ALGOL 68 TRANSPUT

ALGOL 68 TRANSPUT

ACADEMISCH PROEFSCHRIFT

TER VERKRIJGING VAN DE GRAAD VAN
DOCTOR IN DE WISKUNDE EN NATUURWETENSCHAPPEN
AAN DE UNIVERSITEIT VAN AMSTERDAM,
OP GEZAG VAN DE RECTOR MAGNIFICUS,
PROF.DR. J. BRUYN,
HOOGLERAAR IN DE FACULTEIT
DER LETTEREN,
IN HET OPENBAAR TE VERDEDIGEN
IN DE AULA DER UNIVERSITEIT
(TIJDELIJK IN DE LUTHERSE KERK, INGANG SINGEL 411, HOEK SPUI)
OP WOENSDAG 3 OKTOBER 1979 DES NAMIDDAGS TE 4.00 UUR

DOOR

JOHANNES CORNELIS VAN VLIET

GEBOREN TE MIJDRECHT

1979
MATHEMATISCH CENTRUM, AMSTERDAM

PROMOTOR : PROF.DR.IR. A. VAN WIJNGAARDEN

COREFERENT : PROF.DR. J.W. DE BAKKER

Aan mijn vader, die te vroeg stierf
Aan moeder en Marjan

SUMMARY

This treatise contains a description of the input-output ("transput") facilities of the programming language ALGOL 68. The emphasis is on implementability. Although the official report on ALGOL 68 already contains an extensive description of the transput part of the language, it offers little or no help when implementing the language. It can only be seen as a description of the intention.

As a consequence, the implementation of the elaborate ALGOL 68 transput is a considerable task. Each implementer again has to struggle his way through the problems, which easily leads to wrong interpretations and deviations from the official definition.

It is clear that it is not possible to give a description of the ALGOL 68 transput which is completely machine-independent. It is impossible to give an exact description of the operations involving real arithmetic. Also, existing operating systems are very diverse, especially in their input-output facilities.

In the model as defined in part II of this treatise a very precise description is given of a (small) set of primitives, which are expected to be easily implementable on almost any machine. Except for those primitives, the transput is almost completely described in ALGOL 68 itself. In this way, each new implementation of the ALGOL 68 transput becomes relatively simple. After having realized the primitives on a specific machine, a complete implementation of the rest of the transput can be obtained almost mechanically.

The central part of the model is the "buffer". At some stage during the process of transput data must be passed to or from the operating system. It is very natural to pose some black box in between. Both the transput system and the operating system can communicate, via well-defined primitives, with the buffer; they do not directly communicate with each other. The buffer and its primitives are defined to be as flexible as possible. It is not only possible to implement these concepts under very diverse operating systems, but various types of buffers together with the corresponding primitives can easily be realized under a given operating system as well.

Besides a detailed description of the implementation model of the ALGOL 68 transput, as given in part II, part I contains a discussion of the history of the transput and the implementation model. The most important fundamentals of the model and a few interesting aspects of a specific implementation are also covered in part I.

SAMENVATTING

In het proefschrift wordt een beschrijving gegeven van de ALGOL 68 transput, de in- en uitvoermogelijkheden van ALGOL 68, waarbij de nadruk ligt op implementeerbaarheid. In het officiële definiërende rapport van de programmeertaal ALGOL 68 is weliswaar dit onderdeel ook uitvoerig beschreven, doch deze beschrijving biedt weinig houvast bij de implementatie van de taal. Zij kan als het ware alleen gezien worden als een beschrijving van de bedoeling.

Dit heeft tot gevolg dat de implementatie van de ALGOL 68 transput, welke veel uitgebreider is dan die van de meeste programmeertalen, een omvangrijke taak is. Iedere implementator moet zich opnieuw door alle problemen heenworstelen, hetgeen onder meer leidt tot afwijkingen van de officiële definitie en foute interpretaties.

Het is duidelijk dat het niet mogelijk is een beschrijving van de ALGOL 68 transput te geven welke geheel machine-onafhankelijk is. Niet alleen is het onmogelijk de operaties waarbij reële aritmetiek nodig is precies te beschrijven, maar ook zijn bestaande bedrijfssystemen zeer verschillend, zeker waar het de mogelijkheden van in- en uitvoer van gegevens betreft.

In het model van de ALGOL 68 transput, zoals dit in deel II van dit proefschrift is gedefinieerd wordt een zeer preciese beschrijving van een (kleine) verzameling primitieven gegeven, welke geacht worden op eenvoudige wijze op bijna elke computer geïmplementeerd te kunnen worden. De gehele ALGOL 68 transput wordt vervolgens, met uitzondering van deze primitieven, bijna geheel in ALGOL 68 beschreven. Iedere nieuwe implementatie van de ALGOL 68 transput wordt hierdoor een relatief eenvoudige zaak. Nadat de primitieven op de specifieke doelmachine gerealiseerd zijn, kan een volledige implementatie van de verdere transput langs vrijwel mechanische weg verkregen worden.

Het meest centrale concept uit het implementatiemodel is de "buffer". Op zeker ogenblik moet er tijdens transput informatie van of naar het bedrijfssysteem stromen. De buffer fungeert hierbij als een soort zwarte doos. Zowel het transputsysteem als het bedrijfssysteem hebben via primitieven toegang tot de buffer. Er is geen directe uitwisseling van

gegevens tussen transputsysteem en bedrijfssysteem. Deze buffer en de bijbehorende primitieven zijn zo flexibel mogelijk gedefinieerd. Niet alleen is het mogelijk deze concepten onder zeer verschillende bedrijfssystemen te implementeren, ook onder een gegeven bedrijfssysteem kunnen diverse typen buffers met de bijbehorende primitieven eenvoudig gerealiseerd worden.

Naast een gedetailleerde beschrijving van het implementatiemodel van de ALGOL 68 transput zoals gegeven in deel II van dit proefschrift, wordt in deel I een beschouwing gewijd aan de geschiedenis van de totstandkoming van de transput en het implementatiemodel. Tevens wordt in deel I ingegaan op de belangrijkste achtergronden van het implementatiemodel en worden enige interessante aspecten van een specifieke implementatie nader besproken.

CURRICULUM VITAE

De schrijver van dit proefschrift werd op 12 september 1949 geboren in Mijdrecht. In 1967 behaalde hij aan het Keizer Karel College in Amstelveen het HBS-B diploma. Vanaf 1 september 1967 is hij werkzaam op het Mathematisch Centrum te Amsterdam, aanvankelijk als operateur en, na in 1970 het diploma WRA behaald te hebben, als programmeur. Van 1973 tot 1978 studeerde hij part-time wis- en natuurkunde met bijvak econometrie aan de Vrije Universiteit te Amsterdam. Op 21 juni 1978 legde hij het doctoraalexamen af. Sinds 1 augustus 1978 is hij als wetenschappelijk medewerker bij het Mathematisch Centrum werkzaam.

Current address of the author:

J.C. van Vliet
Benonilaan 16
1695 HC Blokker
The Netherlands

ACKNOWLEDGEMENTS

I am deeply indebted to the members of the Task Force on Transput, who commissioned me to design the implementation model of the ALGOL 68 transput. I owe my sincere gratitude to the convener of this Task Force, C.J. CHENEY of the University of Cambridge (UK), for his kind, effective guidance of the Task Force and his ability to negotiate numerous compromises. At the meetings of the Task Force in Oxford (UK), Amsterdam, Cambridge (UK) and Summit (New Jersey, USA), I have had many inspiring discussions with C.H. LINDSEY, H. WUPPER, D. GRUNE, B. LEVERETT, R.G. FISKER and H. BOOM.

My gratitude also concerns my promotor Prof. A. VAN WIJNGAARDEN, director of the Mathematical Centre, and my coreferent Prof. J.W. DE BAKKER, who is also the head of the Computer Science Department of the Mathematical Centre, for their guidance and interest shown in my work.

I am grateful to L.G.L.T. MEERTENS, P. KLINT, S.G. VAN DER MEULEN and the members of the Task Force on Transput who read and commented on the many draft versions of this thesis, and to H. de J. LAIA LOPES for his assistance in testing the implementation model.

The front cover was designed by T. BAANDERS, using an etching called "Earth", made by DICK STOLWIJK. It is one of a series named "The four elements". The thesis has been prepared on a PDP11 under the UNIX Operating System. The final text has been produced on a HCT 300 daisy wheel printer using the standard formatting program ROFF. The printing is done at the Mathematical Centre by D. ZWARST, J. SUIKER, J. SCHIPPERS, J. VAN DER WERF and E. MICHEL.

