RC9000-10/RC8000

SW8585 Compiler Collection

ALGOL8 User's Guide, Part 1

PN 99200389

RC Computer

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

Abstract:
This manual is describes the syntax etc. of the ALGOL compiler for RC9000-10 and RC8000 systems.

Date:
March 1989.

PN: 991 11279

Table Of Contents

Foreword

This edition of the manual is an update to the present state of the ALGOL compiler, ALGOL/FORTRAN runtime system and ALGOL library. Throughout this manual, anything stated about the ALGOL/FORTRAN runtime system and about standard identifiers and procedures in the ALGOL library concern FORTRAN programs as well as ALGOL programs. This means that the chapters 2, 5 (the procedure parts, not the operators), 6, 8, 9 and most of chapter 10 are all proper reading from a FORTRAN user's point-of-view. The extensions of ALGOL8 compared to ALGOL7 concern:

- the activity concept allowing procedures to act as coroutines with an option for concurrent i/o transfers
- format8000 procedures for IBM 3270 compatible transaction processing
- new layout possibilities
- character constants

and a few minor changes.

On top of the ALGOL 8 extensions come the extensions and changes of version 2 of ALGOL, which in highlights are:

- same linker in ALGOL and FORTRAN, making it possible to link into both ALGOL and FORTRAN programs any of the external program units: ALGOL external procedures, FORTRAN subroutines and functions and code procedures made for either ALGOL or FORTRAN use

- same external runtime system to both compilers, linked in at compile time as any other external program unit

- allocation of an extra stack, the high end partition, in address space beyond 1 M halfwords (if available) to contain zone buffer areas and free memory for program segments, leaving other variables to the to the low end partition along with more free memory for program segments

- renewed procedures "read" and "write", among other things making it possible to read and write from or into arrays of any kind instead of just zones, making the way for zone specific conversion tables rather than all-zone block specific conversion tables

- giving parameter alarms on current output as well as on the actual zone

- new record in- and output procedures doing multibuffered input-in-one-and-output-in-many zones, eliminating the need of moving data from the input buffer to the output buffer(s)

- the possibility to "compress" algol and fortran programs into libraries, e.g. by the program "lib", the program "compress" or any other program compressing files and describing them by segment offsets, and execute the "compressed" programs described by auxiliary entries

- procedures in algollib, which used to have a formal parameter specified of type "real array" will now accept arrays of any arithmetic type (long, real, double real, complex and sometimes even integer and boolean), e.g. "movestring", "write", "read", "outvar"....

- procedures in algollib with a parameter array will all consider the array starting in the halfword with halfword index 1 in other words, fielding will now work in all such procedures, e.g. "system", "monitor", "read", "get/setzone" and "get/setshare"

RC Computer A/S, February 1989

1. Introduction

1.1 How to Use the Manual

The scope of this manual is to give a description of all elements in ALGOL8 and a detailed functional description of some elements which are not in ALGOL60.

Along the way, elements shared with RC FORTRAN are pointed out.

The syntactical definition of the language ALGOL8 is given i [14]. The syntactical definition of the language RC FORTRAN is given in [9].

Chapter 2 describes the input/output system of ALGOL8, which is shared with RC FORTRAN, and should be read by anyone who wants to use either of the two languages.

Chapters 3 and 4 are special to ALGOL8.

Chapter 3 describes the FIELD concept of ALGOL8, and is mandatory reading for ALGOL8 users.

Chapter 4 describes CONTEXT BLOCKS and is necessary if you are interested in concepts like multiple incarnations of blocks of code and associated data (variables), reentrant blocks, incarnations of records together with the associated field processing code, virtual memory, survival of variables, etc. useful in connection with e.g. multiple terminal handling, coroutine programming, program restart etc.

Chapter 5 describes the tools for text handling in ALGOL8 and RC FORTRAN.

Chapter 6 gives a list of mathematical procedures available to both languages and chapter 7 gives a list of available operators and standard procedures working as operators in ALGOL8.

Chapter 8 gives a list and a short description of ALGOL8 elements not mentioned in the other chapters.

Chapter 9 shows an example of how ALGOL8 can be used to code an operating system. This chapter implies knowledge of chapter 2.

Chapter 10 gives a description of how the programs are structured, translated and executed. A list of error messages is included.

Chapters 8, 9 and most of 10 describe elements shared with RC FORTRAN.

2. Input/Output System

This chapter describes the use of zones for input/output to any document (device).

An exotic use of zones is programming an operating system in ALGOL. This is described in chapter 9.

A zone array is a group of zones with identical declarations. The following details refer to a zone or to a zone within a zone array.

The ALGOL system contains a set of standard procedures "the high level zone procedures" which take care of normal input/output functions. The "primitive level" zone procedures are described in chapter 9.

The description of the input/output system is fully adaptable to FORTRAN programs with the remarks in 5.5 concerning text handling.

2.1 High Level Zone Procedures

The high level zone procedures are logically split into three groups:

- control procedures
- character procedures
- record procedures

See further description of the mentioned procedures in [15].

2.1.1 Zone Control Procedures

The control procedures will take care of connecting, releasing and positioning the document (cf. 2.3.2) by means of a zone. The following procedures are available:

open
Moves the documentname to the zone, and divides the buffer area into shares of equal size. Specifies how the handling of the document is performed (mode, kind).

close
Terminates the current use of a zone, including the emptying of output buffers and possible releasing of the document.

setposition
Terminates the current use of a zone including emptying of output buffers. A magnetic tape or a backing storage area is then positioned to the file and block specified. The positioning takes no time on a backing storage area, but it may involve tape moving operations for a magnetic tape.

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

A number of more special zone control procedures are available too: *stopzone, initzones, resetzones, openinout, expellinout, closeinout*, cf. [15].

Example 2-1, control procedures

```
<*write something on file 3 on a magnetic tape*>
begin
   zone z(200,1,stderror);
   <*  open will tell the operating system that this program will use the tape
       471100 in connection with zone z *>
   open(z,18,<:mt471100:>,0);
   <*  on magnetic tape a setposition must be perfor-med after open *>
   setposition(z,3,0); <* position to file 3 *>
   write(z,<:something --- :>);
      write(z,<:lastword:>);
   <* empty buffers and terminate *>
   close(z,true); <* suspend tape *>
end
```

2.1.2 Character Input/Output Procedures

The character input/output procedures are used for text handling input/output where the zones are used as buffers.

A conversion is performed for numbers. They are read as decimal numbers and converted to binary numbers, in 'write' the reverse conversion takes place i.e. binary number to decimal number.

read
Inputs a sequence of numbers given in character form on a document (or in an array), converts them to binary numbers, and assigns them to variables.

readchar
Inputs one non-blind character from a document, and supplies the character value and character class.

readstring
Inputs a text string given as characters on a document (or in an array).

readall
Inputs a mixture of numbers in character form, single characters, and text strings from a document (or in an array).

repeatchar
Makes the latest character read from the document available for reading once more.

intable
Exchanges the current input alphabet with an alphabet specified in the program.

tableindex
Used in connection with intable to define the alphabet.

write
Prints texts, numbers, and single characters on a document (or into an array).

writeint
Prints texts, integer numbers, and single characters on a document (or into an array).

The range of character input/output procedures also includes the procedures *outchar, outdate, outinteger, outtext* and the FORMAT8000 input/output procedures *waittrans, readfield, opentrans, writefield* and *closetrans*, cf. [15].

Example 2-2, read with test on terminator.

Program Part:

```
integer number, term;
integer array a(1:3);
<* read number,a(1),a(2),a(3) and terminator*>
read (in,number,a);
<* get terminator *>
repeatchar(in);
<* read terminator once more *>
readchar(in,term);
```

Data: `47, 18p30x 4;5`

Result:

```
number contains: 47
a(1) contains: 18
a(2) contains: 30
a(3) contains: 4
term contains: 59 (59 is the ISO value of ";")
```

Example 2-3, skip characters until digit occurs.

Program Part:

```
integer class, number;
<* read a number skipping all leading nondigit
characters *>
<* read all leading nondigits *>
repeat class := readchar(z,number) until class=2;
<* repeat first digit *>
repeatchar(z);
read(z,number);
```

Data: a+b=487

Result:

```
number contains: 487
```

Example 2-4, read string, with use of intable.

```
begin
   integer array alphabet(0:255);
   long array arr(1:5); integer j;
   <* set space to be a text part class 6 *>
   isotable(alphabet);
   alphabet('sp'):=6 shift 12+'sp'
   alphabet( ... <* the rest of the alphabet is set
                              to standard values cf. intable example, [15] *>
   intable(alphabet)
   <* now we have a new alphabet *>
   j:= readstring(in,arr,2);
end
```

Data: ,,,,,,peter brown

Result:

```
j         contains: 2
arr(2)    contains: peter
arr(3)    contains: brown
```

Normally space is a terminator and the result will be:

```
j         contains: 1
arr(2)    contains: peter
```

Example 2-5, readall

Program Part:

```
<*   read numbe s, strings and delimiters into integer arrays ia and kind, ia
     will contain the numbers, strings or delimiters, kind will contain the
     class of the items stored in ia *>
integer i;
integer array ia, kind(1:20);
i:= readall(in,ia,kind,1);
```

Data: `ab:a1.2c, 17.56 12345678<NL>`

Result:

```
i contains: 12

Index  1 2 3 4 5 6 7 8 9     10 11      12 ia   |ab
|0|58|a1.|2c|44|32|32|18   |32 |great |10 kind | 6 |6| 7| 6 | 6| 7| 7| 7| 2   |
7 |1      | 8 data ab     : a1. 2c  , SP SP 17.56 SP 12345678 NL

'great' indicates the greatest positive integer number.
```

Example 2-6, write

Program Part:

```
i:= write(out,la);
write(out,"sp",18-i,
<:age:>,<<ddd>,la(4),"sp"2,
<:kilos:>,<<ddd.d>,la(5)/1000,
"sp",2,string la(6));
```

Data:

la(1:3) contains:	peter brown
la(4) contains:	36
la(5) contains:	81350
la(6) contains:	m

Result:

```
peter brown    age 36  kilos  81.4 m
```

2.1.2.1 Standard Zones In and Out

Two standard zones, 'in' and 'out', exist. 'in' is used for input on character level, 'out' is used for output on character level. The documents connected to 'in' and 'out' are determined when the program execution starts, i.e. in the FP commands before calling the program.

The zones are declared with a buffer length of 512 halfwords and one share. The blockprocedure is stderror. These standard zones are stored in the top of the memory area occupied by the job.

Example 2-7, this example will show the use of FP commmands to direct 'in' and 'out' (cf. [6]).

Note: this job is executed under the operating system BOSS. Input from a magnetic tape file1 and a backing storage area data1. Output to a convert area convertout. The program name is exampl.

```
convertout=set 1 ; file1=set mth mt123456 0  create convert area
1 ;                                          file describes file1 on magtape
                                             mt123456
o convertout ;                               direct output to convertout
                                             call of program the program will
examp1 file1 ;                             ; write on convertout and connect
                                             input to magtape the program will
                                             read from file1
                                             connect input to the area data1
                          ;                  the program will read from data1
                                             redirect output to
examp1 data1 ;                               console/terminal
                                             will print the convertout on the
                          ;                  printer
                                             end BOSS job
o c                       ;

convert convertout ;


finis                     ;
```

2.1.3 Record Input/Output Procedures

These procedures perform input/output at record level. The zones are used as buffers, but only one record is available for processing at a time.

inrec6
Gets a sequence of the requested number of halfwords from a document and makes them available as a zone record.

outrec6
Creates a zone record with the requested number of halfwords. The contents is initially undefined. The program may then assign values to the record variables, the record will later be output to the document.

swoprec6
Gets a sequence of the requested number of halfwords from a backing storage area and makes them available as a zone record. The program may then modify the record, which is later transferred back to the backing storage area.

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

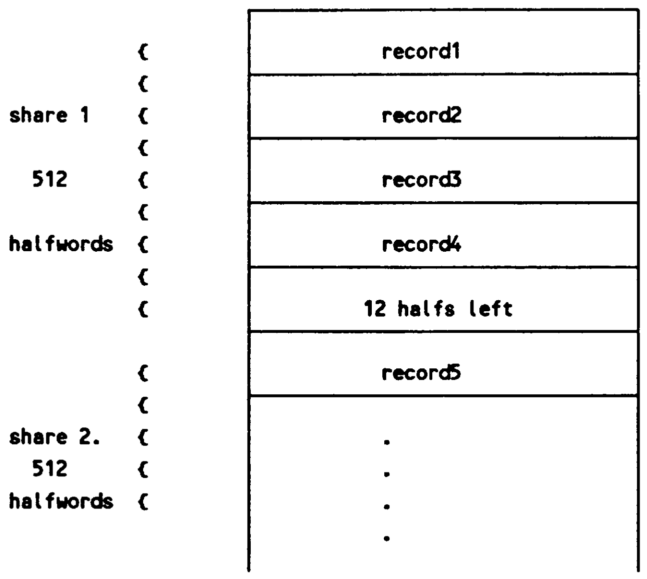
The set of record input/output procedures also includes the procedures *inoutrec* and *changerecio*, cf. [14].

Example 2-8 inrec6 and outrec6.

```
begin
   integer i;
   <* move 10 records from area 'old' to area 'new' *>
   zone zin, zout(128,1,stderror);
   open(zin, 4<:old:>,0);
   open(zout,4,<:new>,0);
   <* open to both areas 'old' and 'new' *>
   for i:=1 step 1 until 10 do
   begin
      inrec6(zin,100);
      <* reads a record from 'old' with 100 halfwords *>
      outrec6(zout,100);
      <* make space for record with 100 halfwords *>
      tofrom(zout,zin,100);
      <* move 100 halfwords to zout from zin *>
   end
   close(zin, true);
   close(zout,true);
end
```

Example 2-9, inrec6, change of share (block).

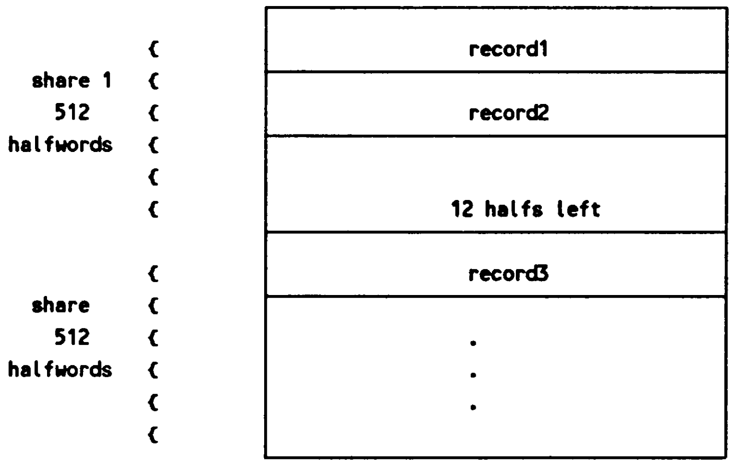
```
begin
   zone z(2*128,2,stderror);
   <* 2 shares each of 128*4=512 halfwords *>
   open(z,4 <:mine:>,0);
   inrec6(z,100);
   <* record1 length 100 *>
   ... <* use the record *>
   inrec6(z,90);
   <* record2 length 90 *>
   ...
   inrec6(z,110);
   <* record3 length 110 *>
   ...
   inrec6(z,200);
   <* record4 length 200 *>
   ...
   inrec6(z,70);
   <*only 12 halfwords left, therefore the next share is used.*>
   <* record5 length 70 *>
   ...
end
```



After inrec6(z,4*n) elements z(1) to z(n) may be addressed either by indexing or by fielding.

Example 2-10, outrec6, change of share (block).

```
begin
   zone z(2*128,2,stderror);
   open (z,4<:new:>,0);
   outrec6(z,200); <* fill(z,200); 1 *>
   outrec6(z,300); <* fill(z,300); 2*>
   outrec6(z,100); <* fill(z,100); 3 *>
   ...
end
```



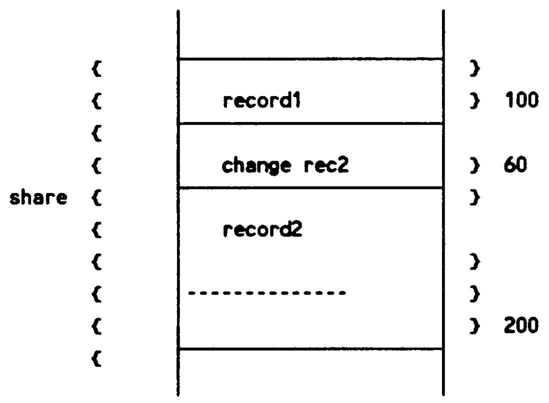
after outrec6(z,4*n) elements z(1) .... z(n) may be stored.

Example 2-11, changerec6.

Program Part:

```
outrec6(z,100);
<* make space for 100 halfwords *>
...
outrec6(z,200);
...
...
<* now we regret last outrec6 and want it replaced by outrec6(z,60) *>
changerec6(z,60);
```

Example 2-11, changerec6 (continued).



invar
Gets a sequence of halfwords from a document as for inrec6, but the number of halfwords is given as the first word in the record. The checksum stored in the second word may be checked.

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

changevar
Changes the length of a record generated by means of outvar. The checksum is computed.

checkvar
Generates a checksum in a record.

Example 2-12, invar, outvar.

```
<* copy 100 records from the area bsfile1 to area bsfile2 *>
begin
   integer i;
   zone zin, zout(128,1,stderror);
   <* open to both areas. Note that setposition is not required *>
   open(zin,4 <:bsfile1:>,0);
   open(zout,4 <:bsfile2:>,0);
   for i:= 1 step 1 until 100 do
   <* copy 100 records *>
   begin
      invar(zin);
      outvar(zout,zin);
   end;
   <* empty buffers and terminate run *>
   close(zin,true);
   close(zout,true);
end
```

Example 2-13, invar

```
<* similar to example 2-9 (inrec) *>
begin
   integer field length, checksum;
   zone z(128*2,2,stderror);
   length:= 2;
   checksum:= 4;
   <* record length and record checksum are found in first record element *>
   open(z,4,<:mine:>,0);
   invar(z); <* 1 use (z,z.length); *>
   ...
   invar(z); <* 2 use (z,z.length); *>
   ...
   invar(z); <* 3 use (z,z.length); *>
   ...
   invar(z); <* 4 use (z,z.length); *>
   ...
   invar(z); <* 5 use (z,z.length); *>
   ...
```

Example 2-14, outvar.

```
<* similar to example 2-10 (outrec) *>
begin
   integer field length;
   zone z(128*2,2,stderror);
   array a(1:100);
   length:= 2;
   open(z,4,<:mine:>,0);
   a.length:= 200; <* assign to the array *>
   outvar(z,a);
   a.length:= 300; <* assign to the array *>
   outvar(z,a);
   a.length:= 100; <* assign to the array *>
   outvar(z,a);
   .
   .
end
```

Example 2-15, changevar.

```
begin
   integer field length;
   array a(1:100);
   zone z(128,1,stderror);
   length:= 2;
   a.length:= 40;
   outvar(z,a);
   ...
   ...
   <* now we want to change the length of the above record *>
   a.length:= 60;
   changevar(z,a);
   <* NB: the checksum is recalculated *>
   .
   .
end
```

changevar is only allowed after outvar, since it would be senseless after invar.

Example 2-16, checkvar.

```
begin
   integer field length, count;
   array a(1:100);
   zone z(128,1,stderror);
   length:= 2; count:= 10;
   a.length:= 40;
   outvar(z,a);
   <* now the checksum is created *>
   ..
   ..
   2.count:= 2.count+1
   <* now the checksum does not correspond to the record *>
   checkvar (z);
   <* now it does *>
   .
   .
end
```

2.2 Zone Elements

Every zone consists of

a **buffer area** or **zone buffer**
a **zone descriptor**
one or more **share descriptors** or **shares**.

2.2.1 Buffer Area

The length of the buffer area is specified with the zone declaration. The length is in reals (each real is 4 halfwords). This buffer can be used as a real array with

```
1 =< subscript =< length
```

Such a subscripted variable is called a record variable.

The **zone record** defines which part of the buffer area is available for subscription.

The buffer area is split up into **shares** usually of equal length.

2.2.2 Zone Descriptor

A zone descriptor consists of the following set of quantities, which specify a process or a document (cf. [3]) connected to the zone, and the state of this connection:

		Set by procedure
process name	A text string specifying the name of a process or a document connected to the zone.	open
mode and kind	An integer specifying mode and kind for a documnet (cf. open).	open
logical position	A set of integers specifying the current position of a document.	open
give up mask	An integer containing ones in positions where a resulting status bit should call the block procedure instead of the standard error action.	open
free parameter	Used by FORTRAN read/write, the varprocedures and the inout procedures.	when declared
state	An integer specifying the latest operation on the zone. (cf. getzone6).	
record	Two integers specifying the part of the buffer area	

	nominated as the zone record (cf. 2.2).	
used share	An integer specifying the share descriptor within the zone, which is used at the moment.	
last halfword	An integer specifying the end of a physical block on a document.	
block procedure	This procedure is third parameter in the zone declaration. The procedure is called when a hard error occurs on the document (standard error actions gave up) or if a status is received with bits corresponding to a bit set in the give up mask.	when dec

2.2.2.1 Zone Record

A number of consecutive halfwords of the buffer area may at run time be nominated as the zone record. The halfwords of the zone record may be available as **record variables**, which may be thought of as a kind of real subscripted variables.

The record variables are numered 1, 2..., record length. All halfwords of the record may be referenced by means of field references, as the zone may be used as a field base.

The position and length of the zone will be specified by the record input/output procedures.

When the character input/output procedures are used the zone record will be a **block**, which is the actual amount of information transferred to or from the share in one operation.

2.2.3 Share Descriptors or Shares

A zone buffer is split into a number of shares, each one described by a share descriptor. The number of shares is specified in the zone declaration.

A share descriptor consists of a set of quantities which describe a transfer of data between a document (process) and a part of the buffer area.

