone state:

Just after the declaration of a zone the state becomes 4, and no document is connected to the zone. The zone record describes the entire buffer area, which has an undefined contents. All the shares are free and each of them describes the entire buffer area.

Zone record:

The zone record may be thougth of as a real array, the elements of the which are numbered 1, 2... record length.

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 ref;

      ref:=6;
      za1(1,1024):= za1(3).rf:= 0;
      open(za2(3),4 shift 12+18,<:mt123456:>,0);
```

A: References

Part numbers in references are subject to change as new editions are issued and are listed as an identification aid only. To order, use package number.

1 PN: 991 11255
RC9000-10 System Software
delivered as part of SW9910I-D, RC9000-10 System Overview
This title is equivalent to the RC8000 manual *Monitor Part 1, System Design* (PN: 991 03577)

2 PN: 991 11259
Monitor, Reference Manual
delivered as part of SW9890I-D, Monitor Manual Set
in the RC8000 documentation this was PN: 991 03588

3 PN: 991 03435
Monitor, Part 3, Definition of External Processes
The cited document is an RC8000 manual. In RC9000-10 systems equivalent manuals are contained in the SW9890I-D, Monitor Manual Set.

5 PN: 991 04162
RC8000 Computer Family, Reference Manual

6 PN: 991 11263
System Utility Programs, Part 1
PN: 991 11264
System Utility Programs, Part 2
These two titles are delivered with SW8010I-D, System Utility Manual Set
PN: 991 11294
System Utility Programs, Part 3
This title is delivered with SW8585-D, Compiler Collection Manual Set

7 PN: 991 11274
BOSS User's Guide
delivered as part of SW8101I-D, RC9000-10 BOSS Manual Set

8 PN: 991 11260
Operating System s, Reference Manual
delivered as part of SW9890I-D, Monitor Manual Set.
In the RC8000 documentation, this manual was PN: 991 03581

9 PN: 991 11292
RC Fortran, User's Manual
delivered as part of SW8585-D, Compiler Collection Manual Set

10 PN: 991 11296
Code Procedures and The Runtime Organisation of ALGOL Programs
this document is not included in any package, but is available on request.

11 R.M. De Morgan et al.:
Modified Report on the Algorithmic Language Algol 60.
The computer Journal, Vol 19, no. 4 pp 364-379.

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

13 ISO: R646 - 1967 (E),