PART I

HISTORICAL REVIEW
AND
DISCUSSION OF THE IMPLEMENTATION MODEL

Table of contents

1. INTRODUCTION

Programming languages cannot be used in a vacuum. If a program is actually to be run on a computer and produce results of some kind, it must contain some type of input/output statements. The input/output statements (also called i/o statements or transput statements) are those commands which relate to getting data in and out of the computer; input gives the data needed for the computation, output consists of the results of that computation.

It appears that there are two essentially different philosophies as regards the style of transput: stream i/o versus record i/o [71]. In general, a programming language contains some hybrid form of transput, with leanings towards one of the basic forms.

In stream i/o, data is represented externally as a sequence (or stream) of characters. Stream i/o is often "formatted", in which case a specification of the external data representation (called a "format") may be given. Sometimes the format may be implicit, in which case it is called unformatted, or free-formatted, i/o. Most scientific languages use stream i/o (FORTRAN, ALGOL 68, PASCAL).

In record i/o, the unit of information being transput is a record, i.e., the information in a contiguous segment of storage. Typically, data is first moved to an i/o area. Conversion is usually done when the data is moved, rather than when i/o statements are executed. The specification of the structure of the i/o area (called picture specification) takes the place of the format specification of stream i/o. COBOL is a typical example of a language having record i/o.

ALGOL 68 uses stream i/o. Around 1972, when work was progressing on the revision of the language, the possibility of introducing some type of record i/o was extensively discussed [26, 28, 29]. However, it has never found its way to the Revised Report. Since the scope of this treatise is restricted to ALGOL 68 transput, the problems of record i/o will not be further addressed.

A fundamental observation that can be made about transput is that it has two, often rather unrelated, aspects:

i) the transmission of data between the program and some external physical device;
ii) the conversion of data from an internal to an external representation, and vice versa.

It is generally the case that the transmission is machine-dependent, while the conversion is language-dependent. If, for instance, some integral value is output using an ALGOL 68 statement of the form 'put(f, x)', then, first, the internal representation of 'x' will be converted to a sequence of characters (composed of zero or more spaces, a sign, and the digits of 'x', in this order), and next this sequence will be transmitted to some physical device. This second step may include machine-dependent actions such as buffering, packing several characters into one machine word, etc.

In the early days of computing, there were very few distinct physical devices (usually just punched cards for input and a typewriter for output) and limited conversion possibilities. Computers nowadays generally provide for many different physical devices, and programming languages allow many types of conversion. Because of this, transput systems have become increasingly more complex.

The burden of implementing a transput system will be greatly relieved if the above two aspects can be separated. This is generally possible, since the conversion and transmission of data are often independent of each other. It is thus worthwhile to try to interpose a logical interface between the conversion and the transmission, as depicted in fig. 1.

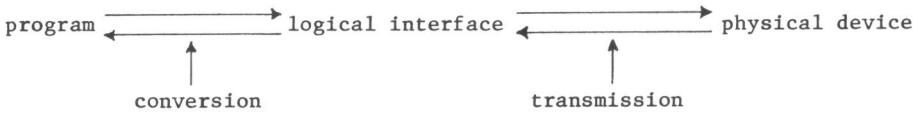


fig. 1. Logical interface between conversion and transmission of data.

Usually, the conversion possibilities are covered in the language, e.g., by formats (ALGOL 68, FORTRAN), picture specifications (COBOL), or by providing specific standard procedures (ALGOL 68, PASCAL). Sometimes, transmission aspects have their counterparts in the language as well, as in COBOL, where the user may specify tape density, blocking factors and the like in his program. Most programming languages, however, do not provide facilities to control the transmission aspects of their i/o.

Language definitions usually just define the various language constructs for i/o and their semantics (sometimes formally, more often rather informally) in terms of mappings from (external) character sequences to internal values, and vice versa.

ALGOL 68 transput provides both standard procedures (like 'print') and formats. The syntax of formats is rigorously defined using the same mechanism (two-level grammars) employed in the definition of the syntax of the remainder of the language. All semantics is defined by means of ALGOL 68 program texts. In addition, the semantics of formatted transput is defined in some kind of formalized English as well, similar to the definition of the semantics of other language constructs. The main drawback of the definition of the ALGOL 68 transput is that the underlying model of the actual computer (the logical interface) is too unrealistic to be of much practical value to an implementer. Moreover, the given program texts, if taken literally, are very inefficient.

The implementation model given in part II of this treatise attempts to formally define a more realistic logical interface. The conversion part has been completely rewritten in order to make it more efficient, and to make it fit the interface. The conversion part is largely defined by means of ALGOL 68 program texts, and can thus be used to create part of the transput implementation automatically. (Obviously, some details of the conversion, like those involving real arithmetic, cannot be defined machine-independently.) In this way, the burden of implementing the rather complex ALGOL 68 transput is greatly relieved.

To be able to fully understand the ALGOL 68 transput, it is helpful to trace the origins of its most important features. Tracing the development of this part of the language may also shed some light on the reasons for the

difficulties that arise when attempting to implement it. In the next chapter, the history of the transput up to 1975 (when the revision of the language was closed) is covered in some detail.

The historic background and the fundamentals underlying the implementation model given in part II of this treatise are discussed in chapter 3.

Finally, two interesting aspects of the implementation under the Control Data NOS/BE operating system are discussed in some detail. This may give an impression of the efficiency and the implementation complexity of the model.

2. HISTORY OF THE ALGOL 68 TRANSPUT

The programming language ALGOL 68 has been designed under the auspices of Working Group 2.1 of the International Federation for Information Processing (IFIP). IFIP is a multinational federation of professional-technical societies concerned with information processing. It has established a number of Technical Committees (TC) and Working Groups (WG). Each Technical Committee supervises a number of Working Groups. Working Group 2.1 (ALGOL) is supervised by TC 2 (Programming).

The original ALGOL 68 Report was published in 1969 [8]; a Revised Report was published in 1975 [12].

It should be emphasized that only the history of the transput part of the language will be addressed. Readers who are interested in the general development of the language are referred to [13] and [14]. In [13], VAN DER POEL, former chairman of Working Group 2.1, describes, in an anecdotic way, the development of the language up to 1968. In [14], SINTZOFF presents a lucid review of the revision of the language.

This history is largely based on other published material. For the period up to 1974, I have made extensive use of issues of the ALGOL Bulletin and the various drafts (penultimate, final) of the (Revised) Report on the Algorithmic Language ALGOL 68 [1-12]. Thereafter, I became personally involved with the ALGOL 68 transput.

In this chapter, the history of the ALGOL 68 transput is divided in four periods. The first period covers its development until the first appearance of some transput in a draft ALGOL 68 report. Next, the transput from the ultimate ALGOL 68 report [8] is described quite extensively, at the same time covering its development through the various drafts of the report. The third period covers the revision of the transput section up to the approval of the Revised Report by Working Group 2.1. Thereafter, considerable changes were again made in the sections on transput. Some of the flavour hereof is treated in the last section, which is euphemistically called "The final polishing".

2.1. ROOTS OF THE TRANSPUT

One of the main drawbacks of ALGOL 60 has been the absence of input-output primitives in its definition. Each implementer again had to define his own set of such primitives. As a consequence, many different input/output systems for ALGOL 60 exist, which renders the porting of programs very difficult. Two i/o proposals for ALGOL 60 have been standardized, one by ISO [15], the other by IFIP [16]. The two proposals complement each other to a large extent.

The ISO standard has been developed by the Subcommittee on ALGOL of the ACM Programming Languages Committee. (After the chairman of this Subcommittee, it is also known as "Knuthput".) It is mainly concerned with formatted i/o. Besides formats for numbers, strings and booleans, the proposal also defines "alignments" (for layout control) and "insertions" (literal strings). Quantities in a format may be replicated. If an unsigned integer n is used as replicator, the quantity is replicated n times. The character X as replicator means a number of times to be specified when the format is called.

Output is essentially printer-oriented. Standard procedures 'h lim' and 'v lim' are provided to control horizontal and vertical layout, respectively. Procedures 'h end' and 'v end' serve to specify that certain event routines provided by the user are to be called at line or page overflow.

The general form of an output statement is:

```
out list(unit number, name of layout procedure,
                      name of list procedure).
```

The procedure 'out list' has a number of local variables that are hidden to the user. These variables indicate the current format, the length of the current line and page, the procedures to be invoked at line or page end, etc. The hidden variables are set when the "layout procedure" is called. If the given layout procedure contains a call of 'h lim', then the hidden variable which controls horizontal layout is set according to the parameters of the call (and otherwise it is set to some default value). If it contains

a call of 'h end', its parameters (which are names of user-defined procedures) are remembered in one of the hidden variables. These procedures are subsequently called if one of the margins, as possibly set by a call of 'h lim', is transgressed. By default, dummy procedures are provided.