The set of quantities forming one share descriptor is:

share state
An integer describing the kind of activity going on in the shared area (cf. getshare6, [15]).

shared area
Two integers specifying the part of the buffer area shared with the process controlling the document.

operation
Specifies the latest operation performed by means of the share descriptor (cf. [3]).

The amount of information transferred to or from a share in one operation is called a block.

On a magnetic tape, a block is a physical block or a tape mark. On a backing storage area, a block is one or more segments. On a paper tape reader, a block is usually one share of characters. The advantage of splitting the zone buffer into shares is, that one or more shares can be used for input/output while another is used for processing.

In case of single share input/output the program has to wait for a data transfer before the processing may continue.

For the choice of number of shares cf. 2.6.2.

2.2.4 Zone after Declaration

When a zone is declared, the following parts of the zone descriptor are defined:

free parameter
Set to 0 (binary zero).

used share
Is the first share, which contains the entire zone buffer area.

zone record
Is the entire buffer area as one record. The entire buffer can be accessed as a real array, where the elements are numbered 1, 2, 3, ..., length declared. The zone record can be used as a real array everywhere in the program (e.g. in a procedure call).

block procedure
The block procedure is defined but will not have any practical effect before a high level zone procedure is called.

Example 2-17, after declaration.

```
begin
   integer i;
   zone z(128,1,stderror);
   comment now it is possible to use the whole zone buffer area or the zone
   record as a real array z(1:128);
   for i:= 1 step 1 until 128 do
      z(i):= i;
   end
```

2.2.5 Zone after Open

Opening a zone with the high level procedure (cf. open)

```
                             ( <string> )
open(<zone>,<mode_kind>,(  <array> ),<give up mask>)
```

will cause the following zone description changes. (cf. 2.2.2).

process name
Is the name specified in the call of open, for a document, which should be connected to the zone.

mode and kind
Is set to modekind from the call. Specifying the mode and kind for the document (cf. open). Kind is as a rule only used in connection with error handling of the document. Mode specifies how to input or output from the document (e.g. even or odd parity).

logical position
Just before the first element of block 0 file 0.

give up mask
Is set to give up mask from the call, and used to specify which status bits, the block procedure should take care of. Normally give up mask is 0, meaning that the block procedure is only called if harderror (1 shift 0) is detected in the status word, i.e. standard error actions gave up (cf. 2.3).

zone state
Is set to 'after open'.

shares
The buffer area is divided evenly among the shares.

The zone is now ready for high level input/output to the specified document.

Note: the document is not connected physically (i.e. the entry on a backing storage is not looked up). Therefore a call of open to a non existing document will not cause an error, but the first call of a high level input/output procedure will show the document as disconnected.

Zone record.
There is no elements in the zone record. A reference to a zone variable will cause an alarm: index, field, or the like.

A high level input/output procedure should be called before a zone record is available.

Example 2-18, zone after open.

```
begin
  zone(128,1,stederror);
  <*  the zone is declared and the zone record is the entire buffer area z(1)
```

```
          to z(128) *>
      open(z,4.<:bsfile:>,0);
      <*  now the zone is opened to the backing storage area bsfile, no zone
          record exists *>
      invar(z);
      <*  now the first record from bsfile is available in the zone record z(1)
          to z(record length) *>
      <*  the first segment of the file is transferred to the share (= the entire
          buffer) *>
      .
      .
      .
      close(z,true);
      <*  now the zone is no longer connected to the bsfile, situation as after
          declaration *>
end;
```

2.3 Error Handling/Checking

Any input/output operation to a document will result in an answer to the program containing a 24 bit logical status word. This status word will describe the success of the operation. If an error occurs (reflected in the status word) a check routine in the runtime system will perform a standard error action depending on the bits set in the status word and the give up mask.

A list of the normal meaning of the status bits is given in 2.3.3.

2.3.1 Block Procedure and Give Up Mask

Call:

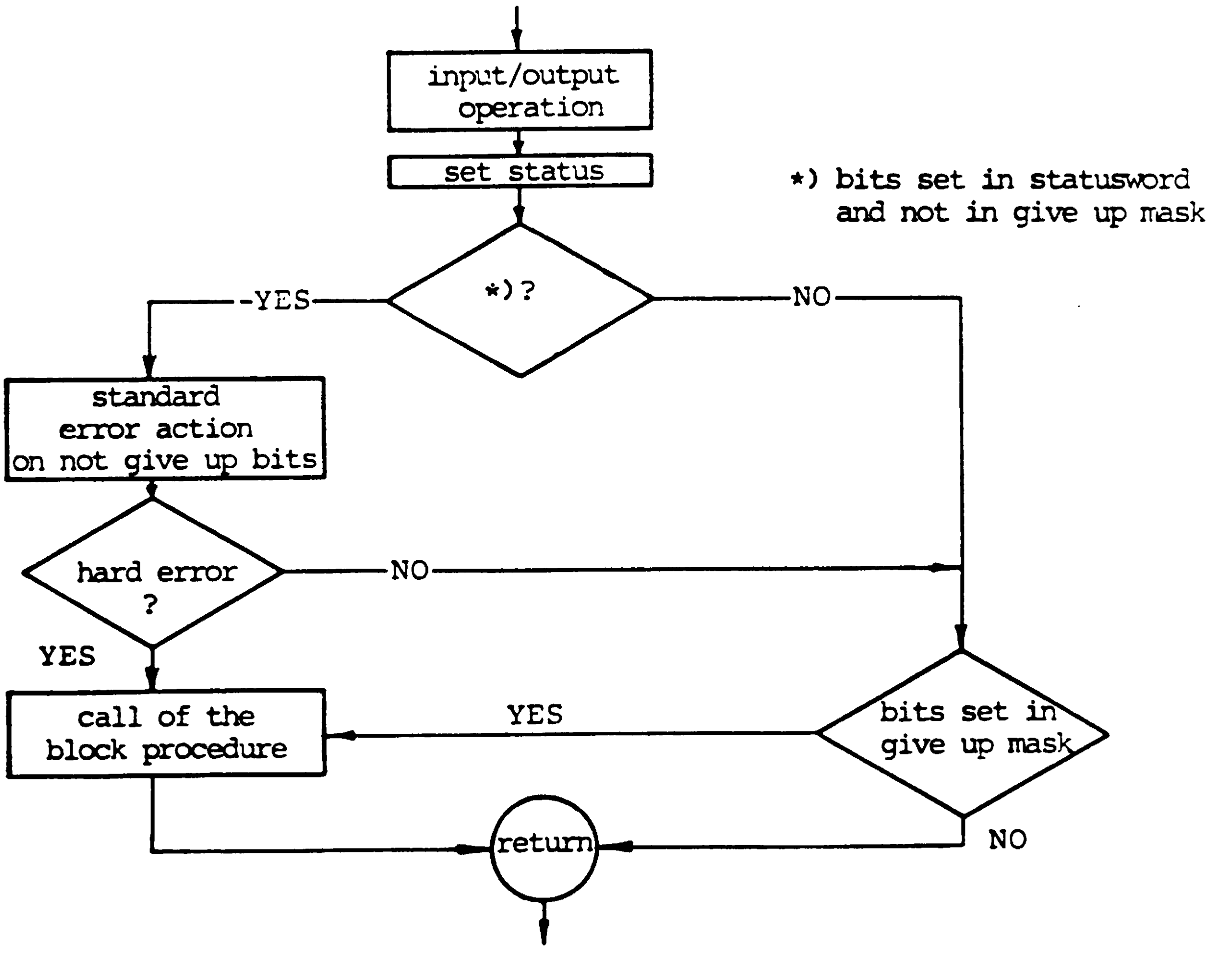
A call of the block procedure specified in the zone declaration will take place in the following cases:

1. When some of the bits set in the give up mask occured in the logical status word (user bits).

2. When the standard error action classifies the situation as a hard error (gives up).

You can tell the difference between the call reasons by mean of the last bit in the logical status word (hard error or give up bit).

The following figure should explain when the block procedure is called.


input/output operation
set status
*) bits set in statusword and not in give up mask
*)?
YES
NO
standard error action on not give up bits
hard error ?
NO
YES
call of the block procedure
YES
bits set in give up mask
NO
return

The block procedure is called with 3 parameters:

block_proc(z,s,b)

z	is the zone. The record of z is the entire shared area available for the transfer.
s	is an integer containing the logical status word.
b	is the number of halfwords transferred in the operation, e.g. the block size.

Normally the give up mask should be '0' and the block procedure 'stderror'. But if a special block procedure is wanted, let the stderror procedure take care of as many of the status bits as possible. Note that a hard error **always** calls the block procedure.

Purpose and return

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

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

You signal the result of the checking back to the high level zone procedure by means of the final block length 'b'. The value of 'b' has no effect when an output operation is checked, but after an input operation you may signal a longer or a shorter block or even an empty block (b=0). However, the value of 'b' at return, must correspond to a block which is inside the shared area specified by the value of used share at return. Otherwise, the execution is terminated with an index alarm. Further details may be found in 2.4.

2.3.1.1 Examples

Example 2-19, reading a file on a magnetic tape.

Instead of creating a block procedure to take care of tapemark, a stop record should be created as the last record in the file.

Example 2-20, rejecting part of a block.

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

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

Example 2-21, copy input.

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

```
procedure copy(z,s,b);
zone z; integer s,b;
<* copy only if not harderror *>
if s extract 1=1 then stderror (z,s,b) else
begin comment this code also works for b=0;
   outrec6(test,b);
   tofrom(test,z,b,);
end;
```

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

Example 2-22, file number checking on magnetic tape.

This example has no relevance, if your parent (operating system) is BOSS. The safety of magnetic tape positioning can be improved by means of file labels. Each of the logical files on the tape are separated by two tape marks surrounding one label block. This block contains the logical file number in text form. The positioning to block 0 of a logical file (counted 1, 2, ...) is started with this procedure:

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

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

Example 2-22, file number checking on magnetic tape (continued).

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

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

Example 2-23, test on end of file

```
zone ztape (300,3,ztape_block_procedure);
procedure ztape_block_procedure (ztape,status,halfs)
zone ztape; integer status, halfs;
integer status, bytes;
   <* if the status is tapemark then nextfile else stderror*>
      if status shift 7 < 0 then
         goto next file
      else
         stderror(ztape,status,bytes);
<* the zone is opened with tapemark in give up mask *>
open(ztape,18,<:tapename:>,1 shift 16);
setposition(ztape,1,0);
for i:= .....
begin
   invar(ztape);
   .
   .
   .
end;
next file:
```

Example 2-24, see example 2 of invar (cf. [15]).

2.3.1.2 Other Calls of the Block Procedure

The runtime system may call the blockprocedure of a zone depending on a bit in the give up mask for another reason, which is not triggered by the receipt of a status word.

If the bit 1 shift 10 is set in the give up mask of a zone at the time the declaration block of that zone is left, either by passing the block end, by goto or by any other exit except runtime alarm, it is checked whether the zone is in a state 'opened and used for input/output but not closed'. If it is, the block procedure of the zone is called. It is imperative that the block procedure should be designed to handle this call reason, too.

This effect of 1 shift 10 in the give up mask depends, however, of a compile time option, zonecheck.yes, with the default value zonecheck.no.

Finally, the user may call the blockprocedure of a zone by issuing the statement: blockproc(z,s,b), where s will be handled as a statusword and b as number of halfwords transferred.

Example 2-25, check that the zone is closed at block exit

```
begin <* declaration block of the zone *>
   zone z (blocklength * no_of_shares,no_of_shares,
                        check_exit_block)
   .
   .
   .
   .
   .
   procedure check_exit_block(z,s,b);
   zone         z                ;
   integer      s,b              ;
      if s extract 1=1 then
         stderror (z,s,b) <* give up on hard error bit *>
      else
      begin <* the zone was not closed at block exit *>
               <* handle the situation e.g. by closing the zone *>
         close (z,true);
      end;
   .
   .
   .
   .
   open(z, modekind, name, 1 shift 10);
   .
   .
   .
end <* declaration block of the zone *>;
<* if the zone is 'in use' the block procedure is called *>
```

2.3.1.3 Another use of the give up mask

The runtime system will use the bit 1 shift 9 in the give up mask of a zone for another purpose than to call the block procedure, and not triggered by the receipt of a status word.

The use concerns concurrent input/output in activities, i.e. instead of waiting for an input/output transfer to complete, control is transferred to activity monitor mode for the possible activation of another activity. For details, cf. section 3.6 i [18].

2.3.2 Documents

A document is a physical medium in which a specific collection of data is stored.

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

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

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

The following sections give a survey of some documents and the way they transfer information to and from the zone buffer. The rules for protection of documents and further details are found in [1], [2], and [3]. The kinds and modes mentioned below are explained under 'open' (cf. [15]).

Internal process (kind 0)
An internal process (another program executed at the same time as your job) may receive or generate a document, which is data to or from your job. If the process just transmits the information to or from a peripheral device, the rules above for that device will hold for the communication with the document too. The kind specified in 'open' of the internal process should then be the kind of the document.

The internal process may also handle the information in its own way, and then no general rules can be given, but usually, the end of the document is signalled as explained in section 2.3.3.

Interval Clock (kind 2)
The interval clock serves delay operations, operations to wait for clockchange and operations to wait for power restart [3]. None of these operations are performed by high level zone procedures, but they may be issued by monitor(16,..., monitor(18,... . The logical position has no meaning as no document is scanned.

Backing storage area (kind 4)
The backing storage consists of a drum or a disc. The term 'drum' is used for a pool of discs, administered by the monitor. A 'drum' is a disc considered to be faster than a normal disc. Details about the various types of devices may be found in [3]. You have no direct access to the entire backing storage, but only to documents which are backing storage areas consisting of a number of 'consecutive' segments (only logically not physically). Each segment contains 512 halfwords (or 128 real variables). The segments are numbered 0, 1, 2, ... within the area, and the block numbers mentioned above are exactly these segments numbers. File numbers are irrelevant.

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

Disk (kind 6)
The document is the entire backing storage [cf. 3], being either a physical disc or a logical disc. Segment numbers on a physical disc specify absolute segments on the disc storage module, while segment numbers on a logical disc specify segments relatively to the start of the logical disc. The block numbers mentioned above are exactly these segment numbers. File numbers are irrelevant.

Terminal (kind 8)
A terminal may be used both for input and output. The sequence of characters input forms one document (infinitely long), and the sequence of characters output forms another document. File numbers and block numbers are irrelevant on a terminal.

One input operation transfers one line of characters (including the terminating New Line character) to the share. If the share is too short, less than a line is transferred, but that is an abnormal situation. The characters are packed in 8 bits to a character with 3 characters to one word, and last word is filled up with binary nulls. One output operation transfers characters packed in the same form to the terminal. Several lines may be output by one operation.

Paper tape reader (kind 10)
A document consists of one roll of paper tape. It may be read in various modes: with even parity, with odd parity, without parity, or with transformation from flexowriter code to ISO code. File numbers and block numbers are irrelevant for a paper tape.

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

Paper tape punch (kind 12)
A document is from the programs point of view infinitely long, even

when the operator divides the output into more paper tapes. A paper tape may be punched in various modes: with even parity, with odd parity, without parity, or with transformation from ISO code to flexowriter code. File numbers and block numbers are irrelevant for a tape punch.

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

Line printer (kind 14)
A document is from the program's point of view infinitely long. File numbers and block numbers are irrelevant on a printer.

One output operation may print any number of characters packed 3 per word.

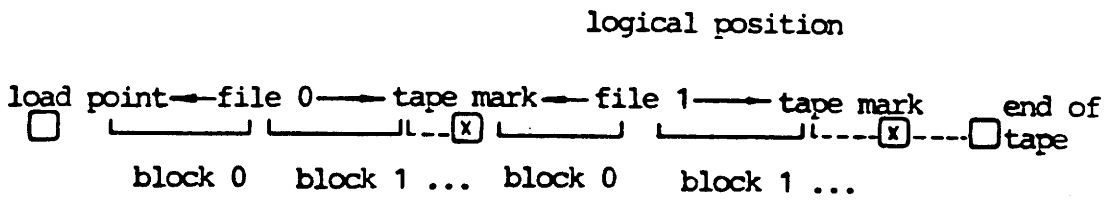
Card reader (kind 16)
A document is one deck of cards.

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

Usually jobs let the operation system read all necessary card decks before they are started. The cards may then be read as a normal text stored on backing storage (cf. [7] for further details).

Magnetic tape (kind 18)
A document is one reel of tape. It consists of a sequence of files separated by a single file mark. Each file consists of physical blocks possibly with variable lengths. The blocks may be input or output in high or low density, with the high or low speed specified, and also in even or odd parity. The files and blocks are numbered 0, 1, 2, ... as shown in the figure.

Magnetic tape document:



The normal magnetic tape station is a 9 track station where a block consists of a sequence of 8 bit characters; one word of the share is here transferred as 3 8-bit characters.

Among the magnetic tape stations, more kinds exist, differing in encoding and speed:

800 bpi	NRZ/1600 bpi PE,	45 ips
1600 bpi	PE/3200 bpi PE,	25/50/100 ips
1600 bpi	PE/6250 GCR,	25/75 ips
1600 bpi	3200 bpi PE/ 6250 GCR	50/100 ips

The share length you use for output to a magnetic tape determines the physical block length. As the blocks are separated by a block gap, the share length has influence on the amount of information the tape can hold and also on the maximum transfer speed.

Details on actual transfer rates and possible densities are found in [3], and the device manuals.

Devices without documents
Some peripheral devices, for instance the clock, do not scan documents, and they cannot be handled by the high level zone procedures. However, the primitive input/output level may handle such devices too.

2.3.3 Standard Error Actions and Modekind

Each standard error action is mainly concerned with a single bit of the remaining bits in the logical status word. The logical status word is 24 bits generated at the end of an operation on the document, it is available as a parameter when the check algorithm has called the block procedure. The first bits from '1 shift 23' until '1 shift 12' are taken from status returned from the external process controlling the document (cf. [3]) . The last bits are a transformation of the result supplied by the monitor, while bits '1 shift 8', '1 shift 7', and '1 shift 6' are generated by the wait transfer routine (cf. 2.4.4). The meaning of the bits is as follows:

Logical status word

External process

`1 shift 23:`
Intervention. The device became deselected or offline during the operation, presumably because the operator changed the paper, etc.

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

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

`1 shift 20:`
Data overrun. The data channel (bus) was overloaded and could not transfer the data.

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

`1 shift 18:`
End of document. Means various things, for instance: Reading or writing outside the backing storage area was attempted, the paper tape reader was empty, the end of tape was sensed on magnetic tape, the paper supply was low on the printer. See [1] and [3] for further details.

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

`1 shift 16:`
Tape mark or Attention. A tape mark was sensed or written on the magnetic tape or the escape key was touched during terminal i/o.

`1 shift 15:`
Write-enable. A write-enable ring is mounted on the magnetic tape, the diskette or the tape cassette was write enabled.

`1 shift 14:`
Mode error. It is attempted to handle a magnetic tape in a wrong mode.

`1 shift 13:`
Read error. Error in input from disk, or card reader error, cf. [3].

`1 shift 12:`
Not connected/disk error/card reject. The printer is not connected, disk error or erased diskette record inpit or card rejected by card reader.

Wait transfer

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

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

`1 shift 6:`
Position error. Generated by the check routine after magnetic tape position operations, when the monitors counts of file and block number differ from the expected values in the zone descriptor (cf. getzone6).

Monitor Result

`1 shift 5:`
Process does not exist. The process communicating with the document is unknown to the monitor, i.e. an area process or a peripheral process.

`1 shift 4:`
Disconnected. The device was somehow disconnected during input/output, e.g. by operator intervention.

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

`1 shift 2:`
Rejected. The program must not use the document, or it should be reserved first. Also if the disk is protected against writing.

`1 shift 1:`
Normal answer. The device has attempted to perform the operation, i.e. '1 shift 5' to '1 shift 2' are not set.

Standard error action

`1 shift 0:`
Hard error. The standard error action has classified the transfer as a hard error (cf. 2.4.4), i.e. the error recovery could not succeed and the standard error action gave up.

The standard error action for 'stopped' cannot be performed successfully if 'users bits' (cf. 2.4.4) contain any one of the following bits: 1 shift 22, 21, 20, 19, 18, 16, 7, 5, 4, 3, or 2. As a consequence, the stopped-bit is ignored by the standard error actions in this case.

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

As a table 2-1 in this section, you will find a quick index on how the standard error actions work for the different devices and status bits. You will also find the translation of the status bits to the messages from FP when the ALGOL program stops because of device errors (stderror is called).

Below follows a more elaborate description of the actions.

The status bits after extraction of the user's bit get a special treatment depending on the kind used in the call of open.

This treatment is shown in the table on the next page.

