stichting
mathematisch
centrum

MC

AFDELING INFORMATICA
(DEPARTMENT OF COMPUTER SCIENCE)

IW 90/77 OKTOBER

J.C. VAN VLIET

TOWARDS AN IMPLEMENTATION-ORIENTED DEFINITION OF THE ALGOL 68 TRANSPUT

2e boerhaavestraat 49 amsterdam

Printed at the Mathematical Centre, 49, 2e Boerhaavestraat, Amsterdam.

The Mathematical Centre, founded the 11-th of February 1946, is a non-profit institution aiming at the promotion of pure mathematics and its applications. It is sponsored by the Netherlands Government through the Netherlands Organization for the Advancement of Pure Research (Z.W.O).

AMS(MOS) subject classification scheme (1970): 4.22, 3.73

ACM-Computer Review-categories: 68A50

Towards an Implementation-Oriented Definition of the ALGOL 68 Transput

by

J.C. van Vliet

ABSTRACT

This report aims at a precise definition of the transput of ALGOL 68, conforming with section 10.3 of the Revised Report on the Algorithmic Language ALGOL 68. Whereas section 10.3 of the Revised Report describes the intention of transput, the emphasis in this report is on implementability.

KEY WORDS AND PHRASES: ALGOL 68, transput, portability, runtime system, ALGOL 68 implementation.

1. INTRODUCTION.

This report aims at a precise definition of the transput of ALGOL 68, conforming with section 10.3 of the Revised Report on the Algorithmic Language ALGOL 68 (henceforth the Revised Report). Whereas section 10.3 of the Revised Report describes the intention of transput, the emphasis in this report is on implementability.

A variety of ALGOL 68 implementations exist or are near completion. They all support some kind of transput, although they all differ slightly from each other and from the Revised Report [2-8]. This diversity renders the porting of programs from one implementation to the other very difficult, if not virtually impossible.

The existence of so many different transput systems may to some extent be due to the fact that the description as given in the Revised Report does not really facilitate implementation of the transput. Each implementer again has to struggle his way through the transput section and locate the problems with the particular operating system. It is hoped that the definition in this report will solve most of these problems once and for all. THOMSON & BROUGHTON [9,10] and R.FISKER of Manchester are working on transput systems of a similar structure. Also, the Subcommittee for ALGOL 68 Support of Working Group 2.1 of IFIP has recently set up a task force to investigate the transput. The present report should be seen in the light of this discussion; it is by no means the last word about transput.

The approach taken is similar to the one in the Revised Report: the transput is described in pseudo-ALGOL 68. The pseudo part can be considered as a language extension which is reasonably implementable. The primitives underlying the model are not defined in ALGOL 68. Instead, their semantics are given in some kind of formalized English, resembling the way in which the semantics in the Revised Report is defined. One advantage of a description in pseudo-ALGOL 68 is that it can largely be tested mechanically. It might even be possible to feed this description of the transput into the compiler, thus automatically creating part of the runtime environment.

Care has been taken to stick to the Revised Report as closely as possible so far as the meaning is concerned. So the full transput is defined, even those features whose value may be questioned (like, e.g., the elaboration of dynamic replicators upon staticizing a picture). At a few places however, a different meaning is assigned to certain features. These differences are clearly indicated as such in the sections headed 'DEVIATIONS'. All errors that have been found in the transput section of the Revised Report have of course been corrected in the present model. (These errors have been collected in [14]; the Subcommittee on ALGOL 68 Support however has not yet agreed upon them; in this respect, the definition given in this report is somewhat running ahead of things.)

The main differences between the transput section presented here and the one from the Revised Report are:

i) Books are considered to form part of the operating-system interface; as such, the mode BOOK is not specified;

ii) On many systems, not all of the text of a file is available at each instant of time. This has been made explicit in the present model by starting from a "buffer" concept. {The consequences of this approach

permeate through the whole transput section!};

iii) The numerous calls of 'undefined' in the Revised Report have been assigned meanings. Hidden kinds of undefined actions like SKIP, UP gremlins and UP bfileprotect have been paid due attention;

iv) The number of tests that is performed for each transput operation is minimized. For many routines, pre- and postconditions have been chosen carefully, so as to achieve security with a minimum of tests;

v) By choosing a different structure for the mode FORMAT, remarkable simplifications can be made in the section on formatted transput.

Acknowledgements. The author has benefited much from his correspondence and discussions with H. Wupper of the Ruhr-University at Bochum, B. Leverett of Carnegie-Mellon University, C. Cheney of Cambridge, R. Fisker of Manchester, L. Meertens, H. Boom and D. Grune of the Mathematical Centre, and last but not least Section 10.3 of the Revised Report itself.

2. UNDERLYING PRIMITIVES.

A model that is intended to be easily implementable on a variety of machines, must be described in such a way that it is clear which aspects are machine dependent, and which are not. There must be a clearly defined set of primitives underlying the transput, and this set must in some sense be small. These primitives then form the operating-system interface. The "meaning" of these primitives must also be defined.

The primitive that lies at the very heart of the present model is the "buffer". In the Revised Report, both the book and the file contain the "text", which is a reference to a three-dimensional character array. For most files on most systems, not all of the three-dimensional character array will be available at any instant of time. This restriction is made explicit in the present model; the piece of text that is available is called the "buffer". Preferably, a buffer corresponds to one line of the text. It is however anticipated that there will be files containing only one page which consists of one huge line (just think of a papertape). In that case, the buffer will probably correspond to a much smaller piece of text. The same holds for files that are used interactively, where the system possibly transputs units of information of the size of a number, say. One immediate consequence of this is that the routine 'backspace', as defined in the Revised Report, cannot be implemented on such a file. The present model deviates from the Revised Report in that an enquiry 'backspace possible' is defined for files, with a function very similar to that of the enquiries 'set possible' and the like.

If buffers are used as units of transput information, there must be ways to write a buffer and read the next buffer. Machine-dependent primitives are needed for this. They are discussed at full length in later sections.

As the internal structure of the buffer is not specified either, primitives for reading and writing single characters from or to the buffer are needed also.

This scheme offers attractive possibilities with regard to "conversion keys": The Revised Report specifies characters to be converted from internal form to external form (and vice versa) before they are actually transput. This conversion is done per character; not only is control given back to the main program if a character cannot be converted, but conversion keys may also be changed per character. It is however not at all sure that characters may be punched alternatingly in ASCII and EBCDIC, say. It is to be expected that on many devices there will be only limited possibilities to change the conversion key. Maybe another conversion key can be provided only if the file is positioned at the very beginning of a book, or at the beginning of a line. Probably, there will also be channels on which a change of conversion key is altogether impossible. On the other hand, there exist line-printers on which a change of conversion key can be achieved on a line to line basis with one hardware instruction. The present model is flexible enough to allow these very different implementations of the conversion-key concept: having primitives both to transput one buffer and to transput single characters to and from the buffer, the implementer is free to do the conversion in either of these. He may even decide to do no conversion at all, or to do it differently on different channels. Changing the conversion key then amounts to changing the above set of primitives.

(Note that, if there are no conversion keys, the buffer need not be made primitive, but can be defined instead as

```
MODE BUFFER = REF [] CHAR.
```

This obviates the need for primitives to transput characters to and from the buffer. If there are conversion keys, the buffer cannot be defined so simply, since there is no reason to expect that internal characters still look like characters after conversion! Also, it may be convenient and efficient to keep the buffer in a machine-oriented form rather than as an ALGOL 68 row-of-character, even if there is no conversion key.) For further details on the buffer and its primitives, see section 4.2.2.1.

Almost all machine dependencies in the present model are incorporated in the channel. A channel is a set of attributes that is common to some set of devices. It is anticipated that most machine dependencies will differ for files opened on different channels, but will be the same for files opened on one and the same channel. {Resetting the position of a file will probably be different for tape and disk, but will probably be identical for two files that are both opened on disk.} It is thus advantageous to incorporate these primitives in the channel. This approach is also taken in the ALGOL68S implementation at Carnegie Mellon University [11,12] and at Manchester [13]. (The ALGOL68S implementation at Carnegie Mellon also uses a buffer concept, albeit a slightly different one [12].)

A full list of the primitives that have been incorporated into the channel is:

- read buffer, write buffer, init buffer;
- get char, put char, get bin char, put bin char;
- newpage, newline;
- (part of) set, (part of) reset.

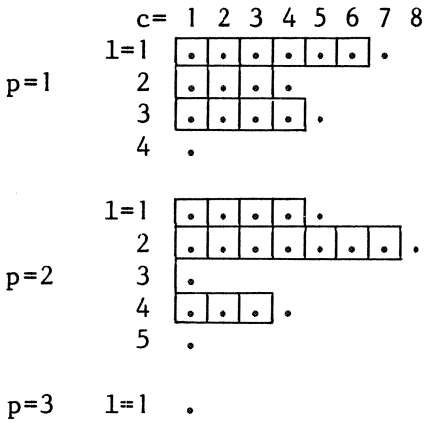
3. POSITIONS

Positions within the text are indicated by a page number, a line number and a character number. Two positions are of importance during transput:

- the "logical end", i.e., the position up to which the book has been filled with information;
- the "current position", i.e., the position where the next transput operation will (normally) operate on.

Before any actual transput operation may take place, the validity of the current position has to be ensured. Whether a given position is "valid" depends on the kind of operation that is desired. {If a newpage is to be given, only the page number has to be within its bounds; if a character is written, both the line number and the character number must be within their respective bounds as well.} If one of the position entities need not be within its bounds, it may be off by one at the upper end; in that case, the line, page or book is said to have "overflowed".

{If p, l and c denote the page number, line number and character number, then a typical text may look as follows:



Possible positions are indicated by ".", although information can only be present at positions within a box. For the dots that are not placed within a box, the condition "line ended" holds. At <1,4,1>, <2,5,1> and <3,1,1> the condition "page ended" holds too, while at <3,1,1> the condition "physical file ended" holds as well.}

{Note that if the page has overflowed, the current line is empty (so the line has overflowed too), and, if the text has overflowed, the current page and line are both empty (so the line and page have both overflowed).}

If the current position has overflowed the line, page or book, then it is said to be outside the "physical file". The position where the book has overflowed is termed the "physical file end". (There is only one such position!)

If, on reading, the current position is at the logical end, then it is said to be outside the "logical file".

{Note that, if "<=" denotes the intuitive lexical ordering, then the invariant

current position <= logical end <= physical file end
always holds.}

If the current position is outside the (physical or logical) file, the next transput operation will call an "event routine" (except possibly when 'set', 'reset', 'set char number' or 'backspace' is called). Event routines are provided by default, but may be changed by the user (for more details, see 4.3.2). The event routines for the overflow conditions correspond to the following "on routines":

- on logical file end;
- on physical file end;
- on page end;
- on line end.

If an overflow condition occurs, the event routine corresponding to the appropriate "on routine" is called. Since more than one overflow condition may occur at one and the same position {e.g., if the page is ended, the line is also ended}, it is important to know which event routine is called in each situation. Obviously, only one out of "physical file ended" and "logical file ended" can occur: on reading the logical end of the file may be reached, on writing the physical end may be reached. The ordering of the calling of event routines is such that physical (logical) file end has preference over page end, while page end has preference over line end.

{One striking difference between the physical file end and the logical file end is that the logical end may occur at each valid position (each position with a dot in the above picture), while the physical end may only occur at the first position on a new page. This will certainly lead to difficulties if straightforwardly implemented: it necessitates knowledge about the size of the current page (which may be compressible) and knowledge about the space that is still available for the text. On many systems, a record that is written to a tape may or may not fit, and this is not known until it is actually tried! If it does not fit, the operating system will usually tell so. It is thus implementation dependent when exactly the event routine corresponding to "on physical file end" is called, and the event routine will have to be called from within the routine that writes a buffer to the book.}

The transput system has to administrate both the logical end and the current position. On reading this is relatively simple: for each character that is read, the current position is advanced over one position until the logical end is reached. Writing is much more complicated. The effect of a write operation depends on:

- whether the current position is before or at the logical end;
- whether the file is random or sequential access;
- whether the file is compressible or not;
- whether it is a layout routine that is called.

It is usually a combination of some of the above possibilities that leads to special effects. These effects are all treated in full detail in later sections.

Writing at the logical end leads to advancing the logical end along with the current position. The logical position ("lpos") will then keep pace with the current position ("cpos"). This is expected to be the normal every-day situation if files are written to. The Revised Report maintains the most important invariant that connects 'cpos' and 'lpos':

```
I:  cpos <= lpos
```

by statements like

```
{I} cpos +:= 1; (cpos > lpos | lpos +:= 1) {I}.
```

In the present model, a new pair of variables (cpos1, lpos1) is introduced. This pair is related to (cpos, lpos) through:

```
cpos = cpos1
lpos = MAX (cpos1, lpos1).
```

The same invariant I is now ensured by

```
{I} cpos1 +:= 1 {I}.
```

Care has to be taken if the current position is set back (which may occur through a call of 'set', 'reset', 'set char number' or 'backspace'). In that case the value of 'cpos1' must first be assigned to 'lpos1' iff 'lpos1' < 'cpos1' (the routine 'mind logical pos' serves this purpose). 'cpos1' and 'lpos1' are again called 'cpos' and 'lpos' in the text. It is expected that a considerable gain in efficiency will result from this change of representation.

4. BOOKS, CHANNELS AND FILES.

{"Books", "channels" and "files" model the transput devices of the physical machine used in the implementation.}

4.1. BOOKS

4.1.1. DEVIATIONS FROM THE REVISED REPORT

{Two kinds of deviations are distinguished: changes in the definitional method and changes in the semantics. These are indicated by prefixing their discussions by {D} and {S}, respectively. Usually, only major changes in the definitional method are mentioned.}

- {D} The various fields of the mode BOOK are not specified. Instead, routines are provided to construct books, or change certain fields of a book.

- {D} It is not prescribed that books be kept in two chains. Books somehow reside in the system and are searched for in some unspecified way if asked for one. In the Revised Report, multiple references are needed to books that allow multiple access. Each time a file is opened on such a book, one of these references is removed from the chain, and a counter in the book is increased. {Note that a book may only be opened on more than one file simultaneously if writing is not possible on any of them.} It is not defined in the present system how this administration is exactly performed. In the Revised Report, critical sections of routines that access the chains of books are protected by the semaphore 'bfileprotect'. In the present system, all these critical sections are behind the curtain, and so is the semaphore.

- {S} The default identification for files which are opened via 'create' is not left undefined. Rather, an operating-system dependent string is assumed, which may differ from case to case.

4.1.2. NEW DEFINITION

Books model the actual devices on which the transput takes place. All information within the system is to be found in a number of such books. The information that is kept in the book is operating-system dependent information, such as the identification string. Via the book, the text may be reached; this text contains the actual information. The text has a variable number of pages, each of which may have a variable number of lines, each of which may have a variable number of characters. Positions within the text are indicated by a page number, a line number and a character number. The book includes a field which either indicates the "logical end" of the book, i.e., the position up to which it has been filled with information, or it indicates that this position is not known yet. Except for 'set', no routine needs exact knowledge of the logical end of the book, as long as it is not in the current line.

Books are either searched for in the system, or constructed according to certain user-defined requisites. Books are searched for by means of the routine 'book in system'. A book is constructed by one of the routines 'construct book' and 'pseudo book'.

```
a) MODE ? BOOK =
     C an actual declarer specifying a mode which contains at least the
       following operating-system dependent information:
       - the identification string;
       - some reference to the actual information;
       - the logical end of the book {, which may however be unknown};
       - information that tells how many other processes have access to this
         book, and whether some process is writing to the book. C;

b) MODE ? POS = STRUCT(INT p, l, c);

c) PRIO ? EXCEEDS = 5,
   OP EXCEEDS = (POS a, b) BOOL:
      p OF a > p OF b OR l OF a > l OF b OR c OF a > c OF b;

d) PRIO ? BEYOND = 5,
   OP BEYOND = (POS a, b) BOOL:
      IF p OF a < p OF b THEN FALSE
      ELIF p OF a > p OF b THEN TRUE
      ELIF l OF a < l OF b THEN FALSE
      ELIF l OF a > l OF b THEN TRUE
      ELSE c OF a > c OF b
      FI;

   {Positions are indicated by values of the mode POS. They are compared by
    the operators EXCEEDS and BEYOND.}

e) PROC ? book in system =
      (STRING idf, CHANNEL chan, REF BOOK book, REF BUFFER buffer) INT:
        C The pool of available books is searched for a book with the
          following properties:
          - the book may be identified by 'idf';
          - the book may be legitimately accessed through 'chan';
          - opening is not inhibited by other users of the book.
            {A book may only be opened on more than one file simultaneously
            if putting is not possible on any of them.}
          If such a book is found, it is assigned to 'book', space for a
          buffer is made available to which 'buffer' is made to refer, and
          the routine yields 0. Otherwise, the routine yields some positive
          integer that corresponds to the appropriate error code. (For a
          list of these error codes, see section 5.2.) C;

f) PROC ? construct book =
      (INT p, l, c, STRING idf, CHANNEL chan, REF BUFFER buffer)
      REF BOOK:
        C A book is constructed with a text of the size indicated by (p, l,
          c), with an identification string 'idf', that may be written to.
          The book is to be accessed via 'chan'. The routine allocates
          space for a buffer to which 'buffer' is made to refer; the buffer
          is not initialized. The logical end of the book is set to (1, 1,
          1). A reference to the book is yielded as result. The validity of
          the parameters has already been checked by the routine
          'establish', which calls 'construct book'. C;
```

```
g) PROC ? pseudo book = (INT p) REF BOOK:
     C A book is constructed whose only interesting information is its
       logical end, which is set to (p + 1, 1, 1). The routine is called by
       'associate'(see 5.2.d). C;

h) PROC ? default idf = STRING:
     C a default identification string for files which are opened via
       'create' C;

i) PROC ? set logical pos = (REF FILE f) VOID:
     BEGIN
       C the logical end of the book of 'f' is set to the current position
         C;
       REF INT(c of lpos OF f):= c OF cpos OF f;
       status OF f ANDAB logical file ended
     END;

     {This routine replaces the pseudo comment in 'put char'.}
```

4.2. CHANNELS

4.2.1. DEVIATIONS FROM THE REVISED REPORT

- {S} A field 'backspace' is added, which determines whether the routine 'backspace', as defined in section 10.3.1.6.b of the Revised Report, is possible on the channel.

- {D} The conversion key is understood to be a set of routines that define how buffers are transput to and from the book, and how characters are transput to and from the buffer.

- {D} The primitives 'do newline', 'do newpage', 'do set' and 'do reset' have been incorporated in the channel.

4.2.2. NEW DEFINITION

A "channel" is a collection of attributes that is common to some set of devices. A channel is a structured value and has two kinds of fields:
- routines returning truth values which determine the available methods of access to a book linked via that channel;
- primitives for device-class dependent actions.

Since the methods of access to a book may well depend on the book as well as on the channel, most of these routines depend on both the channel and the book. The access properties may be examined by use of the environment enquiries for files (4.3.2). Two environment enquiries are provided for channels. These are:
- 'estab possible', which returns true if another file may be "established" (5.2.a) on the channel;
- 'standconv', which may be used to obtain the default "conversion key".

The fields that are primitive actions all depend on the file, since they may have side effects on specific fields of the file {e.g., 'do newpage' will result in a new value for the current position}.

A "conversion key" is a value of the mode specified by CONV, which is used to convert characters to and from the values as stored in "internal" form and as stored in "external" form in a book. A conversion key consists of two parts:
- routines to transput buffers to and from the book and a routine to initialize the buffer;
- routines to transput characters to and from the buffer.

It is not prescribed in the present model whether the actual conversion (if any) is done per line or per character. {This may differ from one implementation to another, or even for two different conversion keys used on one and the same channel.} The implementation may provide additional conversion keys in its library-prelude. After linking a file to a book via some channel, the buffer must be initialized. For files which are "opened" (5.2), this initialization is not done until the first transput operation is initiated, upon which an explicit call of 'init buffer' takes place.