The layout procedure may specify the current format by a call of the form

format n(string, X_1, ..., X_n), (n = 0, 1, 2, ...),

where each X_i is an integer (called by value). The effect is to replace each X in the format string by one of the X_i, with the correspondence defined from left to right. The string thus obtained is assigned to the appropriate hidden variable of 'out list', subsequently to be used when the list procedure is called.

A "list procedure" serves to specify a list of quantities to be output. It has one formal parameter, which is the name of a procedure to be called for each item to be output. When the list procedure is called by the input-output system, some internal system procedure will be "substituted" for it. This internal procedure has access to the hidden variables of 'out list' and performs the actual output operation. The standard sequencing of ALGOL statements in the body of the list procedure determines the order in which the items are output.

A simplified version of the output statement is of the form

output n(unit number, format string, list of n items).
(n = 0, 1, 2, ...)

(In this case, the format string should not contain the character X.)

Input is done in a manner dual to output. One of the hidden variables serves to indicate a label to be jumped to when end of data is reached on input. It is set by a call of the procedure 'no data'. The default is to jump to an implicit label just before the final end of the program.

This fairly complicated scheme sketched above is necessary in order to stay fully within the ALGOL 60 framework. (The number of parameters of a procedure is fixed; there are no procedure or label variables.) Nevertheless, the origins of some important features of the ALGOL 68 transput, like formats and dynamic replicators, or event routines which can be provided by the user, can be traced back to this proposal.

The IFIP report is of a much simpler nature. It does not contain the sophisticated format capabilities of [15]. Rather, a small set of primitive procedures for unformatted input/output is defined. The report defines procedures to transput characters, real numbers, and arrays. The character transput procedures allow some simple kind of conversion from integral numbers to characters and vice versa. These procedures are both given a string as one of the parameters. On input, the value n is assigned to the integer variable given as parameter to the procedure, where n is the position of the first occurrence in the given string of the character read from the foreign medium. On output, n is given, and the n-th character from the string is written. Each of the procedures is given a channel (a positive integer) as parameter.

Garwick considered the IFIP proposal too primitive, and suggested adding a set of procedures to output numbers and strings according to a given primitive format [17]. For example, a routine 'outfix' is proposed which outputs a number in fixed point form, with 'before' digits before, and 'after' digits after the decimal point, where 'before' and 'after' are parameters of 'outfix'. These procedures all output to a buffer at some given position. Such a buffer must be long enough to hold one line. The buffer itself is output on some given channel by an explicit call of another routine (called 'output').

When it came to drawing up a list of features to be expected in ALGOL X, the proposed successor to ALGOL 60, the following statement about its input/output appeared ([18], point 2.15):

> "The data transmission aspects of input/output are to be separated from the data conversion aspects. A set of facilities for the latter are to be specified, intermediate in complexity between those of D.E. KNUTH and J.V. GARWICK. New statement forms and

> delimiters may be introduced by the user for this purpose under specified conditions."

(The references are to [15] and [17], respectively.)

The task to prepare an i/o proposal was assigned to a Subcommittee for ALGOL-X-I-O. Its members were J.V. Garwick, J.M. Merner, P.Z. Ingerman and M. Paul. The Subcommittee presented its report at the Working Group meeting in Warsaw, October 1966 [19]. The report contains proposals for procedures to convert numbers to strings, along the lines sketched by GARWICK [17] (called "fixed form editing"), and procedures to utilize format strings (called "format controlled editing"). The syntax of formats in this proposal is already very close to that in the ultimate ALGOL 68 report [8], and is based on the work of KNUTH et al. [15].

The Subcommittee realized that this proposal was inadequate as a general solution. In the cover letter to [19], it says:

> "The more comprehensive approach which is favoured gets involved with character set declarations and mappings, as well as translation procedures, and physical device characteristics. It is felt that much more effort should be devoted to this area."

It was decided at the Working Group Meeting in Warsaw that the i/o proposal would be incorporated in the ALGOL X document which was presented at this meeting by A. VAN WIJNGAARDEN and B.J. MAILLOUX [1]. After two other drafts [2, 3], the first rudimentary ALGOL 68 transput appeared in [4]. This report appeared in January 1968 as a supplement to ALGOL Bulletin 26. The appearance of the transput coincides with that of C.H.A. Koster as one of the authors.

The document aroused a lot of violent reactions (see, e.g., [20]), but remarkably little was said about transput. C.H. LINDSEY [21] proposed a number of changes to the transput section, some of which were incorporated in the next versions of the report, and others in the revision of the language.

{An interesting reaction came from I.D. HILL [22], who in a long list of remarks on [4] asked:

"Does transput really need to be as complicated as this?"

Things are even worse. The 7 pages devoted to transput in [4] were to grow to 20 pages in the ultimate ALGOL 68 report [8]. In the revised ALGOL 68 report, the transput section comprises over 50 pages (not counting the part defining the syntax and semantics of format texts).}

The ALGOL 68 transput assumed its final shape between January 1968 [4] and February 1969 [8]. In between, three other drafts were published in July 1968 [5], October 1968 [6] and December 1968 [7]. Apart from a few small changes in the area of conversion keys, the transput as given in the version of October 1968 [6] is the same as that given in the ultimate report [8].

Essentially, the transput in [8] follows the lines sketched by the Subcommittee for ALGOL-X-I-O [19]. It also gives a first answer to the problems of character set mappings ("conversion keys") and physical file characteristics that were left unsolved in [19].

2.2. THE TRANSPUT OF THE ORIGINAL REPORT

In this section, the transput as of February 1969 [8] is sketched. Some of the more interesting developments since January 1968 [4] are outlined as well.

When referring to one of the ALGOL 68 reports from this period, I will in the sequel not use the customary square bracket notation; rather, the numbers of the corresponding Mathematical Centre reports will be used. Thus, from now on, MR93 stands for [4], MR101 stands for [8], and so on. This will probably be easier for readers already familiar with the early stages of ALGOL 68.

Transput is modelled in terms of "channels", "backfiles" and "files". The user of ALGOL 68 is mainly concerned with files. Each call of a transput routine is given some file as parameter, and the transput routine then uses that file.

The use of the term "file" is rather confusing here. Usually, it is used to just denote some organized collection of data. In texts on programming

languages, the term generally crops up in the context of i/o only to denote this (external) information. The actual handling of such files falls beyond the definition of the language; it is part of the operating-system environment. In this sense, its closest counterpart in ALGOL 68 is the "backfile", which will be discussed below. In ALGOL 68, the term "file" has a much broader meaning. It encompasses all information necessary for the transput system to work with its backfile; it includes, for instance, the format currently in use.

Before transput using a given file is possible, the file must be "opened" via some channel. A channel corresponds to a device, and each channel is represented by an integer. With each channel, a number of backfiles is associated, which contain the actual information. (If some channel corresponds to a card reader, one backfile will be associated with it, viz., its deck of cards. In the case of a channel which corresponds to a disk, a larger number of backfiles will usually be available.)

In MR93, a fixed number of backfiles is associated with each channel. A chain of available backfiles is built for each channel at the start of the program. Upon opening a given file on a given channel (by means of the routine 'open'), the first backfile available on the channel is taken and associated with the file. This scheme is very primitive; the user does not know how the chains of backfiles are being built, so he has no control whatsoever about the mapping of physical files ("backfiles") to logical files in his program ("files"). In MR 95, each backfile is given an identification in the form of a string. This string is given as parameter to the routine 'open', so that, rather than just taking the first available backfile, a backfile is sought whose identification string is equal to the given string. (Identification strings are not necessarily supported by all channels though.) Apart from opening a file, MR95 also offers the possibility to "create" a file. In that case, a backfile is generated, with a size which is the maximal size allowed by the channel. In MR99, it becomes possible to "establish" a file. If a file is established, a backfile is generated with a size which is given by the user. (This size must of course not exceed the maximal size allowed by the channel.)

The file may subsequently be "closed", in which case the backfile is attached to a chain of closed backfiles for that channel. If a file is

reopened without first closing it, its backfile is obliterated. In MR93, once a file has been closed, its backfile becomes inaccessible for the rest of the program. In MR95 the possibility is offered to indicate explicitly whether or not the backfile should be kept available for reopening ('close' versus 'lock'). There is also an explicit routine in MR95 to obliterate a backfile ('scratch').

Actually, references to backfiles, rather than the backfiles themselves, are chained together. After a call of the routine 'open', the reference is removed from the chain for the given channel. Although not explicitly stated, the model suggests that there is only one such reference for each distinct backfile. (If more than one reference to one and the same backfile is present, there is no protection against writing via different files to the same backfile, possibly even at different positions. Note that such multiple access must be explicitly granted prior to execution of the program. At the time the chains of backfiles are being built, the number of references to one and the same backfile determines how many times the backfile is available for opening.)