Table 2-1, Standard Error actions

			actions for the different kinds							
bit no.	algol equivalent	name	magtape	card reader	line printer	paper tape punch	paper tape reader	type-writer	bs area	internal
			18	16	14	12	10	8	4,6	0,2
0	1 shift 23	intervention	ignore	ignore	ignore	ignore	ignore	ignore	give up	give up
1	1 shift 22	parity error	repeat 15	ignore	give up	give up	ignore	give up	give up	give up
2	1 shift 21	timer	give up	give up	give up	give up	give up	give up or ignore	give up	give up
3	1 shift 20	data overrun	repeat all	give up	error	error	error	error	give up	give up
4	1 shift 19	block length error	repeat 15	error	error	error	error	error	error	give up
5	1 shift 18	end document	give up	ignore or EM	attend	attend	ignore or EM	give up	extend(4) give up(6) or EM(4,6)	give up or EM
6	1 shift 17	load point	ignore	ignore	error	error	ignore	error	error	give up
7	1 shift 16	tapemark or att	EM	ignore	error	error	ignore	ignore	error	give up
8	1 shift 15	writing enabled	ignore	error	error	error	error	error	give up	give up
9	1 shift 14	mode error	give up	error	error	error	error	error	give up	give up
10	1 shift 13	read error	error	ignore	error	error	ignore	error	give up	give up
11	1 shift 12	not connec-ted or disk error	error	ignore	error	error	ignore	error	give up	give up
12	1 shift 11	checksum error	error	error	error	error	error	error	error	error
13	1 shift 10	bit 13	error	error	error	error	error	error	error	error
14	1 shift 9	bit 14	error	error	error	error	error	error	error	error
15	1 shift 8	stopped	repeat all or ring	erro	repeat rest	repeat rest	error	repeat rest	repeat all	repeat rest

16	1 shift 7	word defect	repeat 15	error	error	error	error	error	error	error
17	1 shift 6	position error	repeat 5	error	error	error	error	error	error	error
18	1 shift 5	proc does not exist	mount	give up	give up	give up	give up	give up	create	give up
19	1 shift 4	disconnected	mount	give up	give up	give up	give up	give up	give up	give up
20	1 shift 3	unintelli-gible	give up	give up	give up	give up	give up	give up	give up	give up
21	1 shift 2	rejected	reserve	give up	give up	give up	give up	give up	create	give up
22	1 shift 1	normal	ignore	ignore	ignore	ignore	ignore	ignore	ignore	ignore
23	1 shift 0	hard error								

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

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

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

2.3.3.1 Details of Handling of Device Status

Internal process (kind 0)

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

End of document:
This will only make sense during input. If anything has been input, the bit will be ignored. Otherwise the empty block will be replaced by one word containing three end medium characters. If this bit appears during any other operation, it will cause the block procedure to be called.

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

Interval clock (kind 2)

Any status bit is treated by calling the block procedure. The special actions to be taken (intervention) must be defined by your program.

Backing storage area (kind 4)

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

End of document (i.e. area):
If this happens during input, and if nothing has been transferred, the empty block is replaced by one word containing three end medium characters, otherwise the bit is ignored. During output, the standard action is to try to extend the area. If the extension fails because of lack of resources for that particular disk, a parent message `bs <disk> <segments> 0` is sent. If the FP modebit 'bswait' is 'yes', or FP is not in the process, the process waits. When the process continues, the

extension is tried once more, and if it still fails, the block procedure is called. If the extension succeeds, the output operation is repeated.

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

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

Rejected:
Handled exactly as 'process does not exist'.

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

Logical or physical disc (kind 6)

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

End of document (i.e. disc)
During input, if nothing has been transferred, the empty block is replaced by one word containing three end medium characters, otherwise the bit is ignored. At all other operations, the block procedure is called.

Stopped
May appear at all operations. The operation is repeated, except if it has been overruled by the action for the end of document or the two actions below.

Process does not exist
An area process is created and the action proceeds as for area process.

Rejected
If the disc process does not exist or calling process is not a user, the block procedure is called. If the operation is initialize, cleantrack or output, the disc process is reserved for exclusive access. If this is not possible, the action gives up and calls the block procedure. Now the operation is repeated.

Note that if the process does not exist, an area process will be created only if an entry of that name exists in the main catalog. If not so, the action gives up and calls the block procedure.

Terminal (kind 8)

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

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

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

Paper tape reader (kind 10)

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

End of document (i.e. end of paper tape):
If anything has been input the status is ignored, otherwise a block of one word containing three end medium characters is simulated.

Paper tape punch (kind 12)

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

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

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

Line printer (kind 14)

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

Intervention (i.e. deselected)/End of document (i.e. no more paper):
A message is sent to the parent requesting operator attemtion and the job is stopped until the operator has performed his task and starts the job.

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

Card reader (kind 16)

A parity error status, signalling an error in the conversion, is ignored by the standard error actions, as the monitor substitutes the wrong combination by a substitute character corresponding to the conversion (cf. details in [3]). The 'end hopper full. The of document' status shows end of card deck or 'intervention' status shows stocker full.

End of document (i.e. end of deck):
If anything has been input, the status is ignored, otherwise a block of one word containing three end medium characters is simulated.

Magnetic tape (kind 18)

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

Parity error, word defect, blocklength error, data overrun:
The stopped bit is ignored in this case. An input operation is repeated up to fifteen times, but if the parity persists, the error is a hard one. An output operation is repeated up to fifteen times, i case of parity error preceded by one erase operation the first time, two erase operations the second, and so on. If the parity persists, the standard actions give up and call the block procedure.

Word defect, blocklength error, data overrun:
The actions are as for parity error. Note that if you suppress the 'word defect' action by setting '1 shift 7' in your give up mask, you can read tapes not written on the RC8000 or tapes written with trail < > 0 (cf. open). Of course your block procedure will be called each time the bit occurs. In case of 'word defect', unused character positions are filled with binary nulls.

Position error:
The position reached differs from the position required by a move operation. The operation is repeated up to five times, and if still position error, the standard actions give up and call the block procedure.

Tapemark:
Tapemark is ignored after a sense or a move operation. If tapemark occurs after an input operation, the standard action is to simulate a block of one word containing three end medium characters.

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

Does not exist:
The bit is ignored after a sense operation or a move operation. In other cases, a mount-tape-message is sent to the parent. Next, the tape is reserved for exclusive access and if this is unsuccessful, the

mount-tape-message is sent again. Thirdly, the tape is positioned according to file and block count and the operation is repeated.

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

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

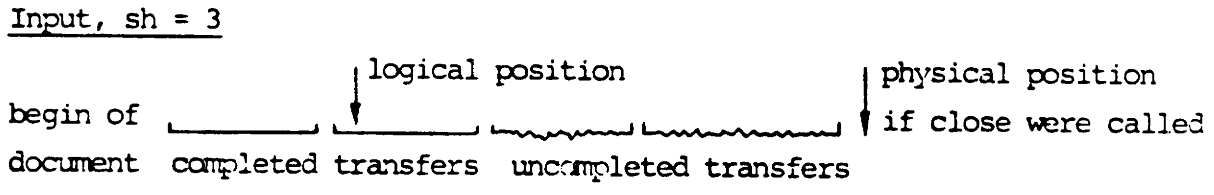
2.4 Multi Buffering and Checking

2.4.1 Multishare Input/Output

Below sh denotes the number of shares in the zone.

Input.
During input from a document via a zone with sh shares, the system uses one of the shares to unpack information and the remaining sh-1 shares for uncompleted input of later blocks (at first input all sh shares are used for uncompleted transfers). The following picture shows the state of the blocks of the document.

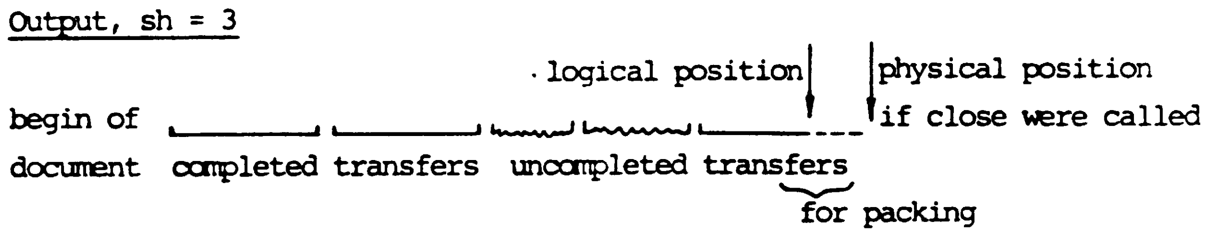
Input, sh = 3



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

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

Output, sh = 3



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

Swoprec6.
The procedure swoprec6 utilizes the shares as follows: One share is used for packing and unpacking of information. If sh > 1, another share is used for uncompleted output. Remaining shares, if any, are used for uncompleted input of later blocks.

2.4.2 Choice of Number of Shares

The advantage of the multishare input/output is that differences in speed between the program and the device may be smoothed to any degree. The most frequent choice is between single or double buffer input/output. Do not use multibuffering if it can be avoided. A decrease of transfer time can be obtained using a larger share size (blocking).

The following rule of thumb may help you calculate time differences in cases where you scan a document sequentially:

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

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

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

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

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

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

2.4.3 Message Buffers Occupied

Input/output by means of sh shares occupies permanently sh-1 of the message buffers available for the job (cf. [1]). First time input, however, occupies momentarily sh message buffers.

From the moment setposition has been called for a magnetic tape and until the first input/output operation is performed, one message buffer is occupied (even when sh = 1).

2.4.4 Algorithms for Multishare Input/Output

The following algorithms are part of ALGOL 8 input/output system.

You must know about these algorithms if you want to interfere with the system in the block procedure of the zone (examples of block procedures are given in 2.3.1). [3] and [8] explain the rules behind the communication with devices.

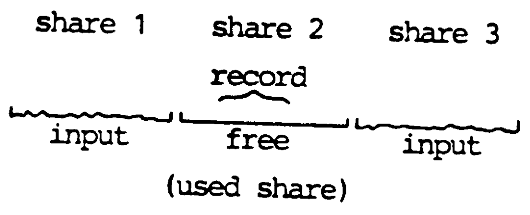
Below sh denotes the number of shares in the zone.

Snapshots of shares in typical situations (sh = 3)

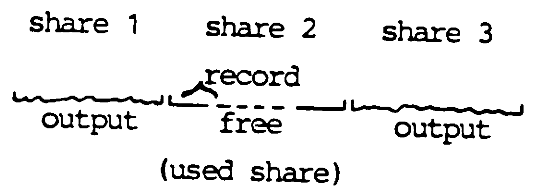
Just after setposition on a magnetic tape:

```
    share 1         share 2   share 3
~~~~~~~~~~~~~~~~  ________  ________
move operation      free      free
for positioning
uses only share 1
```

After inrec:



After several outrecs:



Change of block at input.

Program Part:

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

Change of block at output

```
   if share state(used share <> free then
   begin
      wait transfer(used share);
      comment a positioning operation might be uncompleted;
   end;
```

```
start transfer(output);
used share:= used share mod sh+1;
comment one or more shares behind used share are busy with transfers;
wait transfer(used share);
comment share state becomes free and the share may be filled from record
base to last halfword;
```

Start_transfer (share).

This procedure works only on used share. It sets a part of the message and sends it (written in pseudo ALGOL):

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

Wait_transfer (share).

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

```
record base := abs address of first shared(used share)-1;
last halfword:= abs address of last shared(used share)+1;
record length:= last halfword-record base;
st:= share state(used share);
if st <> running child process then
   share state (used share):= free;
if st <> uncompleted transfer then
   goto return;
wait answer(st);
if kind = magnetic tape and
   not tape mark sensed and
   (status in answer > 0)
   or some words were transferred
   transfer position from
   answer to zone;
compute logical status word;
top transferred(used share):=
if operation = io then
   1+address of last halfword transferred else
   first shared(used share);
users bits:= common ones in logical status and give up mask;
remaining bits:= logical status-users bits;
perform standard error actions for all ones in remaining
bits (cf. 2.3.3);
if hard error is detected then
   logical status:= logical status+1;
if hard error is detected or users bits <> 0 then
```

```
begin <*call blockprocedure*>
   b:= top transferred(used share)-1-record base;
   let record describe the entire shared area from first
   shared to last shared;
   save:= zone state;
   if operation = input and tapemark and b=0 then
      b:= 2;
   blockproc(z,logical status,b);
   zone state:= save;
   if b < 0 or b+record base > last halfword then
      index alarm;
   top transferred(used share):= b+1+record base;
   return
end;
```

Compute logical status word.

The logical status word is 24 bits generated at the end of an operation on the document:

```
<* bits 18 to 22 from monitor *>
result:= monitor (18,z,used share,answer);
<* wait answer result *>
logical status:= 1 shift result;
<* bits 0 to 11 from monitor *>
if result = 1 <* normal answer *> then
   logical status:= logor(logical status, answer(1));
<* add bits 0 to 11 from hardware status in answer *>
<* bits 15 to 17      from wait transfer *>
<* stopped bit 15 *>
if (modekind = bs or operation = output)
   and 'not everything has been transferred' then
   logical status:= logical status add 1 shift 8;
<* word defect bit 16 *>
if modekind = magtape then
   begin
      if 'characters transferred do not fill an integral
      number of words' then
         logical status:= logical status add 1 shift 7;
   end;
<* position error bit 17 *>
if modekind = magtape then
   begin
     if tape mark sensed and
     not move operation then
        transfer answer position to zone;
   if answer position <> filecount, block count then
   logical status:= logical status add 1 shift 6;
end;
```

3. Fields

Field variables is a tool in ALGOL8 which makes it easy to address all types of variables in a zone record or an array.

3.1 File, Record and Field

A file on a document (e.g. a backing storage area) may consist of set of records. A record may be regarded as a set of fields, where a field is the smallest entity which in some connection is considered as a unit of data.

The terms file, record and field will only have a meaning when they are defined together with a specific data set and the operations on it.

Example 3-1, record manipulation.

A use of these concepts could be:

read a record from a file by means of the high level record procedures (cf. chapter 2) change some of the variables using fields to specify the type and length of the record elements, and finally write the record back or to another file.

```
begin
   <* insert record numbers in all records in a file *>
   integer field recordno, rlength, end value;
   zone zin, zout (128,1,stderror);
   integer i;
   open (zin, 4, <:input:>, 0);
   open (zout,4, <:output:>, 0);
   rlength := 2;
   recordno := 6;
   end value := 10;
   i:= 1;
   <* the file has an end record as the last record *>
   repeat invar(zin);
      zin.recordno := i;
      i:= i+1;
      outvar(zout,zin);
   until zin.end value = 9999999;
   close(zin,true);
   close(zout,true); end;
```

Example 3-2, a zone record describes a simple invoice and has the following structure:

Note: z(1) contains length and check sum.

zone element	field name	length in halfwords	field variable type	field value
	customer_no	2	integer	6
z(2)	internal_cust	1	boolean	7
	empty	1		
z(3)-z(5)	customer_name	12	long array	8
z(6) first word	zip code	2	integer	22
z(6) sec. word to z(10) first word	customer_addr	16	long array	22
z(11) first word	item_no	2	integer	42
z(11) sec. word	no_ordered	2	integer	44
z(12)	price	4	long	48
z(13)	discount	4	long	52
z(14)	amount	4	long	56

z.fieldname will address the wanted part of the zone record.

The field variable must point to the rightmost halfword in the required zone part if it is a simple field.

If it is a one dimensional array, the field value must point to its preceding halfword.

The addressing of this zone record with fields is shown in the following table:

Table 3-1.

Record element	Field name	Halfword numbering	Initialized field variable value
z(1)	record_length	2	
	check_sum	4	
z(2)	customer_no	6	customer_no
	internal_cust	7	internal_cust
	- not used -	8	customer_name
z(3)	customer_name	10	
		12	
z(4)		14	
		16	
z(5)		18	
		20	
z(6)	zip_code	22	zip_code= customer_addr
	customer_addr	24	
z(7)		26	
		28	
z(8)		30	
		32	
z(9)		34	
		36	
z(10)		38	
	- not used-	40	
z(11)	item_no	42	item_no
	no_ordered	44	no_ordered
z(12)	price	46	
		48	price
z(13)	discount	50	
		52	discount
z(14)	amount	54	
		56	amount

The following program will read a record like this and write some elements on current output.

```
begin
  integer i;
  zone z(128,1,stderror);
  <* declaration of field variables *>
  integer field customer_no,item_no,no_ordered,zip_code;
  boolean field internal_cust;
  long field price, discount, amount;
  long array field customer_name, customer_addr;
  <* initializtion of field variables is done relative
     to make a change easy *>
  customer_no:=    i:= 6;
  internal_cust:= i:= i+1;
```

```
                  i:= i=1;<* not used *>
  customer_name:= i:= i;
                  i:= i+12;
  zip_code:=      i:= i+2;
  customer_addr:= i:= i;
                  i:= i+16;
                  i:= i+2; <* not used *>
  item_no:=       i:= i+2;
  no_ordered:=    i:= i+2;
  price:=         i:= i+4;
  discount:=      i:= i+4;
  amount:=        i:= i+4;
  <* end init *>
  open(z,4,<:record:>,0);
  <* open to backing storage with records created
     with outvar *>
  invar(z);
  <* now a zone record is available *>
  write(out,"nl",1,<:customer number:>,
    <<dddddd>,z.customer_no,
    "nl",1,<:customer name:>,z.customer_name,
    "nl",1,<:customer address:>,z.customer_addr,
    "nl",1,<:item number:>,z.item_no,
    "nl",1,<total sales price:>,<<dddddddd>,z.amount);
  close(z,true);
end;
```

3.2 Field Elements

A **field reference** consists of a **field base** and a **field variable** separated by a period. A field reference is either a **simple field** or an **array field.**

3.2.1 Field Base

The field base is the array or zone record in which the field is found. Note that the array may be an array field.

3.2.2 Field Variable

The field variables are the pointers to the wanted part of the field base. A field variable can be a simple field variable or an array field variable.

A simple field variable used as reference in the field base will give a simple field, and an array field variable used as reference will give an array field.

3.2.3 Simple Field Variable

A simple field variable points to a single element of the field base. This element is specified to be of type boolean, integer, long or real by the type of the simple field variable. This type specifies therefore the field length of the element as follows:

simple field variable type	length of simple field in halfwords
boolean	1
integer	2
long	4
real	4

3.2.3.1 Value of a Simple Field Variable

The value of the simple field variable is of type integer independent of the type of the variable. This integer value is a halfword pointer, which selects the wanted element, in the field base.

The value of the pointer depends on the type of the simple field variable.

The halfword numeration is the lexicographical ordering of the array (or zone record) (cf. [14]).

The halfword with number 0 (zero) is the last halfword in the (possibly fictive) array element with subscripts (0,...,0).

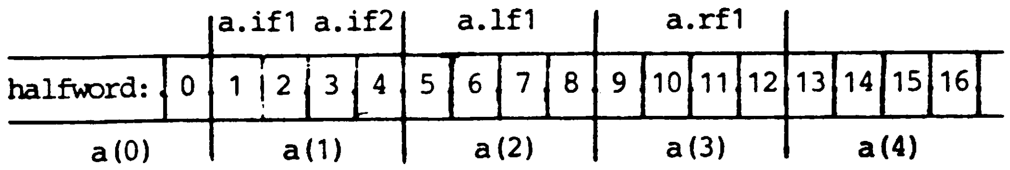
Booleans fields are addressed by their halfword number. Integer fields are addressed by one of the 2 halfwords of the integer word. Long or real fields are addressed by one of the 2 halfwords in the rightmost word.

The field reference is only valid if the simple fields are inside the bounds of the field base.

A simple field variable can be assigned values throughout the program.

Example 3-3,

```
begin integer i;
   <* declaration of the field variables *>
   integer field if1, if2;
   long field lf1;
   real field rf1;
   <* field base a *>
   long array a(1:100);
   <* initializtion of fields *>
   if1:= i:= 2;
   if2:= i:= i+2;
   lf1:= i:= i+4;
   rf1:= i:= i+4;
   <* simple field references *>
   a.if1:= 1;  a.if2:= 2;
   a.lf1:= 3;  a.rf1:= 4;
end
```
The first 12 halfwords of the array a are assigned



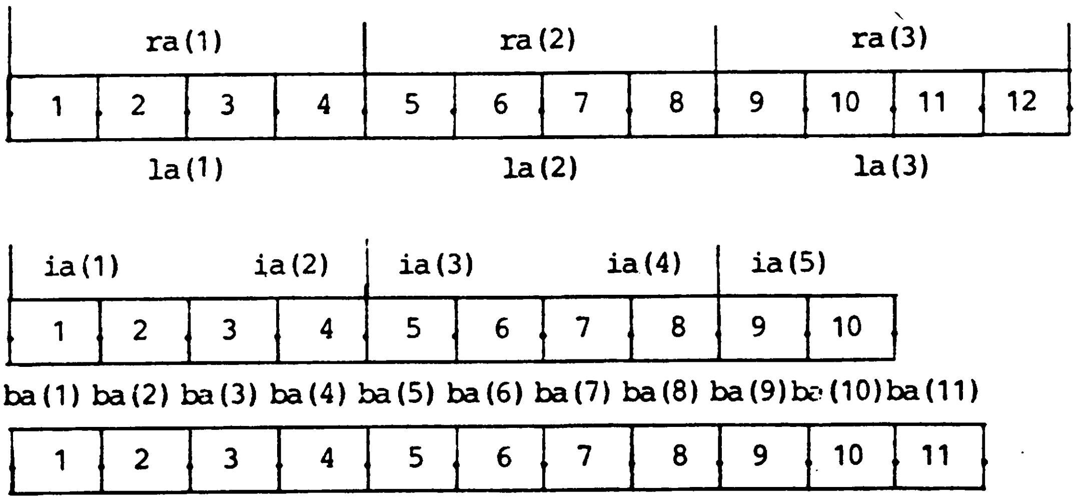
a.if1 will address first word of a(1)
a.if2 will address second word of a(1)
a.lf1 will address a(2)
a.rf1 will address a(3)

Example 3-4, halfword numbering.

If a program contains the declarations

```
real array ra(1:3); long array la(1:3);
integer array ia(1:5); boolean array ba (1:11);
```

then the halfword numeration is according to this scheme:

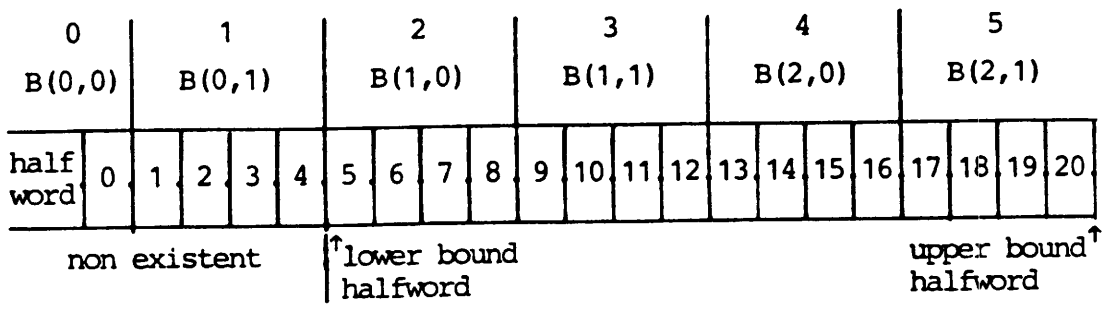


Example 3-5, lexicographical index.

If a program contains the declaration

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

the array will be numbered like this lexicographical index:



3.2.4 Array Field Variables

An array field variable will point to an array within the field base. The elements of this array are specified to be of type boolean, integer, long or real by the type of the array field variable. This type specifies the length of the array elements (cf. 3.2.2).

The array field variable is considered to be declared one dimensional.

3.2.4.1 Value of an Array Field Variable

The value of the array field variable is an integer independent of the type of the variable. This integer value is a halfword pointer selecting a part of the field base. The value should be the halfword number in the field base which defines the possible fictive element (0) of the array field. As the latter is always one dimensional the pointer should be the upper halfword number of the subscript 0 (zero) of the array field.

3.2.4.2 Bounds of Array Fields

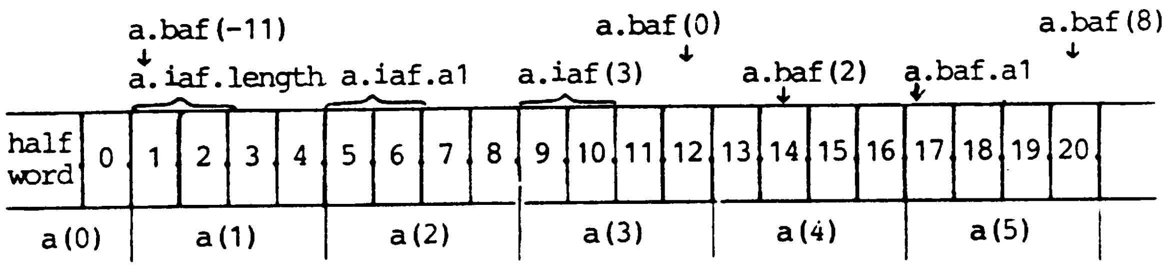
The array must be within the field base. No check is made that the subscripts are outside the array field. But it is checked that the subscripts are not outside the original field base.