The routines 'set', 'reset' and 'newpage' may cause the current position to be moved outside the current line. {Nevertheless, they may to a great extent be written in proper ALGOL 68. For instance, a routine 'newpage' may be written that searches for an end-of-page condition, in the meantime skipping lines that have already been filled, and filling empty lines with blanks.} It is expected that on most systems a newpage is only possible on

devices such as a line-printer, for which a hardware newpage instruction is generally available. Similarly, a fast set routine will probably be available for random-access files that reside on disk, and so on. Therefore, (parts of) these routines have been incorporated in the channel, and each implementation has to define them for each channel it supports.

{Note that a user may still define values of the mode CHANNEL by the way this mode is defined below. An easy way to circumvent this is to define the mode CHANNEL as STRUCT(CHUNNEL c), and to define CHUNNEL as the structured value given below. (The same holds for values of the mode FILE (4.3) and FORMAT (11.2).) For clarity's sake, this has not been done in the present model. Similarly, the user can still get hold of values of the mode specified by CONV, for instance by writing 'standconv(stand in channel)(NIL)'.}

```
a) MODE CHANNEL = STRUCT(
     PROC (REF BOOK) BOOL ? reset, ? set, ? get, ? put, ? bin, ? compress,
                          ? reidf, ? backspace,
     PROC BOOL ? estab,
     PROC POS ? max pos,
     PROC (REF FILE) VOID ? do newline, ? do newpage, ? do reset,
     PROC (REF FILE, INT, INT, INT) VOID ? do set,
     PROC (REF BOOK) CONV ? standconv,
     INT ? channel number);

b) MODE ? CONV = STRUCT(
     PROC (REF FILE) VOID read buffer, write buffer, init buffer,
     PROC (REF FILE, CHAR) VOID put char,
     PROC (REF FILE, BINCHAR) VOID put bin char,
     PROC (REF FILE, REF CHAR) BOOL get char,
     PROC (REF FILE, REF BINCHAR) VOID get bin char);

c) PROC estab possible = (CHANNEL chan) BOOL:
     estab OF chan;

d) PROC standconv = (CHANNEL chan) PROC (REF BOOK) CONV:
     standconv OF chan;

e) CHANNEL stand in channel =
     C a channel value whose field selected by 'get' is a routine which
       always returns true, and whose other fields are some suitable values
       C;

f) CHANNEL stand out channel =
     C a channel value whose field selected by 'put' is a routine which
       always returns true, and whose other fields are some suitable values
       C;

g) CHANNEL stand back channel =
     C a channel value whose fields selected by 'set', 'reset', 'get', 'put'
       and 'bin' are routines which always return true, and whose other
       fields are some suitable values C;
```

4.2.2.1. SEMANTICS OF THE CHANNEL PRIMITIVES

In this section, both the semantics of the routines that comprise the conversion key, and the primitives that can move the current position outside the current buffer ('do newline', 'do newpage', 'do reset' and 'do set') are given.

In the routines given below, the buffer is supposed to have a (boolean) field 'changed' that tells whether the contents of the buffer have changed since its initialization. This information is used by 'do newline': the contents of the buffer of a random-access file is not written back if there is no need to. If this field is not supported, the value of 'put possible(f)' may be used instead. (This field is also inspected by the routines 'close', 'lock' and 'scratch' (5.2.1,m,n).)

If the buffer corresponds to one line of the text, the semantics are somewhat easier to describe than in the case where a buffer corresponds to a smaller unit of information. Therefore, this case is treated first.

```
a) PROC init buffer = (REF FILE f) VOID:
     # The precondition of 'init buffer' is:
            . opened,
            . either read mood or write mood,
            . NOT physical file ended. #
    BEGIN status OF f ORAB buffer filled;
      IF get possible(f) AND status OF f SAYS logical pos ok
      THEN changed OF buffer OF f:= FALSE;
        IF C the page of the book of f has overflowed C
        THEN status OF f ANDAB page end;
          REF INT(char bound OF f):= 0
        ELSE
          buffer OF f:= C the next buffer from the book of f, possibly
                        after conversion C;
          REF INT(char bound OF f):=
                  C the length of the buffer just filled C;
          test line end(f);
          IF C the logical end is in the buffer just read C
          THEN status OF f ANDAB lfe in current line;
            REF INT(c of lpos OF f):= C the position of the logical end C;
            (status OF f SAYS read mood ! test logical file end(f))
          FI
        FI
      ELSE changed OF buffer OF f:= NOT get possible(f);
        IF C the page of the book of f has overflowed C
        THEN status OF f ANDAB page end;
          REF INT(char bound OF f):= 0
        ELSE REF INT(char bound OF f):=
                      C the maximum length of this buffer C;
          test line end(f)
        FI
      FI
    END;
```

```
b) PROC read buffer = (REF FILE f) VOID:
      # The precondition of 'read buffer' is:
              . opened,
              . get possible(f),
              . NOT logical file end in current line,
              . NOT page ended.#
     BEGIN REF POS(cpos OF f):= (p OF cpos OF f, 1 OF cpos OF f + 1, 1);
       (init buffer OF f)(f)
     END;

c) PROC write buffer = (REF FILE f) VOID:
      # The precondition of 'write buffer' is:
              . opened,
              . write mood,
              . NOT physical file ended,
              . NOT page ended.#
     BEGIN C The contents of the buffer of f (up to the position indicated
             by 'c OF cpos OF f') is, possibly after conversion, written to
             the book of 'f' C;
       IF C this fails to succeed (i.e., the physical end of the book is
            reached while writing this buffer) C
       THEN ensure physical file(f, status OF f);
          # Note that, since we are writing to the book, this call can never
          fail; either the situation is mended, or the program is aborted
          after a suitable error message has been issued. #
         (write buffer OF f)(f)
       ELSE REF POS(cpos OF f):= (p OF cpos OF f, 1 OF cpos OF f + 1, 1);
         IF status OF f SUGGESTS lfe in current line
         THEN set logical pos(f)
         FI;
         (init buffer OF f)(f)
       FI
     END;

d) PROC put char = (REF FILE f, CHAR char) VOID:
      # The precondition of 'put char' is:
              . line ok (see 7.2). #
     BEGIN C The character in 'char' is (possibly after conversion) written
             to the buffer of f at the position indicated by 'c OF cpos OF
             f' C;
       c OF cpos OF f +:= 1;
       changed OF buffer OF f:= TRUE
     END;

e) PROC put bin char = (REF FILE f, BINCHAR char) VOID:
     # The precondition of 'put bin char' is:
            . line ok. #
     BEGIN C The binary character in 'char' is written to the buffer of f at
             the position indicated by 'c Of cpos OF f' C;
       c OF cpos OF f +:= 1;
       changed OF buffer OF f:= TRUE
     END;
```

```
f) PROC get char = (REF FILE f, REF CHAR char) BOOL:
      # The precondition of 'get char' is:
             . line ok. #
     BEGIN CHAR c = C the character read (and possibly converted) from the
                     buffer of f at the position indicated by 'c OF cpos OF
                     f' C;
       c OF cpos OF f +:= 1;
       IF C the conversion succeeds, or no conversion takes place C
       THEN char:= c; TRUE
       ELSE FALSE
       FI
     END;

g) PROC get bin char = (REF FILE f, REF BINCHAR char) VOID:
     # The precondition of 'get bin char' is:
            . line ok. #
     BEGIN char:= C The binary character read from the buffer of f at the
                     position indicated by 'c OF cpos OF f' C;
       c OF cpos OF f +:= 1
     END;

h) PROC do newline = (REF FILE f) VOID:
      # The precondition of 'do newline' is:
             . page ok (see 7.2). #
     IF NOT (status OF f SAYS buffer filled)
     THEN C skip over the current line C;
       IF C this causes the physical end to be reached C
       THEN (status OF f ORAB buffer filled) ANDAB
                        physical file end
       ELIF C it causes the logical end to be reached C
       THEN (init buffer OF f)(f)
       ELIF C the page has overflowed C
       THEN status OF f ANDAB page end;
         REF INT(char bound OF f):= 0
       FI
     ELIF status OF f SUGGESTS lfe in current line AND
          status OF f SAYS read mood
     THEN c OF cpos OF f:= c of lpos OF f; test logical file end(f);
       newline(f)
     ELSE
       IF status OF f SUGGESTS lfe in current line
       THEN c OF cpos OF f:= c of lpos OF f;
         C fill the rest of the buffer with spaces if the file is not
           compressible C
       FI;
       IF changed OF buffer OF f
       THEN (write buffer OF f)(f)
       ELSE (read buffer OF f)(f)
       FI
     FI;
```

```
i) PROC do newpage = (REF FILE f) VOID:
      # The precondition of 'do newpage' is:
              . physical file ok (see 7.2). #
     BEGIN
       WHILE NOT (status OF f SAYS page end)
       DO (do newline OF chan OF f)(f) OD;
       REF POS(cpos OF f):= (p OF cpos OF f + 1, 1, 1);
       IF C this causes the physical file to be ended C
       THEN (status OF f ORAB buffer filled) ANDAB physical file end
       ELSE (init buffer OF f)(f)
       FI
     END;

{This routine is incorporated in the channel for optimization purposes only:
it is expected that more efficient implementations will in general be
available.}

j) PROC do reset = (REF FILE f) VOID:
      # The precondition of 'do reset' is:
              . opened,
              . reset possible. #
     BEGIN REF POS(cpos OF f):= (1, 1, 1);
       C the book is physically reset C;
       (status OF f ANDAB NOT buffer filled) ORAB open status
     END;

i) PROC do set = (REF FILE f, INT p, l, c) VOID:
      # The precondition of 'do set' is:
              . opened,
              . set possible. #
     IF POS(1, 1, 1) EXCEEDS POS(p, l, c)
     THEN error("posmin"); abort
     ELIF C The line indicated by 'p' and 'l' is searched for. Note that
           the current position is updated while searching. Searching may
           stop at the following positions:
           - at the physical file end if 'p' exceeds the number of pages in
             the book of 'f';
           - just beyond the page indicated by 'p' if 'l' exceeds the
             number of lines in that page;
           - at the first position of the last (logical) line if the
             position indicated by 'p', 'l' and 'c' is beyond that
             position;
           - at the first position of the line indicated by 'p' and 'l'
             otherwise. C
       POS(p, l, 1) EXCEEDS cpos OF f OR c > char bound OF f + 1
     THEN error("posmax"); abort
     ELIF (init buffer OF f)(f);
       status OF f SUGGESTS lfe in current line
     THEN STATUS reading = state(f);
       IF POS(p, l, 1) BEYOND cpos OF f OR c > c of lpos OF f
       THEN c OF cpos OF f:= c of lpos OF f;
         (reading SAYS read mood | test logical file end(f));
         BOOL mended = (logical file mended OF f)(f);
         ensure state(f, reading);
         (NOT mended | error("wrongset"); abort)
       ELSE c OF cpos OF f:= c; test line end(f);
```

```
      (reading SAYS read mood | test logical file end(f))
    FI
  ELIF c > 1 AND NOT get possible(f)
  THEN error("setmiddle"); abort
  ELSE c OF cpos OF f:= c;
    test line end(f); test logical file end(f)
  FI;
```

In case the buffer does not correspond to one line of the text, the routines essentially remain the same. The value of the current position will in that case not directly lead to a position in the buffer, but some offset is needed (which must be properly set by 'init buffer'); this offset is assumed to be incorporated in the buffer. Normal transput which takes place via 'put char OF f' and 'get char OF f' then assumes that a next buffer is automatically initiated as soon as the current buffer has overflowed.

4.3. FILES

4.3.1. DEVIATIONS

- {D} The file includes a field 'buffer', which contains a reference to the current buffer. Transput is performed on this buffer, rather than on the text of the book (but see section 5.2 on associated files). The field 'char bound' indicates the maximum number of characters that the current buffer can contain.

- {D} The file includes a field 'status', which contains the status information. This status is inspected before, and updated after, each transput operation.

- {D} A field 'c of lpos' is included in the file. If the logical end is in the current line (which can be derived from the status information), this field indicates the position of the logical file end (and otherwise it is left undefined).

- {S} An enquiry 'backspace possible' is provided, which returns true if the file may be "backspaced" (7.2).

4.3.2. NEW DEFINITION

A "file" is the means of communication between a particular-program and a book which has been opened on that file via some channel. It is the most heavily used concept in the transput section. In the present model, it is considered to be largely machine independent.

A file is a structured value which includes a reference to the book to which it has been linked (5.2). The file includes a reference to the text, which is used for associated files (5.2) and a reference to the buffer, which is used for non-associated files. The file also contains information necessary for the transput routines to work with the book, including its current position "cpos" in the text, its current "status", a reference to its current "format list" (11.2) and the channel on which it has been opened.

The "status" of a file contains the following information:
- whether or not the file has been opened;
- whether or not the buffer has been initialized;
- whether or not the line has overflowed;
- whether or not the page has overflowed;
- whether or not the physical file has overflowed;
- whether or not the logical file is ended;
- whether or not the logical file end is in the current line. (If the logical file end is in the current line, the 'c of lpos'-field points to this logical file end.);
- the "mood" of the file, which is determined by four booleans:
  - 'read mood', which is true if the file is being used for input;
  - 'write mood', which is true if the file is being used for output;
  - 'char mood', which is true if the file is being used for character transput;
  - 'bin mood', which is true if the file is being used for binary transput;
- whether or not the file may be set.

{The present model supposes that it is always possible to determine whether a given file has been opened. So, after a declaration
    FILE f,
or even
    REF FILE f = SKIP,
it must be possible to detect that 'f' is not opened. To this end, all user-callable routines may be assumed to start with an implicit test for the 'status'-field being available. It is not defined here how this can be done in an actual implementation.}

The file also contains the current conversion key, i.e., the set of routines that is currently being used to transput buffers to and from the book and characters to and from the buffer. After opening a file, the conversion key from the channel on which it is opened is provided by default. Some other conversion key may be provided by the programmer by means of a call of 'make conv' (m).
{Note that changing the conversion key may depend on the book, the channel, the current position, both the current and the new conversion key, and other environmental factors not defined by this model.}

The routine 'make term' is used to associate a string with a file. This string is used when inputting a variable number of characters, any of its characters serving as a terminator. {For efficient use of this feature, implementation through a bit table seems natural; to this end, the mode TERM (c) is introduced, together with an operator STRINGTOTERM (c) and a routine 'char in termstring' (c).}

The available methods of access to a book which has been opened on a file may be discovered by calls of the following routines (note that the yield of such a call may be a function of both the book and the channel):
- 'get possible', which returns true if the file may be used for input;
- 'put possible', which returns true if the file may be used for output;
- 'bin possible', which returns true if the file may be used for binary transput;
- 'compressible', which returns true if lines and pages will be compressed (7.2) during output, in which case the book is said to be "compressible";
- 'reset possible', which returns true if the file may be reset, i.e., its current position set to (1, 1, 1);
- 'set possible', which returns true if the file may be set, i.e., the current position changed to some specified value; the book is then said to be a "random access" book and, otherwise, a "sequential access" book. For optimization reasons, this information is also incorporated in the status of the file;
- 'backspace possible', which returns true if the file may be backspaced, i.e., the current position set back over one position if it remains within the current line; backspacing will always be possible if the buffer of the file corresponds to one line of the text {but may for instance not be possible for files that are used interactively};
- 'reidf possible', which returns true if the 'idf' field of the book may be changed;
- 'chan', which returns the channel on which the file has been opened (this may be used, for example, by a routine assigned by 'on physical file end', in order to open another file on the same channel).

{Not all combinations of the above set are sensible. For instance, it is

expected that at least one of 'put possible' and 'get possible' holds. Most likely also, 'reset possible' returns true if 'set possible' does and 'estab possible' implies 'put possible'.}

A file includes some "event routines", which are called when certain conditions arise during transput. After opening a file, the event routines provided by default return false when called, but the user may provide other event routines. The event routines are always given a reference to the file as a parameter. If the calling of an event routine is terminated (by a jump), then the transput routine which called it can take no further action; otherwise, if it returns true, then it is assumed that the condition has been mended in some way, and, if possible, transput continues, but if it returns false, then the system continues with its default action. The "on" routines are:

- 'on logical file end'. The corresponding event routine is called when, during input from a book or as a result of calling 'set' (see 7.2), the logical end of the book is reached.
- 'on physical file end'. The corresponding event routine is called when the current page number of the file exceeds the number of pages in the book and further transput is attempted (see 7.2).
- 'on page end'. The corresponding event routine is called when the current line number of the file exceeds the number of lines in the current page and further transput is attempted (see 7.2).
- 'on line end'. The corresponding event routine is called when the current character number of the file exceeds the number of characters in the current line and further transput is attempted (see 7.2).
- 'on char error'. The corresponding event routine is called when, during input, a character conversion was unsuccessful or a character is read which was not "expected" (section 10.3.4.1.11 of the Revised Report). The event routine is called with a reference to a character suggested as a replacement. The event routine provided by the user may assign some character other than the suggested one. If the event routine returns true, then that suggested character as possibly modified is used.
- 'on value error'. The corresponding event routine is called when:
  i) during formatted transput an attempt is made to transput a value under the control of a "picture" with which it is incompatible, or when the number of "frames" is insufficient. If the routine returns true, then the current value and picture are skipped and transput continues; if the routine returns false, then first, on output, the value is output by 'put', and next the program is aborted;
  ii) during input it is impossible to convert a string to a value of some given mode (this would occur if, for example, an attempt were made to read an integer larger than 'max int'). If the routine returns true, transput continues (although no value is assigned to the item being read in); if the routine returns false, an error message is given and the program is aborted.
- 'on format end'. The corresponding event routine is called when, during formatted transput, the format is exhausted while some value still remains to be transput. If the routine returns true, then the program is aborted if a new format has not been provided for the file by the routine; otherwise, the current format is repeated.

```
a) MODE FILE = STRUCT(
     REF BOOK ? book,
     CHANNEL ? chan,
     REF REF FORMATLIST ? plist,
     REF BUFFER ? buffer,
     REF [][][] CHAR ? text, # for associated files only #
     REF POS ? cpos,
     REF INT ? c of lpos,
     TERM ? term,
     PROC (REF FILE) VOID ? read buffer, ? write buffer, ? init buffer,
     PROC (REF FILE, CHAR) VOID ? put char,
     PROC (REF FILE, BINCHAR) VOID ? put bin char,
     PROC (REF FILE, REF CHAR) BOOL ? get char,
     PROC (REF FILE, REF BINCHAR) VOID ? get bin char,
     REF STATUS ? status,
     REF INT ? char bound,
     PROC (REF FILE) BOOL ? logical file mended, ? physical file mended,
          ? page mended, ? line mended, ? format mended,
          ? value error mended,
     PROC (REF FILE, REF CHAR) BOOL ? char error mended);

b) MODE ? BUFFER = C some mode C;

c) MODE ? TERM = C some other mode C;

   OP ? STRINGTOTERM = (STRING s) TERM:
     C The terminator string in 's' is converted to a corresponding value of
       the mode specified by TERM C;

   PROC ? char in termstring = (CHAR k, TERM t) BOOL:
     C This routine returns true if the character 'k' is contained in the
       terminator string 't', and false otherwise C;

d) PROC get possible = (REF FILE f) BOOL:
     IF status OF f SAYS opened
     THEN (get OF chan OF f)(book OF f)
     ELSE error("notopen"); abort
     FI;

e) PROC put possible = (REF FILE f) BOOL:
     IF status OF f SAYS opened
     THEN (put OF chan OF f)(book OF f)
     ELSE error("notopen"); abort
     FI;

f) PROC bin possible = (REF FILE f) BOOL:
     IF status OF f SAYS opened
     THEN (bin OF chan OF f)(book OF f)
     ELSE error("notopen"); abort
     FI;

g) PROC compressible = (REF FILE f) BOOL:
     IF status OF f SAYS opened
     THEN (compress OF chan OF f)(book OF f)
     ELSE error("notopen"); abort
     FI;
```

```
h) PROC reset possible = (REF FILE f) BOOL:
     IF status OF f SAYS opened
     THEN (reset OF chan OF f)(book OF f)
     ELSE error("notopen"); abort
     FI;

i) PROC set possible = (REF FILE f) BOOL:
     IF status OF f SAYS opened
     THEN NOT (status OF f SAYS not set poss)
     ELSE error("notopen"); abort
     FI;

j) PROC backspace possible = (REF FILE f) BOOL:
     IF status OF f SAYS opened
     THEN (backspace OF chan OF f)(book OF f)
     ELSE error("notopen"); abort
     FI;

k) PROC reidf possible = (REF FILE f) BOOL:
     IF status OF f SAYS opened
     THEN (reidf OF chan OF f)(book OF f)
     ELSE error("notopen"); abort
     FI;

l) PROC chan = (REF FILE f) CHANNEL:
     IF status OF f SAYS opened
     THEN chan OF f
     ELSE error("notopen"); abort
     FI;

m) PROC make conv = (REF FILE f, PROC (REF BOOK) CONV c) VOID:
     IF status OF f SAYS opened
     THEN
       C some implementation-dependent tests will probably be needed here:
         whether the conversion key may be changed might depend on the
         current and the newly given conversion key, the book, the channel,
         and other environmental factors.  If the conversion key may be
         changed, the routines in the file that comprise the mode CONV have
         to be exchanged.  One must take care that conversion keys from
         associated files (5.2.d) do not get used as parameter to this
         routine; also, the conversion key of such a file may not be
         changed. C
     ELSE error("notopen"); abort
     FI;

n) PROC make term = (REF FILE f, STRING t) VOID:
     term OF f:= STRINGTOTERM t;

o) PROC on logical file end =
             (REF FILE f, PROC (REF FILE) BOOL p) VOID:
     logical file mended OF f:= p;

p) PROC on physical file end =
             (REF FILE f, PROC (REF FILE) BOOL p) VOID:
     physical file mended OF f:= p;
```

```
q) PROC on page end =
            (REF FILE f, PROC (REF FILE) BOOL p) VOID:
     page mended OF f:= p;

r) PROC on line end =
            (REF FILE f, PROC (REF FILE) BOOL p) VOID:
     line mended OF f:= p;

s) PROC on format end =
            (REF FILE f, PROC (REF FILE) BOOL p) VOID:
     format mended OF f:= p;

 t) PROC on value error =
            (REF FILE f, PROC (REF FILE) BOOL p) VOID:
     value error mended OF f:= p;

 u) PROC on char error =
            (REF FILE f, PROC (REF FILE, REF CHAR) BOOL p) VOID:
     char error mended OF f:= p;

 v) PROC reidf = (REF FILE f, STRING idf) VOID:
     IF status OF f SAYS opened & reidf possible(f) & idf ok(idf)
     THEN
       C The identification string of the book of 'f' is made to be 'idf'. C
     FI;
```