The number of channels in a given implementation is fixed for each program. It is given by the integer 'nmb channels', and the channels are numbered from 1 up to 'nmb channels'. Each channel has a number of properties associated with it, which describe the static, functional properties of some specific device:

- boolean properties that determine the available methods of access to a file linked via this channel. For example, 'get possible[6]' tells whether reading is possible via channel 6.
- integer properties that determine the maximal size of a backfile for this channel.
- an integer giving the maximal number of files the channel can accommodate.
- the standard conversion key of the channel.

Except for the conversion key, all these properties are available to the user.

The most important information contained in a backfile is a three-dimensional array of (small) integers, called the "book". The indices are termed "page", "line" and "char", respectively. The lower bounds of the array are all 1, the upper bounds are determined by the channel via which the backfile is available (or by the user, in the case of a backfile generated by 'establish'). The size of the book is fixed once and for all, and all pages and lines are of the same length.

A backfile also contains three integers comprising the "end of file", i.e., the page number, line number and character number up to which the book of the backfile is filled with information. This "end of file" is set prior to the execution of the program. In case of a sequential-access file (i.e., a file opened on a sequential-access channel) this "end of file" may get changed. In MR93 such a change may only occur when the file is closed, or reset to the beginning. In the case where the file is used for output, all information beyond the "current position" then is lost, i.e., the "end of file" is set to the current position. In MR101, the first output operation on a sequential-access file already causes the "end of file" to be set back, and from then on it keeps pace with the current position.

In MR93, it is never possible to write beyond the "end of file". Thus, if a user wants to write to a book that has not been written to previously, i.e., that contains no information as far as this user is concerned, the "end of file" should point to the position up to which the book <u>may be</u> filled with information. This ambiguity in meaning (physical vs. logical end of file) may lead to strange effects if one of the routines 'set', 'space', 'newline' or 'newpage' is called. Although, for sequential-access files, the book is initialized to spaces from the current position onwards as soon as the first character is written to it, these routines just change the current position and may therefore leave part of the book undefined. This flaw is only partly remedied in MR101. There, the "end of file" is used correctly (in the sense that it is only used to indicate the position up to which the book is actually filled with information), but the layout routines may still leave part of the book undefined.

A "file" controls the access to a backfile via some channel; it contains references to both. In MR93 a file also contains a reference to the "current position", i.e., the position in the book at which the next transput

operation will take place. In MR101, this current position is incorporated in the backfile, rather than in the file (thus ensuring that different files accessing the same backfile always do so at the same position).

The file also contains a conversion key, which, in MR93, is a reference to a string. After opening a file, the standard conversion key of the channel is used. The user may provide a different conversion key by assigning some other value to the corresponding field of the file (in MR93, this field is the only one accessible to the user). Conversion keys map characters to (small) integers and vice versa. The characters are used inside the program, the integers appear in the book. On output, each character is mapped to the integer which is equal to the position of its first occurrence in the conversion key. On input, the character returned is the character which appears in the conversion key at the position indicated by the integer. In fact, this is just the mapping already defined in [16], although it works in the opposite direction. In this way, conversion keys are not symmetric; it is possible to map two different integers onto the same character, but it is not possible to map two different characters onto the same integer (as was already reported in [20]). This is remedied in MR99. There, a conversion key is an array of integers, and the operators ABS and REPR are used to map characters to integers and vice versa. In MR100 a special routine 'make conv' is provided to associate a conversion key with a file, and in MR101 the corresponding field of the file is made inaccessible to the user. In this last model, all possible conversion keys must be present in the standard prelude.

The interface suggested by the above model is rather unrealistic. In real-life operating systems, physical files are, in general, looked at through a window, via which only part of the information is visible at each instant of time. Also, many properties which are assumed to be fixed as soon as execution of the program starts can in fact only be determined at run-time. For instance, whether or not writing is possible to a given file does not only depend on the channel via which the file is opened, but also on the (external) physical file it is linked to. Whether or not another file can be opened via a given channel is another typical run-time property.

In MR93 the user of transput has no possibility to trap any erroneous situation. If any such situation occurs, the further elaboration is left

undefined. In MR95 the user is given the possibility to trap some of the possible error conditions. To this end a number of procedures are incorporated in the file. These procedures may be provided by the user; he may simply assign something to the corresponding fields of the file. (Note that these procedures are not given the file as parameter, nor deliver any indication of the result back to the transput system.) In MR95, four such procedures are provided:

- 'logical file ended', which traps end of information on sequential-access files;
- 'physical file ended', which traps the physical end;
- 'disagreement', which occurs when conversion fails, or when during formatted transput a character is transput which is not expected;
- 'incompatibility', to trap formatted transput of values incompatible with the item of the format at hand.

In MR99, the flexibility of these error routines is further enhanced. Firstly, all error routines now deliver a boolean indicating whether or not the user feels that he has mended the situation. The routine 'disagreement' (now called 'char error') is given a reference to a character as parameter. Its value is a character suggested by the transput system as replacement, but the user may give some other character instead of the suggested one. (The routine 'incompatibility' is renamed 'value error'.) Two additional error routines are provided:

- 'format end', which is called when the end of the format is reached while there still remains some value to be transput (formats will be dealt with later on);
- 'other error'. This routine is given some integer as parameter. The meaning of the parameter is not defined, and the routine is nowhere called in the transput section.

(Given the one-way flow of information from the operating system to the program which is envisaged in this transput model, where all information is also fixed completely before the execution of the program starts, this 'other error' routine comes in rather unexpectedly. The fact that it is nowhere called also hints at the apparent attitude that the transput section is nothing but a description of the intention, and has to be reprogrammed

for each implementation anyway. From a language designer's point of view, I consider this to be an appalling mistake.)

Transput is either unformatted, formatted or binary. In unformatted (or formatless) transput, some standard scheme is followed. Layout control is automatic. In MR93 numbers are always followed by one space on output (and also have to be followed by one space on input if not at the end of a line); in MR95 they are preceded by a space, and in MR99 they are preceded by a space if not at the beginning of a line. The user can get some control over the layout of numbers by first converting numbers to strings by means of one of the routines 'int string', 'real string' and 'dec string', and subsequently outputting these strings. Actually, these routines are, with default values for some of the parameters, also used inside 'put'. These conversion routines essentially also appear in [19].

In MR93, if inputting to a string variable, the number of characters read is equal to the length of the string referred to by the variable at the time of the call, even for flexible names. From MR95 onwards flexible string variables are handled differently: reading stops either at the end of a line, or at some character from the "terminator string" of the file. Normally, the terminator string of a file is the empty string, but the user may assign some appropriate string to the corresponding field of the file. (In MR101 there is offered the possibility to pass line boundaries when reading strings by assigning an appropriate routine to the 'physical file end'-field of the file.)

Formatted transput uses so-called "formats" to describe the exact layout. In MR93 each call of one of the formatted-transput routines (termed 'in' and 'out') is given a format and a list of elements to be transput. The first item of the format is used to transput the first element of the data list, and so on. The format has to contain enough items to process the complete data list. In MR95 it is possible to subsequently use that format in more than one call of 'in' or 'out'. To this end, the format has to be incorporated in the file, together with a pointer to the current item of the format. In MR99 the event routine 'format end' is introduced. This routine is called when the current format is exhausted while there still remains some item to be transput. The default action is to repeat the current format.

The syntax and semantics of formats as given in MR93 (or MR101) is based on [19]. A format consists of a number of "primaries", each of which is either an elementary "item", or a list of primaries to be replicated a certain number of times. For each kind of simple value (integer, real, boolean, complex, string) there is a different kind of item. For integers there are two possibilities: besides the possibility to transput it as a sequence of digits in some radix system (base 2, 4, 8, 10 or 16), it is also possible to transput an integer as one of a list of literals. I.e., in the case of output the j-th literal is output if the given integer has the value j, and in the case of input the value j is assigned to the name given if the j-th literal from the list is read. Booleans are always output as one of a list of two literals (the default being ("1", "0")).

At the lowest level, formats consist of "frames", "replicators", "literals" and "alignments". A "frame" is a character specifying how some other character is to be transput, e.g., a "d" for a digit of a number, an "a" for a character from a string. A "literal" may be any sequence of characters placed between quotes. An "alignment" is a character to explicitly control layout (e.g., a "p" causes a page eject). In formatted transput all layout must be indicated explicitly by the user, i.e., new lines and pages are not generated by default.