```
lower bound of array field =
lower bound of field base - array field variable

upper bound of array field =
upper bound of field base - array field variable
```

Example 3-6, array fields.

```
begin
   <* declaration of field variables *>
   integer       field length, a1;
   integer array field iaf;
   boolean array field baf;
   <* field base a *>
   long array a(1:5);
   <* initialization of fields *>
   length:= 2;
   a1    := 6;
   iaf   := 0;
   baf   := 12;
   <* array field references *>
   a.iaf(5) := 1;   a.iaf.length:= 20;
   a.baf(8) := "3"; a.baf.a1    := 4;
end;
```



Example 3-7, array fields.

The index bounds for the array field baf from the above example are as if it was 'declared':

```
boolean array a_baf(-11:5);
```

3.2.5 Fields as Parameters to Procedures

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

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

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

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

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

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

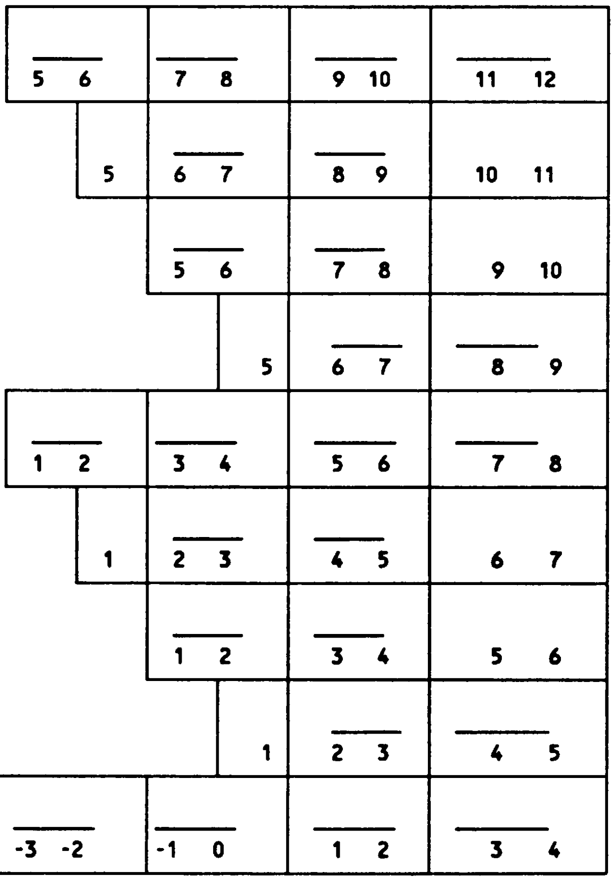
is true.

Example 3-8,

Consider a program like this:

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

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



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

Halfwords with equal locations are shown in the same column.

The reference halfword numbers corresponding to indexing in la are bold and underlined, and the halfwords accessible by direct indexing are shown in boldface. The word boundaries are shown as vertical lines going from line to line.

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

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

4. Context Blocks

The context concept is a very useful tool in application programs handling on-line terminal systems, and for programming 'coroutines' (cf. 4.2), and various other things.

A program that is to serve several terminals does not know from which terminal to take the input at a given time. This is not known until the input operation is completed, which means that the communication initiative is taken by the terminal operator, and not by the program. Two consecutive transactions may thus quite well originate from different terminals, which circumstance makes it difficult to express the processing of a sequence of transactions from the same terminal.

In such program the terminals usually have a set of terminal variables. These variables describe the state, function, type, and the like of the operations performed on the terminal in question. These variables are normally the same for all the terminals in the system. Therefore a set of equally defined variables is needed.

It should be easy for the application program to access the terminal variables belonging to the terminal from which input is actually read. The context blocks will solve these problems. A declaration of a context block will make the context variables available in a number of incarnations i.e. one per terminal. When the context block is entered the variables of one of the terminal incarnations is selected for processing.

Example 4-1, transaction series from a terminal.

In the following example two procedures are assumed to exist:

```
wait_trans(z,device,...); and
read_field(z,...);
```

wait_trans awaits the arrival of a transaction from zone z by jump-out, the value of 'device' is the identification of the terminal from which the input was received, whereas the zone z is positioned to the first character of the transaction.

read_field positions to the first character of the next field of the transaction procured by calling wait_trans. The example is now as follows:

```
next_transaction:
   wait_trans(in,device,...);
   ...
   begin context(device,10,mode);
      integer array a(1:20); boolean more_fields;
      real x,y; integer i,j,k; long l;
      ...
      next field:
         read_field(in,...);
         read(in,x,y,i,j,k,l);
         read_string(in,a,1);
         ...
      if more_fields then goto next_field;
      ...
   end context block;
goto next_transaction;
```

After the call of wait_trans the device number is defined. This identifies an incarnation of the context block shown, which has thus one incarnation for each of the 10 terminals.

The example outlines a way of programming the processing of a sequence of transactions from the same terminal.

4.1 Context Block Elements

A **context block** is an ordinary ALGOL block, semantically extended with certain properties. A context block is declared as follows:

```
begin
   context(incarnation,no_of_incarnations,context_mode);
   <* declaration of simple variables,arrays,etc. *>
   ...
end;
```

An ordinary ALGOL block and a context block differ as follows:

Immediately after the block 'begin' follows the context declaration:

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

where the three declaration parameters: incarnation, no_of_incarnations, and context_mode are all integer expressions, the values of which must be defined at the block entry.

Declaration of zones and zone arrays in the head of the block is **not** permitted.

The context declaration is not permitted to appear more than once in the head of the block.

The local variables (simple as well as arrays) declared in the head of a context block are also called **context variables**.

Context blocks are permitted anywhere in an ALGOL program (or external procedure), where ordinary blocks are permitted. However, they cannot appear as a procedure body, i.e. they have to be enclosed in the begin-end of the procedure body.

The context declaration specifies the number of incarnations of the context variables, the actual incarnation and the way the context variables should be handled when the context block is entered and left.

Example 4-2, context declaration.

```
begin
   context(i,n,3);
   <* context variables: s,t,x,a,b *>
   real x; integer s,t; real array a,b(1:m);
   ...
   begin
      real y,z; integer j,mode;
      real array p(1:20); zone z(128,1,stderror);
      ...
      begin
         context (j,10,mode);
         <* context variables: s,t1,l *>
         integer s,t1; long 1;
         ...
      end;
      ...
   end;
   ...
end;
```

In this example all block declarations are permissible constructions. Note that the zone declaration in the 2'nd block is permissible, because, it does not appear among context variables.

4.1.1 Incarnations

The context declaration defines a number of incarnations of the declared block, the incarnations being numbered: 1,2,3,...,<no of incarnations'. The declaration parameter <incarnation> represents such an incarnation number. An incarnation number is related to each execution of a context block. If two different executions of a context block have the same incarnation number, they define the same incarnation of the context block. The number of different incarnations is thus equal to the value of the declaration parameter: <no of incarnations>.

4.1.2 Incarnation Interval

When a context block is executed for the first time, the actual value of <no of incarnations> defines the number of differen' incarnations of the block. This value will then remain frozen for the repeated use of this binary program, although the value may be changed by the program during the execution. The value can be changed using the context mode 'new block bit' in a later execution of the context block (cf. 4.1.4).

The incarnation interval:

```
1 <= incarnation <= no of incarnations
```

is then defined.

4.1.3 Context Variables

The context variables are the local variables declared in the head of a context block. Zones cannot be declared as context variables.

When a context block is declared, space for the context variables is reserved the number of times specified by 'no_of_ incarnations', but only one of the incarnations is accessible at a time.

Before the first statement in the context block is executed, the context variables of the context block are initialized to values depending on which incarnation is to be executed. When the block is left (via the last end in the block or by means of a goto statement) the values of its context variables are stored in such a manner that the value of the variables can be retsored by the stored values in the next execution of the same incarnation of the block. The declaration parameter <context mode> governs the initialization and storing of context variables (cf. 4.1.4).

4.1.3.1 Initialization of Context Variables

At the entry point of a context block, and before the first statement is executed, the following is done:

a) The declaration parameter <incarnation> is evaluated and it is verified that it is within the incarnation interval. This value remains unchanged throughout the execution of the context block, although it may be changed through assignments, etc., (it has thus the same status as value parameters in procedures). In this way an incarnation of the context block is selected. To the incarnation is related just one record, the fields of which are identical to the context variables of the block. To each context block are thus connected <no of incarnations> records. The records are called context records.

b) Next, the context variables of the block are initialized. If the incarnation has been executed before, the contents of the relevant context record will be transferred to the context variables of the block (depending on context mode). If it is a first time execution of the incarnation, all the context variables of the block are zero-set (binary 0). This zero-setting is also performed when 'new incarnation bit' is specified in the context mode.

The topical array lengths define at the same time the maximum array lengths applicable to this incarnation. This means that an array length, in all subsequent executions of this incarnation, shall be less than or equal to the maximum length.

The transfer of values between context records and context variables is done in accordance with the normal lexicographical ordering.

4.1.3.2 Storage of Context Variables

At the exit an incarnation of from a context block (via the last end in the block, an exit operator, or a goto statement) the context variables of the incarnation are stored (depending on context mode) in the context record belonging to the incarnation, as follows:

* If the incarnation was executed for the first time this means that the corresponding context record does not exist. Such a record is now established in the virtual store connected to the program (cf. 4.4).
* The values of the context variables of the incarnation are transferred to the context record belonging to the block and its incarnation.

If several context blocks are nested into one another, and if jumps occur out of several context block levels, the process described above will take place for each of the context blocks thus being left.

4.1.4 Context Mode

The actual value of <context mode> affects the action described under 4.1.2, 4.1.3.1, and 4.1.3.2. The value is regarded as a bit pattern:

Read bit: 1 shift 0
The action described under 4.1.3.1 b) is not executed unless 'read bit' is set. If 'read bit' is not set, only the zero-setting of context variables is performed.

Write bit: 1 shift 1
The action described under 4.1.3.2 is not executed unless 'write bit' is set. 'Write bit' = 0 is therefore useful for references to and searching in context records.

Save bit: 1 shift 2
The context record is written into the file containing the virtual store every time the context block is left (cf. 4.5). If only the 'write bit' is set the context records are not updated in the virtual storage file.

New block bit: 1 shift 3
If 'new block bit' is set, the action is as if this context block was executed first time. Even in the case of context records already established, such records are abandoned.

New incarnation bit: 1 shift 4
If 'new incarnation bit' is set, the action is as if this incarnation was executed first time.

4.2 Coroutines

With the context concept are connected two **context statements** and one **standard procedure**, permitting a context block to be considered a coroutine in <no_of_incarnations> incarnations.

The two statements are described below.

4.2.1 Exit

The exit statement is a goto statement, which leaves a context block in such a way that the same incarnation can restart execution next time with the statement immediately following the exit call.

The syntax is:

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

where it is required that:

- The statement shall be found in a context block only,
- The block level at which an exit statement is found shall be identical to the level of the context block,
- The statement shall be found outside repetitive statements embedded in context blocks.

The exit statement has the following effect:

The restart point (also called the continuation point), i.e. the logical address of the statement immediately following the exit statement, is stored in an anonymous context variable. This variable is called the context label belonging to the incarnation.

Jumping to the point of <designational expression>, is done exactly the same way as via a goto statement, i.e. the action described under 4.1.3.2 is performed.

More exit statements are permitted in the same context block, and each incarnation has its own specific context label.

The value of a context label is either a continuation point or 0 (zero). A continuation point can be created by an exit statement only. The zero value is found in the following situations:

- first time the incarnation is executed,
- when 'read bit' has not been set,
- when 'new block bit' or new incarnation bit is set.

4.2.2 Continue

The context statement continue is a goto statement which jumps to a context label.

The syntax is:

```
continue
```

where it is required that the statement is found in a context block. The block level at which it is found may be different from the context block level (i.e. inner block).

The statement has the following effect:

- if the context label belonging to the incarnation is zero, the statement is blind. This means that the execution goes on with the next statement,
- if the context label belonging to the incarnation has the value of a continuation point, the execution continues in that point.

4.2.3 Resume

The standard procedure resume works as the context statement continue, except that the context label is cleared (zero-set) after execution. This means the a context label can only be used once by the procedure resume.

4.3 Examples

In the following are given a number of examples of context blocks.

4.3.1 Continue and Exit

The following example illustrates one way of using exit and resume.

Example 4-3, continue and exit

```
next:
...
begin
   context(i,n,mode);
   array a(1:10); real x,y;
   ...
   continue;
   <* goto l1 or l2 depending on context label *>
   ...
   exit(next);
   <* goto next, set context label to l1 *>
   l1:
   ...
   exit(next);
   <* goto next, set context label to l2 *>
   l2:
   ...
end context block;
```

4.3.2 Relation to the concept of owns

In the following example the context declaration has the same effect as an own declaration of x,y, and a (although an array cannot be declared own). The context concept can thus be considered a generalization of the own concept in the ALGOL60 sense (cf. warning in example 4-6, however).

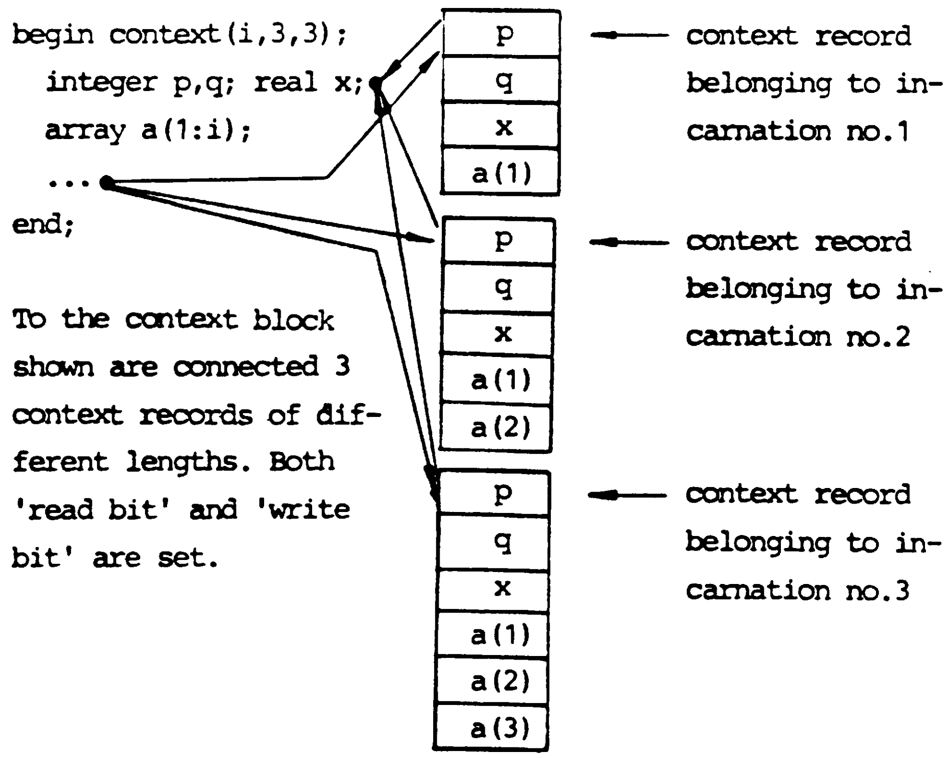
Example 4-4, relation to the concept of owns

```
begin
   context(1,1,3);
   real x,y; real array a(1:n);
   ...
end;
```

4.3.3 Context Records

The following example illustrates the interrelation between context variables and context records.

Example 4-5, context records



```
begin context(i,3,3);
   integer p,q; real x;
   array a(1:i);
   ...
end;
```

To the context block shown are connected 3 context records of different lengths. Both 'read bit' and 'write bit' are set.

p
q
x
a(1)
context record belonging to incarnation no.1
p
q
x
a(1)
a(2)
context record belonging to incarnation no.2
p
q
x
a(1)
a(2)
a(3)
context record belonging to incarnation no.3

4.3.4 Record Classes

The following is an example of the use of context blocks in connection with the handling of record classes with appurtenant code.

Example 4-6, record classes

```
integer procedure next_record(action,record,last_record);
integer action,record,last_record;
begin
   integer mode;
   mode:= case action of (3,1,3,2,...);
   begin
      context(record,1000,mode);
      long array text(1:50);
      integer version, chars,chain;
      case action of
      begin
         comment,only 4 actions shown;
         <*1*> begin
                     version:= version+1;
                     chars:= read_string(in,text,1)*6;
                     next_record(3,last_record,record);
                  end;
         <*2*> begin
                     write(out,<:<10>version:>,version,<:<10>:>);
                     write (out,text);
                  end;
         <*3*> chain:= last_record;
         <*4*> <* dummy action = clear record *>
         <*5*> ...
      end case;
      next_record:= chain;
   end context block;
end procedure next_record;
```

The procedure defines a class consisting of 1000 records. A record contains a text field (array text), a version number (version), and a chain to the next record (chain). The 1000 records constitute a number of single chained chains. The vaule of the procedure call is the number of the following record in the chain.

The following statements will output the chain of records starting with record no. <first_record> and ending with a record having no successor:

```
next:= first_record;
for next:= next_record(2,next,0) while next <> 0 do;
```

The statement:

```
next_record(1,n,m);
```

will change the contents of record no. n (or establish one, if it is not found), and link it to the chain, so that it follows after record no. m, and increase the version number by 1.

Note that action no. 1 in the context block calls the procedure recursively.

Caution:
If the procedure is called recursively by the **same incarnation**, the relation to own variables referred to in example 4-4 will no longer be applicable, because a context record is not updated until block_exit. Own variables are in fact always directly updated (in assignments, etc.), whereas the fields in context records are updated through context variables, which during the updating are separated from the record. Only at block_exit are these variables transferred to the record.

Recursive calls of context blocks, using the same incarnation, should therefore be subject to caution.

4.4 Virtual Storage File

All context records in the program are kept in a Virtual Storage File, related with the program. When the program is called, this virtual storage file is the extension of the program file itself. Segment transfers to and from this storage file are integrated into the paging algorithm used to transfer normal program segments. The program is thus, in a way, selfmodifying. If the program is called more times, the context records will assume the values stored in the virtual storage file by the previous execution.

The file containing the program must have a scope giving write access to the file.

The upper limit of the volume of the virtual storage file (program not included) is 24 000 000 characters (= 16M halfwords).

4.4.1 Openvirtual, Virtual

Another backing storage area can be connected as virtual storage file to the program by means of the procedures:

```
openvirtual(name_of_file);
string name_of_file;
```

or

```
virtual(name_of_file);
string name_of_file;
```

where name_of_file is the name of the file intended for virtual storage file. The function is as follows:

In case of openvirtual, the file connected as virtual storage file is disconnected, i.e. all own variables, internal descriptions, etc., are transferred into the file.

Both procedures then connect the new file to the program as virtual storage file in a way depending on the contents of the catalog entry of the file, cf. [6].

Contents= 0:
The file is considered a file which has not previously been used as a virtual storage file in connection with context. 'openvirtual' or 'virtual' is to be called before execution of any context block. The catalog entry of the file gets contents key = 9, and all own variables, internal descriptions, etc. are written at the start of the file. Now the file is the virtual storage file of the program.

Contents= 2:
The file is considered a program having as an extension the virtual storage file of this program. The own variables and internal descriptions of the calling program are initialized by values fetched from this virtual storage file. It is a prerequisite hereof that the structures of own variables and context blocks must be equal in the two programs. After 'openvirtual' or 'virtual' has been called, the calling program will continue with the new virtual storage file; i.e. context records established by the other program are now used by this program.

Contents= 9:
The file is considered a virtual storage file which has previously been used by this program (or other programs) in connection with context. The function is, otherwise, the same as for contents= 2.

NOTE:
'openvirtual' and 'virtual' are not to be called within a context block.

4.5 Program Restart

It is guaranteed in the implementation that a context record is explicitly written back into the file connected as virtual storage file, provided that 'write bit' and 'save bit' are set, even if the segments, in which the record is stored, are found in memory. The virtual storage file, which is thus currently updated, survives the program termination. When the program is later called in the normal way, it will be in a position to continue with the virtual storage file in its most recently updated condition.

If 'save bit' is not set, the writing into the file is only done when the paging algorithm of the runtime system requires segment space in memory. However, at program termination, just before exit, all context data segments, which have been updated, are written back into the file. This ensures that the virtual storage file contains updated values of the context records at program end.

5. Text Handling

Text or string variables are not implemented in ALGOL8, but strings can be parameters in procedure calls (e.g. open).

5.1 Storing of Texts

Text constants belong to either of two groups: short texts and long texts, cf. [14].

Short texts are texts with less than 6 characters, which may be moved to a long or a real variable. The last position of the text is a null character.

Long texts are texts with 6 or more characters, which may be moved to an array, usually of type long.

The value of a long text string is a reference to the address where it was stored by the compiler.

All texts are stored with 3 characters per word.

The standard procedures read, readall, readstring and movestring may move a text string into an array.

The operators add, long, and real may help to store a text directly in a variable.

Example 5-1, storing of texts in variables.

The following piece of code will cause the variable 'time' to contain the text 'time', the variable 'number' to contain the text 'number' and array 'address' to contain the text 'Lautrupbjerg 1-3, Ballerup'.