5. OPENING AND CLOSING FILES

5.1. DEVIATIONS

- {S} The numerous calls of 'undefined' in the Revised Report have been assigned meanings. Hidden kinds of undefined actions like SKIP, UP gremlins and UP bfileprotect have been paid due attention. At places where 'undefined' was called in the Revised Report, the present system issues an error message. In case no sensible continuation is possible, the elaboration of the particular program is aborted by means of a jump to the label 'abort'. Following this label, all buffers still need to be emptied, and all files are subsequently closed. The "gremlins" are not activated in the present system either; rather, opening a file is treated as a special kind of assignment.

- {S} If opening is not successful, non zero error codes are returned, and 'undefined' is not called.

- {D} The validity check for the parameters 'p', 'l' and 'c' in 'establish' is performed differently, since the operator BEYOND used in the Revised Report does not trap all erroneous combinations of 'p', 'l' and 'c'.

- {D} The routines 'close', 'lock' and 'scratch' are considered primitive actions.

- {S} If a file is "associated", all upper bounds of the multiple value referred to must be at least 1.

5.2. NEW DEFINITION

A book is "linked" with a file by means of 'establish' (a), 'create' (b) or 'open' (c). The linkage may be terminated by means of 'close' (1), 'lock' (m) or 'scratch' (n).

When a file is "established" on a channel, then a book is constructed (4.1.2.f) with a 'text' of the given size and the given identification string, that may be written to. The logical end of the book is set to (1, 1, 1). {An implementation may require (f) that the characters forming the identification string should be taken from a limited set and that the string should be limited in length. It may also prevent two books from having the same string.} If the establishing is completed successfully, then the value 0 is returned; otherwise, some non zero integer is returned, which indicates why the file was not established successfully. A list of these error codes is given below.

When a file is "created" on a channel, then a file is established with a book whose text has the default size for the channel, and whose identification string is a default identification string (4.1.2.h) for the implementation.

When a file is "opened", then the pool of available books is searched for a book with the following properties:
- The book may be identified by the given identification string;
- The book may be legitimately accessed through the given channel;

- Opening is not inhibited by other users of the book.

If such a book cannot be found, an appropriate error message is given and a non zero integer is returned. Otherwise, the file variable is initialized properly, and the value 0 is returned.

The routine 'associate' may be used to "associate" a file with a value of the mode specified by either REF CHAR, REF [] CHAR, REF [][] CHAR or REF [][][] CHAR, thus enabling such variables to be used as the book of a file. All lower bounds of the multiple value referred to must be 1, and all upper bounds should be at least 1. Note that the scope of the multiple value must not be newer than the scope of the given file 'f'.

For associated files, the buffer mechanism cannot be used: here it is necessary to transput directly to or from the associated multiple value, since the user also has direct access to it. It is for this reason that a separate reference to a 'text' is included in the file. The various primitives that have been incorporated in the channel also directly access the text in this case.

As a consequence of linking a book with a file, the fields of the file are initialized as follows:

- 'book'.
  In 'establish' and 'create', a book is constructed (4.1.2.f) according to the requirements of the user. In 'open', the pool of available books is searched for a matching one (4.1.2.e). In 'associate', a pseudo book (4.1.2.g) is used;
- 'chan'.
  In 'establish', 'create' and 'open', the channel is provided by the user. In 'associate', a channel value is constructed
  i) whose fields selected by 'reset', 'set', 'get', 'put' and 'backspace' always return true,
  ii) whose fields selected by 'bin', 'compress', 'reidf' and 'estab' always return false,
  iii) whose field selected by 'max pos' returns a value of mode POS whose fields selected by 'p', 'l' and 'c' return 'max int',
  iv) whose field selected by 'standconv' is such that transput takes place directly on the multiple value, without any intermediate conversion,
  v) whose fields selected by 'do newline', 'do newpage' and 'do reset' simply move the current position and update the status information accordingly,
  vi) whose field selected by 'do set' tests whether the user-supplied position is valid (possibly after calling the event routine corresponding to 'on logical file end'), and, if so, moves the current position to the supplied one, but if not, gives an appropriate error message, whereupon the elaboration of the particular program is aborted (see also 4.2.2.1.i),
  vii) whose field selected by 'channel number' is left undefined;
- 'plist'.
  The field selected by 'plist' is assigned a nil name;
- 'buffer'.
  As a result of calling 'establish' or 'create', a reference to the buffer is yielded by the routine that constructs the book; this buffer is initialized. When 'open' is called, the buffer is yielded by 'book in system' along with the book. It is not initialized; initialization has to await the first transput operation. The field 'buffer' is left

undefined (and is not used) for associated files;
- 'text'.
The field 'text' is only used for associated files. It then contains a reference to the associated multiple value; otherwise, it is left undefined;
- 'cpos'.
The current position is always initialized to (1, 1, 1);
- 'c of lpos'.
The 'c of lpos' field is linked up with the logical end of the book. If the logical end is in the current (i.e., lst) line, 'c of lpos' points to the position where the logical end is. Otherwise it is left undefined. Whether the logical end is in the current line can be derived from the status information via the 'lfe in current line' flag. If a file is established or created, the logical file end is at (1, 1, 1) initially; when a file is opened it depends on the book and the read/write mood; in this case initialization has to await the first transput operation ('init buffer' will then take care of it). When a file is associated with a multiple value, the logical end is just beyond the extreme end of the associated multiple value;
- 'term'.
Initially, the terminator string is the empty string;
- 'read buffer', 'write buffer', 'init buffer', 'put char', 'put bin char', 'get char', 'get bin char'.
For associated files, these routines are built by the routine 'associate'. They do not make use of an intermediate buffer, nor do they perform any conversion. For the other opening routines they are copied from the given channel;
- 'status'.
In all cases, the initial information is such that the file is opened, and the line, page and physical file are not ended. The logical file end is at the current position if a file is established or created. So in that case the 'lfe in current line' flag is raised. {The 'logical file ended' flag is not raised since the write mood will be true.} For associated files they are both not raised, since in that case the text is assumed to have at least one page. If 'open' is called, then setting these flags has to await the first transput operation. The read mood is set to true if a file is linked to a book via 'open', and writing is not possible on the given channel; otherwise it is set to false. The write mood is set to true if a file is linked to a book via 'establish' or 'create', or via 'open' in case reading is not possible on the given channel; otherwise it is set to false. The char mood is set to true if a file is linked to a book, and binary transput is not possible on the given channel; otherwise it is set to false. Lastly, the bin mood is always initialized to false. The 'set poss' flag indicates whether the file is random access or not. It is incorporated in the status information to simplify some of the tests;
- 'char bound'.
The length of the current buffer differs from case to case:
  - it is given if 'establish' is called (the parameter 'c'),
  - it is the maximum buffer length of the given channel if 'create' is called;
  - it depends on the book if 'open' is called;
  - it depends on the associated multiple value if 'associate' is called (then it is equal to UPB sss[1][1]);

- the 'on routines'.
  The 'on routines' are all initialized to routines returning false.

Files may be "closed", "locked" or "scratched". For associated files, these all just amount to updating the status information such that the file is no longer said to be opened. For files that are "opened", "created" or "established", the resulting action is different for each routine. If a file is closed, the book is put back into the pool of available books, so that the book may be re-opened at some later stage (e.g., if the channel corresponds to a tape-unit, the tape is not removed yet). If a book is locked, it cannot be re-opened until some subsequent system-task has put the book back into the pool (in this case, the tape is removed and put on the shelf). If a file is scratched, the book is disposed of by the system in some way (i.e., its information generally gets lost). In all cases, the current buffer may still have to be transput to the book, possibly resulting in an end-of-page or end-of-file condition and the like. The status information is updated such that the file is no longer said to be opened.

If opening is not successful, a non zero error code is returned. These error codes consist of two parts: a "general" part and a "specific" part. The general part can in a loose way be associated with the various validity tests; the specific parts depend on the particular operating environment, and they are not further specified in this report. {This strategy is borrowed from [11]}. The general error codes used have been given the following symbolic names:

(1) 'badidf' - The string argument to 'open' or 'establish' is wrong; in 'establish', the call of 'idf ok' returns false, in 'open' no matching book is found.

(2) 'nowrite' - Writing cannot be done.

(3) 'notavail' - 'file available' returns false. {This may be a temporary problem which can be handled by re-trying the calling of 'open' or 'establish' - for instance, there is no room in the directory system for a new file.}

(4) 'noestab' - 'estab OF chan' returns false: files simply cannot be established on this channel.

(5) 'posmax' - One of the dimension arguments to 'establish' or 'set' is too large.

(6) 'posmin' - One of the dimension arguments to 'establish' or 'set' is too small (i.e., ≤ 0).

(7) 'notopen' - A transput routine is called with a file as parameter which is not opened.

(8) 'noread' - Reading cannot be done.

(9) 'noset' - An attempt is made to set a file for which 'set possible' returns false.

(10) 'noreset' - An attempt is made to reset a file for which 'reset possible' returns false.

(11) 'nobackspace' - An attempt is made to backspace on a file for which 'backspace possible' returns false.

(12) 'noshift' - An attempt is made to shift from 'bin mood' to 'char mood' or vice versa on a sequential-access file.

(13) 'nobin' - Binary transput is not possible.

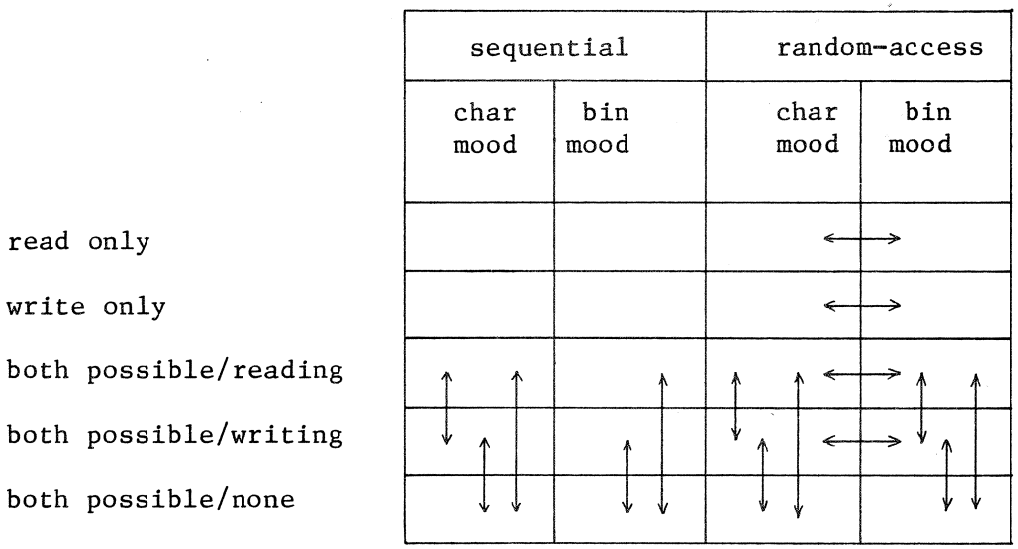
(14) 'noalter' - An attempt is made to shift from 'read mood' to 'write mood' or vice versa on sequential-access files in 'bin mood'.

(15) 'nomood' - Transput is attempted while no mood has been set yet.
(16) 'wrongmult' - 'associate' is called with a reference to a multiple value as parameter whose bounds are incorrect.
(17) 'wrongset' - 'set' is called with a position as parameter which is beyond the logical end, and this situation is not mended by the user.
(18) 'nocharpos' - No "good" character position can be ensured.
(19) 'noline' - No "good" line can be ensured.
(20) 'nopage' - No "good" page can be ensured.
(21) 'wrongpos' - 'set char number' is called with a wrong character position as parameter.
(22) 'wrongchar' - 'get' or 'getf' encounters a character which is not expected, or one which cannot be converted, and this situation is not mended by the user.
(23) 'wrongval' - In 'get', it is impossible to convert a string to a value of some given mode, or during formatted transput an attempt is made to transput a value under the control of a picture with which it is incompatible, or whose number of frames is insufficient, and this situation is not mended by the user.
(24) 'setmiddle' - An attempt is made to set a write-only file to a position which is not at the beginning of a line (see also section 7.1).
(25) 'wrongbacksp' - Backspace is called with the current position at the beginning of a line.
(26) 'smallline' - An attempt is made to output a number to a line which is too small to contain that number.
(27) 'noformat' - 'putf' or 'getf' is called while no format is associated with the given file, or the user has not provided a new format while the event routine corresponding to 'on format end' returns true.
(28) 'wrongformat' - During the elaboration of the first insertion of a 'collitem', the user has changed the current format of the file.
(29) 'wrongbin' - During binary transput, it is not possible to input a value of some given mode.

If more than one validity test needs to be performed, they are performed one after the other; if one test fails, the corresponding error code is returned and the remaining tests are not performed.

Files can be categorized according to various different properties:
- a file may be random access or sequential;
- a file may be read-only, write-only, or both may be possible;
- a file may be used for both character and binary transput, or for character transput only.

Below, a state diagram is given in which the various possible changes in the mood are depicted for both sequential and random-access files (binary transput is supposed to be possible here).



{There is one striking irregularity in the above picture: reading and writing may not be alternated on sequential-access files if the file is used for binary transput.}