Frames, literals and alignments may be replicated any number of times by putting a "replicator" in front of it (the default value being 1). A replicator is either a plain integer, or it is "dynamic", in which case its value is determined at run-time. Thus, '10d' indicates that the frame "d" should be replicated 10 times, and 'n(i)d' indicates that it should be replicated 'i' times, where 'i' is elaborated at run-time. (A dynamic replicator is written as an "n" followed by a closed clause delivering an integer.)

Upon a call of one of the formatted-transput routines, the given format is "transformed", i.e., all of its dynamic replicators are elaborated and some internal structure is built. This internal structure is hidden from the user; like in [15] and [19], a string is used in the description. This string is subsequently inspected by the formatted-transput routines. An implementer may, for efficiency reasons, choose another approach. He may, for example, generate code segments for each of the items of a format and

then use some coroutine-like mechanism for formatted transput.

For an extensive comparison of the formatted transput of ALGOL 68, PL/I and FORTRAN, see CARROLL [72].

Lastly, binary transput may be used to transput values in a more efficient way. On output, each simple value is converted to a sequence of integers in some unspecified way, and this sequence is subsequently copied into the book. On input, the reverse scheme is followed. Here, the number of integers read is, in an obscure way, determined by the name being read into. Binary transput is likely to be rather machine-dependent.

2.3. TOWARDS THE REVISION OF THE TRANSPUT SECTION

Flexibility of a transput system has several aspects. Firstly, there is the flexibility with which values may be converted. Most of what the transput section in MR101 offers in this respect goes back to the work done by the Subcommittee for ALGOL-X-I-O [19]. Secondly, there is the flexibility of interaction of the transput system with the user program and the operating-system environment. MR101 offers little in this respect. Its view of the outside world (the operating system) is very static. At a late stage, some interaction with the user became possible through the introduction of some event routines. Not surprisingly, the major part of the revision of the transput section has to do with the enhancement of its flexibility, most notably in the area of interaction with the user program and the operating system.

At the time Working Group 2.1 approved the ALGOL 68 report, one or more revisions were already envisaged, as is emphasized by the cover letter to MR101. First of all, however, experience with ALGOL 68 implementation had to be gained. Various conferences on ALGOL 68 implementation were organized from 1969 onwards. The transput section, alas, did not inspire many people, as is evidenced by the proceedings of these conferences (see, e.g., [23] and [24]). Not one paper on transput is to be found there; very few people even mention it at all. When it was decided at the Working Group meeting in Habay-la-Neuve (July, 1970) that at one moment a new report would be prepared, the Working Group justly felt that it needed more guidelines for possible improvements of the transput [25]. For this purpose, it established

a new standing subcommittee, the "Subcommittee on ALGOL 68 data processing".

The Subcommittee met for the first time in Manchester, March 23-26, 1971. The meeting was convened by C.H. Lindsey (University of Manchester). Other members present were C.H.A. Koster, D.P. Jenkins (who implemented the ALGOL 68R transput) and R.J. Gilinsky (Computer Sciences Corporation, California). A list of topics discussed at that meeting can be found in [26]. The changes proposed there mostly did not address global questions, but were rather concerned with relatively local changes and additions. For example, the following proposals were made:

- an escape facility in formatted transput, the so-called "general pattern". At that time only a plain "g" was proposed; the present routines 'whole', 'fixed' and 'float' did not exist yet.

- a method to construct formats inside formats, the "included pattern" (later renamed "format pattern").

- a routine 'set char number', as the unformatted equivalent of the "k"-alignment.

- a flexible style of printing in formatless transput, like it is available in ALGOL 68R [27]. This proposal has later been replaced by one containing an approximation of the present conversion routines 'whole', 'fixed' and 'float' [31].

Some questions of a more general nature addressed the length of a line, which in MR101 is the same for each line, and the possibility of allowing transput to and from strings (which is presently possible through the routine 'associate').

At the meeting in Manchester the Subcommittee also discussed "record transfer", which they considered to be the most vital part of their work as far as data-processing applications of ALGOL 68 are concerned. (Record transfer does not operate on simple values, but on records, i.e., values of some, in general, more complicated mode. For instance, if some structure containing multiple values is output and subsequently re-input to a name with flexible bounds, then those bounds will be adjusted to the value

actually received.) Of course, one needs a different notion of the term "position" here. This is especially true for random-access files, where one needs the "address" of a record, or its "key".

The Subcommittee held another meeting in Amsterdam, August 9-11, 1971 [28]. The items of the previous report [26] were considered again; a new set of conversion routines was proposed, and the string-transput proposal was still felt unsatisfactory. The greater part of the meeting was devoted to a discussion of record transfer (see [28]). It was felt that much more work was needed in this area. {A formal proposal, based on the discussions of the Subcommittee ([26, 28]) can be found in [29]. However, record transfer has never found its way to the Revised Report. Recently, various people have again shown interest in this area.}

The following proposals arose from discussions between C.H. Lindsey and A.J. Fox (Royal Radar Establishment) at the transput-subcommittee meeting in Manchester, January 1972 [30]:

- The (user-callable) routines 'file ended', 'page ended', etc., are removed from the standard prelude. The motivation for this change is that many operating systems are loath to provide such information. At the same time, extra fields are added to the mode FILE, so that the same events are now reported to the user in a roundabout way.

- The error-procedure fields of the file will be given a REF FILE parameter (in this way, the same routine can be associated with different files).

- A new mode CHANNEL is introduced, and the environment enquiries become of the mode PROC (CHANNEL) BOOL. This is to make it possible to tell at compile time which channels are going to be used by a program.

The Subcommittee cooked up a long list of proposals for revision of the transput section (the main sources hereof are [26], [28], [30]) and presented it at the meeting of the Working Group at Fontainebleau, April 7-11, 1972 [31]. Unfortunately, no decisions on transput could be taken at this meeting. (Once again, transput is treated stepmotherly.) At the next meeting of the Working Group, which took place in Vienna, September 11-15,

1972, decisions were taken on most of the proposals from [31]. The results can be found in [32].

A few of the more interesting points from [31] and [32], not yet mentioned, are:

- Each replicator from a format is to be elaborated only when it is encountered during a call of one of the formatted-transput routines.

- The standard prelude will contain no declarations involving field-selectors known to the user (except for COMPL). This is to facilitate the writing of preludes in French.

- Various proposals aim at ensuring that the report does not appear to insist on facilities that specific operating-system environments may be completely incapable of supplying. Some of these problems are "solved" by the introduction of the "gremlins".

Finally, a draft revised report appears in February 1973 [9], and this draft is discussed at the meeting of the Working Group at Dresden, April 3-7, 1973. History repeats itself. Contrary to the first drafts of the original report, this one does contain some transput, albeit just a copy of the first few pages of the transput section from MR101. At the meeting itself, a document with part of the revised transput routines becomes available [33]. (From now on, the terms "backfile" and "book" are replaced by "book" and "text", respectively.) The changing of the guard becomes apparent from the heading of this document, which reads:

> "Prepared by C.H. Lindsey and R. Fisker
> after discussions with C.H.A. Koster."

During the summer of 1973, Lindsey and Fisker worked hard on transput. The draft Revised Report dated July 1973 [10] contains a complete set of transput routines, but not a single pragmatic remark. This document is discussed at the meeting of the Working Group at Los Angeles, September 3-7, 1973.

The Revised Report on the Algorithmic Language ALGOL 68 is approved in Los Angeles and passed to TC 2, being the next higher level of IFIP.

In the ALGOL Bulletin that appears in November 1973, the Editor states:

> "all that remains now is for a few final polishings of the text."

As for the transput part, these "final polishings" will include writing all of the pragmatics, and rewriting a considerable part of the routine texts. This rewriting is partly caused by a drastic change agreed upon in Los Angeles: the appropriateness of the current position will be checked at the entering of the various routines, rather than at the exit.

For the transput section, one of the most turbulent periods has yet to come.

2.4. THE FINAL POLISHING

After the Working Group meeting in Los Angeles, September 1973, the editors of the Revised Report hold a "final vetting session" in Manchester during the last week of November. During this meeting, much time is spent on the transput section; the system is by no means foolproof yet. It transpires that the user, by admittedly wicked manipulations in the event routines, can easily break through the defence. For instance, it turns out to be possible to write to a read-only file.

From December 1973 onwards, a group of people at the Mathematical Centre (D. Grune, L.G.L.T. Meertens, J.C. van Vliet and R. van Vliet) concern themselves with transput. Initially, L.G.L.T. Meertens starts to look how the system can be implemented in a tolerably efficient way, making use of the fact that tests that have been performed once need not be repeated if no call of an event routine intervenes. MEERTENS' own account of this effort states [34]:

> "However, I did not make any progress with this work, since the philosophy behind the tests kept escaping me. [...] During these vain attempts I found various cases which were clearly intolerable, and which I was tempted to attribute to a supposed

lack of philosophy."