```
begin
   long time, number;
   long array address(1:5);
   time:= long <:time:>;
   number:= long <:numbe:> add 'r';
   <* 'r' is the value of the letter r, this
```

```
        letter is added to the position of the
        null character.
        Number is now containing a text portion and
        can be written out as follows:
   write (out,string number) *>
   movestring (address, 1, <: Lautrupbjerg 1-3, Ballerup :>
   <* address is now containing a text packed in an array
        and may be written: write (out, address; *>
end;
```

5.2 Isolation of Text Parts

Fields can be used to single out parts of a text string stored in an array.

If a part of a text stored in a long or real variable is to be moved to another variable this may be done by use of the operators: shift and extract.

Example 5-2, text parts.

Let the long variable number contain 'ab1234', where ab is an item specification and 1234 is a serial number, these two parts of number should be split into two variables, an integer item and a long partnr.

```
begin
  integer item;
  long partnr;
  <* number is assigned outside this block *>
  item:= (number shift (-32)) shift 8 extract 24;
  partnr:= number shift 16;
    <* Note, this is a textstring not a number *>
end
```

5.3 Comparison of Texts

As a text constant - string literal - is represented by an address made by the compiler, a simple compare between long texts is not possible. The texts must be stored in variables and the variables may then be compared (for other possibilities cf. 5.4).

Example 5-3, comparing texts.

Cf. example 5-2. Test that the long partnr is equal to the text <:1234:>.

```
if partnr= long <:1234:> then ... else ...;
```

If a text should be compared with an integer the following statement will work

```
if item= long <:ab:> shift(-24) extract(24)
   then ... else ...;
```

Example 5-4, comparing long texts

A test like the following will never be true

```
t:= long <:a12345:>; <*long string*>
...
if t= long <:a12345:> then goto equal;
...
```

In this case it is neccesary to move both text strings to arrays:

```
real array ra1,ra2(1:n);
movestring (ra1,1,<:a12345:>);
...
movestring (ra2,1,<:a12345:>);
...
if ra1(1)= ra2(1) then
   begin
    if ra1(2)= ra2(2) then
    goto equal;
   end;
...
```

5.4 Text Handling Procedures

Procedures to handle texts are found in the algol library, and are:

read, readall, readchar, readstring (read, readall and readstring may read from arrays)

write, writeint, outchar, outdate, outtext (write and writeint may write into arrays)

movestring, tofromchar, pos, len

isotable, intable, outtable, tableindex, outindex

The procedures read, readall, readstring, write and writeint, which may read from/write into arrays are specially helpful when handling texts.

The subpackage Algol Text Procedures is a collection of text handling procedures for more advanced text handling and text comparison.

5.5 Text Handling in FORTRAN Programs

The present chapter may be fully adapted to FORTRAN programs with the following corrections:

Short texts are texts with at most 6 characters (Hollerith text or apostrophed text), which may be assigned to long or real variables (internal type is long). The characters are left justified with NULL filling (so the last position need not be zero).

Extended texts are only allowed in DATA statements, i.e. apostrophed texts of any number of characters are assigned to data elements in a COMMON block, e.g. to arrays.

Since string parameters and string expressions are unknown in FORTRAN, the procedure movestring is of no use and the procedure write cannot be called with an actual string parameter.

The procedures read and write may be used if called through alias names, since read and write are reserved names in FORTRAN.

6. Mathematical Procedures

In this chapter is shown a list and a short description of the available mathematical procedures within the ALGOL8 system. See further descriptions in [15]. The standard RC8000 software mathematical and statistical package [16] will contain additional procedures.

The procedures here and in [16] are usable in FORTRAN programs as well as in ALGOL programs. For some of the procedures in [16], precautions should be made for the ALGOL parameter type "call by name" when the so-called "Jensens Device" is exploited.

arcsin	Is the mathematical function arcsin(r).
arctan	Is the mathematical function arctan(r).
arg	Is the argument in radians of the complex number(u,v).
cos	Is the mathematical function cos(r).
exp	Is the exponential function e**r.
ln	Is the natural logarithm function ln(r), base e.
random	Computes two pseudorandom numbers, a real and an integer.
sgn	Yields -1 or 1 according to the sign of the parameter.
sign	As sgn but = 0 if parameter is 0.
sin	Is the mathematical function sin(r).
sinh	Is the mathematical function sinh(r).
sqrt	Is the square root function sqrt(r).
tan	Is the same as arctan, the name is for use in fortran programs
log	Is the same as ln, the name is for use in fortran programs

7. Operators And Standard Procedures Working As Operators

This chapter contains a list and a short description of the available operators defined in the language.

Furthermore, standard procedures with related functions are listed. See further description in [15].

7.1 Arithmetic Operators

The arithmetic operators:

+	`plus`
-	`minus`
*	`multiply`
/	`divide`
//	`integer divide`
**	`exponentiation`

have their usual meaning.

Operators performing arithmetic operations:

`abs`
This operator yields the absolute value of integer, long or real expressions.

`entier`
This operator transfers a real expression to the largest integer not greater than the expression.

`mod`
This operator yields the remainder corresponding to an integer division.

7.1.1 Transfer Functions

The transfer functions will cause a type change of the operand. Assignment of a variable to an expression of different type, will cause a type transfer, except for type boolean.

extend
This operator converts an integer expression into a type long.

extract
This pattern operator extracts a number of the rightmost bits, the result is of type integer.

long
This operator changes the type of a string or real expression into type long.

real
This operator changes the type of a string or long expression into a type real.

round
This operator rounds the value of a real or long expression to the nearest integer.

string
This operator changes the type of a real or long expression into type string.

7.2 Logical Operators

The logical or boolean operators:

and	logical and
or	logical or
->	implication
==	equivalence
not	logical negation

have their usual meaning.

Standard procedures performing logical operations:

exor
Is the logical function exlusive or.

logand
Is the logical function and. More types can be used as operands.

logor
Is the logical function or. More types can be used as operands.

7.3 Pattern Operators

These operators can be used in both arithmetic and boolean expressions:

`add`
This dyadic pattern operator will perform a binary addition. The type of the left hand operand will determine the type of the result.

`extract`
This pattern operator extracts a number of the rightmost bits, the result is of type integer.

`shift`
This pattern operator will perform a logical shift of the left hand operand.

7.4 Relational Operators

The relational operators:

<	`less than`
<=	`less than or equal`
=	`equal`
>=	`greater than or equal`
>	`greater than`
<>	`not equal`

have their usual meaning.

8. Standard Identifiers And Procedures

This chapter contains a short description of the standard identifiers and procedures not mentioned in the previous chapters. Further description in [15], [17] and [18].

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

The elements are listed after functions.

8.1 Run Time Survey

Procedures to control the program at run time:

systime
Systime gives access to the real time clock in the monitor and to the CPU time used by the job.

trap
A standard procedure changing the traplabel in an ALGOL block. The traplabel defines the program returnpoint in case of an error detected in runtime system check routine.

getalarm
A standard procedure to grap the runtime alarm which would have been printed, if the program had terminated and trapmode had allowed it to be output.

alarmcause
Contains the cause of a suppressed runtime alarm.

trapmode
This bitpattern may cause the output of an runtime error message to be skipped.

endaction
The value governs the action when the program terminates:
- return to the file processor 'end program'
- return to the file processor 'break routine'
- send a 'finis' message to the parent

errorbits
The value defines the 'end program condition' when returning to the file processor.

overflows
This integer variable controls the actions on floating point overflow.

underflows
This integer variable controls the action on floating point underflow.

blocksread
This integer variable counts the number of times segments have been transferred from the backing storage to the memory. The value of this variable is printed after the program is finished.

progsize
This integer variable holds the size of the program as it was just after compilation (program segments + 1 segment for owns).

blocksout
This integer variable counts the number of times segments have been transferred from virtual store to the virtual storage file.

rc8000
This boolean variable tells whether or not the program is executing on an RC8000 (as opposed to RC9000-10).

8.2 Locking of Segments in Core

lock
Transfers a number of program segments to core and locks them i.e. they will not be overwritten by other program segments.

locked
Transfers the segment numbers of the segments locked for the moment to an integer array.

progmode
This standard variable specifies whether the current segment should be locked or not.

8.3 Miscellaneous

increase
This procedure will automatically increase the argument by one.

movestring
Copies a literal text string to an array.

tofrom
The procedure will copy a set of data from one array or zone record to another.

tofromchar
The procedure will copy a number of characters from a given position in one array or zone record to a given position in another.

algol
This compiler directive directs the compiler to read source texts from various sources during compilation.

pos
The procedure will locate a substring in another string of characters, stored in arrays.

len
The procedure will identify the length of a given string of characters, stored in an array.

8.4 Procedures for System Communication

monitor
This procedure is the Algol/fortran equivalent to the monitor procedures (monitor interface).

system
This procedure gives access to various system and job parameters (runtime system interface).

fpproc
This procedure gives access to a selection of the most useful routines embedded in the file processor (file processor interface).

8.5 Program to Program Communication

fpmode
Tests an fp mode bit.

setfpmode
Sets or removes an fp mode bit.

8.6 Zone Handling Procedures

getzone, getzone6
Transfers the contents of a zone descriptor to an array.

setzone, setzone6
Transfers the contents of an array to a zone descriptor.

getshare, getshare6
Transfers the contents of a share descriptor to an array.

setshare, setshare6
Transfers the contents of an array to a share descriptor

getstate
Gets the zone state from a zone descriptor.

setstate
Sets the zone state in a zone descriptor.

getposition
Gets the logical position from a zone descriptor.

setposition
Sets the logical position in a zone descriptor and positions the document if it is a magnetic tape.

initzones
Changes the buffersizes and number of shares of the zones in a zone array.

resetzones
Resets the buffersizes and number of shares of the zones in a zone array.

open
Initializes the zone with a given document name and prepares the zone for input/output.

close
Terminates the current use of a zone and makes it ready for a new call of open.

stopzone
Terminates the current input/output in the zone making the zone ready to resume input/output.

8.7 Input/Output Procedures

read, readall,readchar, readstring, repeatchar
Inputs one or more items in character form from a document or an array, possibly with character conversion, converts the items to algol values, and assigns them to variables or arrays.

write, writeint, outchar, outdate, outinteger, outtext
Outputs one or more algol values in character form to a document or an array, possibly with character conversion.

replacechar
Changes the special characters of a layout field used by the algol number to character output procedures.

inrec, inrec6, invar
Makes the next sequence of halfwords or the next variable length record from a document available as a zone record.

outrec, outvar, outrec6
Creates a zone record, initially undefined, or a variable length record with contents from an array, intended for output to a document at the next change of block.

changerec, changerec6, changevar
Regrets the latest call of inrec/inrec6, outrec/ outrec6 or swoprec/swoprec6 or outvar and makes a new zone record available (initializes a new variable length record).

swoprec, swoprec6
Makes a zone record available, with contents taken from the next halfwords in the sequence from a backing storage document, intended for output to the same position of the document at the next blockchange.

checkvar
Calculates the record checksum of a variable length record as generated by outvar.

stderror
The standard error block procedure intended for use in zones where you are satisfied with the standard error actions for the given kind of document. The procedure just gives up and terminates the program with a runtime device alarm.

blockproc
Calls the blockprocedure of a given zone.

check
Checks a transfer to or from a document in the way used by the high level zone procedures.

8.8 Character Conversion

isotable
Initializes a given integer array with the standard ISO 7 bit character table for use in intable or outtable.

intable, tableindex
Exchanges the current input alphabet or modifies the current input alphabet used by all character reading procedures.

outtable, outindex
Exchanges the current output alphabet or modifies current output alphabet used by all character writing procedures.

8.9 Sorting Procedures

The procedures perform in-memory sorting. For further details cf. [17].

newsort
Creates a zone record in memory with initially undefined contents, intended to be an active record in the next call of any sorting procedure.

deadsort
Creates a zone record in memory, with initially undefined contents, intended to be an inactive record in the next call of any sorting procedure, i.e. it will not be made available by the procedure outsort.

lifesort
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 as well as the inactive records in the zone, and all inactive records are made active.

initsort
Initiates a sorting process in a zone to be used by newsort, outsort, deadsort and lifesort.

outsort
Makes available a zone record, which is the winner in a sorting process among the set of active records in the zone, intended to be taken away from the zone before the next call of any sorting procedure.

initkey
Generates a piece of code for comparison of two records in a zone to be used by newsort, outsort and lifesort. The procedure may be replaced by changekey6 and startkey6.

sortcomp
Compares two records, using the key comparison code generated by startkey6, changekey6 or initkey.

startsort6
Initiates a sorting process in a zone, and generates a piece of key comparison code to be used by newsort, outsort, deadsort, lifesort and sortcomp.

changekey6
Makes it possible to change the key code generated by startsort6.

8.10 Procedures in connection with context

openvirtual	cf. 4.4.1
virtual	cf. 4.4.1
resume	cf. 4.2.3

8.11 FORMAT8000 Procedures

A set of procedures to process IBM 3270 compatible transactions, for further details cf. [15] and [18].

waitttrans
Awaits the arrival of a FORMAT8000 transaction and gets the fields of the transaction head.

readfield
Inputs the next field designator of a FORMAT8000 transaction.

opentrans
Outputs a transaction head of FORMAT8000 transaction.

writefield
Outputs a field designator in a FORMAT8000 transaction.

closetrans
Terminates the current FORMAT8000 output transaction.

getf8000tab
Gives access to the FORMAT8000 character input table.

f8000table
Exchanges the current input alphabet with the FORMAT8000 character input table.

8.12 Coroutines

The procedures implement a coroutine concept in ALGOL/FORTRAN, called activities. For further details, cf. [15] and [18].

activity
Creates a number of empty activities.

newactivity
Initiates an empty activity with a procedure (coroutine) and starts the activity.

passivate
De-activates the executing activity establishing its restart point.

activate
Restarts a non-empty activity at its restart point.

w_activity
Waits for an event in the event queue (message or answer), and supplies the identification of of the activity responsible for the event.

8.13 Input in One/Output in Many Zones

The procedures offer the possibility for input_in_one/ output_in_many zones in parallel, multibuffered and without need of in-memory data movement.

buflengthio
For a given blocklength and a given number of shares, the procedure returns the bufferlength to be used in a zone array declaration intended for inoutrec.

openinout
Prepares a zone array for input in one/output in many zones, each zone already being connected to a document.

expellinout
Expells a zone among the set of output zones in the zone array from further use until another closeinout or openinout.

closeinout
Terminates the use of the zones in the zone array making them ready for other use as before openinout.

inoutrec
Makes a record available in the zones of the array. The contents of the record is obtained by input of the next halfwords from the input document, and it is available for modification before it is output to all outputzones.

changerecio
Regrets the latest call of inoutrec and makes a new zone record available, maybe a shorter one, part of the present record, or a new one, obtained by blockchange.

9. Operating System Created In ALGOL8

The chapter is fully adaptable to FORTRAN programming, so whenever an ALGOL program is mentioned, it could as well be a FORTRAN program.

9.1 Primitive Level Zone Procedures

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

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

getzone6
Transfers the contents of a zone descriptor to an array.

setzone6
Transfers the contents of an array to a zone descriptor.

getshare6
Transfers the contents of a share descriptor in a zone to an array.

setshare6
Transfers the contents of an array to a share descriptor in a zone.

initzones
May change the buffersize and number of shares of each zone in a zone array.

monitor
This procedure is the ALGOL equivalent to all the functions of the monitor. It looks up, changes, creates or removes catalog entries, it starts and stops communication with peripheral devices, it creates, starts, stops, and removes child processes, etc..

blockproc
Calls the block procedure of a given zone.

check
Checks a transfer to or from a document in the way used by the high level zone procedures.

9.2 Document Driver

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

1. Use entry 20, 'wait message'(or entry 24 'wait event'), in 'monitor' to wait for messages sent to the ALGOL program. The sender of the message assumes that the ALGOL program is a document.

2. Copy the block of information described in the message into a zone buffer area by means of 'system', entry 5, 'move data', or use entry 84, 'general copy', in 'monitor'.

3. Send the answer to the message by means of entry 22, 'send answer', in 'monitor'.

4. Output the block of information to the document.

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

9.3 Operating System

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

1. Use entry 56 in 'monitor' to create the child process in a zone buffer area. It may be necessary to use entry 72 in 'monitor' to set your own catalog base in order to define the base of the process name.

2. Include the process as a user of some peripheral devices by means of entry 12 in 'monitor', and give the process access to the backing storage by means of entry 78 in 'monitor'.

3. Initialize the child process area with a suitable binary program for example the File Processor code which may be read directly from the backing storage area, fp, into the zone buffer area.

4. Set the machine registers of the child process by means of entry 62 in 'monitor'. See [6] if fp is used.

5. Start the child process by means of entry 58 in 'monitor'. Now, the child process starts executing the instructions of the binary program. We say that it runs in parallel with the other processes in the computer (including your ALGOL program). If fp is the executive system, the user base is communicated so that this is the catalog base at which the child process was started. fp will as its first action set the catalog base to standard.

6. When you want to stop the child, use entry 60 in 'monitor'.

7. Wait for the completion of the stop by means of entry 18 or 24 in 'monitor'. Now, all modifications of the child process area are ceased, and you may for instance store the area on the backing storage, use the area for something else, later reestablish the process area and start the child again by means of entry 58 in 'monitor' so that it continues as if nothing had happened.

8. When you want to get rid of the child and withdraw its resources, you use entry 64 of 'monitor'. Remember the process must be stopped first.

In order to make an operating system, which handles several child processes, serves as a driver for peripheral devices, and communicates with the operator, you have to mix the principles of 10.1, 10.2, and 10.3. In this mixing, entry 24 of 'monitor' is very useful to help the program to serve the first arriving event first. An event is here the arrival of a message or an answer, or the completion of a stop.

10. Program Translation And Execution

This chapter describes the structure of ALGOL programs, the ALGOL compiler, and the execution system.

Since the structure of a FORTRAN program is the same as for an ALGOL program, and the execution system is one and the same, most of the present chapter adapts to FORTRAN programs.

10.1 Translatable and Linkable Program Structures

Program structures to be translated by the ALGOL compiler are:

- ALGOL programs
- ALGOL external procedures

Program structures to be translated by the FORTRAN compiler are:

- FORTRAN programs
- FORTRAN external procedures and functions

ALGOL external procedures may be linked into an ALGOL program, which it will be as a result of any reference to its name in the translated program text, just as code procedures (cf. [10]) may be linked into an ALGOL program.

FORTRAN external subroutines and functions (cf. [9]), too, may be linked into an ALGOL program, just as ALGOL external procedures may be linked into a FORTRAN program, along with code procedures.

One special program structure to be linked into programs by both compilers is the ALGOL/FORTRAN RUNTIME SYSTEM, in short: RTS. RTS, which is normally kept in a file called 'algftnrts' contains:

- the runtime system entries (variables and routines) described in [10], embedded in the memory residing part

- routines for alarm printing, zone declaration, common data and zone common block initialization, input/output system, checksystem, standard error handling, power function (**) and others, distributed over a number of non-resident program segments.

- a program initialization part executed at program start

Other linkable program structures are taken from files or libraries, which are files shared by a number of linkable program structures, 'compressed' by programs like compresslib, lib, contract etc. The files 'algollib' and 'fortranlib' are such files, both names also being the names of external procedures doing nothing but writing on current out the list of names belonging to the library.

Translated ALGOL and FORTRAN programs, too, may be 'compressed' into libraries of programs, e.g. by programs like compress, lib, contract etc.

10.1.1 Program Structure

An ALGOL program is a block.

The syntax of a program is:

```
begin
   D1; ...; DM;
   S1;
   ...
   SN;
end;
```

Where S1, ..., SN are ALGOL statements, which therefore may be blocks (cf. [14], and D1, ..., DM are declarations.

10.1.2 External Procedure Structure

An external procedure in ALGOL is a separately translated procedure, included in the catalog. It can be used as a standard procedure for later translated programs and externals.

External procedure syntax:

```
external
      <procedure declaration>;
end
```

where <procedure declaration> is a declaration of a procedure, (cf. [14]).

10.1.3 Examples

Example 10-1, external procedure

```
external
   procedure heading(line,page,date);
   integer line,page; long array date;
   <* date contains yyyy.mm.dd *>
   begin
    write(out,"ff",1,date,"sp",20,<:RC COMPUTER:>,
          "sp",20,<<bddd>,page,"nl",3);
    page:= page+1; line:= 4;
   end;
end
```

Example 10-2, calling external procedure

This is an example of an ALGOL8 program. It reads some records and writes a message depending on the record type. The external procedure from example 10-1 is used.