{Note that arrows going downwards into the category "both possible/none" correspond to transitions caused by a call of 'reset'.}

```
a) PROC establish =
       (REF FILE f, STRING idf, CHANNEL chan, INT p, l, c) INT:
     IF NOT file available(chan) THEN error("notavail")
     ELIF NOT estab OF chan THEN error("noestab")
     ELIF POS(p, l, c) EXCEEDS max pos OF chan THEN error("posmax")
     ELIF POS(1, 1, 1) EXCEEDS POS(p, l, c) THEN error("posmin")
     ELIF NOT idf ok(idf) THEN error("badidf")
     ELSE
       HEAP BUFFER buffer;
       HEAP BOOK book:=
            construct book(p, l, c, idf, chan, buffer);
       IF NOT (put OF chan)(book) THEN error("nowrite")
       ELSE
         CONV cc = (standconv OF chan)(book);
         STATUS st:= establish status OR write mood;
         (NOT (bin OF chan)(book) | st ORAB char mood);
         (NOT (set OF chan)(book) | st ORAB not set poss);
         f:=
           (book, chan, REF REF FORMATLIST(NIL), buffer, SKIP,
            HEAP POS:= (1, 1, 1), HEAP INT:= 1,
            STRINGTOTERM "",
            read buffer OF cc, write buffer OF cc, init buffer OF cc,
            put char OF cc, put bin char OF cc,
            get char OF cc, get bin char OF cc,
            HEAP BITS:= st,
                 #i.e., opened &
                  lfe in current (lst) line & buffer initialized #
            HEAP INT:= c,
            false, false, false, false, false, false,
            (REF FILE f, REF CHAR c) BOOL: FALSE);
         0
```

```
      FI
    FI;

b) PROC create = (REF FILE f, CHANNEL chan) INT:
    BEGIN POS max pos = max pos OF chan;
      establish(f, default idf, chan, p OF max pos, 1 OF max pos,
                c OF max pos)
    END;

c) PROC open = (REF FILE f, STRING idf, CHANNEL chan) INT:
    IF NOT file available(chan) THEN error("notavail")
    ELIF REF BOOK book, HEAP BUFFER buffer;
      INT er = book in system(idf, chan, book, buffer);
      er ≠ 0
    THEN er
    ELSE CONV c = (standconv OF chan)(book);
      STATUS st:= open status;
      (NOT (put OF chan)(book) | st ORAB read mood);
      (NOT (get OF chan)(book) | st ORAB write mood);
      (NOT (bin OF chan)(book) | st ORAB char mood);
      (NOT (set OF chan)(book) | st ORAB not set poss);
      f:=
        (book, chan, REF REF FORMATLIST(NIL), buffer, SKIP,
         HEAP POS:= (1, 1, 1), HEAP INT:= SKIP,
         STRINGTOTERM "",
         read buffer OF c, write buffer OF c, init buffer OF c,
         put char OF c, put bin char OF c,
         get char OF c, get bin char OF c,
         HEAP BITS:= st,
         HEAP INT:= SKIP,
         false, false, false, false, false, false,
         (REF FILE f, REF CHAR c) BOOL: FALSE);
      0
    FI;

d) PROC associate = (REF FILE file, REF [][][] CHAR sss) VOID:
    IF LWB sss ≠ 1 OR UPB sss < 1
    THEN error("wrongmult"); abort
    ELIF LWB sss[1] ≠ 1 OR UPB sss[1] < 1
    THEN error("wrongmult"); abort
    ELIF LWB sss[1][1] ≠ 1 OR UPB sss[1][1] < 1
    THEN error("wrongmult"); abort
    ELSE
      PROC t = (REF BOOK a) BOOL: TRUE;
      PROC f = (REF BOOK a) BOOL: FALSE;

      PROC new buffer = (REF FILE f) VOID:
        # page is ok #
        (INT l = 1 OF cpos OF f +:= 1;
         c OF cpos OF f:= 1;
         IF l > UPB (text OF f)[1]
         THEN status OF f ANDAB page end
         FI);
```

```
CONV c = (
 # read buffer #
   new buffer,
 # write buffer #
   new buffer,
 # init buffer #
   (REF FILE f) VOID:
     REF STATUS(status OF f):= associate status OR char mood,
 # put char #
   (REF FILE f, CHAR char) VOID:
     BEGIN REF POS cpos = cpos OF f;
       (text OF f)[p OF cpos][1 OF cpos][c OF cpos]:= char;
       c OF cpos +:= 1; TRUE
     END,
 # put bin char #
   SKIP,
 # get char #
   (REF FILE f, REF CHAR char) BOOL:
     BEGIN REF POS cpos = cpos OF f;
       char:= (text OF f)[p OF cpos][1 OF cpos][c OF cpos];
       c OF cpos +:= 1; TRUE
     END,
 # get bin char #
   SKIP);

CHANNEL chan =
    (t, t, t, t, f, f, f, t, BOOL: FALSE,
     POS: (max int, max int, max int),
     # newline #
     new buffer,
     # newpage #
     (REF FILE f) VOID:
       (REF POS(cpos OF f):= (p OF cpos OF f + 1, 1, 1);
        STATUS reading = state(f);
        REF STATUS(status OF f):=
          IF p OF cpos OF f = UPB text OF f + 1
          THEN REF INT(c of lpos OF f):= 1;
            IF reading SAYS read mood
            THEN associate status & logical file ended
            ELSE associate status
            FI
          ELSE associate status
          FI OR reading),
     # reset #
     (REF FILE f) VOID:
       (REF POS(cpos OF f):= (1, 1, 1);
        REF INT(c of lpos OF f):= SKIP;
        REF STATUS(status OF f):= associate status
                                  OR char mood),
```

```
            # set #
            (REF FILE f, INT p, l, c) VOID:
              IF INT up = UPB text OF f + 1;
                POS lpos = (up, 1, 1), STATUS reading = state(f);
                POS(p, 1, c) BEYOND lpos
              THEN REF POS(cpos OF f):= lpos;
                (reading SAYS read mood | test logical file end(f));
                BOOL mended = (logical file mended OF f)(f);
                ensure state(f, reading);
                (NOT mended | error("wrongset"); abort)
              ELIF
                INT ul = UPB (text OF f)[1] + 1,
                    uc = UPB (text OF f)[1][1] + 1;
                BOOL fne = p < up;
                BOOL pne = (fne | 1 < ul | FALSE);
                BOOL lne = (pne | c < uc | FALSE);
                BOOL lfe = NOT fne & reading SAYS read mood;
                POS(p, 1, c) EXCEEDS
                   POS(up, (fne | ul | 1), (pne | uc | 1))
              THEN error("posmax"); abort
              ELIF POS(1, 1, 1) EXCEEDS POS(p, 1, c)
              THEN error("posmin"); abort
              ELSE REF POS(cpos OF f):= (p, 1, c);
                REF INT(c of lpos OF f):= (fne | SKIP | 1);
                REF STATUS(status OF f):=
                    bits pack((TRUE, TRUE, lne, pne, fne, lfe, fne,
                               FALSE, FALSE, FALSE, FALSE, FALSE))
                         OR reading
              FI,
            (REF BOOK b) CONV: c,
            SKIP);

        file:=
         (pseudo book(UPB sss), chan, REF REF FORMATLIST(NIL), SKIP, sss,
          HEAP POS:= (1, 1, 1), HEAP INT:= SKIP,
          STRINGTOTERM "",
          read buffer OF c, write buffer OF c, init buffer OF c,
          put char OF c, put bin char OF c,
          get char OF c, get bin char OF c,
          HEAP STATUS:= associate status OR char mood,
          HEAP INT:= UPB sss[1][1],
          false, false, false, false, false, false,
          (REF FILE f, REF CHAR c) BOOL: FALSE)
    FI;

e) PROC ? file available = (CHANNEL chan) BOOL:
     C true if another file, at this instant of time, may be "opened",
       "established" or "created" on 'chan', and false otherwise C;

f) PROC ? idf ok = (STRING idf) BOOL:
     C true if 'idf' is acceptable to the implementation as the
       identification of a new book and false otherwise C;

g) PROC ? false = (REF FILE f) BOOL: FALSE;
```

```
h) PROC ? set write mood = (REF FILE f) VOID:
     # opened and (in general) NOT write mood #
     IF NOT (put OF chan OF f)(book OF f)
     THEN error("nowrite"); abort
     ELIF status OF f SAYS read to write not possible
     THEN error("noalter"); abort
     ELSE status OF f ANDAB NOT read mood;
       status OF f ORAB (write mood OR logical pos ok)
     FI  # opened & write mood #;

i) PROC ? set read mood = (REF FILE f) VOID:
     # opened and (in general) NOT read mood #
     IF NOT (get OF chan OF f)(book OF f)
     THEN error("noread"); abort
     ELIF status OF f SAYS write to read not possible
     THEN error("noalter"); abort
     ELSE status OF f ORAB read mood;
       IF status OF f SAYS write mood
       THEN status OF f ANDAB NOT write mood;
         mind logical pos(f)
       FI
     FI  # opened & read mood #;

j) PROC ? set char mood = (REF FILE f) VOID:
     # opened and (in general) NOT char mood #
     IF status OF f SAYS bin to char not possible
     THEN error("noshift"); abort
     ELSE status OF f ORAB char mood;
       status OF f ANDAB NOT bin mood
     FI  # opened & char mood #;

k) PROC ? set bin mood = (REF FILE f) VOID:
     # opened and (in general) NOT bin mood  #
     IF NOT (bin OF chan OF f)(book OF f)
     THEN error("nobin"); abort
     ELIF status OF f SAYS char to bin not possible
     THEN error("noshift"); abort
     ELSE status OF f ORAB bin mood;
       status OF f ANDAB NOT char mood
     FI  # opened & bin mood #;

l) PROC close = (REF FILE f) VOID:
     BEGIN
       IF changed OF buffer OF f
       THEN (write buffer OF f)(f)
       FI;
       status OF f ANDAB closed;
       C
         The information on the number of users is updated.  Some system-
         task is activated to actually close the book; in this case, the
         book may be re-opened. C
     END;
```

```
m) PROC lock = (REF FILE f) VOID:
     BEGIN
       IF changed OF buffer OF f
       THEN (write buffer OF f)(f)
       FI;
       status OF f ANDAB closed;
       C The information on the number of users is updated.  Some system-
         task is activated to actually lock the book; in this case, it is
         not possible to re-open the book (except possibly after some
         subsequent system-task). C
     END;

n) PROC scratch = (REF FILE f) VOID:
     BEGIN
       IF changed OF buffer OF f
       THEN (write buffer OF f)(f)
       FI;
       status OF f ANDAB closed;
       C The information on the number of users is updated.  The book is
         disposed of in some way by the system. C
     END;
```

6. POSITION ENQUIRIES

6.1. DEVIATIONS

- {D} The present model makes no use of the routines 'current pos', 'book bounds', 'line ended', 'page ended', 'physical file ended' and 'logical file ended'. In the present model, the status of the file is inspected rather than its current position.

6.2. NEW DEFINITION

The current position of a book opened on a given file is the value referred to by the 'cpos' field of that file. It is advanced by each transput operation in accordance with the number of characters written or read.

If c is the current character number and lb is the (maximum) length of the current line, then at all times $1 \leq c \leq lb+1$. c=1 implies that the next transput operation will be on the first position of the line, and c = lb+1 implies that the line has overflowed and that the next transput operation will call an event routine. If lb = 0, then the line is empty and is therefore always in the overflowed state. Corresponding restrictions apply to the current line and page numbers. Note that, if the page has overflowed, the current line is empty, and, if the book has overflowed, the current page and line are both empty.

The user may determine the current position by means of the routines 'char number', 'line number' and 'page number' (a, b, c).

The "status" of a file contains the following information:
- whether or not the file has been opened;
- whether or not the buffer has been initialized;
- whether or not the line has overflowed;
- whether or not the page has overflowed;
- whether or not the physical file has overflowed;
- whether or not the logical file is ended;
- whether or not the logical file end is in the current line;
- whether or not the file is being used for input;
- whether or not the file is being used for output;
- whether or not the file is being used for character transput;
- whether or not the file is being used for binary transput;
- whether or not the file is random access.

In order to achieve an efficient implementation, the status is not defined as a set of separate booleans. Rather, the status is defined to be of the mode BITS. (On a machine where 'bits width' is less than 12, some trivial modifications have to be made in the definitions given below (d, e and f).)

As a consequence of a call of one of the routines 'set', 'reset', 'set char number' and 'backspace', the current position may be set back. Following the philosophy sketched in Chapter 2, the current position 'cpos' and the logical position 'lpos' are related through

```
cpos = cpos1
lpos = MAX(cpos1, lpos1),                    (*)
```

where 'cposl' and 'lposl' are the variables maintained by the system described in this report ('cposl' and 'lposl' are termed 'cpos' and 'lpos' in the text again). If one of the above mentioned routines is called, relation (*) may no longer be valid if lposl < cposl. For this purpose, the routine 'mind logical pos' has been provided. Since the logical position is of importance when reading and writing is alternated, 'mind logical pos' is also called from within 'set read mood'.

The status is inspected before each transput operation. This inspection generally proceeds in two steps:

i) one overall test (which depends on the transput operation); the "normal" situation will be detected by this test, so that transput may often be continued after one single test;
ii) in case the overall test fails, a chain of tests is activated to detect the specific condition that fails to hold.

After each transput operation, the status of the file is updated. Routines are provided to update the 'line end' and 'logical file end' information. Updating 'buffer filled', 'page end', 'physical file end' and 'logical file end in current line' information obviously is one of the tasks of the routines 'read buffer', 'write buffer' and 'init buffer'.

```
a) PROC char number = (REF FILE f) INT:
     IF status OF f SAYS opened
     THEN c OF cpos OF f
     ELSE error("notopen"); abort
     FI;

b) PROC line number = (REF FILE f) INT:
     IF status OF f SAYS opened
     THEN l OF cpos OF f
     ELSE error("notopen"); abort
     FI;

c) PROC page number = (REF FILE f) INT:
     IF status OF f SAYS opened
     THEN p OF cpos OF f
     ELSE error("notopen"); abort
     FI;

d) MODE ? STATUS = BITS;
     # The bits in the status have the following meaning
       (they are numbered from left to right):
       bit  1 = 1 <=> the file is opened;
       bit  2 = 1 <=> the buffer is initialized;
       bit  3 = 1 <=> NOT line ended;
       bit  4 = 1 <=> NOT page ended;
       bit  5 = 1 <=> NOT physical file ended;
       bit  6 = 1 <=> NOT logical file ended;
       bit  7 = 1 <=> NOT lfe in current line;
       bit  8 = 1 <=> read mood;
       bit  9 = 1 <=> write mood;
       bit 10 = 1 <=> char mood;
       bit 11 = 1 <=> bin mood;
       bit 12 = 1 <=> NOT set possible. #
```

```
e) PRIO ? ORAB = 1,
   OP ORAB = (REF STATUS s, STATUS t) REF STATUS: s:= s OR t;

   PRIO ? ANDAB = 1,
   OP ANDAB = (REF STATUS s, STATUS t) REF STATUS: s:= s AND t;

   PRIO ? SAYS = 6,
   OP SAYS = (STATUS s, t) BOOL: s ≥ t;

   PRIO ? SUGGESTS = 6,
   OP SUGGESTS = (STATUS s, t) BOOL: s ≤ t;
```

{Sometimes one wants to know whether certain bits are on (then SAYS is used), sometimes whether they are off (then SUGGESTS is used).}

```
f) # Some constant-declarations. #
   STATUS ? put char status             = 2r 1 0 000 00 0110 0,
          ? get char status             = 2r 1 0 000 00 1010 0,
          ? put bin status              = 2r 1 0 000 00 0101 0,
          ? get bin status              = 2r 1 0 000 00 1001 0,

          ? line ok                     = 2r 1 1 111 10 0000 0,
          ? page ok                     = 2r 1 0 011 10 0000 0,
          ? physical file ok            = 2r 1 0 001 10 0000 0,

          ? logical pos ok              = 2r 1 0 000 10 0000 0,
          ? logical pos not ok          = 2r 0 0 000 01 0000 1,
          ? logical file ended          = 2r 1 1 111 00 1111 1,

          ? opened                      = 2r 1 0 000 00 0000 0,
          ? closed                      = 2r 0 0 000 00 0000 0,
          ? buffer filled               = 2r 0 1 000 00 0000 0,
          ? not lfe in current line     = 2r 0 0 000 01 0000 0,
          ? lfe in current line         = 2r 1 1 111 10 1111 1,
          ? line end                    = 2r 1 1 011 11 1111 1,
          ? page end                    = 2r 1 1 001 11 1111 1,
          ? physical file end           = 2r 1 1 000 11 1111 1,
          ? associate end               = 2r 1 1 000 11 0000 1,

          ? not set poss                = 2r 0 0 000 00 0000 1,

          ? establish status            = 2r 1 1 111 10 0000 1,
          ? open status                 = 2r 1 0 111 11 0000 0,
          ? associate status            = 2r 1 1 111 11 0000 0,

          ? read mood                   = 2r 0 0 000 00 1000 0,
          ? write mood                  = 2r 0 0 000 00 0100 0,
          ? char mood                   = 2r 0 0 000 00 0010 0,
          ? bin mood                    = 2r 0 0 000 00 0001 0,
          ? read or write mood          = 2r 0 0 000 00 1100 0,
          ? mood part                   = 2r 0 0 000 00 1111 0,

          ? read to write not possible = 2r 1 0 000 00 1001 1,
          ? write to read not possible = 2r 1 0 000 00 0101 1,
          ? bin to char not possible   = 2r 1 0 000 00 0001 1,
          ? char to bin not possible   = 2r 1 0 000 00 0010 1;
```

```
g) PROC ? mind logical pos = (REF FILE f) VOID:
     IF status OF f SAYS not lfe in current line
     THEN SKIP
     ELSE
       IF c OF cpos OF f > c of lpos OF f
       THEN REF INT(c of lpos OF f):= c OF cpos OF f
       FI;
       test logical file end(f)
     FI;

h) PROC ? test line end = (REF FILE f) VOID:
     IF c OF cpos OF f > char bound OF f
     THEN status OF f ANDAB line end
     FI;

i) PROC ? test logical file end = (REF FILE f) VOID:
     IF c OF cpos OF f = c of lpos OF f
     THEN status OF f ANDAB logical file ended
     FI;
```

7. LAYOUT ROUTINES

7.1. DEVIATIONS

- {S} The Revised Report specifies that 'space', 'newline' and 'newpage' act as skip operations if the logical file end is not yet reached (is not in the current line or page, respectively). This requirement will be difficult to fulfil on write-only files as far as 'space' is concerned. In the present model, blanks are written in that case.

- {S} A write-only file can only be set to a position which is at the beginning of a line (so 'c' equals 1); it is not expected that the system can start to write in the middle of a line that cannot be read (see also the description of 'do set' in section 4.2.2.1).

- {S} 'backspace' is not assumed to be possible on all files. For files having very long lines, the buffer may correspond to a smaller piece of information. In that case, 'backspace' is not allowed in the present model. The user may discover whether backspacing is possible through the environment enquiry 'backspace possible' (4.3.2.j). If 'backspace' is called with a file as parameter for which backspacing is not possible, an error message is given and the program is aborted.

- {D} The 'get good'-routines from the Revised Report are termed 'ensure'-routines in the present model. For clarity's sake, they are written (right-) recursively rather than with the aid of a while loop. {Of course, this can easily be optimized out.}

- {D} The routine 'get good file' in the Revised Report serves a twofold purpose: on reading, it is tested whether the current position is still within the logical file; on writing, the current position is checked against the physical file end. This task has been split up into two separate routines: 'ensure logical file' and 'ensure physical file'.

7.2. NEW DEFINITION

A book input from an external medium by some system-task may contain lines and pages not all of the same length. Contrariwise, the lines and pages of a book which has been established (5.2.a) are all initially of the size specified by the user. However if, during output to a compressible book (4.3.2), 'newline' ('newpage') is called with the current position in the same line (page) as the logical end of the book, then that line (the page containing that line) is shortened to the character number (line number) of the logical end. Thus 'print(("abcde", newline))' could cause the current line to be reduced to 5 characters in length. Note that it is perfectly meaningful for a line to contain no characters and for a page to contain no lines.

The routines 'space' (a), 'newline' (c) and 'newpage' (d) serve to advance the current position to the next character, line or page, respectively. They do not alter the contents of the positions skipped over, except during output with the current position at the logical end of the book.

If, during character output with the current position at the logical end

of the book, 'space' is called, then a space character is written (similar action being taken in the case of 'newline' and 'newpage' if the book is not compressible). Note that on sequential-access files, 'space' is only treated as a skip operation if the logical end is in the current line! Thus, 'print(("a", backspace, space))' has a different effect from 'print(("a", backspace, blank))', while 'print(space)' may well have the same effect as 'print(blank)', even if the current position is not at the logical end of the book.

The current position may be altered also by calls of 'set char number' (o) and, on appropriate channels, of 'backspace' (b), 'set' (m) and 'reset' (n).

A call of 'set' which attempts to leave the current position beyond the logical end results in an error message, after which execution of the particular program is aborted. There is thus no defined way in which the current position can be made to be beyond the logical end, nor in which any character within the logical file can remain in its initial undefined state.

The mood of the file (4.3.2) controls some effects of the layout routines. If the read/write mood is reading, the effect of 'space', 'newline' and 'newpage', upon attempting to pass the logical end, is to call the event routine corresponding to 'on logical file end'; the default action then is to give an error message and abort the elaboration of the particular program. If the read/write mood is writing, the effect is to output spaces (or, in bin mood, to write some undefined character) or to compress the current line or page. If the read/write mood is not determined on entry to a layout routine, an error message is given and the program is aborted. On exit, the read/write mood present on entry is restored.

A reading or writing operation, or a call of 'space', 'newline', 'newpage', 'set' or 'set char number', may bring the current position outside the physical or logical file (6.2, 3), but this has no immediate consequence. However, before any further transput is attempted, or a further call of 'space', 'newline' or 'newpage' (but not of 'set' or 'set char number') is made, the current position must be brought to a "good" position. It is ensured that the position is "good" by calling one of the "ensure" routines: 'ensure logical file' (g), 'ensure physical file' (h), 'ensure page' (i) and 'ensure line' (j). On writing, the page is "good" if the (physical) file has not overflowed (6.2, 3); on reading, the page is "good" if the logical file has not overflowed (6.2, 3). The line (character position) is "good" if the page (line) has not overflowed (6.2, 3). The event routine corresponding to 'on physical file end', 'on logical file end', 'on page end' or 'on line end' is therefore called as appropriate. Except in the case of formatted transput (which makes use of the routine 'check pos', see below), the default action, if the event routine returns false, is to give an error message and stop execution of the program in the first two cases, or to call 'newpage' or 'newline', respectively. After this (or if the event routine returns true), the position is tested again (recursively).

The routines 'next pos' (k) and 'check pos' (l) also belong in the category of "ensure" routines. 'Next pos' is very similar to 'ensure line'; instead of returning false if the position can not be ensured, an error message is issued and the program is aborted. It is mainly used in the section on unformatted transput. The routine 'check pos' differs from

'ensure line' in that no default line mending is performed (although, by default, 'newpage' may well be called). It is used during unformatted input of strings or numbers and during formatted transput. An important characteristic of 'check pos' is that as soon as an event routine returns false, no other event routine is called.

Most routines in the transput section obey certain well-defined pre- and post conditions. The various conditions that may hold can be summarized as follows:

- opened: the file has been opened.
- mood ok: the file has been opened and the read/write mood is correct (in general, this means that the read/write mood is as on entry).
- physical file ok: the file has been opened, the read/write mood is correct, and the book has not overflowed (i.e., the page number is within its bounds).
- logical file ok: the file has been opened, the read/write mood is correct, and, on reading, the current position is within the logical file.
- page ok: the file has been opened, the read/write mood is correct, and both the book and the current page have not overflowed.
- line ok: the file has been opened, the read/write mood is correct, and the book, the current page and the current line have not overflowed (i.e., the current position is within the physical file).