In a letter to Lindsey and Fisker, dated December 10, 1973, MEERTENS writes [35]:

> "The philosophy behind get good page escapes me: [...]. In the present text, I can only go through the routine texts with a particular situation in mind, but I do not succeed in constructing a conception of the intended meaning."

The problems are mainly concerned with the definition and application of the routines that control the state ("get good"-routines). These routines are very critical; in the revision of the transput, the event routines may be called at every odd place, and after each such call the state must be ensured again. The transput system at that time seemed to perform these tests in a very haphazard way. On March 1, 1974, the situation is described by MEERTENS as follows [34]:

> "For a good month we have been working on the transput. In this time we have succeeded in unearthening a number of unacceptable phenomena. We feel now that we understand more or less what the basic cause of the error is, and we have reached an infinitesimal confidence that an implementable and watertight system may be reached by mending, one by one, each specific case. Instead, we have started to design, from scratch, a system of checks that is clear to work from the beginning. Our experience has been, that this task is much tougher than it looks at the start. The basic problem is that, after a call of an event routine, we may assume nothing whatsoever about the state of the file, and the number of (implicit) assumptions turns out to be much larger than the few about physical file, etc.. Also, for quite some time, we have worked along the assumption that it would be possible to formulate some invariant assertions, such as that it is not possible to get far outside the logical file in some defined manner, which may simplify some tests. However, it has become apparent that if the user tries hard enough, there is no way to prevent him from doing so, so that this also has to be checked after each call of an event routine. We have good hope now, however, that our present

approach is watertight, does not have intolerable surprises for the user, is implementable with reasonable efficiency and is understandable."

This approach can be described very concisely as a system of routines of the form:

```
PROC ensure condition on some level = VOID:
  WHILE ensure condition on next higher level;
    event occurs on this level
  DO (NOT event mended | appropriate action) OD.
```

On March 18, 1974, the ALGOL 68 group at the Mathematical Centre sends a document to the editors which, apart from shocks and surprises such as:

- how to read from a write-only file
- how to read beyond the logical end
- how to write beyond the logical end
- how to get an automatic alternation of defined and undefined characters,

also contains an outline of their philosophy and a sketch of a set of position enquiry- and layout routines, based on this philosophy [36].

A somewhat more elaborate version of this document [37] is discussed at the first meeting of the Subcommittee on ALGOL 68 Support, held in Cambridge, U.K., April 7-10, 1974. The current state of the report at that time is [11], together with a list of errata. Many of the points mentioned in [37] were accepted (for a discussion of this meeting, see [38]), and parts of the transput section had to be rewritten yet another time.

After another year, the Revised Report appears [12].

3. ABOUT THE IMPLEMENTATION MODEL

Many programs that are written in ALGOL 68 as defined originally [8] look pretty much the same when rewritten according to the revision of the language [12]. The two defining documents however differ considerably. This is the more true for the sections on transput. For the casual user, transput has not changed so drastically since 1968. Nevertheless, the transput sections in [8] and [12] are completely different; both the pragmatics and the program texts have been fully rewritten.

The revision has introduced a few very useful routines, like 'whole', 'fixed' and 'float'. It has also enriched the language with a number of (in my opinion) useless features, most notably the run-time elaboration of dynamic replicators and the calling of event routines at every odd place. These features greatly complicate implementation, and, if not looked at very carefully, considerably slow down the system as a whole. Moreover, since a file is a normal ALGOL value, the use of event routines may easily lead to unexpected scope violations.

In the revision, an attempt has also been made to enhance the flexibility towards the operating-system environment. The unrealistic view of the outside world is indeed one of the weak points of the original transput. The introduction of the semaphore 'gremlins' does offer the possibility to answer many nasty questions in an easy way, but this is hardly of any help to an implementer. Most computers have no built-in gremlins.

The original transput routines are seemingly written in the pre-structured programming era. Since then, we have all read Dijkstra, and we are now living according to a different paradigm. In his thesis, FISKER, being one of the authors of the revised transput sections, writes [39]:

> "When the transput section was being rewritten, certain aims were born in mind. The major aim has been to achieve the principle stated in the Revised Report that "the declarations are intended to describe their effect clearly". In addition, one of the directives to the editors of the Revised Report, given by WG 2.1 was "to endeavour to make its study easier for the uninitiated

> reader". To this aim, a standard layout was adopted, and the principles of structured programming applied."

Neat layout and jump removal may to some extent enhance the ease of understanding and clarity of a program, but it does not make it a good program. The basis for a good program is a good design, a clear statement of the philosophy behind it, a concise and orthogonal description of its intended functioning.

One cannot simply blame the authors of the transput sections for its failing on these criteria. Both in 1968 and in 1974, transput came in very late. On both occasions it was not properly discussed in time. Quite fundamental changes were agreed upon in a late stage, sometimes even long after the Working Group had already approved the document; it is hard to hit a running rabbit. Moreover, there were no proper tools available to thoroughly test the correctness of the program texts.

Nevertheless, transput is likely to be a substantial part of the run-time system of an ALGOL 68 implementation. It is tempting to just feed the transput routines from the Revised Report into the compiler, thus creating part of the run-time system automatically. This does not work.

At some stage there has to be some interaction with the operating-system environment. There has to be a (hopefully small) set of primitives that the transput is based on; these primitives then form the operating-system interface. The transput sections of the Revised Report offer little or no help in finding these underlying primitives. It may only be looked upon as a description of the intention of transput. It has been remarked before that this is very unfortunate, since, as a consequence, the burden of finding all tricky spots is placed upon the shoulders of each individual implementer. This effectively means that the transput has to be rewritten for each implementation.

Even if the routines could be used straightaway, the resulting system would be hopelessly inefficient. If the Revised Report is bluntly followed, the very simple writing of one single character in the middle of a normal line of a very normal file will lead to ("->") the following chain of tests and procedure calls:

```
put(f, "a")
-> opened OF f?
   set write mood(f)
   -> put possible(f)
      -> opened OF f?
         (put OF chan OF f)(book OF f)
      set possible(f)
      -> opened OF f?
         (set OF chan OF f)(book OF f)
   set char mood(f)
   -> set possible(f)
      -> opened OF f?
         (set OF chan OF f)(book OF f)
   next pos(f)
   -> get good line(f, read mood OF f)
      -> get good page(f, reading)
         -> get good file(f, reading)
            -> set mood(f, reading)
               -> set write mood(f)
                  -> put possible(f)
                     -> opened OF f?
                        (put OF chan OF f)(book OF f)
                     set possible(f)
                     -> opened OF f?
                        (set OF chan OF f)(book OF f)
                  physical file ended(f)
                  -> current pos(f)
                     -> opened OF f?
                     book bounds(f)
                     -> current pos(f)
                        -> opened OF f?
               page ended(f)
               -> current pos(f)
                  -> opened OF f?
                  book bounds(f)
                  -> current pos(f)
                     -> opened OF f?
            line ended(f)
```

```
            -> current pos(f)
               -> opened OF f?
               book bounds(f)
               -> current pos(f)
                  -> opened OF f?
      put char(f, k)
      -> opened OF f?
         line ended(f)
         -> current pos(f)
            -> opened OF f?
            book bounds(f)
            -> current pos(f)
               -> opened OF f?
         set char mood(f)
         -> set possible(f)
            -> opened OF f?
               (set OF chan OF f)(book OF f)
         set write mood(f)
         -> put possible(f)
            -> opened OF f?
               (put OF chan OF f)(book OF f)
            set possible(f)
            -> opened OF f?
               (set OF chan OF f)(book OF f).
```

It is checked 18 times whether the file is opened, and 44 procedure calls are needed.

This inefficiency was known. In his thesis, FISKER writes [39]:

> "A consequence of the aim to make the transput declarations as clear as possible is that they are inefficient. This is no great disadvantage since in any implementation the transput would be implemented either by writing it in some other language, or by rewriting it in order to improve its efficiency."

In my opinion, the above kind of inefficiency is largely due to the plugging of loopholes after every reported bug, which action by the way does

not improve clarity.

Also, rewriting the complete transput part is advocated again. This is likely to be a big task. In his review of ALGOL 68, SINTZOFF states [40]:

> "The writing of the full transput package, its integration into a real operating system and a complete documentation for the compiler users seem to require a work of the same order of magnitude as for the language kernel."

Faced with such an enormous task, implementers will be tempted to deviate from the Revised Report, or drop certain difficult parts (like formatted transput) entirely.

3.1. HISTORY OF THE IMPLEMENTATION MODEL

I made my first plea for a machine-independent description of the transput at the Strathclyde ALGOL 68 Conference in March 1977 [41]. At that time I had already been working on such a description for about a year. I first attacked the so-called "conversion routines": 'whole', 'fixed' and 'float'. A first version of a new set of conversion routines appears in [42].