```
begin
   <* declaration *>
   integer line, page, maxlines, stop_element;
   long array date(1:2);
   long array field rec_name;
   integer field record_type, rec_ident, update_type;
   zone zin(128,1,stderror);
   procedure insert;
   ...
   procedure delete;
   ...
   procedure update;
   ...
   <* initialization *>
   maxlines:= 50;
   stop_element:= 999999;
   page:= 1;
   line:= maxlines;
   rec_name:= 8;
   record_type:= 4;
   rec_ident:= 5;
   update_type:= 6;
   movestring (date, 1, <:1987.08.24:>);
   <* program start *>
   <* connect the zone to the area <:records:> *>
   open(zin,4,<:records:>,0);

   repeat

     invar(zin);
     if line >= maxlines then
        heading(line,page,date)
     else
        line:= line+1;
```

```
      case zin.record_type of
      begin
        begin <* record type=1:insert *>
          write(out,"nl",1,<:insert record:  :>,
             zin.rec_ident,"sp",3,zin.rec_name);
          <* call insert procedure *>
          insert;
        end <* record_type=1 *>;
        begin <* record_type=2:deletion *>
           write(out,"nl",1,<:delete record:  :>
              zin.rec_ident,"sp",3,zin.rec_name);
           <* call delete procedure *>
           delete;
         end <* record_type=2 *>;
         begin <* record_type=3:update *>
           write(out,"nl",1,<:update record:  :>,
             zin.rec_ident,"sp",3,zin.rec_name,
             "sp",3,zin.update_type);
           <* call update procedure *>
           update;
         end <* record_type=3 *>
      end case;
   until zin.record_type=stop_element;
   close(zin,true);
end;
```

Example 10-3, calling a FORTRAN external subroutine

This is an example of a (very simple) ALGOL program calling a FORTRAN external subroutine, which again calls an ALGOL external procedure:

```
begin
 write_ftnlib;
end;
```

where the external FORTRAN subroutine looks like:

```
SUBROUTINE WRITE_FTNLIB
CALL FORTRANLIB
END
```

Example 10-4, FORTRAN program calling an ALGOL external procedure

This is an (also very simple) example of a FORTRAN program calling an ALGOL external procedure:

```
PROGRAM XXX
CALL ALGOLLIB
END
```

10.2 Translation

Most of the section is valid information for the FORTRAN compiler as well as the ALGOL compiler, only 10.2.2 is specific for ALGOL, although many parameters are found in the call of the FORTRAN compiler with the same meaning.

The compiler works in your job process and you start the translation by means of an FP command specifying the source text of a program or an external procedure, the compilation options, and the file where the resulting object program should end (cf. 10.2.2.1).

The result of the translation is either a complete, self-contained, binary program or a binary external procedure. In the first case, the program may be executed as described in 10.3. In the second case, the procedure may be used as a standard procedure in later translations. If you permanent the program or the procedure (give it scope user or scope project), you can use it in later jobs.

10.2.1 The Compiler.

The compiler occupies about 16000 (FORTRAN about 19000) instructions divided into 11 passes, either on backing storage or on magnetic tape. In the first case, it may be used for simultaneous translation in several job processes.

The 11 passes of the compiler perform the following tasks: Pass 0 is a common administration routine. Pass 1 to 8 perform the translation into binary code by means of 8 scans of the source program. The intermediate program text is stored in the place later occupied by the binary program. Pass 9 rearranges the binary program, inserts references to standard procedures, and includes the code of the runtime system (RTS) and the code of the external procedures used in the program. When an external procedure is translated, pass 9 only rearranges the binary procedure and RTS is not included.

Pass 11 does not exist, but in ALGOL a pass 12 may make crossreferences of the different names used.

In FORTRAN it takes the preprocessor XFORTRAN to make crossreference lists.

10.2.2 Call of the Compiler

Here is shown the File processor commands used in connection with call of the compiler. Further description of the File Processor is given in [6]. The notation is described in [14]. The call of the FORTRAN compiler is described in [9].

10.2.2.1 Syntax

```
                   { <s> <source>   } *
<bs file> = algol  {                }
                   { <s> <modifier> } 0


<source> ::= <text file>


                    {  { blocks     }                                                    }
                    {  { bossline   }                                                    }
                    {  { connect    }                                                    }
                    {  { fp         }                                                    }
                    {  { index      }                                                    }
                    {  { spill      }   { yes }                                          }
                    {  { list       } . {     }                                          }
                    {  { message    }   { no  }                                          }
                    {  { survey     }                                                    }
                    {  { ix         }                                                    }
                    {  { zonecheck  }                                                    }
                    {  { code       }                                                    }
                    {                                                                    }
                    {                   { on  }                                          }
                    {     list        . {     }                                          }
                    {                   { off }                                          }
                    {                                                                    }
                    {                                    *                               }
<modifier>  ::=     {     copy        . { <copy source> }                                }
                    {                                    1                               }
                    {                                                                    }
                    {     rts         . <rts file>                                       }
                    {                                                                    }
                    {                   { yes         }                                  }
                    {     stop        . { no          }                                  }
                    {                   { <last pass> }                                  }
                    {                                                                    }
                    {                   { no                                             }
                    {     xref        . { yes                                     1   1  }
                    {                   { <connections> { .<intervals> { <sortarea> }  } }
                    {                                                             0   0  }
                    {                                                                    }
                    {                   { yes                                          } }
                    {                   { no                                           } }
                    {     details     . {                                              } }
                    {                   { <first pass> . <last pass>                   } }
                    {                   { <first pass> . <last pass> . <first line> . <last line> } }

<copy source> ::= <name>
                  { all     } *
                  { declare }
<connections> ::= {         }
                  { assign  }
                  { use     } 1
                                                                                     1
<intervals> ::= <firstline> . <last line> . { <first name line> . <last name line> }
                                                                                     0
{ <first line>      }
{ <last line>       }
{ <first name line> }
{ <last name line>  }  ::= <integer>
{ <first pass>      }
{ <last pass>       }

<sortarea> ::= <name>
```

10.2.2.2 Semantics

<bs file>
A file descriptor describing a backing storage area. It is used as working area for the compilation, and the object code ends up here and will be described in the file descriptor. If <bs file> does not exist an area is created on the disc where the job has maximum temporary resources. After a possible creation, the area is made as large as possible leaving 1 slice on the device. When the compilation terminates the area is cut to the used number of segments. An existing area, however, is never cut.

<source>
The list of source files specifies the input files to the compiler. If no

source file is specified, the compiler reads the source text from current input.

<modifier>
The list of modifiers is scanned from left to right. Each modifier changes the variables which control the compilation. When the scan starts, the variables are initialized to the values stated below.

blocks.yes
In case the program text is listed, a blocknumber will be listed in a column at the left hand margin. The blocknumber is increased by one, if a begin is found in the input - and decreased by one if an end is found.

The initial setting is 'blocks.no'.

bossline.yes
Implies that listing or messages besides the linenumber will state the BOSS linenumber. The initial setting is 'bossline.no'.

connect.no
Implies that a text parameter following the call of the translated program will not be connected as current input (cf. 10.3.2.2).

The initial setting is 'connect.yes'.

copy.<copy sources> >
This parameter will make it possible to copy the specified <copy sources> into the main source text. Further description in 10.2.2.3.

details.yes
This parameter is mostly used for compiler debugging. Intermediate output from all the passes of the compiler is printed on current output. The output may be restricted to an interval of pass numbers and to an interval of line numbers. The output from pass 8 (for instance caused by 'details.8.8') consists of a list of those line numbers which correspond to segment boundaries in the object program.

The initial setting is 'details.no'.

code.yes
The parameter is mostly used for compiler debugging. The code generated by pass 8 will be listed on current output in disassembled form (SLANG-like) as it is produced: backwards over the segments, starting in address 502 or 500 of the last program segment. The lines of the output has the format:~ <rel. addr.> <rel. segm.> <code>~ where <rel. addr.> is the relative address on the segment and <rel. segm.> is segment number - no. of segments. The last program segment, then, will be -1, the preceding one -2, and so forth. The option will override the option "details.8.8".

The default setting is 'code.no'.

fp.yes
Implies that the call of the translated program may use an integer as first parameter without removing the File Processor (cf. 10.3.2.2).

The initial setting is 'fp.yes'.

ix.yes
Code for dynamic indexing/index check, using the "ix" double-word instruction, is generated. Compared to the traditional suite of instructions doing indexing and index check, 5 instructions will be replaced by one double-word instruction by this option. Note, that index check is embedded in the instruction "ix" and cannot be omitted. The option is intended, of course, for execution on CPU's having this instruction in its instruction set, but may be executed on any other CPU. In case of no "ix" instruction available on the CPU executing, it will be emulated by the runtime system, the penalty being an execution time of 4-5 times that of the normal execution for the "ix" instruction. If the program zeroes the "floating point exception active" bit, indexing outside the bounds by the "ix" instruction (in-line code or emulated) will lead to indexing the upper bound element.

Default setting is ix.no, execpt when an entry named "algolix" is visible, in which case the default setting will be

- ix.yes, if the translating CPU can execute an "ix" instruction
- ix.no otherwise

The existence of an entry "algolix" is entirely an installation or user matter. It may be set by the command algolix=assign algol, in which case it becomes another name of the compiler.

index.no
Code for dynamic check of subscripts against bounds is omitted. The code is reduced by 3 out of 5 instructions generated for each indexing and simple fielding (+ zone indexing and zone simple fielding) and execution becomes considerably faster. The initial setting is 'index.yes'.~
The option is not available with ix.yes, cf. above.

list.yes
The entire source text is listed on current output with line numbers in front of each line. See further description in 10.2.2.4.

The initial setting is 'list.no'.

list.on
The following source text is listed on current output with line numbers in front of each line. See further description in 10.2.2.4.

The initial setting is 'list.off'.

message.no
Normally, the text preceding the first begin and all comments denoted by message in the source text are listed with line numbers. With 'message.no' this listing is omitted. The initial setting is 'message.yes'.

spill.yes
Dynamic check of integer overflow is performed. Even if the external procedures referenced were translated with spill.no, a partial check of integer overflow is performed when they are executed.

The initial setting is 'spill.no'.

stop.<last pass>
The translation is terminated after the pass specified. Stop.yes terminates the translation after pass 9. The translation is regarded as unsuccessful. The following shortcut is useful in case of large programs and/or heavy system loads: If only a listing and a crossreference is wanted the translation can be stopped after the second pass: stop.2.

If error messages are wanted stop after pass 6.

The initial setting is 'stop.no'.

survey.yes
A summary is printed on current output after the completion of each pass of the translation. The meaning of the summary from pass 9 is explained in [10] and in 10.2.7.

The initial setting is 'survey.no'.

xref.yes
A crossreference listing (xref-list) is printed on current output after a possible listing. The xref-list is a listing of the identifiers used in the program. The list contains an occurence list for each identifier. Further description in 10.2.2.5.

The initial setting is 'xref.no'.

rts.<rts file>
The parameter is intended for runtime system debugging and testing. The file <rts file> will be linked into the program as runtime system by the linker, pass 9. The tail of the catalog entry for <rts file> is supposed to contain the values:

```
tail (1)      : number of rts segments+1
tail (2-5)    : documentname (= name of disc)
tail (6)      : 1<23 + rel.addr. of rts init code
tail (7)      : kind <18 + size of rts table
tail (8)      : 0
tail (9)      : 4<12 + start external list
tail (10)     : no. of rts segments + size of rts own
                area
```

The kind in tail (7) must be 15 = 'illegal procedure' and the size in tail (1) must be > 0, i.e. the runtime system cannot be part of a 'compressed library'.

The default setting is 'rts.algftnrts'.

zonecheck.yes
The parameter is used to check that zones are not still 'active' when the block of declaration is abandoned, i.e. if the bit 1 shift 10 is set in the give up mask of the zone and the state of the zone is not 'after declaration' = 'after close' = 4 and it is not 'in sort' = 9 at the time the declaration block of the zone is left, then the blockprocedure of the zone is called. It is imperative that the blockprocedure should be designed to handle this situation.

The default setting is 'zonecheck.no'.

10.2.2.3 The Algol and Copy Concept

The **compiler parameter:**

```
     {                 } *
copy {.<copy sources>}
     {                 } 1
```

makes it possible to copy the specified <copy sources> into the main source text.

The source text will specify when the copy is to take place.

The following **compiler directive** is used in the source text:

```
              { on } 1         { <copy source> }   1
algol { list.{     }}  { copy.{                 } }
              { off} 0         { <integer>      }   0
```

```
<copy source>::= <name>
```

In case of

```
copy .<copy source>
```

the content of the specified copy source will be copied to the place specified in the source text.

In case of

```
copy .<integer>
```

the integer parameter is matched with the copy parameters of the compiler call which are numbered 1, 2, ... The matched copy source is copied to the place specified in the source text.

The source text delimiter 'algol' is by the compiler treated as message, i.e. it is listed unless the parameter mesage.no is specified, and the delimiter must follow either begin or semicolon, and it must be terminated by semicolon.

Description of the modifier 'list' cf. 10.2.2.4.

Example 10-3

See example 10-6 in 10.2.2.4.

10.2.2.4 Details on Listing

There are three levels of listing of an ALGOL program.

1. The compiler parameter:

```
          ( yes )
     list.(     )
          ( no  )
```

```
list.yes     the total source text is listed.

list.no      nothing is listed.

                           ( yes )
The last (right most) list.(     )
                           ( no  )

modifier in the call is used.
```

2. The compiler parameter:

```
      ( on  )
list. (     )
      ( off )
```

```
                                ( yes )
This modifier is blind if list. (     )
                                ( no  )

is used.
```

list.on
will cause the listing of the following source texts, until a possible list.off is met.

list.off
will cause no listing of the following source texts, until a possible list.on is met.

Default value depends on the File Processor mode bit:

```
         ( yes )
listing. (     )
         ( no  )
```

listing.yes
everything is listed until a list.off is met.

listing.no
nothing is listed until a list.on is met.

The initial setting is listing.no.

3. The compiler directive list.(on/off)

This directive may be written in the source text:

```
       {      on  } 1          { <copy source> } 1
algol {{ list.    }}  { copy.{{                }}
       {      off } 0          { <integer>     } 0
```

Cf. 10.2.2.3.

If a list parameter is not followed by a copy source parameter, it means that the listmode of the actual source is changed.

If a list parameter is followed by a copy source, the list parameter relates only to the copy source.

If no list parameter is specified for the copy source, the copy source will be listed in case the actual source is listed.

A list parameter in front of copy.<integer> will be blind. The list mode specified in the call will be used.

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

Example 10-4, listing.

```
p = algol text1 text2
nothing is listed.

p = algol text1 text2 list.yes
everything is listed.
```

Example 10-5, listing.

```
p = algol text1 list.on text2
If fp mode listing.no text2 is listed.
If fp mode listing.yes text1 and text2 are listed.
```

Example 10-6, listing.

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

source 1      = t0 copy source 1 = t1 copy source 2 = t2
copy source 3 = t3


File    Contents                               Note

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

t1:        comment copy source no.1;              listed
t2:        comment copy source no.2;              listed
t3:        comment copy source no.3;              not listed
t4:        comment copy source no.4;              listed
```

Output:

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


t0  d.8    71111.1436
      10       1 begin
  1. lin   e    30    3 algol list.on;
      40       4 comment 1;
      50       5 algol list.off;
      lin  e    70    7  algol copy.1<         *t1*>;
t1  d.8    71111.1433
      10       7 comment copy source no        .1;
      20       8

t0
      lin  e    90    9  algol list.on          copy.t4;
t4  d.8    71111.1433
      10       9 comment copy source t4        ;
      20      10

t0
      lin  e   110    11  algol copy.2<        *t2*>;
t2  d.8    71111.1433
      10       11 comment copy source n        o.2;
```

```
     20       12

t0
     lin e   120   12  algol copy.3<            *t3*>;
t3  d.8    71111.1433
     lin e    20   13  end medium
t0
algol e    nd
```

10.2.2.5 Details on Crossreference

The crossreference listing (xref-list) is a listing of the identifiers used in the program. The list contains an occurrence list for each identifier listing the linenumbers in which it is represented. These line numbers are split up in 3 different groups, each group starts with a group letter in the listing:

D: meaning the identifier is found in a declaration or specification. A label is considered declared in the line where it is defined.

A: meaning the identifier occured in an assigment, i.e. in front of := . A switch declaration is indicated with a D.

U: meaning all other occurrences.

The xref-list is made with no regard to the block structure of the program. The identifier names are sorted according to the collating sequence.

```
abcdefghijklmnopqrstuvwxyzæøå
ABCDEFGHIJKLMNOPQRSTUVWXYZÆØÅ
0123456789
```

Connections and Intervals

The modifier xref.yes will list all groups of occurrences. But it is possible to select certain groups of occurrences, to specify which part of the program and which part of the identifiers the xref-list should contain.

```
                                  Select occurence group:

connections

declare }    The occurence-lists will only contain the
assign  }    specified groups D, A, U respectively.
use     }

all        This connection is equivalent to the connections
           declare.assign.use.

           xref.yes is equivalent to xref.all.
```

Select program and identifier interval

<first line>.<last line>
The occurrence lists will only contain numbers belonging to the specified interval. If not specified, the line interval will include the entire program.

<first name line>.<last name line>
Only those identifier names, which appear in the specified part of the program, are listed in the xref-list. This parameter restricts the set of identifier names in the xref-list. If not specified, the name line interval will include the entire program.

Examples cf. 10.2.2.6

The *sortarea* is usually created by the compiler. This area is used for sorting the occurrences of the identifiers. A part of the area used for compilation of the program is used. For very large programs it may be necessary to create a specific sortarea. The name of this area may then be specified at the end of the list of modifications to the xref-parameter.

10.2.2.6 Examples

Example 10-7, call of compiler.

```
o lp
sl1=algol list.yes sl2 sl3
```

The final program is stored in sl1. The source is taken from the file described in sl2 followed by the file sl3. The entire source text and all error messages appear on lp (line printer).

```
sl1=algol list.yes stop.1
```

The source text is read from current input and listed on current output. The translation stops after pass 1, i.e. just after the listing. If xref was wanted too stop.2 should be specified.

The following examples show the calls of the compiler with xref-parameter and the corresponding output. The bold headed lines are the commands.

```
algol text list.yes xref.yes
  text d.871111.0849
1  begin integer i, j;
2  procedure pip(a,b);
3  value a;
4  real a; integer array b;
5  b(a):=a;
6
7
7  A:pip(i,ia);
8
8  end
<an FF character is printed here>
```

```
a      D:  4
       U:  2  5
b      D:  4
       A:  5
       U:  2
i      D:  1
       U:  7
ia     D:  6
       U:  7
j      D:  1
pip    D:  2
       U:  7
A      D:  7
No. of identifiers=7
algol end 17
```

```
algol text xref.assign
 text d.871111.0849
1  begin
8  end
  < a FF character is printed here >
  a
  b      A:  5
  i
  ia
  j
  pip
  A
  No. of identifiers=7
  algol end 17
```

```
algol text xref.all.2.10.1.1
  text d.871111.0849
1  begin
8  end
   < an FF character is printed here >
   i      U:  7
   j
   No. of identifiers=2
   algol end 17
```

```
algol text xref.all.4.7.6.6
  text d.871111.0849
1  begin
2  end
    < an FF character is printed here >
    ia
    No. of identifiers=1
    algol end 17
```

10.2.3 Storage Requirements, etc.

The compiler requires a job process with a memory space of 13000 halfwords with 4 message buffers and with 8 area processes (6 if current input and output are not backing storage areas).

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

10.2.4 Speed, Length of Object Code

After a basic time of 2 seconds, the total translation speed is about

1000 characters/second (for RC8000-45),
3000 characters/second (for RC8000-55),
7200 characters/second (for RC9000-10),

500 final instructions per second (for RC8000-45),
1500 final instructions per second (for RC8000-55),
3600 final instructions per second (for RC9000-10),

for an average program.

The final program of, say, i segments, consists of

- the segments of the runtime system (segments 0-15)
- the segments with the code corresponding to the source text (segments 16...n-1)
- the segments of the standard procedures linked into the program (segments n...i-2)
- the first virtual data segment with initial owns (rts owns) (segment i-1)

The length of the code, measured in segments, corresponding to a source text is about 1.5 times the length of the source text, measured in segments.

10.2.5 Error Checking

The compiler performs extensive syntax and type checking, but a few errors may pass undetected as described in 10.3.6.3.

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

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

10.2.6 Messages from the Compiler

In this section, only the messages in 10.2.6.2 coming from other passes than pass 9 are special for the ALGOL compiler. The rest of the section goes for FORTRAN as well.

Four formats of error message exist:

```
1. <pass number>.line <line number>.<operand
number> <text>
   (e.g. 6. line 12.6 type)

2. <pass number>.<text>
   (e.g. 8. program too big)

3. <pass 9>.<name> <text>
   (e.g. 9. write program too big)

4. ***algol <text>
   (e.g. ***algol param)
```

Below, the error messages are sorted according to <text>. The messages are classified as:

(alarm)
The translation will terminate immediately as an unsuccessful execution. The program cannot be executed.

(warning)
The message has no effect. The erroneous construction is skipped.

Nothing
The message allows the translation to continue and the program to be executed until the erroneous construction is met or until an undeclared or doubly declared identifier is used.

10.2.6.1 Line and Operand Numbers

The lines of the program are counted 1, 2, 3, ... where line 1 contains the first 'begin' or 'external'. Only lines containing visible (printing) symbols are counted.

The operands within a line are counted 1, 2, 3, ... An operand is an identifier, a constant, or a string.

The point of the program where an error of form 1 is detected, is specified by the line number and the number of operands passed within the line, for example:

```
source line 12:  if a<-1.5 then b(i):= real<:cd:>; else
operand numbers:    1  2       3 4          5
error message:   6. line 12.5 termination
```

10.2.6.2 Alphabetic List of Error Texts

algol end <i>
This is not an error message. The ALGOL program has been translated. The object code occupies <i> segments. The ok bit (cf. [6]) is set to yes. The warning bit is set to no if no error message has occurred, otherwise it is to yes.

algol sorry <i>
An alarm has occurred. The ok bit is set to no, cf. [6]. The integer i shows the number of segments the compiler has attempted to make.

area
(pass 9). In the start up phase of pass 9, an area process for the actual runtime system could not be created. The result of create area process is shown.

bases
(alarm, pass 9) The bases of the area process created for the next external procedure to be linked do not macth the entry bases for the entry looked up. The alarm is closely related to the alarm 'catalog' from pass 9).

block proc
(pass 6). Error in the declaration of the block procedure of a zone.

blocks
(alarm, pass 5). More than 62 nested blocks.

call
(pass 6). A procedure call has a wrong number of parameters.

case elements
(pass 6). More than 2046 case elements in one element list. Applies to case expressions, case statements and switch declarations.

catalog
(alarm, pass 2). Trouble reading the backing storage catalog. May be caused by too few area claims.
(alarm, pass 9). Trouble with
- catalog lookup, result is printed
- create area entry, result is printed
- size >= 0 but contents < 32 and contents <> 4, size is printed

The trouble could e.g. be caused by a standard identifier disappearing
- catalog 3: Could be caused by a previously translated external procedure or a code procedure containing references to prodedures, which have been changed.

Remedy: retranslate the external procedure.
- catalog 4: Entry not in compressed library or base trouble.

char or illegal
(warning, pass 6). Illegal character or wrong use of a graphic.

comment
(pass 6). Comment or message not after begin or semicolon.

common
(pass 9). In linking a fortran program unit, a common block or zone common block does not match the definition already given by a previoiusly called fortran program unit. May happen linking two or more fortran external program units into an algol program.

compiler directive syntax
(pass 6). Syntactical error in the compiler directive "algol". The erroneous constructions are skipped.

constant
(pass 6). Syntactical error in a constant number.

context zone
(pass 6). A zone is declared among context variables.

context label
(pass 6). The exit operator is at an erroneous block level, or in a for-, repeat-, or whilestatement located within a context block.

+declaration
(pass 6). Identifier declared two or more times in the same block. The message appears at each place of declaration.

delimiter
(pass 6). Impossible sequence of delimiters.

-delimiter
(pass 6). Two operands follow each other.

entry
(alarm, pass 9). A standard identifier has been changed in the catalog the execution of pass 9, or the identifier is described by an auxentry with entry bases differing from the bases of the main entry.

error at source
(alarm, pass 1). Trouble with input from the source file specified. Following possibilities:
<name> unknown
<name> not textfile (contents key <> 0)
<name>.<integer> not magtape
<name> illegal kind
<name> connect error (hard error)
<name> not text
<name> hard error, followed by:
device status <name>
<cause>

ext param
(alarm, pass 5). More than 7 parameters in an external procedure.

external
(pass 6). External-end does not surround a procedure declaration.

for label
(pass 6). Label which labels a statement inside a for-, repeat-, or while-statement from the is used outside of the statement.

head
(pass 6). Impossible procedure head. The line number points to the first symbol of the procedure body.

kind
(pass 9). A standard identifier has been changed in the catalog since the translation started. This is most likely to happen in connection with an external procedure which was translated assuming a certain standard identifier, but now this identifier has been changed in the catalog, beyond the scope of changes accepted by pass 9, cf. the program upextlists. The two kindwords are shown in decimal representation (expected values), in the form <half1>.<half2>, as shown by the program lookup.

In the start up phase of pass 9, an area process for the actual runtime system could not be created. The result of create area process is shown.

layout
(pass 6). Impossible layout.

local
(pass 6). Local variable used in array or zone declaration.

mode kind <i>
(pass 9). The modekind, i.e. catalog entry tail word 1, of an external procedure entry is illegal, e.g. an auxiliary entry to a shared main entry. The modekind is shown in decimal representation.

name trouble <i>
<alarm, pass 9). An external procedure is being translated, but the resulting catalog entry could not be created/changed. The result of change entry is shown.

not text
(pass 1). A source text contains a character > 127.

object area
(alarm, ***algol). The file specified for the object code does not exist, cannot be used, or cannot be created (cf. 10.2.2.2).

operand
(pass 6). Operand appears in wrong context or is missing.

-operand
(pass 6). Operand missing at end of construction.

overflow
(pass 7). Integer or real overflow during evaluation of a constant expression.

owns <i>
(alarm, p ss 9). A reference to an own memory location is based upon an own base big enough to give an own reference exceeding the limit of 4095 halfwords. The own reference is shown.

param
(warning, ***algol). Illegal parameter in the FP command. The parameter is ignored.

pass trouble
(alarm, pass 1-12). The job area is too small to load the next pass or the next pass has been destroyed.

give up, one of above errors too severe
(alarm, pass 7). Insufficient clearing of the source text errors in earlier passes. Will disappear together with these.

program too big
(alarm, pass 1-12). The backing storage area specified cannot hold the object code.

relative
(alarm, pass 9). An undebugged code procedure is assembled. The procedure contains a relative reference outside the interval
`0 <= r <= 510.`
The value of r is shown.

remove process
(pass 9). The area process for the latest linked external procedure could not be removed. The result is shown.

right par improper
(pass 6). The construction (<letter string> is not followed by :(.

rs entry <i>
(pass 9). An unknown runtime system entry was referred, maybe an undebugged code procedure is being linked. The name of the external procedure which contained the reference is shown, and the name of the illegal runtime system entry is shown.

size
(pass 9). An external procedure is being linked, which extends
- the number of program segments beyond the limit of 4095 segments
- the number of halfwords for owns beyond the limit of 4095 halfwords

segs <i>
(alarm, pass 9). A segment reference in a program unit is based on a segment base big enough to give a segment reference beyond the limit of 4095 segments. The segment reference is shown.

sorry <i>
(alarm, ***algol). The translation is unsuccessful, because of an alarm or because the FP parameter 'stop' was used. See also 'algol sorry <i>'.

source exhausted
(pass 1). The source text is exhausted before the program was complete. A clue to the missing termination is printed:
in comment
in comment string

in string
ends missing is output.

sort area
(alarm, pass 12). Cross references could not be made because the sort area could not be created or connected.

stack
(alarm, pass 2-12). The job area is too short for the translation tables (see 10.2.3).

subscripts
(pass 6). A subscripted variable has a wrong number of subscripts.

termination
(pass 6). Parentheses or bracket like structures do not match.

text
(warning, pass 6). Illegal constituent of text string, usually <: or digits in < >.

type
(pass 6). The declaration or type of an operand is not in accordance with its use.

too many parameters
(alarm, pass 7). More than
- 509 actual parameters in a procedure call, or
- 2044 halfwords used in the stack for a procedure call (6 for return information + 4 for formal locations for each actual parameter + 4 for each literal location, i.e. expression computed at the call side).

undeclared
(pass 6). The identifier is not declared. Later occurrences of the identifier in the same block will not result in a message.

unknown
(pass 9). An external is to be linked, but is not found in the catalog. The name of the external is shown.

variables
(alarm, pass 2). The program contains more than 3484 distinct names of identifiers.
(alarm, pass 5). More than 1951 halfwords of simple variables and simple zones in one block, or more than 2047 halfwords of owns in the entire source text, or more than 2047 labels and procedures in the entire source text.

works
(alarm, pass 7). More than 96 halfwords of working locations in one block, e.g. more than 15 nested repetitive statements.

wrong version <i>
(pass 9). The internal version number of an external procedure (algol or fortran) which states the version number of the compiler having translated it, is less than "the smallest version number in an

external procedure acceptable by pass 9", i.e. the procedure needs retranslation. The version number of the external procedure is shown. A survey from pass 9 will expose the 'pseudo external' *version:

```
0 1 0 2 *version
```

where 1 is the "smallest version number of an external procedure acceptable by this version of the compiler" and 2 is the version number of the compiler translating.

wrong version 0
(pass 9). Pass 9 is linking an
- external algol procedure translated with version 0 of the compiler (older than release 11.0, 1979.11.22)
- external fortran subroutine or function translated with version 0 of the compiler (older than release 2.3, 1985.11.01)
- a code procedure which violates the rule that the last word on any segment must end with a 'null' character

Notice that pass 9 continues and that the program will be executable except for alarm printing in the segments of the external (this statement holds up to and including version 2 of the compiler).

xref too big
(alarm, pass 12). The area used for sorting is not large enough.

zone
(pass 6). Wrong number of subscripts after zone or zone array.

zone declaration
(pass 6). Wrong number of commas in zone array declaration.

10.2.7 Assembly, Index, Spill

In this section, only specific information about passes other than pass 0 and pass 9 is special to the ALGOL compiler, the rest is valid for the FORTRAN compiler as well.

10.2.7.1 Assembly

Pass 9 performs the assembly or linking of standard and other external procedures into the main program and if these external procedures refer to other external procedures the assembly continues recursively. All **standard** identifiers must exist in the catalog at this stage.

The pass output from pass 9, caused by the compiler option survey.yes (or details.9.9) will be of interest dealing with the messages from pass 9 or the limitations imposed by the formats handled by pass 9 (cf. 10.2.8).

The output from pass 9 contains one line for the main program and one line for each assembled and linked external, ending with one line of pass information (summary output). If the program unit translated is an external procedure, the summary output decreases to this last line of pass information, where only the second field, used segments, is significant, cf. below.

The output from pass 9 is explained with below example.

```
pptx1 d.890208.1104
        1 begin
       19 end;
1.       -       -               -       -       -
2.       -       -               -       -       -
3.       -       -               -       -       -
4.       -       -               -       -       -
5.       -       -               -       -       -
6.       -       -               -       -       -
7.       -       -               -       -       -
8.       -       -               -       -       -
9.        16    168      16       4    890208    133522 pp1
           .      .       .       .    .         .      .
           .      .       .       .    .         .      name of
           .      .       .       .    .         .      main program
           .      .       .       .    .         time of translation
           .      .       .       .    date           -       -
           .      .       .       own halfword base main program
           .      .       segment                -   -      -
           .      relative address program entry point
           segment number             -      -     -

          6      0    1989    0201     algftnrts
        .      .       .       .    .
        .      .       .       .    name of runtime system
        .      .       .       date of release
        .      .       year
        .      subrelease number (internal rts number)
          release                   (internal rts number)

        0      1       0       2    *version
        .      .       .       .    .
        .      .       .       .    pseudo external *version
        .      .       .       version number of compiler
        .      .       unused
        .      smallest acepted version in any external
          unused
```

```
18    176       17       4    890208    144819 write
 .      .        .       .     .          .      .
 .      .        .       .     .          .      name external
 .      .        .       .     .          .      procedure area
 .      .        .       .     .        time from external lis
 .      .        .       .    date                -    -     -
 .      .        .      own halfword base this external
 .      .      segment                        -   -     -
 .    relative address entry point
   segment number              -   -

 0    410        0       0    out
 .      .        .       .     .
 .      .        .       .    name external zone
 .      .        .      unused
 .      .      unused
 .    addr relative to memory base
   unused

 1    634       17       4    outindex
 .      .        .       .     .
 .      .        .       .    name own core variable
 .      .        .      own halfword nase stille
 .      .      segment
 .    right half of final address
   left

18    178       17       4    writeint
 .      .        .       .     .
 .      .        .       .    name external
 .      .        .       .    entry into write
 .      .        .      own halfword base this external
 .      .      segment                       -    -     -
 .    relative address entry point
   segment number              -   -

 0   1642        0       0    blocksout
 .      .        .       .     .
   .      .        .       .    name of rts variable
 .      .        .      unused
 .      .      unused
 .    right half of final address
   left         -   -   -     -
```

```
       0    183      24     138    0 ....  summary output pass 9
       .      .       .       .    .
       .      .       .       .    unused
       .      .       .     total no of own halfwords (0 if external
       .      .     total no of segments                  procedure)
       .    max (no of globals + no of externals)
               in all externals linked into program
               (in this example the procedure write)
          no of externals in
               max (no of globals + no of externals)
               in all externals linked into program
               (in this example still write)

algol end 24

lookup ppp1

ppp1 = set 24 disc d.890208.1336 6.142 0  2.3248 3930 ; -
 .          . .      .          . . . . . .    .
 .          . .      .          . . . . .      rts load
 .          . .      .          . . . . .      length
 .          . .      .          . . . . rel. rts entry
 .          . .      .          . . . segm.  -  -
 .          . .      .          . .  segment number
 .          . .      .          . .  first program segment
 .          . .      .          . relative address
 .          . .      .          . program descriptor vector
 .          . .      .          segment number
 .          . .      .          program description vector
 .          . .     date and time of translation
 .          .  name of backing storage document
 .         no of program segments (rts + program + init owns)
 name of translated program
```

The summary output from each pass, as seen in the example, is composed of one line with the field:

`<pass no.>. <sum0> <sum1> used segments <int1> <int2>`

Just for the record is here a list of the used fields:

<sum0> is the sum of the halfwords read by the pass

<sum1> is the double sum of the halfwords read by the pass

`used segments` is the number of segments produced by the pass, counting upwards for forward passes, downwards for reverse passes.

`<int1> and <int2> are used by some passes only:`

```
2. <int1> -  result of remove process (<: catalog :>)
3. <int1> -  no. of lexicographical blocks ine the
             program.
4. <int1> -  max. halfwords in stack (internal stack)
   <int2> -  max. words in use (internal stack)
5. <int1> -  no. of occurences of identifiers in
```

```
               statements and expressions
   <int2> -    no. of redeclarations of identifiers
8. <int1> -    (greatest top - stack top)*1000 + sigma
               (last constant - segment base) // no.of code
               segments
  (int2> -     max. line change * 1000 + total moves // no.
               of code segments
```

The summary output of pass 9 is different, as explained in the example.

10.2.7.2 Index

At run time, subscript check will be omitted during the execution of all program parts compiled with ix.no and index.no. All standard procedures may be thought of as compiled with ix.no and index.yes. For further details cf. 10.2.2.2.

10.2.7.3 Spill

If the main program is compiled with spill.yes (cf. 10.2.2.2) , a partial check of integer overflow is performed in procedures compiled with spill.no. If the main program is compiled with spill.no, integer overflow at multiplication will still be detected in subroutines compiled with spill.yes. None of the standard procedures can cause an integer overflow.

10.2.7.4 Program Descriptor Vector

When a program has been translated, it will contain the runtime system as its first segments. As part of the runtime system will be found the program descriptor vector, which is a shared dara area, where the compiler gets information about the runtime system and where it hands over information to the runtime system to be used at program start-up and during program execution. The address of the program descriptor vector is found in tail word 7 of the program catalog entry (cf. 10.2.7.1), and its format is:

```
; PROGRAM DESCRIPTOR - USED FOR COMMUNICATION OF
; VALUES BETWEEN PASS 9 AND THE RUNTIME SYSTEM:

;+ 0 modebit word 1                                         (pass 0)
;+ 2 modebit word2                                          (pass 0)
;+ 4 compiler version                                       (pass 0)
;+ 6 compiler release<12 + compiler subrelease              (pass 0)
;+ 8 compiler release year<12
     + compiler release date                                (pass 0)
;+10 rts version                                             (rts)
;+12 rts release<12 + rts subrelease                         (rts)
;+14 rts release year<12 + rts release date                  (rts)
;+16 interrupt mask                                         (pass 0)
;+18 entry point to main program                            (pass 9)
;+20 length of own area                                     (pass 9)
;+22 length of data table (fortran)                         (pass 9)
;+24 length of zone common table (fortran)                  (pass 9)
```

```
;+26 segment no for first own segment                        (pass 9)
;+28 length of common area                                   (pass 9)
```

10.2.8 Limitations

As can be seen from the compiler messages certain limitations exist, that should be considered when developing programs, especially large programs. Below follows a survey of the limitations, where possible with the message given from the compiler if they are exceeded, and the number of the pass giving the message.

The limitations with a message from pass 9 or with no message are valid limitations for the FORTRAN compiler as well.

Limitation	**Message**	**Pass**
Minimum memory size 13000 halfwords (cf. 10.2.4)	stack pass trouble	2,5,8 all
No more than 3484 distinct names of identifiers	variables	2
No more than 1951 halfwords of simple variables and simple zones in one block	variables	5
No more than 2047 halfwords of owns in the program unit being translated (program or external procedure)	variables	5
No more than 2047 - external procedures - local procedures - labels - rts entries in the program unit being translated (number of globals + number of externals, cf.10.2.7).	variables	5
No more than 1023 segments occupied by one single local procedure declaration or by a suite of local procedure declarations.	no message	
No more than 62 nested blocks	blocks	5
No more than 7 parameters in an external procedure	ext param	5
Number of case elements in on list no more than 2046 (case expressions, case statements	case element	6

and switch declarations)		
No more than 97 halfwords of working locations in one block, e.g. no more than 15 repetitive nested statements	works	7
No more than 509 actual parameters in a procedure, or no more than 2044 halfwords used in the stack for a procedure call (return information + actual parameters + values computed at the call stack, cf. 10.2.6.2).	too many parameters	7
No more than 4095 segments in the final program, counting runtime segments and segments of external procedures but not counting segments of virtual storage (cf.10.2.7)	size segs	9 9
No more than 4095 halfwords occupied by owns in the final program, counting runtime system owns and owns in external procedures, but not counting commons or zone commons from fortran program units (cf.10.2.7)	size owns	9 9
Line number information will be counted modulo 8192*32=2621444 = 256 K	no message	

10.3 Execution

Except for the ALGOL specific figures in 10.3.5.4, (of which some are valid for FORTRAN too) the rest of the section is valid information for FORTRAN, too.

A binary object program is executed in the job process and started by means of an FP command as described in 10.3.2. The program must at that moment exist in a backing storage area.

10.3.1 Segmentation

The object program consists of independent program segments of 512 halfwords. Whenever the running program demands a program segment which is not in memory, it is transferred from the backing storage, possibly replacing another segment in memory. The number of segments held in the partition of memory is increased gradually until the limit posed by the variables is met. If no higher partition of memory exists (cf. 10.3.3), segment places will be chosen in lower partition in increasing order with a wraparound in lower partition when the stack of variables is met. If a high end partition of memory exists segment places will be chosen first in lower partition, then in higher partition in increasing order with a wraparound to the other partition when filled. If a higher partition exists, the lower partition may be filled with the variable stack, leaving only the minimum of two segment places in lower partition. If more variables are declared, some segments will be released from lower partition memory, and the next segment place will be taken in the high end partition, if it exists.

This scheme works satisfactorily as long as the program segments involved in the current part of the algorithm are kept in memory. Under these circumstances a jump to another segment is performed in

- 7 microseconds for RC8000-45,
- 4.5 microseconds for RC8000-50/60,
- 2 microseconds for RC8000-55/65 with cachehit
- 0.5 microseconds for RC9000-10 with cachehit

while a jump within one segment is performed in

- 3 microseconds for RC8000-45,
- 1.3. microseconds for RC8000-50/60
- 0.8 microseconds for RC8000-55/65 with cachehit
- 0.2 microseconds for RC9000-10 with cachehit

When the number of variables is increased so that the active segments cannot stay in memory, the program can still execute, but a jump to another segment will often cause a transfer from the backing storage resulting in a jump time of 15 000-25 000 microseconds. The standard identifier blocksread will show how these situations may be detected. You will see from this that it is very important to avoid crowding the program with variables.

As a minimum, you should have room for at least 8 segments in memory, corresponding to 4000 halfwords.

As a further aid, the ALGOL compiler may print a list of line numbers corresponding to the segment boundaries in the object program, if the compiler is called with details.8.8 (cf. 10.2.2.2), while the FORTRAN compiler will list the final instructions in a disassembled form, segment by segment, and a list of external procedures telling their segment boundaries in the program, if the compiler is called with details.9.9 (or survey.yes, cf. 10.2.7.1). The procedure lock may help to ensure the permanent presence of selected segments of the program in memory.

10.3.2 Call of Object Program

10.3.2.1 Syntax

```
                         ( <empty>                  )
           1             ( <s><source><anything>  )
( <name> =)  <bs file> (                          )
           0             ( <s><integer>             )
                         ( <s><param><anything>   )


                    ( <integer>)  ( <integer> )
          <param>::=(           ). (           )
                    ( <name>   )  ( <name>    )
```

10.3.2.2 Semantics

<name> =
Has no direct significance. However, <name> may be accessed from the executing program by means of 'system'.

<bs file>
A file descriptor describing a backing storage area which contains an object program from an ALGOL translation.

<empty>
The program is called with the zone 'in' connected to current input file.

<source>
Specifies a text file to be 'in'. Current input file is not touched in this case.

Note: if the program was translated with connect.no the text parameter following the program call, will not be connected to the zone 'in', unless the program itself does it.

<integer>
The file processor is overwritten from the process. The program cannot use the zones 'in' and 'out' and it cannot print error messages. When the program terminates, it sends a parent message corresponding to a 'break' and specifying the cause of the termination. On the other hand, 3000-4000 halfwords more are available in this way. This possibility is mainly intended for operating systems, which 'never' are terminated, never use 'in' and 'out', and work satisfactorily in a limited memory space.

Note: if the program was translated with fp.yes the file processor is not cleared.

<param>
Works as <empty>. The command parameters <param> and <anything> may be accessed from the running program by means of 'system' and interpreted in any way. <param> must obey the FP syntax (cf. [6].

<anything>
See <param>

10.3.2.3 Examples, translation and execution