{The following inclusions may be observed:

line ok => page ok => physical file ok => mood ok => opened,

and

logical file ok => physical file ok.
}

In terms of the above conditions, the pre- and post-conditions (if the routine does not fail) of the ensure routines are:

	PRE	POST
ensure state	---	mood ok
ensure logical file	mood ok	logical file ok
ensure physical file	mood ok	physical file ok
ensure page	mood ok	page ok
ensure line	mood ok	line ok
next pos	mood ok	line ok
check pos	mood ok	line ok

```
a) PROC space = (REF FILE f) VOID:
     IF STATUS reading = state(f);
       IF status OF f SAYS line ok
       THEN TRUE
       ELSE ensure line(f, reading)
       FI
     THEN  # line ok #
       REF INT c = c OF cpos OF f;
       IF status OF f SAYS not lfe in current line
       THEN c +:= 1; test line end(f)
       ELIF c of lpos OF f > c
       THEN c +:= 1; test line end(f); test logical file end(f)
       ELIF status OF f SAYS bin mood
       THEN (put bin char OF f)(f, SKIP)
       ELSE
         IF NOT (status OF f SAYS char mood)
         THEN set char mood(f)
         FI;
         put char(f, " ")
       FI
     ELSE error("nocharpos"); abort
     FI;

b) PROC backspace = (REF FILE f) VOID:
     IF NOT backspace possible(f)
     THEN error("nobackspace"); abort
     ELSE mind logical pos(f);
       REF INT c = c OF cpos OF f;
       (c > 1 | c -:= 1 | error("wrongbacksp"); abort);
       status OF f ORAB logical pos ok  # logical file ok #
     FI;

c) PROC newline = (REF FILE f) VOID:
     IF STATUS reading = state(f);
       IF status OF f SAYS page ok
       THEN TRUE
       ELSE ensure page(f, reading)
       FI
     THEN  # page ok #
       (do newline OF chan OF f)(f)
     ELSE error("noline"); abort
     FI;

d) PROC newpage = (REF FILE f) VOID:
     IF STATUS reading = state(f);
       IF status OF f SAYS physical file ok
       THEN TRUE
       ELSE ensure physical file(f, reading)
       FI
     THEN  # physical file ok #
       (do newpage OF chan OF f)(f)
     ELSE error("nopage"); abort
     FI;
```

```
e) PROC ? state = (REF FILE f) STATUS:
     IF NOT (status OF f SAYS opened)
     THEN error("notopen"); abort
     ELIF status OF f SUGGESTS NOT read or write mood
     THEN error("nomood"); abort
     ELSE status OF f & mood part
     FI;

f) PROC ? ensure state = (REF FILE f, STATUS reading) VOID:
     IF NOT (status OF f SAYS opened)
     THEN error("notopen"); abort
     ELSE
       IF reading SAYS read mood
       THEN set read mood(f)
       ELSE set write mood(f)
       FI;
       IF reading SAYS char mood
       THEN set char mood(f)
       ELSE set bin mood(f)
       FI
     FI;

g) PROC ? ensure logical file = (REF FILE f, STATUS reading) BOOL:
     BEGIN  # logical file ended #
       BOOL mended = (logical file mended OF f)(f);
       ensure state(f, reading);
       IF status OF f SAYS logical pos ok
       THEN TRUE
       ELIF mended
       THEN ensure logical file(f, reading)
       ELSE FALSE
       FI
     END;

h) PROC ? ensure physical file = (REF FILE f, STATUS reading) BOOL:
     # The mood is correct, the file generally not #
     IF
       IF status OF f SAYS logical pos ok
            OR reading SAYS write mood
       THEN TRUE
       ELSE ensure logical file(f, reading)
       FI
     THEN  # logical file ok #
       IF status OF f SAYS physical file ok
       THEN TRUE
       ELSE  # physical file ended #
         BOOL mended = (physical file mended OF f)(f);
         ensure state(f, reading);
         IF mended
         THEN ensure physical file(f, reading)
         ELSE error("nopage"); abort
         FI
       FI
     ELSE FALSE
     FI;
```

```
i) PROC ? ensure page = (REF FILE f, STATUS reading) BOOL:
     # The mood is ok, the page generally not #
     IF
       IF status OF f SAYS physical file ok
       THEN TRUE
       ELSE ensure physical file(f, reading)
       FI
     THEN # physical file ok #
       IF status OF f SAYS page ok
       THEN TRUE
       ELSE  # page ended #
         BOOL mended = (page mended OF f)(f);
         ensure state(f, reading); (NOT mended | newpage(f));
         ensure page(f, reading)
       FI
     ELSE FALSE
     FI;

j) PROC ? ensure line = (REF FILE f, STATUS reading) BOOL:
     # The mood is ok, the line generally not #
     IF
       IF status OF f SAYS page ok
       THEN TRUE
       ELSE ensure page(f, reading)
       FI
     THEN # page ok #
       IF NOT (status OF f SAYS buffer filled)
       THEN (init buffer OF f)(f); ensure line(f, reading)
       ELIF status OF f SAYS line ok
       THEN TRUE
       ELSE  # line ended #
         BOOL mended = (line mended OF f)(f);
         ensure state(f, reading); (NOT mended | newline(f));
         ensure line(f, reading)
       FI
     ELSE FALSE
     FI;

k) PROC ? next pos = (REF FILE f) VOID:
     IF NOT ensure line(f, status OF f)
     THEN error("nocharpos"); abort
     FI;
```

```
1) PROC ? check pos = (REF FILE f) BOOL:
     # the mood is ok, the line generally not #
     IF STATUS reading = status OF f;
       IF status OF f SAYS page ok
       THEN TRUE
       ELSE ensure page(f, reading)
       FI
     THEN  # page ok #
       IF NOT (status OF f SAYS buffer filled)
       THEN (init buffer OF f)(f); check pos(f)
       ELIF status OF f SAYS line ok
       THEN TRUE
       ELSE  # line ended #
         BOOL mended = (line mended OF f)(f);
         ensure state(f, reading);
         (mended | check pos(f) | FALSE)
       FI
     ELSE FALSE
     FI;

m) PROC set = (REF FILE f, INT p, l, c) VOID:
     IF NOT (status OF f SAYS opened)
     THEN error("notopen"); abort
     ELIF status OF f SAYS not set poss
     THEN error("noset"); abort
     ELSE mind logical pos(f);
       (do set OF chan OF f)(f, p, l, c)
     FI;

n) PROC reset = (REF FILE f) VOID:
     IF NOT reset possible(f)
     THEN error("noreset"); abort
     ELSE mind logical pos(f);
       REF STATUS st = status OF f;
       st ANDAB NOT mood part;
       (NOT (put OF chan OF f)(book OF f) | st ORAB read mood);
       (NOT (get OF chan OF f)(book OF f) | st ORAB write mood);
       (NOT (bin OF chan OF f)(book OF f) | st ORAB char mood);
       (do reset OF chan OF f)(f)
     FI;

o) PROC set char number = (REF FILE f, INT c) VOID:
     IF NOT (status OF f SAYS opened)
     THEN error("notopen"); abort
     ELIF c < 1 OR c > char bound OF f + 1
     THEN error("wrongpos"); abort
     ELSE
       WHILE c OF cpos OF c ≠ c
       DO
         IF c > c OF cpos OF f
         THEN space(f)
         ELSE backspace(f)
         FI
       OD
     FI;
```

8. CONVERSION ROUTINES

8.1. DEVIATIONS

- {D} In the set of conversion routines given in the Revised Report, it is not always clear exactly when 'undefined' is called. It seems to be the intention to call 'undefined' only when it is obvious that no string can be delivered satisfying the constraints set by the parameters. However, when 'x' and 'i' are of the mode REAL and INT, respectively, 'whole(x, 1)' calls 'undefined', while 'whole(i, 1)' does not. Similarly, 'float' may call 'undefined' due to its recursive nature. In the present model, 'undefined' is never called; rather, just 'error characters' are returned.

- {D} No use has been made of the routine 'L standardize'. In general, the number of places where real arithmetic comes in is kept minimal: only the routine 'subfixed', and a few lines in 'string to L real' use real arithmetic.

- {D} The routine 'char dig' does not replace spaces by zeroes, since 'indit string' caters for that.

8.2. NEW DEFINITION

{An earlier version of this Chapter appeared in the ALGOL Bulletin [15].}

From the routines 'whole', 'fixed' and 'float' given below, the following may be observed:

- The routines do not make use of real arithmetic. All real arithmetic is delegated to 'subfixed'. Unless the exponent in 'float' is of the order of magnitude of 'max int', which is very unlikely, the integer arithmetic presents no trouble either;
- The routines do not distinguish between various lengths of numbers; these numbers are just passed down to 'subwhole' and 'subfixed';
- Numbers are first converted to strings of sufficient length, after which the rounding is performed on the strings. This seems to be the only reasonable way to ensure that numbers like 'L max real' may be converted using 'fixed' or 'float';
- The routines are written non-recursively.

The routines 'whole', 'fixed' and 'float' have the following parameters:

- 'v', the value to be converted,
- 'width', whose absolute value specifies the length of the string that is produced,
- 'after', whose value specifies the number of digits desired after the decimal point (for 'fixed' and 'float' only), and
- 'exp', whose absolute value specifies the desired width of the exponent (for 'float' only).

The routine 'whole' is intended to convert integer values. Leading zeroes are replaced by spaces and a sign is normally included. The user may specify that a sign is to be included only for negative values by specifying a negative or zero width. If the width specified is zero, then the shortest possible string is returned. The routine uses 'subwhole' for the actual conversion.

Examples:
'whole(i, -4)' might yield " 0", " 99", " -99", "9999", or, if 'i' were greater than 9999, "****", where "*" is the yield of 'errorchar';
'whole(i, 4)' would yield " +99" rather than " 99";
'whole(i, 0)' might yield "0", "99", "-99", "9999" or "99999".

The routine 'fixed' may be used to convert real values to fixed point form (i.e., without an exponent part). It has an 'after' parameter to specify the number of digits required after the decimal point. From the value of the 'width' and 'after' parameter, the amount of space left in front of the decimal point may be calculated. If the space left in front of the decimal point is not sufficient to contain the integral part of the number being converted, digits after the decimal point are sacrificed. If the number of digits after the decimal point is reduced to zero and the number still does not fit, 'error characters' are returned. The routine uses 'subfixed' for the actual conversion.

Examples:
'fixed(x, -6, 3)' might yield " 2.718", "27.183", "271.83" (one place after the decimal point has been sacrificed in order to fit the number in), "2718.3", " 27183" or "271833" (in the last two examples, all positions after the decimal point are sacrificed);
'fixed(x, 0, 3)' might yield "2.718", "27.183" or "271.828".

The routine 'float' may be used to convert real values into floating point form. It has an 'exp' parameter to specify the width of the exponent. A sign is normally included in both the mantissa and the exponent. Just as in the case of the 'width' parameter, the sign of the 'exp' parameter specifies whether or not a plus-sign is to be included. If the value of the 'exp' parameter is zero, 'float' acts as if minus one were specified, i.e., the exponent is converted to a string of minimal length. The value of the 'width' parameter, however, may not be zero in this case. Note that 'float' always leaves the first position for a sign; if no sign is to be included, a space will be given. From the value of 'width', 'after' and 'exp' it follows how much space is left in front of the decimal point. From this, the value of the exponent follows; this exponent has to fit in a string whose length is bounded by the width specified by the 'exp' parameter. If this is not possible, the digits after the decimal point are sacrificed one by one; if there are no more digits left after the decimal point and the exponent still does not fit, digits in front of the decimal point are sacrificed too. (Note that this has repercussions on the value of the exponent, and thus possibly on the width of the exponent.) During this process, it has to be checked at each step whether all digits have been consumed, in which case 'error characters' are returned. The routine makes use of 'subfixed' for the actual conversion.

Examples:
'float(x, 9, 3, 2)' might yield "-2.718e+0", "+2.72e+11" (one place after the decimal point has been sacrificed in order to make room for the exponent);
'float(x, 6, 1, 0)' might yield "-256e1", "+26e12" or "+1e -9" (in case 'x' has the value 0.996e-9).

The routine 'subfixed' performs the actual conversion from numbers to strings. When called from 'fixed', it must return a string containing all digits from the integral part of the value submitted, and 'after+1' digits

from the fractional part. When called from 'float', it must return a string containing the first 'after+1' significant digits. In both cases, the last digit is truncated, and not rounded. (The rounding is done later on, and rounding the number twice may give wrong results.) Considering this string as a number, the value of the parameter 'p' will be the shift of the decimal point from before the first digit. The parameter 'neg' will indicate the sign of the value submitted (true iff negative).

The routine 'subfixed' must be completely accurate: it will be used to measure the accuracy of numerical algorithms, and these measurements must not be downgraded by the inaccuracy of the conversion. Rather than an ALGOL-68 routine, a semantic definition is given below.

The (hidden) routine 'round' is used for rounding. The parameter 's' refers to the string to be rounded, the parameter 'k' is the index of the last element of 's' that will be returned. The routine yields true if the rounding causes a carry out of the leftmost digit.

a)
```
MODE ? NUMBER = UNION(⦃L REAL⦄, ⦃L INT⦄);
```

b)
```
PROC whole = (NUMBER v, INT width) STRING:
     CASE v IN
       (UNION(⦃L INT⦄) x):
         (BOOL neg; STRING s:= subwhole(x, neg);
          (neg | "-" |: width > 0 | "+" | "") PLUSTO s;
          IF width = 0
          THEN s
          ELIF INT n = ABS width - UPB s; n ≥ 0
          THEN n * " " + s
          ELSE ABS width * errorchar
          FI),
       (UNION(⦃L REAL⦄) x): fixed(x, width, 0)
     ESAC;
```

c)
```
PROC fixed = (NUMBER v, INT width, after) STRING:
     IF after < 0 OR (width ≠ 0 AND after ≥ ABS width)
     THEN (width = 0 | 1 | ABS width) * errorchar
     ELIF INT point, BOOL neg;
       STRING s:= subfixed(v, after, point, neg, FALSE);
       width = 0
     THEN (round(UPB s - 1, s) | point +:= 1);
       (s = "" | "0" PLUSTO s; point:= 1);
       (neg | "-" | "") +
          (point = UPB s | s | s[ : point] + "." + s[point + 1 : ])
     ELIF INT space = ABS width - (neg OR width > 0 | 1 | 0);
       space ≤ after OR point > space
     THEN ABS width * errorchar
     ELIF INT digits = (space = point | space | space - 1);
       IF digits > UPB s - 1
       THEN (round(UPB s - 1, s) | point +:= 1)
       ELSE (round(digits, s) | point +:= 1;
                          (point ≠ space | s:= s[ : digits]))
       FI;
       (point = 0 & digits > UPB s | "0" PLUSTO s; point:= 1);
       point > UPB s
```

```
      THEN ABS width * errorchar
      ELSE s:= (neg | "-" |: width > 0 | "+" | "") +
               (point = UPB s | s | s[ : point] + "." + s[point + 1 : ]);
        (ABS width - UPB s) * " " + s
      FI;

d) PROC float = (NUMBER v, INT width, after, exp) STRING:
      IF INT before:= ABS width - (after ≠ 0 | after + 1 | 0)
                                - (ABS exp + 1) - 1,
               exponent, aft:= after, expspace:= ABS exp;
        SIGN before + SIGN aft ≤ 0
      THEN (width = 0 | 1 | ABS width) * errorchar
      ELIF BOOL neg, rounded:= FALSE, possible:= TRUE;
        STRING s:= subfixed(v, before + after, exponent, neg, TRUE),
               expart;
        (neg OR width > 0 | before -:= 1);
        exponent -:= before;
        WHILE expart:= (exponent < 0 | "-" |: exp > 0 | "+" | "") +
                       subwhole(ABS exponent, LOC BOOL);
          IF SIGN before + SIGN aft ≤ 0
          THEN possible:= FALSE
          ELIF UPB expart > expspace
          THEN expspace +:= 1;
            (aft > 0 | aft -:= 1;
                       (aft = 0 | before +:= 1; exponent -:= 1)
                     | before -:= 1; exponent +:= 1); TRUE
          ELIF rounded THEN FALSE
          ELIF round(before + aft, s)
          THEN exponent +:= 1; rounded:= TRUE
          ELSE FALSE
          FI
        DO SKIP OD;
        NOT possible
      THEN ABS width * errorchar
      ELSE (neg | "-" |: width > 0 | "+" | "") +
           s[: before] +
           (aft = 0 | "" | "." + s[before + 1 : before + aft]) +
           "e" + (expspace - UPB expart) * " " + expart
      FI;

e) PROC ? subwhole = (UNION({L INT}) x, REF BOOL neg) STRING:
      CASE x IN
        {(L INT x):
          BEGIN STRING s:= "", L INT n:= ABS x; neg:= x < L 0;
            WHILE dig char(S(n MOD L 10)) PLUSTO s;
              n OVERAB L 10; n ≠ L 0
            DO SKIP OD;
            s
          END}
      ESAC;
```

```
f) PROC ? subfixed =
       (NUMBER v, INT after, REF INT p, REF BOOL neg, BOOL floating) STRING:
     C A unit which, given values V, AFTER and FLOATING (where AFTER is at
       least zero), yields a value S and makes 'p' and 'neg' refer to values
       P and B, respectively, such that:

       • B is true if V is negative, and false otherwise;
       • it maximizes

                          UPB s
                   M =     ∑       ci*10^(P-i)
                          i=LWB s

         under the following constraints:

         • P is at least zero if FLOATING is false, and S[1] ≠ "0" otherwise
         (i.e., the representation of V in S is "normalized");
         • LWB S = 1;
         • UPB S = P + AFTER + 1 if FLOATING is false, and AFTER + 1
         otherwise;
         • for all i from LWB S to UPB S:
             0 ≤ ci ≤ 9, where ci = 'char dig(S[i])';
         • M ≤ |V|.
     C;
```

{So that one need not know the storage allocation techniques used by a specific implementation (which are needed to build the string), one may construct an embedding like:

```
  PROC ? subfixed =
      (NUMBER v, INT after, REF INT p, REF BOOL neg, BOOL floating) STRING:
    BEGIN INT size; guess storage(v, after, size, floating);
          # size:= some sufficiently large integer, an upper bound for the
            number of digits that will result #
      [1 : size] CHAR s;
      do subfixed(v, after, p, neg, floating, size, s);
          # the actual conversion; the characters are placed in s.  As a
            side effect, size indicates the number of digits placed in s #
      s[ : size]
    END;}

g) PROC ? round = (INT k, REF STRING s) BOOL:
     IF BOOL carry:= char dig(s[k + 1]) ≥ 5; s:= s[: k]; carry
     THEN
       FOR j FROM k BY -1 TO 1 WHILE carry
       DO INT d = char dig(s[j]) + 1; carry:= d = 10;
         s[j]:= (carry | "0" | dig char(d))
       OD;
       (carry | "1" PLUSTO s); carry
     ELSE FALSE
     FI;

h) PROC ? dig char = (INT x) CHAR:
     "0123456789abcdef"[x + 1];
```