6 and 7 bit coded character set for information processing interchange.

14 PN: 991 11278
ALGOL8, Reference Manual
delivered as part of SW8585-D, Compiler Collection Manual Set

15 PN: 991 11279
ALGOL8, User's Guide, Part 1
delivered as part of SW8585-D, Compiler Collection Manual Set

16 PN: 991 11288
Mathematical and Statistical Routines, Reference Manual
delivered as part of SW8585-D, Compiler Collection Manual Set

17 PN: 990 00168
Format8000 Display System

18 *IBM 3270 Information, Display systems, component description*
GA 27-2749-4

20 PN: 991 11277
Parent Messages in BOSS
delivered as part of SW8101I-D, RC9000-10 BOSS Manual Set

21 PN: 990 00469
A Coroutine System Written in ALGOL8

22 PN: 991 11290
ALGOL Coroutine System, User's Guide
delivered as part of SW8585-D, Compiler Collection Manual Set

23 PN: 991 04985
RC855 IBM 3270 BSC Emulator, Reference Manual

24 PN: 991 11262
LAN Device Processes, Reference Manual
delivered as part of SW9890I-D, Monitor Manual Set

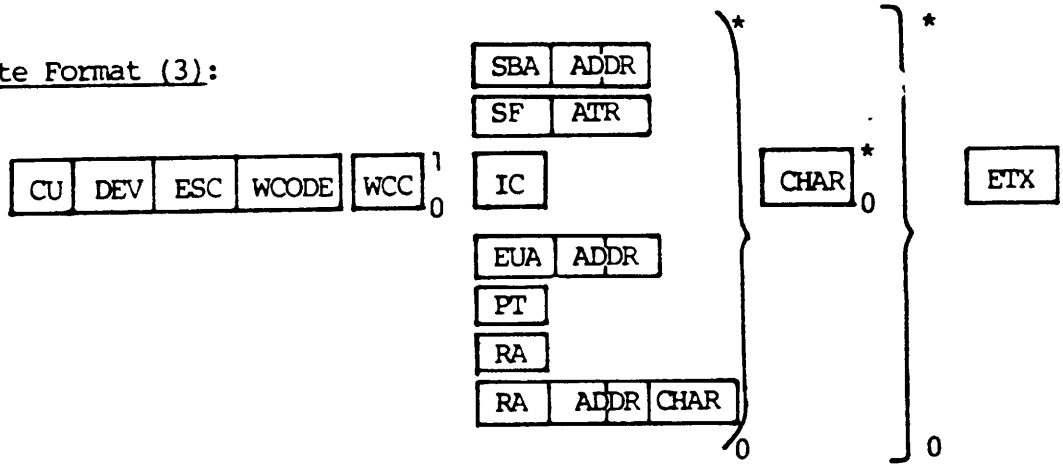
B: Survey Of Format8000 Transactions

Read Modified Format (1):

Short Read Format (2):

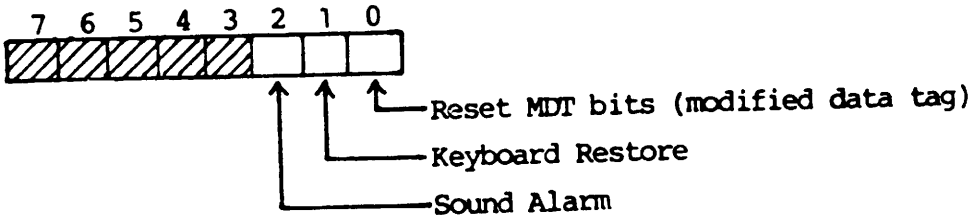
```
CU  DEV  AID  ETX
```

Write Format (3):

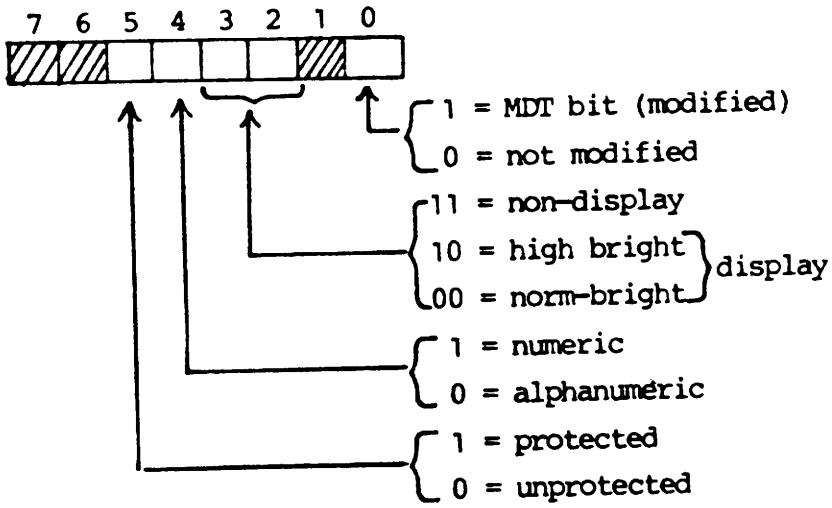


(SBA must be the first command)

Write Control Character (WCC)



Attribute Character (ATR):



Field Type (in readfield/writefield)

Command	type	char	meaning
SBA	1	17	set buffer address
SF	2	29	start field
IC	3	19	insert cursor
EUA	4	18	erase unproctected to addr
PT	5	9	program tab
RA	6	20	repeat to addr
ETX	7	3	end of text
data	8	-	
EM	9	25	end medium
USM	10	31	unsolicited message
ESC	-	27	escape

Attention Ident (AID):

Name	type	char	format
SEND	1	129	read modified
PF1	2	130	
PF2	3	131	
PF3	4	132	
.	.	.	
.	.	.	
.	.	.	
PF12	13	141	
PF13	19	147	
PF14	20	148	
CURSEL	21	149	
MSR	42	170	
3278MSR	43	171	
PF15	45	173	
.	.	.	
.	.	.	
.	.	.	
PF24	54	182	
PA1	14	142	short read
PA2	15	143	
PA3	16	144	
CLEAR	17	145	
PA4	22	150	
PA5	23	151	
USM	24	152	
RESET	27	155	
PA6	34	162	
.	.	.	
.	.	.	
.	.	.	
PA10	38	166	
*PF13	18	146	read buffer
NO AT	32	160	
NO AP	33	161	
STATUS	28	156	read status
CONNECT	29	157	connect

Write Command (WCODE):

Command	char	format
write	49	write
erase/write	53	
copy	55	
erase unprotected	63	
read buffer	50	
read modified	54	
read modified all	62	