During this research, quite a number of problems in the transput sections came to light. I prepared a list of 42 such problems [43] and submitted it to the Subcommittee on ALGOL 68 Support for discussion at its meeting in Kingston, Ontario, August 1977. Other material submitted to this meeting includes a first draft of my machine-independent description of the transput [44] and remarks on it by FISKER [45], answers to the list of reported errors by LINDSEY [46] and two papers on an implementation model of THOMSON & BROUGHTON [47, 48].

The Subcommittee did not feel entitled to judge these questions in an appropriate manner and therefore set up a "Task Force on Transput". The Task Force was set up in order to "reasonably interpret the transput sections of the Revised Report" and it was "not merely [to] give specific answers to ... bugs, but should consider them and their implications more generally" [49].

The initial membership list mainly contains people who are (have been) involved in the implementation of ALGOL 68, such as:

C.J. Cheney (convener of the Task Force,
University of Cambridge, ALGOL68C compiler),
C.H. Lindsey, R.G. Fisker (University of Manchester,
Manchester implementation),
B. Leverett (Carnegie Mellon University,
ALGOL 68S compiler),
J. Schlichting (Control Data Corporation, CDC implementation),
H.J. Boom (Mathematical Centre, ALGOL 68H compiler),
S.R. Bourne (Bell Laboratories, ALGOL68C compiler),
C.M. Thomson (CHION Corporation, FLACC compiler),
J.C. van Vliet (Mathematical Centre, MC compiler).

The Task Force met for the first time in Oxford, U.K., December 14-15, 1977. Most of the problems from [43] were solved at this meeting. For a discussion of the solutions, see [50]. The Task Force also discussed a paper from C.H. Lindsey describing some interface problems between the ALGOL 68 transput as defined in the Revised Report and the features provided by real-life operating systems [51].

Considerable attention was paid to a discussion of a possible machine-independent description of the transput. Documents from three different sources were available:

- The text of the Manchester implementation of the transput by FISKER [52];
- Reports from THOMSON & BROUGHTON on the FLACC implementation of the transput [47, 48];
- A second version of my model [53];

At the basic level, there is no fundamental difference between these proposals. For instance, they all emanate from a buffer concept. However, the proposal in [53] was by far the most complete. After some discussion, this report was chosen as a basis for a machine-independent model of the transput.

Two major objections were made against the model as described in [53]: the model makes extensive use of string-processing operations, and the formatted-transput part was considered to be unacceptably inefficient.

Building up a string of n characters one by one using standard ALGOL 68 operators is likely to result in the allocation of about (n**2)/2 storage cells, most of which are garbage. This is the more dramatic for sublanguage implementations which do not have a garbage collector.

The inefficiency of the formatted-transput part was known to me. Due to my wish to present a complete model at the Oxford meeting, I had no time to look into this part in sufficient detail. Therefore, I just gave a correct description conforming with the rest of the model. Apart from this, the formatted-transput part in [53] completely follows the lines of the corresponding part of the Revised Report.

The Task Force commissioned me to modify the model from [53], to be the required reasonable interpretation of the transput sections of the Revised Report. A draft version of this model [54] was presented to the Task Force at its meeting in Amsterdam, August 25-26, 1978.

Both the above objections were paid due attention to in this draft. The model in [54] does not employ string-processing operations; rather, an upper bound on the number of characters required is determined in each case and a row of characters of that length is used thereafter. The efficiency of formatted transput is enhanced in two ways:

- pictures that do not contain any dynamic replicators (i.e., replicators which are to be elaborated at run-time) are handled in a different, more efficient, way.

- redundant information in the form of default values for insertions, replicators and the like is no longer stored explicitly in the various data structures that are used inside the formatted-transput routines. This saves both space and time.

The technical contents of the model as described in [54] is accepted at this meeting. A number of minor changes is agreed upon and still have to be

incorporated. (For example, the mode CHANNEL is defined in [54] as

STRUCT(INT ? channel number).

Since to each channel a number of properties is attached, the above definition directs one towards some specific implementation techniques to retrieve these properties. It was felt desirable to give the implementer more freedom in this respect. Therefore, it was agreed to define the mode CHANNEL by means of a pseudo-comment.)

Quite some experience had been gained in implementing the model. H. Ehlich & H. Wupper from the Ruhr-University at Bochum (FRG) had implemented the model as described in [53] on a TR440. Their experiences are described in [55]. This implementation provided fruitful feedback, especially as regards the implementability of the primitives underlying the model. Work was also progressing on an implementation under the CDC NOS/BE operating system at the Mathematical Centre (J.C. van Vliet, H. de J. Laia Lopes). All routine texts were thoroughly tested using the CDC ALGOL 68 implementation [56].

Other related topics discussed at the Amsterdam meeting include:

- interactive terminals, such that the terminal can be modelled as a single ALGOL 68 book rather than separate input and output books;

- provision of a simple means of producing overprinted characters to produce, for example, characters not present in the standard set available at the installation;

- "record input-output" which includes the indexed sequential facilities commonly used at commercial installations;

Another draft of the implementation model [57] is submitted to the Task Force at its meeting in Cambridge, U.K., December 18-19, 1978. The main difference between the models in [54] and [57] is the description of the basic model, viz., the buffer and its primitives. Much attention is paid to make this part as clear and understandable as possible. The resulting model is accepted by the Task Force, and the next version, in which the text is

once more scrutinized, was presented to the Task Force and to the Subcommittee on ALGOL 68 Support for discussion at their meetings in Summit, New Jersey, April 3-4 and April 4-6, 1979 [58].

This version was, together with a few minor changes, formally accepted by the Subcommittee at its meeting in Summit. The Task Force on Transput was subsequently discharged, its charter of reasonably interpreting the transput section of the Revised Report having been fulfilled.

At the subsequent meeting of the Working Group in Summit, April 6-10, 1979, the latest version of the implementation model was accepted and the Working Group recommended to TC 2 to authorize the following statement for publication with the model:

> "This implementation model of the ALGOL 68 transput has been reviewed by IFIP Working Group 2.1. It has been scrutinized to ensure that it correctly interprets the transput as defined in section 10.3 of the Revised Report. This model is recommended as the basis for actual implementations of the transput."

The related topics mentioned above (interactive terminal i/o and the like) were again discussed in detail in Cambridge and Summit. No convincing solutions to these problems could be found; further study was felt to be needed. Moreover, most of the active members of the Task Force had no wide experience in these particular fields. The implementation model presented to the Subcommittee on ALGOL 68 Support did not attempt to present solutions to these areas.

3.2. FUNDAMENTALS OF THE IMPLEMENTATION MODEL

3.2.1. DESIGN CRITERIA

One of the most basic requirements which must be fulfilled before one starts constructing a piece of software is a very clear and complete description of the problem that is to be solved. Critical problems stemming from a lack of a good requirements specification include [59]:

- top-down design is impossible, for lack of a well-specified "top";
- testing is impossible, because there is nothing to test against;
- management is not in control, as there is no clear statement of what is being produced.

Such a complete specification was by no means available when one started writing the revised transput sections, as can be seen from the historic review given in the previous chapter. In my opinion, this is one of the prime reasons for its impracticability.

When I started working on the implementation model, I considered it my first task to destill such a complete description from the definition as given in [12]. In doing so, a number of problems were discovered. These were reported to the Subcommittee on ALGOL 68 Support [43] and subsequently solved by the Task Force on Transput [50]. The results have been incorporated in the problem definition. In particular, the orthogonality of various parts of the transput has been increased (see for example Commentaries 10 and 27 from [50]).

The problem definition is partly implicit in the implementation model. The crucial parts, however, have been given ample exposition in the model given in part II (such as the notion of a position, the default actions taken when an event routine is called and the precise initialization of file variables upon opening).

A first prime requirement of the implementation model was that it should not deviate from the Revised Report (as modified by the Commentaries [50]). Such deviations would most certainly lead to portability problems; each implementer would be tempted to deviate even further, which would result in completely different transput implementations. It has undoubtedly also precluded endless discussions on which changes would be most valuable.

It has been clear from the start that the implementation model should fit very diverse operating systems, i.e., that the larger part of it should be easily portable. It is extremely important that such a question be raised at an early stage of the development. Experience has shown that, if portability is not considered beforehand, it may be easier to rewrite the whole system for a new machine or new operating system rather than to try to

modify the old one [60]. The amount of work needed to rewrite the whole transput is estimated to be considerable [40].

Other prime requirements were that the model should be precise and understandable. Various mechanisms have been employed to achieve these aims. First of all, the increase in orthogonality has contributed considerably to the clarity and ease of understanding. Much attention has been paid to the pragmatic descriptions. Invariant assertions have been chosen carefully and the validity of these assertions is ensured at crucial spots. For many routines, pre- and postconditions have been chosen with great care.