Strings are converted to numbers by the routines 'string to L int' and 'string to L real'. These routines are hidden from the user, so it can safely be assumed that the layout of the string supplied is correct. The first element of the string contains the sign of the number. If conversion is successful, the value computed is assigned to the 'i' (or 'r') parameter, and the routine returns true; otherwise the routine returns false. The conversion is unsuccessful if the absolute value of the result is greater than 'L max int' or 'L max real' (for 'string to L int' and 'string to L real', respectively). The version of 'string to L real' that is given below compares the string 's' against 'max', which contains a representation of 'L max real'. It merely is an outline of how things might be done; the routine needs real arithmetic, and therefore has to be rewritten for a particular implementation.

```
i) PROC ? string to L int = (STRING s, INT radix, REF L INT i) BOOL:
        # returns true if the absolute value of the result is ≤ L max int #
     BEGIN
       L INT lr = K radix; BOOL safe:= TRUE;
       L INT n:= L 0, L INT m = L max int OVER lr;
       L INT ml = L max int - m * lr;
       FOR j FROM 2 TO UPB s
       WHILE L INT dig = K char dig(s[j]);
         safe:= n < m OR n = m & dig ≤ ml
       DO n:= n * lr + dig OD;
       IF safe
       THEN i:= (s[1] = "-" | -n | n); TRUE
       ELSE FALSE
       FI
     END;

j) PROC ? string to L real = (STRING s, REF L REAL r) BOOL:
     BEGIN INT e:= UPB s + 1; char in string("e", e, s);
       INT p:= e; char in string(".", p, s); INT expart:= 0;
       BOOL safe:= (e < UPB s | string to int(s[e+1 : ], 10, expart)
                              | TRUE);
       IF safe
       THEN INT j:= 1;
         FOR i FROM j + 1 TO e - 1
         WHILE s[i] = "0" OR s[i] = "."
         DO j:= i OD;
         expart +:= p - 2 - j; # exponent of most significant digit #
         L REAL x:= L 0, INT length:= 0, max exp;
         STRING max = subfixed(L max real, UPB s, max exp, LOC BOOL, TRUE);
            # 'max' contains the first |s| significant digits
              of 'L max real' #
         IF safe:=
            (expart > max exp | FALSE
            |: expart = max exp
            | s[j+1:e-1] > (max[:p-j] + "." + max[p-j+1:e-j])
            | safe)
         THEN
           FOR i FROM j+1 TO e-1 WHILE length < L real width
           DO
             IF s[i] = "." THEN SKIP
             ELSE x +:= K char dig(s[i]) * L 10.0 ** expart; expart -:= 1
             FI
```

```
          OD;
          r:= (s[1] = "-" | -x | x)
        FI
      FI;
      safe
    END;

k) PROC ? char dig = (CHAR x) INT:
     (INT i; char in string(x, i, "0123456789abcdef"); i-1);

l) PROC char in string = (CHAR c, REF INT i, STRING s) BOOL:
     (BOOL found:= FALSE;
      FOR k FROM LWB s TO UPB s WHILE NOT found
      DO (c = s[k] | i:= k; found:= TRUE) OD;
      found);

m) INT L int width =
     # the smallest integral value such that 'L max int' may be converted
       without error using the pattern n(L int width)d #
     (INT c:= 1;
      WHILE L 10 ** (c-1) < L max int OVER L 10 DO c +:= 1 OD;
      c);

n) INT L real width =
     C the smallest integral value such that different values yield
       different strings using the pattern d.n(L real width - 1)d C;

o) INT L exp width =
     C the smallest integral value such that 'L max real' may be converted
       without error using the pattern
       d.n(L real width - 1)d e n(L exp width)d C;
```

9. TRANSPUT MODES AND STRAIGHTENING

9.1. DEVIATIONS

- {S} Rather than giving a description of 'outtype' and 'intype', a formal definition using meta-syntax is given. This results in a different mode for INTYPE. As it is defined in the Revised Report, 'reference to flexible row of character' may only occur immediately after 'reference', and may not be stowed. Thus, values of a mode specified by STRUCT(BOOL b, STRING s), say, cannot be input. This restriction is very much against the way in which stowing is normally handled; also, straightening is well capable of handling this task.

9.2. NEW DEFINITION

a) MODE ? SIMPLOUT = UNION(⦃ L INT ⦄, ⦃ L REAL ⦄, ⦃ L COMPL ⦄, BOOL,
⦃ L BITS ⦄, CHAR, [] CHAR);

b) {Here, uppercase stands for metanotions.}
OUTTYPE:: union of OUTTYPERS mode.
OUTTYPERS:: OUTTYPER; OUTTYPER OUTTYPERS.
OUTTYPER:: PLAIN;
structured with OUTTAGS mode;
ROWS of OUTTYPER.
OUTTAGS:: OUTTYPER field TAG; OUTTYPER field TAG OUTTAGS.

c) MODE ? SIMPLIN = UNION(⦃ REF L INT ⦄, ⦃ REF L REAL ⦄, ⦃ REF L COMPL ⦄,
REF BOOL, ⦃ REF L BITS ⦄, REF CHAR, REF [] CHAR, REF STRING);

d) INTYPE:: union OF INTYPERS mode.
INTYPERS:: INTYPER; INTYPER INTYPERS.
INTYPER:: PLAIN;
flexible row of character;
structured with INTAGS mode;
ROWS of INTYPER.
INTAGS:: INTYPER field TAG; INTYPER field TAG INTAGS.

e) OP ? STRAIGHTOUT = (OUTTYPE x) [] SIMPLOUT:
C the result of "straightening" 'x' C;

f) OP ? STRAIGHTIN = (INTYPE x) [] SIMPLIN:
C the result of "straightening" 'x' C;

{Straightening is defined in the Revised Report in 10.3.2.3.c,d.}

10. FORMATLESS TRANSPUT

10.1. DEVIATIONS

- {S} In the Revised Report, the routines 'put' and 'get' start with a test for the file being opened. This test is only needed for the case where the second parameter is an empty row, as in 'put(f, ())'. In all other cases, the file is tested each time around the loop. In the present model, this test has been omitted; it can now be stated that the test for the file being opened is performed only once for each item to be output except when control is given back to the user in between.

- {S} In the Revised Report, an empty string written at the end of a page cannot be read back. To achieve compatibility between getting and putting strings, 'ensure page' is called explicitly before a row of character is output. Note however that this introduces an incompatibility between putting and getting rows of characters! Therefore, 'ensure page' is also called before a row of characters is input using 'get'. This makes no difference as long as the row of characters is not empty; for an empty row, a good page will be found, however.

- {S} In the present model it is assumed that internal characters can always be converted to external ones. Thus, 'char error mended' is never called by 'put char'. As a consequence, the test for the file being opened and the current position being good can be omitted from 'put char'.

- {D} As the present model does not assume that backspacing is possible on each file, the character that may have been read ahead by 'get' must be restored in a different way. A primitive 'back char' is introduced for this purpose.

- {S} If a complex number is read, no value is assigned to the complex variable if either conversion to a real number fails. In the Revised Report, a value may in that case be assigned to one of the subnames of the complex variable.

10.2. NEW DEFINITION

In formatless transput, the elements of a "data list" are transput, one after the other, via the specified file. Each element of the data list is either a routine of the mode specified by PROC (REF FILE) VOID or a value of the mode specified by OUTTYPE (on output) or INTYPE (on input). On encountering a routine in the data list, that routine is called with the specified file as parameter. Other values in the data list are first straightened (9.2) and the resulting values are then transput via the given file one after the other.

Transput normally takes place at the current position but, if there is no room on the current line, then first, the event routine corresponding to 'on line end' (or, where appropriate, to 'on page end' or 'on physical file end' or 'on logical file end') is called, and next, if this returns false, the next "good" character position of the book is found, viz., the first character position of the next nonempty line.

For formatless output, 'put' (a) and 'print' (or 'write') (section 10.5.1 of the Revised Report) may be used. Each straightened value V from the data list is output as follows:

If the mode of V is specified by L INT, L REAL or L COMPL, output has to fit on one and the same line. Moreover, if output does not take place at the beginning of a line, a space is given first. The length of the string that is output is such that 'L max int' ('L max real', ('L max real', 'L max real')) is output without error if the mode of V is specified by L INT (L REAL, L COMPL). So, if the current position is at the beginning of a line, the length of these strings is:

```
L int width + 1,
L real width + L exp width + 4, or
2 * (L real width + L exp width + 4) + 2,
```

respectively, and one more otherwise. If the length of the string happens to be greater than the length of the current line (i.e., the string would not fit even if the line were empty), an error message is given and the program is aborted. Otherwise, if there is not enough room for a string of this length on the current line, then a good position is found on a subsequent line, and the test is repeated until the number will fit. Then, when not at the beginning of a line, a space is given and V is output as if under the picture

```
n(L int width-1)z + d,
+d.n(L real width-1)d e n(L exp width-1)z + d, or
+d.n(L real width-1)d e n(L exp width-1)z + d " " i
  +d.n(L real width-1)d e n(L exp width-1)z + d,
```

respectively.

If the mode of V is specified by BOOL, then first, if the current line is full, a good position is found on a subsequent line; next, if V is true (false), the character yielded by 'flip' ('flop') is output (with no intervening space).

If the mode of V is specified by L BITS, then the elements of the only field of V are output (as if of the mode specified by BOOL) one after the other (with no intervening spaces, and with new lines being taken as required).

If the mode of V is specified by CHAR, then first, if the current line is full, a good position is found on a subsequent line; next V is output (with no intervening space).

If the mode of V is specified by [] CHAR, then first a good page is found; next the elements of V are output (as above) one after the other (with no intervening spaces, and with new lines being taken as required).

```
a) PROC put = (REF FILE f, [] UNION(OUTTYPE, PROC (REF FILE) VOID) x) VOID:
     FOR i TO UPB x
     DO
       IF NOT (status OF f SAYS put char status)
       THEN ensure state(f, put char status)
       FI;
       CASE x[i] IN
         (PROC (REF FILE) VOID pf): pf(f),
         (OUTTYPE ot):
           BEGIN [] SIMPLOUT y = STRAIGHTOUT ot;
             ¢ PROC L real conv = (L REAL r) STRING:
               float(r, L real width + L exp width + 4,
                        L real width - 1, L exp width + 1) ¢;

             FOR j TO UPB y
             DO
               CASE y[j] IN
                 (UNION(NUMBER, ¢ L COMPL ¢) nc):
                   BEGIN STRING s =
                     CASE nc IN
                       ¢(L INT k): whole(k, L int width + 1)¢,
                       ¢(L REAL r): L real conv(r)¢,
                       ¢(L COMPL z): L real conv(RE z) + " i" +
                                     L real conv(IM z)¢
                     ESAC;
                     INT n = UPB s;
                     WHILE
                       IF NOT (status OF f SAYS line ok)
                       THEN next pos(f)
                       FI;
                       (n > char bound OF f | error("smallline"); abort);
                       c OF cpos OF f +
                          (c OF cpos OF f = 1 | n | n + 1) >
                              char bound OF f + 1
                     DO BOOL mended = (line mended OF f)(f);
                       ensure state(f, put char status);
                       (NOT mended | newline(f))
                     OD;
                     IF c OF cpos OF f ≠ 1
                     THEN (put char OF f)(f, " ")
                     FI;
                     FOR k TO UPB s
                     DO (put char OF f)(f, s[k]) OD;
                     IF status OF f SAYS logical pos not ok
                     THEN set logical pos(f)
                     FI;
                     test line end(f)
                   END # numeric #,
                 (BOOL b):
                   (IF NOT (status OF f SAYS line ok)
                    THEN next pos(f)
                    FI;
                    put char(f, (b | flip | flop))),
```

```
                ¢(L BITS lb):
                  ([] BOOL b = lb;
                   FOR k TO UPB b
                   DO
                     IF NOT (status OF f SAYS line ok)
                     THEN next pos(f)
                     FI;
                     put char(f, (b[k] | flip | flop))
                   OD)¢,
                (CHAR k):
                  (IF NOT (status OF f SAYS line ok)
                   THEN next pos(f)
                   FI;
                   put char(f, k)),
                ([] CHAR ss):
                  (IF NOT (status OF f SAYS page ok)
                   THEN ensure page(f, put char status)
                   FI;
                   FOR k FROM LWB ss TO UPB ss
                   DO
                     IF NOT (status OF f SAYS line ok)
                     THEN next pos(f)
                     FI;
                     put char(f, ss[k])
                   OD)
            ESAC
          OD
        END
    ESAC
  OD;

b) PROC ? put char = (REF FILE f, CHAR char) VOID:
   BEGIN (put char OF f)(f, char);
     IF status OF f SAYS logical pos not ok
     THEN set logical pos(f)
     FI;
     test line end(f)
   END;
```

For formatless input, 'get' (a) and 'read' (section 10.5.1 of the Revised Report) may be used. Values from the book are assigned to each straightened name N from the data list as follows:

If the mode of N is specified by REF L INT, then first, the book is searched for the first character that is not a space (finding good positions on subsequent lines as necessary); next, the largest string is read from the book that could be "indited" (section 10.3.4.1.1.kk of the Revised Report) under the control of some picture of the form +n(k1)" "n(k2)dd or n(k2)dd (where k1 and k2 yield arbitrary nonnegative integers); this string is converted to an integer and assigned to N; if the conversion is unsuccessful, the event routine corresponding to 'on value error' is called.

If the mode of N is specified by REF L REAL, then first, the book is searched for the first character that is not a space (finding good positions on subsequent lines as necessary); next, the largest string is read from the book that could be "indited" (section 10.3.4.1.1.kk of the Revised Report)

under the control of some picture of the form +n(k1)" "n(k2)d or n(k2)d followed by .n(k3)dd or by ds., possibly followed again by en(k4)" "+n(k5)" "n(k6)dd or by en(k5)" "n(k6)dd; this string is converted to a real number and assigned to N; if the conversion is unsuccessful, the event routine corresponding to 'on value error' is called.

If the mode of N is specified by REF L COMPL, then first, a real number is input (as above); next, the book is searched for the first character that is not a space; next, a character is input and, if it is not "|" or "i", then the event routine corresponding to 'on char error' is called, the suggestion being "|"; finally, a second real number is input. Note that if either conversion to a real number is unsuccessful, the event routine corresponding to 'on value error' is called. In that case, no value is assigned to either subname of N. If conversion is successful, the first (second) number read is assigned to the first (second) subname of N.

{Numbers are input using a finite-state machine that largely follows the syntax of numbers as given in sections 8.1.1 and 8.1.2 of the Revised Report. However, spaces within numbers are explicitly allowed after a sign and after a 'times-ten-to-the-power-symbol' (although no good positions are found on a subsequent line).}

If the mode of N is specified by REF BOOL, then first, the book is searched for the first character that is not a space (finding good positions on subsequent lines as necessary); next, a character is read; if this character is the same as that yielded by 'flip' ('flop'), then true (false) is assigned to N; otherwise, the event routine corresponding to 'on char error' is called, the suggestion being 'flop'.

If the mode of N is specified by REF L BITS, then input takes place (as for booleans, see above) to the subnames of N one after the other (with new lines being taken as required).

If the mode of N is specified by REF CHAR, then first, if the current line is exhausted, a good position is found on a subsequent line; next, a character is read and assigned to N.

If the mode of N is specified by REF [] CHAR, then input takes place (as above) to the subnames of N one after the other (with new lines being taken as required).

If the mode of N is specified by REF STRING, then characters are read until either

i) a character is encountered which is contained in the string associated with the file by a call of the routine 'make term', or

ii) the current line is exhausted, whereupon the event routine corresponding to 'on line end' (or, where appropriate, to 'on page end' or 'on logical file end') is called; if the event routine moves the current position to a good position, then input of characters is resumed. Note that, if the page has overflowed, a new page is given by default, but, if the line has overflowed, no default action is taken.

The string consisting of the characters read is assigned to N (note that, if the current line has already been exhausted, or if the current position is at the start of an empty line or outside the logical file, then an empty string is assigned to N).

```
a) PROC get = (REF FILE f, [] UNION(INTYPE, PROC (REF FILE) VOID) x) VOID:
     FOR i TO UPB x
     DO
       IF NOT (status OF f SAYS get char status)
       THEN ensure state(f, get char status)
       FI;
       CASE x[i] IN
         (PROC (REF FILE) VOID pf): pf(f),
         (INTYPE it):
           BEGIN [] SIMPLIN y = STRAIGHTIN it;
             CHAR k, STRING t:= "", BOOL k empty;

             OP ? = (STRING s) BOOL:
               (IF NOT k empty THEN SKIP
                ELIF status OF f SAYS line ok
                THEN get char(f, k); k empty:= FALSE
                FI; # k is filled, if possible #
                IF k empty
                THEN FALSE
                ELSE k empty:= char in string(k, LOC INT, s)
                FI);

             OP ? = (CHAR c) BOOL: ? STRING(c);

             PRIO ! = 8;

             OP ! = (STRING s, CHAR c) CHAR:
               (IF NOT k empty THEN SKIP
                ELIF status OF f SAYS line ok
                THEN get char(f, k)
                ELIF check pos(f) THEN get char(f, k)
                ELSE error("nocharpos"); abort
                FI; # k is filled #
                k empty:= TRUE;
                IF char in string(k, LOC INT, s)
                THEN k
                ELSE CHAR sugg:= c;
                  IF BOOL mended = (char error mended OF f)(f, sugg);
                    ensure state(f, get char status); mended
                  THEN (char in string(sugg, LOC INT, s)
                        | sugg
                        | error("wrongchar"); c)
                  ELSE error("wrongchar"); c
                  FI
                FI);

             PROC skip initial spaces = VOID:
               WHILE
                 IF NOT (status OF f SAYS line ok)
                 THEN next pos(f)
                 FI;
                 get char(f, k); k = " "
               DO SKIP OD;
```

```
  PROC fixed point numeral = VOID:
    (t PLUSAB "0123456789" ! "0";
     WHILE ? "0123456789" DO t PLUSAB k OD);

  PROC fractional part = BOOL:
    IF ? "."
    THEN t PLUSAB "."; fixed point numeral; TRUE
    ELSE FALSE
    FI;

  PROC integer = VOID:
    (t PLUSAB (? "+-" | k | "+");
     WHILE ? " " DO SKIP OD;
     fixed point numeral);

  PROC intreal = VOID:
    (t PLUSAB (? "+-" | k | "+");
     WHILE ? " " DO SKIP OD;
     IF NOT fractional part
     THEN fixed point numeral; fractional part
     FI;
     IF ? "\e"
     THEN WHILE ? " " DO SKIP OD;
       t PLUSAB "e"; integer
     FI);

FOR j TO UPB y
DO k empty:= TRUE;
  CASE y[j] IN
    (UNION( ⦃ REF L INT ⦄, ⦃ REF L ⦃ REF L COMPL ⦄) irc):
      BEGIN skip initial spaces; BOOL ok =
        CASE irc IN
          ⦃(REF L INT ii):
            (integer; string to L int(t, 10, ii))⦄,
          ⦃(REF L REAL rr):
            (intreal; string to L real(t, rr))⦄,
          ⦃(REF L COMPL zz):
            (L COMPL z; intreal;
             BOOL ok:= string to L real(t, re OF z);
             WHILE ? " " DO SKIP OD; "⊥i" ! "i";
             WHILE ? " " DO SKIP OD; t:= "";
             intreal;
             ok:= ok & string to L real(t, im OF z);
             (ok | zz:= z); ok)⦄
        ESAC;
        (NOT k empty | back char(f));
        IF NOT ok
        THEN BOOL mended = (value error mended OF f)(f);
          ensure state(f, get char status);
          (NOT mended | error("wrongval"); abort)
        FI
      END,
    (REF BOOL bb):
      (skip initial spaces;
       bb:= (flip + flop) ! flop = flip),
```

```
        ¢(REF L BITS lb):
          ([1 : L bits width] BOOL b;
           FOR i TO UPB b
           DO skip initial spaces;
             b[i]:= (flip + flop) ! flop = flip
           OD;
           lb:= L bits pack(b))¢,
        (REF CHAR cc):
          (IF NOT (status OF f SAYS line ok)
           THEN next pos(f)
           FI;
           get char(f, cc)),
        (REF [] CHAR ss):
          (IF NOT (status OF f SAYS page ok)
           THEN ensure page(f, get char status)
           FI;
           FOR i FROM LWB ss TO UPB ss
           DO
             IF NOT (status OF f SAYS line ok)
             THEN next pos(f)
             FI;
             get char(f, ss[i])
           OD),
        (REF STRING ss):
          BEGIN STRING t:= "";
            WHILE
              IF
                IF status OF f SAYS line ok
                THEN TRUE
                ELSE check pos(f)
                FI
              THEN get char(f, k);
                IF char in termstring(k, term OF f)
                THEN back char(f); FALSE
                ELSE TRUE
                FI
              ELSE FALSE
              FI
            DO t PLUSAB k OD;
            ss:= t
          END
      ESAC
    OD
  END
ESAC
OD;
```

```
b) PROC ? get char = (REF FILE f, REF CHAR char) VOID:
     char:=
       IF CHAR k; BOOL conv ok = (get char OF f)(f, k);
         test line end(f);
         IF status OF f SUGGESTS lfe in current line
         THEN test logical file end(f)
         FI;
         conv ok
       THEN k
       ELIF CHAR sugg:= " ";
         BOOL mended = (char error mended OF f)(f, sugg);
         ensure state(f, get char status); mended
       THEN sugg
       ELSE error("wrongchar"); " "
       FI;

c) PROC ? back char = (REF FILE f) VOID:
     C If the current position is not at the beginning of a buffer, it is
       set back over one position; otherwise, an error message is given and
       the elaboration of the particular program is aborted.  (This can only
       be caused by a call of the event routine corresponding to 'on char
       error' while reading a string; this call must then have caused the
       current position to be moved to the first position of a new buffer,
       while it returned a character from the terminator string of 'f'.
       This case is assumed to be exceedingly rare.) C;
```

11. FORMATTED TRANSPUT

11.1. DEVIATIONS

- {D} A different structure has been chosen for the mode specified by FORMAT. In the Revised Report, the mode specified by FORMAT almost exactly mirrors the structure of format texts. This mode FORMAT serves two purposes: it contains a description of some format denotation (which is completely static), and it contains some administration on how this format is being used (which is dynamic). These two parts have been separated in the present model. Here, the mode specified by FORMAT only contains a static description of a format denotation; thus, formats need no longer be copied upon assignment. The dynamic information is collected in the 'plist' field of the file to which the format is associated. This change results in remarkable simplifications in the routines 'get next picture', 'do fpattern' and 'associate format'.
  Care has been taken not to change the semantics of 'get next picture', although there are reasons to do so. In the Revised Report, the insertion I following the last picture P of a format-pattern is linked up with the (second) insertion of that picture. Consequently, their replicators are elaborated collaterally. If the elaboration of some replicator from I is terminated abnormally (by a jump), the value intended to be transput using P is not transput, which is likely to surprise the user. For instance,

```
printf(($ $5d, 6d$ n(GOTO label) "a" $, i, j))
```

  will only result in the value of 'i' to be printed, after which a jump to 'label' results. Something similar happens with collections grouped together in a (replicated) collection-list-pack: the insertion following this pack is linked to the second insertion of its last picture the last time the pack is used.