```
p = algol ptx
p infile
p
```

Translates the source program in ptx. Executes it once with input from infile and once with input from current input file.

10.3.3 Storage Requirements

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

1) The job area is entirely below the limit of 1 M = 1048576 halfwords.

Length in halfwords	Contents
1536	File Processor
3106	Runtime System
Depends on program	Own/common space variables
2*L	Segment table, L-total number of program/virtual memory segments.
minimum: 1024 reasonable: 4096	Space for program segments currently in memory, lower partition
	Space for variables, arrays, zones.
1024+2*16	Buffers for 'in' and 'out'.

2) The job area crosses the limit of 1 M = 1048576 halfwords:

Length in halfwords	**Contents**
1536	File Processor
3106	Runtime System
Depends on program	Own variables/common space
2*L	Segment table, L= total number of program/virtual memory segments.
Minimum: 1024	Space for program segments, lower partition
Limit 1 M: Minimum: 1024 Reasonable: 4096	Space for variables, arrays, zones
1024+2*16	
	Space for program segments, higher partition
	Space for zone buffers and share descriptors
	Buffers for 'in' and 'out'

When the program is called with the parameter 0 (cf. 10.3.2.2), the space occupied by the file processor and buffers for in and out becomes 16 halfwords.

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

Lengths of memory are usually given in halfwords (one halfword = 12 bits), sometimes in words or double words (4 halfwords = 2 words = 1 double word).

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

Quantity:	**Number of halfwords reserved:**
simple boolean variable, field variable, simple integer variable	2
simple long variable, simple real variable	4
simple double precision simple complex	8 (FORTRAN)

Array segment	2*(number of array identifiers +1+number of subscripts)+space for total number of array elements.
	1 2
array element, boolean	
array element, integer	4
array element, real or long	
array element, double	8 (FORTRAN)
precision or complex zone	50+24*number of shares+4*bufferlength + 18
zone array working	2+space for all the zones.
locactions	Depends on structure of program, usually about 10 for each block.
block, procedure body	2*number of statically surrounding blocks+(if normal block then 4 else if type procedure then 14 else 10);
parameter	8 if the actual parameter is constant, or expression computed at the call side, 4 otherwise.

10.3.4 Message Buffers, Area Processes, etc.

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

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

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

10.3.5 Execution Times

The times given below represent the total physical times in microseconds for execution of algorithmic constituents on the RC8000/45 computer. The relative speed factors of the computers RC8000/15, RC8000/(50 or 60), RC8000/(55 or 65) and RC9000-10 are in average 0.6, 1.3, 2.6 and 10.4, which means that the total execution times should be divided by on of these factors for the specific computer.

The total time to execute a program part is the sum of the times for the constituents. The times are only valid under the following assumptions:

1. The time for transfer of program segments from the backing storage is negligible (cf. 10.3.1).
2. The program is not waiting for peripheral devices (cf. 3.3.2).
3. The time slice interval is 25.6 milliseconds or more (cf. [1]).
4. The program is executing in the only internal process running in the computer.

When the computer is time shared, assumption 4 is not fulfilled, but then the times represent the CPU time used by the program.

10.3.5.1 Operand References

Reference to local identifiers and constants	0
Reference to non-local identifiers(variable,zone, or array)	0-2.5
An array parameter is referenced as if it was declared locally in the outermost block of the procedure. If a sequence of identifiers from the same non-local block are referenced without intervening references to other non-local blocks, the first reference costs 4 microseconds and the later one usually 0.	
Reference to name parameter, actual is simple	6.5
Reference to name parameter, actual is composite	115
Reference to own variable	2.5

10.3.5.2 Constant Subexpressions

Operations are performed during the translation and thus do not contribute to the execution time in the following cases:

+ - * / shift extend working on constant operands. Conversion of an integer constant to a real constant, or vice versa.

real string long working on all operands.

The result of an operation performed during translation is again treated as a constant. Examples:

A(-2+6/5)	is reduced to A(-1)
1+0.5-0.25	is reduced to 1.25
p+1/2-1/4	is only reduced to p+0.5-0.25 because p+0.5 must be evaluated first.

10.3.5.3 Saving Intermediate Results

By the term 'composite expression' we shall mean any expression involving operations to be executed at run time. Examples:

A(2) b+1 a shift 8 pr (i,i,<:ab:>)	are composite
11.5 real<:ab:> 5 shift 20	are not composite

During evaluation of expressions, one intermediate result is saved in the following cases:

+, *, and, or,	all relations, shift, extract when working on 2 composite expressions.
-, /, //, mod	when the rigth hand expression is composite.
add	when both operands are composite or when the left hand operand is a composite real.

The saving of one intermediate result takes

integer or boolean value saved	5
real value saved	8

Examples:

A(i)+B(i)+C(i)	uses 2 savings (+,+)
a<b and b<d	uses 1 saving (and)
a<b+c and t	uses 0 savings
a+b+2-e	uses 0 savings
a-f*(g+h)	uses 1 saving (-)

10.3.5.4.1 Operators, ALGOL

integer+integer, integer-integer	2.7
long+long, long-long	4.2
real+real, real-real	13.6
and, or	2.5
integer*integer, spill.no	8.2
integer*integer, spill.yes	13.2
integer//integer, integer mod integer	14.2
real*real	34.2

real/real	25.8
long*long	139.2
long//long, long mod long	153.2
p extract<constant>	2.3
p extract i	6.9+i*0.6
real add i, long add i, string add i..	3.8
integer add i, boolean add i	3
real shift i, integer shift i	2+abs(i)*0.5
boolean shift i	2+abs(i)*0.5
entier real	22.6
round real	9.0
round long	1.5
extend integer	2.1
abs real	35.3
abs integer	2.8
abs long	9.8
subscripted variable with check against bounds one subscript	19.2
subscripted variable without check against bounds, one subscript	10.2
subscripted variable for each extra subscript add	7.0
integer:= integer	5.0
integer:= long, spill.yes	11.5
integer:= long, spill.no	6.5
integer:= real	14.1
long:= integer	10.0
long:= long	7.9
long:= real	60.6
real:= integer	18.3
real:= long	76.4
real:= real	7.9
goto local label	7.0
for i:= 1 step <constant> until n do, each loop	14.8
if true then ... else	7.7
if true then	5.1
if false then ... else	6.2
if false then	6.2
i<j, other connections	6.3
r<q, other connections	18.6
case i of	17.6
call of procedure with empty body, no parameters	113.0
parameter, for each value parameter add	25.8
parameter, for each name parameter add	6.3

10.3.5.4.2 Operators, FORTRAN specific

double precision + double, double - double	279.2
complex + complex, complex - complex	125.4
and, or	5.7
long * long	144.3
long / long	145.0
double * double	327.2
double / double	531.7
complex * complex	286.2
complex / complex	411.2
real ** integer	235.9
real ** real	625.0
subscripted variable, with check against bounds, 1 subscript	14.5
do., without check against bounds	7.5
do., for each extra subscript add	7.0
logical = logical	7.0
double = double	39.6
complex = complex	39.6
goto	2.6
assigned goto	23.4
computed goto	16.3
logical if (true)	5.7
logical if (false)	2.9
arithmetical if	6.4
continue	0.0
do (1 cycle)	22.6
integer.rel.interger	5.8
real.rel.real	17.8
call subroutine	91.6
for each param., add	6.3

10.3.5.5.1 Execution Times for Standard Externals, FORTRAN

ABS	164.7
IABS	117.6
CABS	386.4
DABS	214.7
AMOD	296.3
MOD	182.1
AMAXO(,)	295.4
AMAX1(,)	279.8
MAXO(,)	310.7
MAX1(,)	287.1
AMINO(,)	295.2
AMIN1(,)	275.9
MINO(,)	303.2
MIN1(,)	567.7
REAL	59.2

AIMAG	69.2
COMPLX	93.7
EXP	533.2
CEXP	1582.7
ALOG	455.5
CLOG	1309.8
SIN	569.5
CSIN	2143.9
COS	546.6
CCOS	2248.8
SQRT	243.5
CSQRT	649.5
ATAN	527.5
CANG	661.5
DSIGN	171.0

10.3.5.5.2 Execution Times for Certain Standard Procedures

arcsin	613.3
arctan	354.0
arg	686.7
cos	628.5
exor	228.3
exp	564.2
ln	485.6
logand	225.9
logor	226.8
random	176.3+ double words moved
sgn, sign	154.5
sin	625.2
sinh	648.7
sqrt	274.5
tofrom	302+7.5* double words moved

10.2.5.5.3 Example

The follwoing example shows the computation of the time for the following loop:

```
for i:= n step -1
                until j+1 do           14.8 (for do)
                                        2.7 (j+1)
if ia(i)= 3 and                        19.2 (ia(i))
                                        6.3 (=3)
                                        7.5 (save, and)
ra(i+1)>1 then                         21.9 (ra(i+1),+)
                                       18.6 (>1)
                                 (5.1-) 6.2 (if then)
                                 ______________________
                                       97.2

p:= p+ra(i+1);                         21.9 (ra(i+1),+)
                                       13.6 (real +)
```

```
                                7.9 (p:-)
                          ___________________
                             140.6
```

The result is that the loop takes about 140 microseconds when the last statement is executed, 97 otherwise.

10.3.6 Messages from the Running Program

10.3.6.1 Initial Alarm

Before the first begin of the program is entered, the alarm

```
***<program name> init: <explanation>
```

may appear, and the program terminates unsuccesfully. The <explanation> is either of:

- 'not bs', i.e. the program is not on backing storage,
- 'connect in', i.e. the zone 'in' could not be connected to the file parameter (<program> <file>),
- 'infile not text', i.e. the file parameter is not a text file,
- 'process too small', i.e. the job process is too small, or
- 'at restart wrong size', i.e. the program performs a restart from data in virtual memory, but the virtual memory in the program file extension is inconsistent, probably it has been reused by another program.

10.3.6.2 Normal Form

When the program is called with <program> <integer>, a possible run time alarm will appear as a parent message (cf. 10.3.2.2).

If the program is successful, the message:

```
end <blocks read>
```

will terminate the execution.

A run time alarm will terminate the program with a message of the form:

```
<cause>     <alarm address>
called from <alarm address>
called from ...
```

A run time message of the same form may appear, after which the program continues, cf. the procedure write.

If a run time alarm occurs in activity mode, the fist line in the error message is supplied witl the activity number. If notrap label is present at the moment of the alarm, an implicit passivate statement will be executed, rather than terminating the program.

If the alarm occurs in disable mode, the error message will include an alarm address for the corresponding disable statement.

If a traplabel is present at the moment of the alarm, the program will continue execution of the traplabel after having given the message.

The printing of the message may be omitted, governed by the standard variable trapmode. The value of <cause> may be obtained, though, through the standard variable alarmcause and the procedure getalarm.

A list of the possible alarm causes is given in 10.3.6.4.

If the program terminates, it does so unsuccessfully except after the message 'end'.

An alarm address shows where the error occurred. If it was a procedure or a name parameter, a line specifying the call address or the point where the name parameter was referenced is printed too. The action is repeated if several calls or references were active at the time of the alarm. If more than 10 calls or references are active, the process stops after having printed the last 'called from', but before the last alarm address is printed.

An alarm address may take 3 forms:

1. name of a standard procedure or of a set of standard procedures

2. line <first line>-<last line>

3. ext <first line>-<last line>

Form 2 specifies a line interval in the source text of the main program. Form 3 specifies a line interval in an external ALGOL or FORTRAN procedure or subroutine. The accuracy of a line interval corresponds to about 17 instructions of generated code. The first line number may sometimes be 1 too great if the line is not terminated with a delimiter. The line number of a procedure call points to the end of the paranthesis.

The following alarm addresses from standard procedures are used:

```
activity        (activity, newactivity, activate,
                passivate, disable, enable, wactivity)
alarm segm0     (the code in rts preparing the alarm
                print and the termination or resumption
                of the progra m execution, rts segment)
alarm segm1     (the code in rts actually printing the
                alarm text, rts segment)
algolcheck      (the subprocedure in the check procedure
                in rts calling the user's block
                procedure. May als o be the code
                performing certain operations o n long
                operands, rts segment)
buflengthio     (buflengthio)
char input      (read, readall, readchar, readstring,
                repeat- char, intable)
check           (the check procedure in rts used by all
                high level zone procedures, rts segment)
```

checkspec	(the rts standard error handling actions in the check procedure, rts segment)
ch/outvar	(changevar, outvar, checkvar)
close/term	(close, setposition, stopzone)
fpmode	(fpmode, setfpmode)
fpproc	(fpproc)
f8000table	(f8000table, getf8000tab)
inoutrec	(inoutrec, changerecio)
invar	(invar)
lock	(lock, locked)
monitor	(monitor)
open	(open)
open/stop	(open, stopzone)
openinout0	(openinout)
openinout1	(openinout)
openinout2	(closeinout, expellinout, resetzones)
outchar	(outchar, outtext, outinteger)
outdat/move	(outdate, movestring)
pos/state	(getposition, setposition, getstate, setstate) power func. (real ** integer, real ** real, rts segment) recprocs (changerec, inrec, outrec, swoprec)
pos	(pos, len)
recprocs6	(changerec6, inrec6, outrec6, swoprec6)
resume	(resume)
sorting	(newsort, deadsort, lifesort, outsort, initsort, initkey, sortcomp)
sorting6	(startsort6, changekey6)
std.fct.1	(exp, ln, alog, sign, sgn, random, sinh)
std.fct.2	(arctan, atan, arg, sin, cos)
std.fct.3	(arcsin, sqrt)
system0	(system, int all entries, entries 1-8)
system1	(system, entries 9-11, 13)
system2	(system, entries 12 and 14, increase, check, blockproc, stderror)
systime	(systime, logand, logor, exor)
tofrom1	(tofrom, init and exit, move half instr)
tofrom1	(tofrom, data moving segment)
tofromchar	(tofromchar)
transinput	(waittrans, readfield)
transoutput	(opentrans, writefield, closetrans)
virtual	(virtual, openvirtual)
write ext.	(write)
write	(write, writeint, replacechar, outtable, isotable)
zone declar	(the rts code taking care of zone and zone array declarations, rts segment, resetzones)
zone, share	(getzone, setzone, getshare, setshare)
zone, share6	(getzone6, setzone6, getshare6, setshare6)
zones/trap	(initzones, getalarm, trap)

A number of alarm addresses in standard FORTRAN subroutines and functions may be found in [9], D.2.2.

10.3.6.3 Undetected Errors

If all parts of a program have been translated with index.yes and spill.yes, the following errors may still pass undetected.

1. Parameters in the call of a procedure which is a formal parameter do not match the declaration of the corresponding actual procedure. Any reaction may result.

2. Number of subscripts of a formal array do not match the number of subscripts of the actual array. Wrong results may be produced, but the control of the program remains intact.

3. A subscript may exceed the bounds in an array declaration with more dimensions as long as the lexicographical index is inside its bounds. The control of the program remains intact.

4. The program may write into the backing storage area occupied by the program itself. Any reaction may result.

5. Undebugged standard procedures in machine language may cause any reaction.

6. The location of an array given as actual parameter to a procedure is established at the call side (call by value) but the array is nevertheless changed in the procedure (the actual array is a zone record). Any reaction may result.

The monitor and the operating system will usually limit the consequences of errors in such a way that no other job or process in the computer can be harmed (see [1]).

10.3.6.4 Alphabetic List of Alarm Causes

The error messages below cover only the standard procedures described in this manual. Error messages coming from FORTRAN standard subroutines and functions may be found in [9], D.2.4.

The set of messages is expected to grow in step with the growth of the standard procedure library.

abled
In disabled mode
1) alarms in disabled mode are terminated by the message abled... The alarm address shows the call of disable
2) jump out of a disable statement
3) leaving a disable statement in activity mode. This may happen if a jump is done into a disable statement. The address shows the end of the disable statement.

actno <i>
System(12 ... is called with an illegal activity number.

arcsin 0
Illegal argument to arcsin

block <i>
Too long record in call of changerec6, inrec6, outrec6, or swoprec6. The block length is shown.

break <i>
An internal interrupt is detected. <i> is the cause of the interrupt, usually meaning:
0 index error in program translated with index.no
6 too many message buffers used (cf.10.3.4)
8 program breaked by the parent, often because it is looping endlessly. In this case, the alarm address should be taken with some reservation.
The break alarm will often be called as a result of the undetected errors described in 10.3.6.3.

bufsize <i>
Initzones. The original total buffer size for za is exceeded by allocating space for za(i), or buffersize(i) <= 0, or shares(i) <= 0.

case <i>
Case index outside range. The index is shown. The line number points to 'of'.

connect
openvirtual. Number of owns in calling program different from number of owns defined in the file connected.

context
openvirtual. The procedure is called inside a context block.

create
openvirtual. A file is attempted created as virtual storage after some context blocks in the calling program have been executed.

c.array
Actual array length in a context block exceeds the maximum.

c.expand
The file containing the virtual storage cannot be extended, i.e. there is no space available for further context records.

c.incarn
Incarnation number not positive, or exceeds number of incarnations.

end <i>
The program has passed the final end. The integer printed after 'end' shows the value of the standard identifier 'blocksread' as the program terminated.
This is not an error message.

entry <i>
Illegal function code or entry conditions in a call of monitor, system, or systime. The function code is shown.

exp 0
Illegal argument to exp.

field <i>
Field reference outside bounds. The illegal halfword address is shown.

goto
It is not allowed to jump (by a goto statement) out of an activity or out of a disabled statement.

giveup <i>
Printed by stderror. The number of halfwords transferred is shown. The File Processor prints the name of the document and the logical status word.

index <i>
Subscript outside bounds. The lexicographical index is shown. This message also occurs for subscripted zones or record variables.
The character input procedures call the index alarm if they cannot assign a single result to their return parameters or if a character outside the current alphabet is met.
The procedure 'check' calls the index alarm if a block procedure specifies a too long block. In this case, the value of the parameter 'b' is shown.

In the procedure write or writeint, a parameter could not be 'properly classified'. The integer i is parameter no.*100+kind, where kind is the parameter kind, cf. [10], page 39. The program execution continues.
In the procedure getalarm the parameter array does not offer 8 halfwords from halfword index 1.
In the procedure initzones, bufsize or shares cannot be referenced with 1< = index < = number of zones.
In one of the procedures newactivity, activate or activity, the activity number, i, or the shared activity number, i, defines a not declared activity, or it is an illegal parameter value in call of activity.

Other procedures may report an index error. Consult the proper manual pages found following the clue given by the alarm address.

integer
Integer overflow.

killed
activate. An activity has been killed, i.e. activated by a call of activate with a negative parameter.

kind <i>
Illegal modekind in call of open. The kind is shown.

lenght<i>
Negative record length in call of inrec6, outrec6, or swoprec6. The length is shown.

level
Trap level is global to the calling block.

ln 0
Argument to ln is < = 0.

mode
One of the procedures activity, newactivity, passivate, activate or wactivity is called in a wrong mode:

activity	not	in neutral	mode
newactivity	not	in monitor	mode
passivate	not	in activity	mode
activate	not	in activity/monitor	mode
wactivity	not	in monitor	mode

movesize <i>
Tofrom is called with the number of halfwords to be moved < 0. The size is shown.

movefld <i>
Tofrom is called with an array where the halfword numbered 1 or the halfword numbered size does not exist.

oddfield <i>
Tofrom, read, readall, readstring, write, writeint or movestring was called with an array where the word boundaries are not between an even numbered halfword and its odd numbered successor. The wrong parameter number (1 or 2) is shown.

param
Wrong type or kind of a parameter.

param <i>
In the procedure write or writeint, a parameter could not be properly classified. The integer i is

```
parameter number*100 + kind
```

where kind is the parameter kind, cf. [10], page 39. The program execution continues.

param <n> lock
Means that an inconsistent set of parameters is used, i.e. parameter 2 of a pair is missing, parameter 1 > parameter 2, or the segment number specified is < 0 or >4095. The value of <n> designates the type and the parameter causing the error:

```
<n> = paranumber * 10 + type
type: 1 integer
      2 label
      3 procedure
      4 parameter list exhausted
      5 type not integer or label.
```

proc
Parameter number 3 in a call of new_activity is not a typeless procedure identifier.

reclen <i>
Changevar or outvar was called with a length word < 0 or 0 < length word < 4.

real
Floating point overflow or underflow.

replace <i>
In the procedure replacechar, the first parameter, special, is outside the range 0....7.

segment
A text seems to be a long string but could not be found as a text constant.

share <i>
An illegal share number is specified. The share number is shown.

sh.state <i>
A share in an illegal state is specified. The share state is shown.
> 1 message buffer address for an uncompleted transfer or a stopping child process.
< 0 process description address for a running child process.
= 0 for a free share
= 1 for a ready share

sinh 0
Illegal argument to sinh.

sqrt 0
Argument to sqrt is < 0.

stack <i>
The number of variables exceeds the capacity of the job area, or an array or a zone is declared with a nonpositive number of elements. The number of halfwords in the reservation of storage is shown.

string <i>
As for param <i>...

syntax
The program is terminated at a point where an error was detected during the translation.

value <i>
The contents of ia(i) in setzone/setzone6(z,ia) or setshare/setshare6 (z,ia,sh) is illegal. The value is shown.

virtual
The shared virtual activity, defined by the second parameter in new_activity is not initialized (by new_activity), or it is not a virtual activity.

z.kind
Swoprec is used on a document, which is not a backing storage area.

z.length <i>
The buffer length is too short. The actual buffer length is shown.

z.state <i>
A high level zone procedure is called in an illegal zone state. The actual state is shown.
Initzones is called with zonestate not equal to 4.
Concerning zonestate, cf. [15], getzone.

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 manual is the equivalent to the RC8000 document called *Monitor, Part 1, System Design* (PN: 991 03577).

2 PN: 991 11259
Monitor, Reference Manual delivered as part of SW9890I-D, Monitor Manual Set.

3 PN: 991 03435
Monitor, Part 3, External Processes
This manual is part of the documenttaion for RC8000. The RC9000-10 relevant processes are described in manuals part of 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 manuals are delivered as part of SW8010I-D, System Utility Manual Set. PN: 991 11294
System Utility Programs, Part 3
This manual is delivered as part of 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.

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 Run Time Organisation of ALGOL Programs
this document is not part of any package, but it is available on demand.

11 R.M. De Morgan et al.:
Modified Report on the Algorithmic Language ALGOL60.
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 11280
ALGOL8, User's Guide, Part 2
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.

B. Index