11.2 NEW DEFINITION

The description of collections and pictures, as given in section 10.3.4.1. of the Revised Report remains to a large extent valid. Below only a new set of mode declarations is given, together with a new definition of "transforming", which replaces sections 10.3.4.1.2.a and b.

{The yield of a format-text is that of its collection-list, by way of pre-elaboration.}

The yield N of a collection-list C, in an environ E, is determined as follows:

- N is a newly created name {whose mode is 'FORMAT'};
- N is equal in scope to the environ necessary for C in E;
- N is made to refer to a value V {whose mode is 'row of COLLECTION'}, having a descriptor ((1, m)), where m is the number of constituent collections of C, and elements determined as follows:
  For j = 1, ..., m, letting Cj be the j-th constituent collection of C,
  Case A: The direct descendents of Cj include a picture P:
  - the j-th element of V is a structured value, whose mode is 'PICTURE', and whose fields, taken in order, are the yields of
    - {p} the constituent pattern of P, if any, and, otherwise, a

void-denotation;
- {i} the insertion of P.

Case B: The direct descendents of Cj include a first insertion I1, a replicator REP, a collection-list-pack P and a second insertion I2:
- the j-th element of V is a structured value whose mode is 'COLLITEM' and whose fields, taken in order, are the yields of
  - {i1} I1,
  - {rep} REP,
  - {p} the collection-list of P,
  - {i2} I2.

```
a) MODE FORMAT = REF [] COLLECTION;

b) MODE ? COLLECTION = UNION(PICTURE, COLLITEM);

c) MODE ? COLLITEM =
     STRUCT(INSERTION i1,
            PROC INT rep, # replicator #
            FORMAT p,
            INSERTION i2);

d) MODE ? INSERTION =
     FLEX [1 : 0] STRUCT(PROC INT rep, UNION(STRING, CHAR) sa);

e) MODE ? PICTURE =
     STRUCT(UNION(PATTERN, CPATTERN, FPATTERN, GPATTERN, VOID) p,
            INSERTION i);

f) MODE ? PATTERN =
     STRUCT(INT type, # of pattern #
            FLEX [1 : 0] FRAME frames);

g) MODE ? FRAME =
     STRUCT(INSERTION i,
            PROC INT rep,
            BOOL supp, # true if suppressed #
            CHAR marker);

h) MODE ? CPATTERN =
     STRUCT(INSERTION i,
            INT type, # boolean or integral #
            FLEX [1 : 0] INSERTION c);

i) MODE ? FPATTERN =
     STRUCT(INSERTION i,
            PROC FORMAT pf);

j) MODE ? GPATTERN =
     STRUCT(INSERTION i,
            FLEX [1 : 0] PROC INT spec);
```

{The corresponding metaproduction rules are not given here since they follow trivially from the above mode declarations.}

During formatted transput, values are transput using the current format of the file. This current format, together with its administration, is incorporated in the 'plist' field of the file. The mode of that field is 'reference to FORMATLIST', and it refers to the following information:

- (count) the number of times the current collection list is to be repeated;
- (cp) the number of the collection to be executed next;
- (p) the current collection list;
- (next) a reference to a chain of (embracing) collection lists with which to continue after the current one is finished.

Upon associating a format with a file, a name W is created, which is (initially) made to refer to a value of the mode specified by FORMATLIST, whose fields are:

- (count) 1 (since the collection list comprising the format is to be repeated once);
- (cp) 1;
- (p) is set to the collection list of the given format;
- (next) a nil name.

When, during formatted transput to a file 'f', a collection is encountered, which itself contains a collection list c, then further transput uses the collections of c, and c is repeated r times, where r is the integer returned by the replicator of the collection containing c. In that case, a new name W is created, which is made to refer to a value of the mode specified by FORMATLIST, whose fields are:

- (count) r;
- (cp) 1;
- (p) c;
- (next) the 'plist' field of 'f'.

Subsequently, the 'plist' subname of 'f' is made to refer to W.

Something very similar occurs when a format pattern fp is encountered: A new name W is created, which is made to refer to a value of the mode specified by FORMATLIST, whose fields are:

- (count) 1;
- (cp) 1;
- (p) the collection list returned by the 'pf' field of 'fp';
- (next) the 'plist' field of 'f'.

Subsequently, the 'plist' field of 'f' is made to refer to W.

In all three cases, a special generator is needed: the newly created name must have a scope which is equal to the newest of the scopes of the 'p' and 'next' fields of the value to which it is made to refer. {This scope is that of the file with which the format is associated; it does not change with the other manipulations.}

a)
```
MODE ? FORMATLIST =
    STRUCT(INT count, cp, FORMAT p, REF FORMATLIST next);
```

```
b) PROC ? get next picture = (REF FILE f, REF PICTURE picture) VOID:
     IF plist OF f :=: REF REF FORMATLIST(NIL)
     THEN error("noformat"); abort
     ELSE BOOL picture found:= FALSE, STATUS reading = status OF f;
       WHILE NOT picture found
       DO
         IF cp OF plist OF f = 0      # format ended #
         THEN BOOL mended = (format mended OF f)(f);
           ensure state(f, reading);
           IF NOT mended
           THEN cp OF plist OF f:= count OF plist OF f:= 1
           ELIF cp OF plist OF f = 0
           THEN error("noformat"); abort
           FI
         ELSE REF REF FORMATLIST plist = plist OF f;
           CASE (p OF plist)[cp OF plist] IN
             (COLLITEM cl):
               (REF FORMATLIST pl = plist; plist:= NIL;
                [1 : UPB i1 OF cl] SINSERT si, INT count;
                (staticize insertion(i1 OF cl, si), count:= rep OF cl);
                IF reading SAYS read mood
                THEN get insertion(f, si)
                ELSE put insertion(f, si)
                FI;
                IF REF FORMATLIST(plist) :≠: NIL
                THEN error("wrongformat"); abort
                FI;
                plist:= C a newly created name which is made to refer to the
                          yield of an actual-formatlist-declarer and whose
                          scope is equal to the scope of 'f' C
                           := (count, 0, p OF cl, pl);
                IF count ≤ 0
                THEN picture found:= TRUE; picture:= (EMPTY, ());
                  cp OF plist:= UPB p OF plist
                  # this forces the yielding of a 'void-pattern' whose
                    insertion is the second insertion of the collitem 'cl' #
                FI),
             (PICTURE pict):
               (picture found:= TRUE; picture:= pict)
           ESAC;
           WHILE cp OF plist = UPB p OF plist
           DO
             IF (count OF plist -:= 1) > 0
             THEN cp OF plist:= 0 # repeat this piece #
             ELIF REF FORMATLIST next = next OF plist;
               next :=: REF FORMATLIST(NIL)
             THEN cp OF plist:= -1  # format ended #
             ELSE plist:= next;
               INSERTION extra =
                 CASE (p OF plist)[cp OF plist] IN
                   (COLLITEM cl): i2 OF cl,
                   (PICTURE pict): i OF pict
                 ESAC;
               i OF picture:=
                 (INT m = UPB i OF picture, n = UPB extra;
                  [1 : m+n] STRUCT(PROC INT rep, UNION(STRING, CHAR) sa) c;
```

```
                c[ : m]:= i OF picture; c[m + 1 : ]:= extra; c)
            FI
          OD;
          cp OF plist +:= 1
        FI
      OD
    FI;

c) MODE ? SINSERT =
    STRUCT(INT rep, UNION(STRING, CHAR) sa);

d) PROC ? staticize insertion =
       (INSERTION ins, REF [] SINSERT sins) VOID:
    # calls collaterally all the replicators in 'ins' #
    IF UPB ins = 1
    THEN rep OF sins[1]:= rep OF ins[1];
      sa OF sins[1]:= sa OF ins[1]
    ELIF UPB ins > 1
    THEN (staticize insertion(ins[1], sins[1]),
            staticize insertion(ins[2 : ], sins[2 : ]))
    FI;

e) MODE ? SFRAME =
    STRUCT(FLEX [1 : 0] SINSERT si, INT rep, BOOL supp, CHAR marker);

f) PROC ? staticize frames =
       ([] FRAME frames, REF [] SFRAME sframes) VOID:
    # calls collaterally all the replicators in 'frames' #
    IF UPB frames = 1
    THEN [1 : UPB i OF frames[1]] SINSERT si;
      (staticize insertion(i OF frames[1], si),
         rep OF sframes[1]:= rep OF frames[1]);
      si OF sframes[1]:= si;
      supp OF sframes[1]:= supp OF frames[1];
      marker OF sframes[1]:= marker OF frames[1]
    ELIF UPB frames > 1
    THEN (staticize frames(frames[1], sframes[1]),
           staticize frames(frames[2 : ], sframes[2 : ]))
    FI;

g) PROC ? put insertion = (REF FILE f, [] SINSERT si) VOID:
    BEGIN ensure state(f, put char status);
      FOR k TO UPB si
      DO
        CASE sa OF si[k] IN
          (CHAR a): alignment(f, rep OF si[k], a),
          (STRING s):
            TO rep OF si[k]
            DO
              FOR i TO UPB s
              DO (check pos(f) | put char(f, s[i])
                              | error("nocharpos"); abort)
              OD
            OD
        ESAC
      OD
```

```
    END;

h) PROC ? get insertion = (REF FILE f, [] SINSERT si) VOID:
    BEGIN ensure state(f, get char status);
      FOR k TO UPB si
      DO
        CASE sa OF si[k] IN
          (CHAR a): alignment(f, rep OF si[k], a),
          (STRING s):
            (CHAR c;
             TO rep OF si[k]
             DO
               FOR i TO UPB s
               DO (check pos(f) | get char(f, c)
                                | error("nocharpos"); abort);
                 IF c ≠ s[i]
                 THEN BOOL mended = (char error mended OF f)(f, c:= s[i]);
                   ensure state(f, get char status);
                   (NOT mended | error("wrongchar"); abort)
                 FI
               OD
             OD)
        ESAC
      OD
    END;

i) PROC ? alignment = (REF FILE f, INT r, CHAR a) VOID:
    IF a = "x" THEN TO r DO space(f) OD
    ELIF a = "y" THEN TO r DO backspace(f) OD
    ELIF a = "l" THEN TO r DO newline(f) OD
    ELIF a = "p" THEN TO r DO newpage(f) OD
    ELIF a = "k" THEN set char number(f, r)
    ELIF a = "q"
    THEN BOOL read = status OF f SAYS read mood;
      TO r
      DO
        IF read
        THEN CHAR c;
          (check pos(f) | get char(f, c) | error("nocharpos"); abort);
          IF c ≠ blank
          THEN BOOL mended = (char error mended OF f)(f, c:= blank);
            ensure state(f, get char status);
            (NOT mended | error("wrongchar"); abort)
          FI
        ELSE (check pos(f) | put char(f, blank)
                           | error("nocharpos"); abort)
        FI
      OD
    FI;
```

```
j) PROC ? do fpattern = (REF FILE f, FPATTERN fpattern) VOID:
     BEGIN FORMAT pf, [1 : UPB i OF fpattern] SINSERT si;
       STATUS reading = status OF f;
       (staticize insertion(i OF fpattern, si), pf:= pf OF fpattern);
       IF reading SAYS read mood
       THEN get insertion(f, si)
       ELSE put insertion(f, si)
       FI;
       REF REF FORMATLIST(plist OF f):= C a newly created name which is made
          to refer to the yield of an actual-formatlist-declarer and whose
          scope is equal to the scope of 'f' C
                       := (1, 1, pf, plist OF f)
     END;

k) PROC ? associate format = (REF FILE f, FORMAT format) VOID:
     plist OF f:= C a newly created name which is made to refer to the yield
                    of an actual-reference-to-formatlist-declarer and whose
                    scope is equal to the scope of 'f' C
                := C a newly created name which is made to refer to the yield
                    of an actual-formatlist-declarer and whose scope is
                    equal to the scope of 'f' C
                := (1, 1, format, NIL);

1) PROC putf = (REF FILE f, [] UNION(OUTTYPE, FORMAT) x) VOID:
     FOR k TO UPB x
     DO
       IF NOT (status OF f SAYS put char status)
       THEN ensure state(f, put char status)
       FI;
       CASE x[k] IN
         (FORMAT format): associate format(f, format),
         (OUTTYPE ot):
           BEGIN INT j:= 0, PICTURE picture, [] SIMPLOUT y = STRAIGHTOUT ot;
             WHILE (j +:= 1) ≤ UPB y
             DO BOOL incomp:= FALSE;
               get next picture(f, picture);
               [1 : (p OF picture | (FPATTERN): 0
                                  | UPB i OF picture)] SINSERT sinsert;

               CASE p OF picture IN

                 (PATTERN pattern):
                   BEGIN INT rep, sfp:= 1;
                     [1 : UPB frames OF pattern] SFRAME sframes;
                     (staticize frames(frames OF pattern, sframes),
                        staticize insertion(i OF picture, sinsert));
                     STRING s:= "";

                     OP ? = (STRING s) BOOL:
                         # true if the next marker is one of the elements
                           of 's', and false otherwise #
                       IF sfp > UPB sframes
                       THEN FALSE
                       ELSE SFRAME sf = sframes[sfp];
                         IF char in string(marker OF sf, LOC INT, s)
                         THEN rep:= rep OF sf; sfp +:= 1; TRUE
```

```
    ELSE FALSE
    FI
  FI;

 OP ? = (CHAR c) BOOL: ? STRING(c);

PROC int pattern = (REF BOOL sign mould) INT:
  (INT l:= 0;
   WHILE ? "uv" DO (rep ≥ 0 | l +:= rep) OD;
   sign mould:= ? "+-";
   WHILE ? "zd" DO (rep ≥ 0 | l +:= rep) OD; l);

¢PROC edit L int = (L INT i) STRING:
  (BOOL sign mould, neg;
   INT l = int pattern(sign mould);
   STRING t = subwhole(i, neg);
   IF l < UPB t OR neg & NOT sign mould
   THEN incomp:= TRUE; SKIP
   ELSE (sign mould | (neg | "-" | "+") | "") +
            (l - UPB t) * "0" + t
   FI)¢;

¢PROC edit L real = (L REAL r) STRING:
  (BOOL sign, neg;
   STRING t:= "", point:= "", expart:= "";
   INT b = int pattern(sign);
   INT a = (? "."
           | point:= "."; int pattern(LOC BOOL)
           | 0);
   IF ? "e"
   THEN INT exp;
     t:= subfixed(r, a + b, exp, neg, TRUE);
     IF round(UPB t - 1, t)
     THEN t:= t[ : UPB t - 1]; exp +:= 1
     FI;
     exp -:= b; expart:= "e" + edit int(exp)
   ELSE INT bb; t:= subfixed(r, a, bb, neg, FALSE);
     IF round(UPB t - 1, t) THEN bb +:= 1 FI;
     IF bb > b
     THEN incomp:= TRUE
     ELSE (b - bb) * "0" PLUSTO t
     FI
   FI;
   IF NOT (incomp:= incomp OR a + b = 0
                          OR neg & NOT sign)
   THEN (sign | (neg | "-" | "+") | "") +
             t[:b] + point + t[b+1:] + expart
   FI)¢;

¢PROC edit L compl = (L COMPL z) STRING:
  edit L real(RE z) + (sfp +:= 1; "⊥") +
    edit L real(IM z)¢;
```

```
    ¢PROC edit L bits = (L BITS lb, INT radix) STRING:
      (sfp +:= 1;  # skip r-frame #
       L INT n:= ABS lb; STRING t:= "";
       WHILE dig char(S(n MOD K radix)) PLUSTO t;
         n OVERAB K radix; n ≠ L 0
       DO SKIP OD;
       IF INT l = int pattern(LOC BOOL);
         UPB t > l
       THEN incomp:= TRUE; SKIP
       ELSE (l - UPB t) * "0" + t
       FI)¢;

    PROC charcount = INT:
      (INT l:= 0;
       WHILE ? "a" DO (rep ≥ 0 | l +:= rep) OD; l);

    CASE type OF pattern IN

# integral #
      (y[j]
      | ¢(L INT i): s:= edit L int(i)¢
      | incomp:= TRUE),

# real #
      (y[j]
      | ¢(L REAL r): s:= edit L real(r)¢,
        ¢(L INT i): s:= edit L real(i)¢
      | incomp:= TRUE),

# boolean #
      (y[j]
      | (BOOL b): s:= (b | flip | flop)
      | incomp:= TRUE),

# complex #
      (y[j]
      | ¢(L COMPL z): s:= edit L compl(z)¢,
        ¢(L REAL r): s:= edit L compl(r)¢,
        ¢(L INT i): s:= edit L compl(i)¢
      | incomp:= TRUE),

# string #
      (y[j]
      | (CHAR c):
          (charcount = l | s:= c | incomp:= TRUE),
        ([] CHAR t):
          IF charcount = UPB t - LWB t + l
          THEN s:= t[@ l]
          ELSE incomp:= TRUE
          FI
      | incomp:= TRUE)

    OUT
```

```
    # bits #
        (y[j]
        | ¢(L BITS lb):
            s:= edit L bits(lb, type OF pattern - 4)¢
        | incomp:= TRUE)

      ESAC;
      IF NOT incomp THEN edit string(f, s, sframes) FI
    END,

  (CPATTERN choice):
    BEGIN [1 : UPB i OF choice] SINSERT si;
      staticize insertion(i OF choice, si);
      put insertion(f, si);
      INT l = CASE type OF choice IN
          # boolean #
             (y[j] | (BOOL b): (b | 1 | 2)
                   | incomp:= TRUE; SKIP),
          # integral #
             (y[j] | (INT i): i
                   | incomp:= TRUE; SKIP)
              ESAC;
      IF NOT (incomp:= incomp OR l ≤ 0
                              OR l > UPB c OF choice)
      THEN [1 : UPB (c OF choice)[l]] SINSERT ci;
        staticize insertion((c OF choice)[l], ci);
        put insertion(f, ci)
      FI;
      staticize insertion(i OF picture, sinsert)
    END,

  (FPATTERN fpattern): do fpattern(f, fpattern),

  (GPATTERN gpattern):
    BEGIN [1 : UPB i OF gpattern] SINSERT si;
      [] PROC INT spec = spec OF gpattern;
      INT n = UPB spec; [1 : n] INT s;
      (staticize insertion(i OF gpattern, si),
         staticize insertion(i OF picture, sinsert),
         s:= (n |  spec[1],
                  (spec[1], spec[2]),
                  (spec[1], spec[2], spec[3])
                | ()));
      put insertion(f, si);
      IF n = 0 THEN put(f, y[j])
      ELSE
        NUMBER yj = (y[j]
                     | ¢(L INT i): i¢,
                       ¢(L REAL r): r¢
                     | incomp:= TRUE; SKIP);
```

```
                    IF NOT incomp
                    THEN
                      CASE n IN
                        put(f, whole(yj, s[1])),
                        put(f, fixed(yj, s[1], s[2])),
                        put(f, float(yj, s[1], s[2], s[3]))
                      ESAC
                    FI
                  FI
                END,

              (VOID):
                (j -:= 1; staticize insertion(i OF picture, sinsert))

            ESAC;
            IF incomp
            THEN ensure state(f, put char status);
              BOOL mended = (value error mended OF f)(f);
              ensure state(f, put char status);
              (NOT mended | put(f, y[j]); error("wrongval"); abort)
            FI;
            put insertion(f, sinsert)
          OD
        END
    ESAC
  OD;

m) PROC ? edit string = (REF FILE f, STRING s, [] SFRAME sf) VOID:
    BEGIN BOOL supp, zs:= TRUE, signput:= FALSE, INT j;
      INT sign:= (  s[1] = "+" | j:= 1; 1
                 |: s[1] = "-" | j:= 1; 2
                               | j:= 0; SKIP);

      PROC copy = (CHAR c) VOID:
        (NOT supp | (check pos(f) | put char(f, c)
                                  | error("nocharpos"); abort));

      FOR k TO UPB sf
      DO SFRAME sfk = sf[k]; CHAR marker = marker OF sfk;
        supp:= supp OF sfk; put insertion(f, si OF sfk);
        TO rep OF sfk
        DO j +:= 1; CHAR sj = s[j];
          IF marker = "d"
              THEN copy(sj); zs:= TRUE
          ELIF marker = "z"
              THEN (sj = "0" | copy((zs | " " | sj))
                             | zs:= FALSE; copy(sj))
          ELIF marker = "u" OR marker = "v"
              THEN (sj = "0" | copy((zs | " " | sj))
                             | (NOT signput
                               | copy((sign | (marker = "u" | "+" | " "),
                                              "-"));
                                 signput:= TRUE);
                               copy(sj); zs:= FALSE)
```

```
          ELIF marker = "+" OR marker = "-"
              THEN (NOT signput | copy((sign | (marker = "+" | "+" | " "),
                                                "-")));
                   j -:= 1
          ELIF marker = "."
              THEN copy(".")
          ELIF marker = "e" OR marker = "i"
              THEN copy(sj); zs:= TRUE; signput:= FALSE;
                   sign:= (  s[j+1] = "+" | j +:= 1; 1
                          |: s[j+1] = "-" | j +:= 1; 2
                                         | SKIP)
          ELIF marker = "a" OR marker = "b"
              THEN copy(sj)
          ELIF marker = "r"
              THEN j -:= 1
          FI
        OD
      OD
    END;

n) PROC getf = (REF FILE f, [] UNION(INTYPE, FORMAT) x) VOID:
     FOR k TO UPB x
     DO
       IF NOT (status OF f SAYS get char status)
       THEN ensure state(f, get char status)
       FI;
       CASE x[k] IN
         (FORMAT format): associate format(f, format),
         (INTYPE it):
           BEGIN INT j:= 0;
             PICTURE picture, [] SIMPLIN y = STRAIGHTIN it;
             WHILE (j +:= 1) ≤ UPB y
             DO BOOL incomp:= FALSE;
               get next picture(f, picture);
               [1 : (p OF picture | (FPATTERN): 0
                                  | UPB i OF picture)] SINSERT sinsert;

               CASE p OF picture IN

                 (PATTERN pattern):
                   BEGIN [1 : UPB frames OF pattern] SFRAME sframes;
                     (staticize frames(frames OF pattern, sframes),
                       staticize insertion(i OF picture, sinsert));
                      STRING s:= "";
                      INT radix = (type OF pattern ≥ 6 | type OF pattern - 4
                                                       | 10);
                      indit string(f, s, sframes, radix);

                      CASE type OF pattern IN

                  # integral #
                        (y[j]
                        | ¢(REF L INT ii):
                             incomp:= NOT string to L int(s, 10, ii)¢
                        | incomp:= TRUE),
```

```
# real #
      (y[j]
      | ¢(REF L REAL rr):
          incomp:= NOT string to L real(s, rr)¢
      | incomp:= TRUE),

# boolean #
      (y[j]
      | (REF BOOL bb): bb:= s = flip
      | incomp:= TRUE),

# complex #
      (y[j]
      | ¢(REF L COMPL zz):
          (INT i, BOOL ok, L COMPL z;
           char in string("I", i, s);
           ok:= string to L real(s[: i-1], re OF z);
           ok:= ok OR string to L real[i+1 :], im OF z);
           (ok | zz:= z); incomp:= NOT ok)¢
      | incomp:= TRUE),

# string #
      (y[j]
      | (REF CHAR cc):
           (UPB s = 1 | cc:= s[1] | incomp:= TRUE),
        (REF [] CHAR ss):
           (UPB ss - LWB ss + 1 = UPB s | ss[@ 1]:= s
           | incomp:= TRUE),
        (REF STRING ss): ss:= s
      | incomp:= TRUE)

    OUT

# bits #
      (y[j]
      | ¢(REF L BITS lb):
           IF L INT i; string to L int(s, radix, i)
           THEN lb:= BIN i
           ELSE incomp:= TRUE
           FI
      | incomp:= TRUE)
    ESAC
  END,

(CPATTERN choice):
  BEGIN [1 : UPB i OF choice] SINSERT si;
    staticize insertion(i OF choice, si);
    get insertion(f, si);
    INT c = c OF cpos OF f, CHAR kk;
    INT k:= 0, BOOL found:= FALSE;
    WHILE k < UPB c OF choice & NOT found
    DO k +:= 1;
      [1 : UPB (c OF choice)[k]] SINSERT si;
      staticize insertion((c OF choice)[k], si);
      BOOL bool:= TRUE;
      ensure state(f, get char status);
```

```
                    STRING s:= "";
                    FOR i TO UPB si
                    DO s PLUSAB (sa OF si[i] | (STRING ss): ss) *
                                        rep OF si[i]
                    OD;
                    FOR jj TO UPB s
                    WHILE bool:= bool & status OF f SAYS line ok
                    DO get char(f, kk); bool:= kk = s[jj] OD;
                    (NOT (found:= bool) | set char number(f, c))
                  OD;
                  IF NOT found THEN incomp:= TRUE
                  ELSE
                    CASE type OF choice IN
                  # boolean #
                      (y[j]
                        | (REF BOOL b): b:= k = 1
                        | incomp:= TRUE),
                  # integral #
                      (y[j]
                        | (REF INT i) i:= k
                        | incomp:= TRUE)
                    ESAC
                  FI;
                  staticize insertion(i OF picture, sinsert)
                END,

              (FPATTERN fpattern): do fpattern(f, fpattern),

              (GPATTERN gpattern):
                BEGIN [1 : UPB i OF gpattern] SINSERT si;
                  (staticize insertion(i OF gpattern, si),
                     staticize insertion(i OF picture, sinsert));
                  get insertion(f, si);
                  get(f, y[j])
                END,

              (VOID):
                (j -:= 1; staticize insertion(i OF picture, sinsert))

            ESAC;
            IF incomp
            THEN ensure state(f, get char status);
              BOOL mended = (value error mended OF f)(f);
              ensure state(f, get char status);
              (NOT mended | error("wrongval"); abort)
            FI;
            get insertion(f, sinsert)
          OD
        END
    ESAC
OD;
```

```
o) PROC ? indit string =
       (REF FILE f, REF STRING s, [] SFRAME sf, INT radix) VOID:
     BEGIN BOOL zs:= TRUE, sign found:= FALSE, space found:= FALSE,
           INT rep, sp:= 1;

       PRIO ! = 8;

       OP ! = (STRING s, CHAR c) CHAR:
         # expects a character contained in 's'; if the character read is
           not in 's', then the event routine corresponding to 'on char
           error' is called with the suggestion 'c' #
         IF CHAR k;
           (check pos(f) | get char(f, k) | error("nocharpos"); abort);
           char in string(k, LOC INT, s)
         THEN k
         ELSE CHAR sugg:= c;
           BOOL mended = (char error mended OF f)(f, sugg);
           ensure state(f, get char status);
           (mended | (char in string(sugg, LOC INT, s)
                        | sugg
                        | error("wrongchar"); c)
                   | error("wrongchar"); c)
         FI;

       OP ! = (CHAR s, c) CHAR: STRING(s) ! c;

       [] CHAR good digits = "0123456789abcdef"[: radix];
       s:= (char in string(marker OF sf[1], LOC INT, "ab") | "" | "+");
       FOR k TO UPB sf
       DO SFRAME sfk = sf[k], BOOL supp = supp OF sfk;
         CHAR marker = marker OF sfk;
         get insertion(f, si OF sfk);
         TO rep OF sfk
         DO
           IF marker = "d"
               THEN s PLUSAB (supp | "0" | good digits ! "0");
                    zs:= TRUE
           ELIF marker = "z"
               THEN s PLUSAB (supp | "0"
                                   | CHAR c = ((zs | " " | "") +
                                                        good digits) ! "0";
                                     (c ≠ " " | zs:= FALSE; c | "0"))
           ELIF marker = "u" OR marker = "+"
               THEN IF sign found
                    THEN zs:= FALSE; s PLUSAB good digits ! "0"
                    ELSE CHAR c = ("+-" + (marker = "u" | " " | "")) ! "+";
                      IF c = "+" OR c = "-"
                      THEN sign found:= TRUE; s[sp]:= c
                      FI
                    FI
```

```
        ELIF marker = "v" OR marker = "-"
            THEN IF sign found
                 THEN zs:= FALSE; s PLUSAB good digits ! "0"
                 ELSE
                   CHAR c =
                      ("+- " + (space found | good digits | "")) ! "+";
                   IF c = "+" OR c = "-"
                   THEN sign found:= TRUE; s[sp]:= c
                   ELIF c = " "
                   THEN space found:= TRUE
                   ELSE zs:= FALSE; sign found:= TRUE; s PLUSAB c
                   FI
                 FI
        ELIF marker = "."
            THEN s PLUSAB (supp | "." | "." ! ".")
        ELIF marker = "e"
            THEN s PLUSAB (supp | "e" | "\e" ! "e"; "e");
                 sign found:= space found:= FALSE; zs:= TRUE;
                 s PLUSAB "+"; sp:= UPB s
        ELIF marker = "i"
            THEN s PLUSAB (supp | "⊥" | "i⊥" ! "⊥"; "⊥");
                 sign found:= space found:= FALSE; zs:= TRUE;
                 s PLUSAB "+"; sp:= UPB s
        ELIF marker = "b"
            THEN s PLUSAB (flip + flop) ! flop
        ELIF marker = "a"
            THEN s PLUSAB
                   (supp | " "
                         | CHAR c;
                           (check pos(f) | get char(f, c)
                                         | error("nocharpos"); abort);
                           c)
        ELIF marker = "r"
            THEN SKIP
        FI
      OD
    OD
END;
```

12. BINARY TRANSPUT

12.1. DEVIATIONS

- {D} Rather than assuming that binary transput goes via elementary values of the mode CHAR, a special mode BINCHAR is used as a primitive in this model.

- {D} In the Revised Report, the number of characters to be input to a name N is determined as follows:
  - Let 'yj' be the value referred to by N;
  - The number of characters that is input is equal to the number of characters output by 'put bin(f, yj)' {i.e., 'UPB to bin(f, yj)'}.

  It is anticipated that in an actual implementation there will be smarter ways to determine that number. Therefore, a separate routine 'bin length' (d) has been introduced.

12.2. NEW DEFINITION

In binary transput, the values obtained by straightening the elements of a data list are transput, via the specified file, one after the other. The manner in which such a value is stored is defined only to the extent that a value of mode M (being some mode from which that specified by SIMPLOUT is united) output at a given position may subsequently be re-input from that same position to a name of mode 'REFERENCE TO M'. Note that, during input to the name referring to a multiple value, the number of elements read will be the existing number of elements referred to by that name.

The current position is advanced after each value by a suitable amount and, at the end of each line or page, the appropriate event routine is called, and next, if this returns false, the next good character position is found.

For binary output, 'put bin' (e) and 'write bin' (section 10.5.1 of the Revised Report) may be used and, for binary input, 'get bin' (f) and 'read bin' (section 10.5.1 of the Revised Report).

```
a) MODE ? BINCHAR =
     C The elementary mode of binary transput; each value to be transput is
       so via some 'row of BINCHAR', the length of the row being determined
       by the file on which the transput takes place, the mode of the value
       to be transput (and its length in case of a multiple value). C;

b) PROC ? to bin = (REF FILE f, SIMPLOUT x) [] BINCHAR:
     C The lower bound of the resulting multiple value is 1, the upper bound
       depends on 'f' and on the mode and the value of 'x'; furthermore,
       x = from bin(f, x, to bin(f, x)). C;

c) PROC ? from bin = (REF FILE f, SIMPLOUT y, [] BINCHAR c) SIMPLOUT:
     C A value, if one exists, of the mode of the value yielded by 'y', such
       that c = to bin(f, from bin(f, y, c)). If such a value does not
       exist, an error message 'wrongbin' is given and the program is
       aborted. C;
```

```
d) PROC ? bin length = (REF FILE f, SIMPLIN y) INT:
     C The upper bound of the multiple value which is needed to input a
       value into 'y'. C;
     # the following ALGOL-68 unit will do:

       (SIMPLOUT yj =
        CASE y IN
          ⦃(REF L INT i): i⦄,
          ⦃(REF L REAL r): r⦄,
          ⦃(REF L COMPL z): z⦄,
          (REF BOOL b): b,
          ⦃(REF L BITS lb): lb⦄,
          (REF CHAR c): c,
          (REF [] CHAR s): s,
          (REF STRING ss): ss
        ESAC;
        UPB to bin(f, yj)) #

e) PROC put bin = (REF FILE f, [] OUTTYPE ot) VOID:
     FOR k TO UPB ot
     DO
       IF NOT (status OF f SAYS put bin status)
       THEN ensure state(f, put bin status)
       FI;
       [] SIMPLOUT y = STRAIGHTOUT ot[k];
       FOR j TO UPB y
       DO [] BINCHAR bin = to bin(f, y[j]);
         FOR i TO UPB bin
         DO next pos(f); (put bin char OF f)(f, bin[i]);
           IF status OF f SAYS logical pos not ok
           THEN set logical pos(f)
           FI;
           test line end(f)
         OD
       OD
     OD;

f) PROC get bin = (REF FILE f, [] INTYPE it) VOID:
     FOR k TO UPB it
     DO
       IF NOT (status OF f SAYS get bin status)
       THEN ensure state(f, get bin status)
       FI;
       [] SIMPLIN y = STRAIGHTIN it[k];
       FOR j TO UPB y
       DO [1 : bin length(f, y[j])] BINCHAR bin;
         FOR i TO UPB bin
         DO next pos(f); (get bin char OF f)(f, bin[i]);
           test line end(f);
           IF status OF f SUGGESTS lfe in current line
           THEN test logical file end(f)
           FI
         OD;
```

```
    CASE y[j] IN
      ¢(REF L INT ii):
         (from bin(f, ii, bin) | (L INT i): ii:= i)¢,
      ¢(REF L REAL rr):
         (from bin(f, rr, bin) | (L REAL r): rr:= r)¢,
      ¢(REF L COMPL zz):
         (from bin(f, zz, bin) | (L COMPL z): zz:= z)¢,
      (REF BOOL bb):
         (from bin(f, bb, bin) | (BOOL b): bb:= b),
      ¢(REF L BITS lb):
         (from bin(f, lb, bin) | (L BITS b): lb:= b)¢,
      (REF CHAR cc):
         (from bin(f, cc, bin) | (CHAR c): cc:= c),
      (REF [] CHAR ss):
         (from bin(f, ss, bin) | ([] CHAR s): ss:= s),
      (REF STRING ss):
         (from bin(f, ss, bin) | (STRING s): ss:= s)
    ESAC
  OD
OD;
```

REFERENCES

[1] WIJNGAARDEN, A. VAN, et al (eds.), Revised Report on the Algorithmic Language ALGOL 68, Acta Informatica 5 (1975), pp 1-236.

[2] WOODWARD, P.M. & S.G. BOND, ALGOL 68-R Users Guide, Royal Radar Establishment, Malvern, England, 1975.

[3] ALGOL 68 Version I Reference Manual, Control Data Services B.V., Rijswijk, The Netherlands, 1976.

[4] HILL, U., H. SCHEIDIG & H. WOESSNER, An ALGOL 68 Compiler, Technische Universität München and University of British Columbia, 1972.

[5] ROBERTSON, A. & G.E. HEDRICK, A Portable Compiler for an ALGOL 68 Subset, in G.E. Hedrick (ed.), Proceedings of the 1975 International Conference on ALGOL 68, Oklahoma State University, Stillwater, Oklahoma, June 10-12, 1975, pp 59-64.

[6] BERRY, R.D., A practical Implementation of Formatted Transput in ALGOL 68, MS Thesis, Oklahoma State University, Stillwater, Oklahoma, July 1973.

[7] LEROY, A., et al, On the Adequacy of the ALGOL 68 Environment Compared with an Existing Current Operating System and Problems of I/O Implementation, in G.E. Hedrick (ed.), Proceedings of the 1975 International Conference on ALGOL 68, Oklahoma State University, Stillwater, Oklahoma, June 10-12, 1975, pp 202-220.

[8] BROUGHTON, C.G. & C.M. THOMSON, Aspects of Implementing an ALGOL 68 Student Compiler, in G.E. Hedrick (ed.), Proceedings of the 1975 International Conference on ALGOL 68, Oklahoma State University, Stillwater, Oklahoma, June 10-12, 1975, pp 23-38.

[9] THOMSON, C.M., A description of the FLACC Transput System, preliminary version, Edmonton, Canada, August 1977.

[10] THOMSON, C.M. & C.G. BROUGHTON, A description of a New Transput System, preliminary version, Edmonton, Canada, August 1977.

[11] CMU ALGOL-68 Transput, Users's Guide, Carnegie Mellon University.

[12] ALGOL68S implementation -- file system, Carnegie Mellon University.

[13] FISKER, R., private communication.

[14] VLIET, J.C. VAN, Compilation of problems and errors in section 10.3 of the Revised Report, Mathematical Centre, June 1977.

[15] VLIET, J.C. VAN, On the ALGOL 68 Transput Conversion Routines, ALGOL Bulletin 41, July 1977, pp 10-24.

Alphabetic listing of all defining occurrences of mode-indications and identifiers. If a mode-indication or identifier is prefixed with an *, this means that it is not (completely) defined in ALGOL 68.