3.2.2. DESIGN METHODOLOGY

Software design is still almost completely a manual process. It is mostly done bottom-up, where software components are developed before interface and integration issues are addressed. I have tried to design the implementation model in a top-down fashion.

The operating-system interface has been designed first. The design of the rest of the transput together with the data structures to be used was completed before coding started. After this initial design, only two major constituents had to be overhauled. These correspond to the two objections raised against an early version of the model [53] at the meeting of the Task Force in Oxford, December 1977. As a consequence of one of these objections, viz., the inefficiency of formatted transput, the corresponding part has been completely redesigned and rewritten. The other objection, viz., the excessive use of string-processing operations, has been accommodated by adjusting the program texts at appropriate places. One may argue that, because of this, the numerical interface of the model does not fit the program texts as neatly as one might wish.

Just like jump removal and neat layout do not automatically lead to structured and well-understandable programs, the above techniques do not automatically result in better programs. "Good programming is not learned from generalities, but by seeing how significant programs can be made clean, easy to read, easy to maintain and modify, human engineered, efficient and reliable, by the application of common sense and good programming practices." [61]. The construction of good programs cannot yet be viewed as

an "engineering" activity, whereby the proper application of simple rules almost automatically leads to the desired object. There is a tendency to not just give the principles of "structured programming", "top-down design", "egoless programming" in textbooks on programming. Rather, good programming practices are becoming illustrated by giving examples of by no means trivial programs [61, 62, 63].

There is at least one reasonable ground to support the claim that the implementation model given in part II meets the standards of clarity, understandability, and the like: one, non-professional, programmer succeeded in implementing the transput model within 4 months, solely from the description of an earlier version [53]. This concerned an implementation on a TR440, which is a machine I am not familiar with, let alone that the model was tuned to this particular machine or its operating system.

3.2.3. DESIGN OF THE MODEL

The model given in part II is largely written in ALGOL 68. As a result, it could be tested thoroughly using the CDC ALGOL 68 implementation [56]. Also, the reliability is increased because of the powerful mechanisms ALGOL 68 provides to ensure security (mode checks and the like).

At a number of places in the program texts, pseudo comments appear. These indicate that the corresponding actions cannot be properly expressed in ALGOL 68. (Many of these actions could be described in ALGOL 68, but to do so would make too many assumptions about the underlying machine, thereby making the model unnecessarily restrictive.) The pseudo comments are mainly of two kinds:

i) those that relate to the operating-system interface: books, channels and buffers, together with routines to cope with them;
ii) primitives that involve real arithmetic.

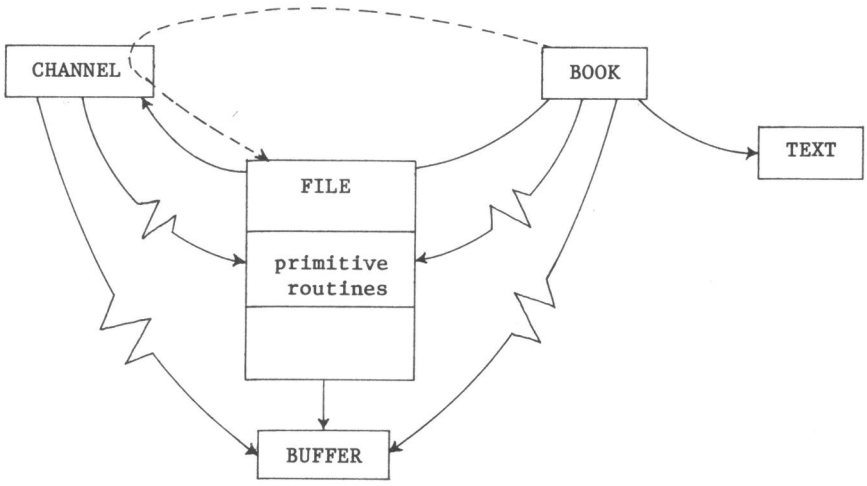
Various other miscellaneous comments (like the definitions of the modes INTYPE and OUTTYPE and some special generators in the formatted-transput sections) will not be further discussed here. These will sometimes necessitate special provisions in the compiler.

3.2.3.1. OPERATING-SYSTEM INTERFACE

The most central part of the operating-system interface is the buffer. At some stage during the process of transput data must be passed to or from the operating system. It is most natural to pose some black box in between. Both the transput system and the operating system can communicate, in a very restricted way, with this black box; they do not directly communicate with each other. This black box is called the "buffer". The buffer can contain any number of characters. The number of characters it may contain (the "length" of the buffer) may vary from case to case; it is determined upon opening. In the model given in part II, the length of the buffer is taken to be the length of the line, though this is not very essential. The various primitives associated with the buffer are defined with this notion of length in mind. The changes to be made to the primitives in the case where the buffer does not correspond to one line of the text are described in a special section.

Primitive routines are defined to write or read one single character to or from the buffer (this is the way the transput system communicates with the buffer), and to write or read complete buffers (the communication with the operating system).

The relations between files, channels and books are as follows:



An arrow A ⟶ B indicates that A contains a reference to B; A ⇝ B indicates that A (partly) determines B. The broken line which connects the book and the file indicates that, after opening, the book is linked with the file via some channel (the book has been opened on the file via some channel). Note that, in general, a channel may be used to open many files, and that a book may be opened on many files simultaneously.

The precise structure of the buffer (i.e., its ALGOL 68 mode) is not specified. It is tempting to define its mode as being REF [] CHAR. This would make things simpler on the ALGOL 68 side and offer possibilities for optimization (it becomes possible to move sequences of characters in one statement, the transput system may directly write into, or read from, the buffer). On the other hand, it would also unnecessarily restrict implementations. It may for instance become impossible to incorporate control characters in the buffer, which then have to be padded to it when the buffer is ultimately moved. It may also prevent implementation of obscure conversion keys.

The flexibility aimed at when this part of the operating-system interface was being defined, is essential. This flexibility is a fundamental property of the model. It is achieved by isolating certain properties (the mode and length of the buffer, the type of conversion, etc.) so that changes in those decisions do not transgress the boundaries of the primitives that embody them and by providing fixed interfaces for those primitives. The ability to defer decisions concerning the implementation of these primitives is essential in obtaining flexibility [65].

3.2.3.2. REAL ARITHMETIC

Real arithmetic is a notoriously troublesome area. It is hard to find anything common between different machines in this area. Moreover, transput makes critical use of it. If numbers are printed or read, this should be done as accurately as possible. There is no chance to achieve such accuracy when the corresponding algorithms are expressed in ALGOL 68. Therefore, the model in part II does not contain any real arithmetic; rather, it is all delegated to a few primitives which perform the necessary conversions from numbers to sequences of characters and vice versa.

One specific troublespot is rounding, which for instance occurs when numbers are to be output. This is especially true for very large numbers, since these may cause overflow problems. To circumvent this problem, I have taken a very strict line: the primitives do not perform any rounding. Numbers are first converted to a sequence of characters of sufficient length, and the rounding is performed in ALGOL 68 on this character sequence.

3.2.3.3. EFFICIENCY AND CLARITY

As has been emphasized before, the inefficiency of the transput part of the Revised Report, if the text is taken literally, makes it impracticable. I have tried to improve on this. Various methods have been employed to achieve reasonable efficiency. It is not surprising that the techniques used to improve clarity and understandability of the system have had a positive effect on its efficiency as well, and vice versa.

Efficiency is seriously affected by the chain of tests that is to be performed at each transput call. The same tests must also be repeated after intervention of the user through the calling of an event routine. Most of these tests are performed by calling one of the "ensure" routines (these are to be found in Chapter 7 of the implementation model). Appropriate pre- and postconditions for these routines have been established carefully, so that calling such a routine can often be circumvented by first checking its postcondition.

Many of the tests are boolean tests: either the line is ended, or it is not. All these booleans are collected in the status of the file. In the model given in part II, the status is defined to be of the mode specified by BITS, but this may easily be changed by the implementer (the only affected definitions are in 6.2.d, e and f of the model). The introduction of the mode specified by STATUS, the operators SAYS and SUGGESTS and a number of named constants of the mode specified by STATUS have provided a very powerful aid. Not only is clarity greatly enhanced by statements of the form

```
IF status SAYS write sequential
THEN ...
```

(one need not look up the definitions to see what is meant, since this is immediately clear from the statement at hand), but efficient implementation can easily be obtained as well.

The operators SAYS and SUGGESTS, as defined in 6.2.e of part II, deliver a boolean value. These operators are mostly applied in simple if statements like the one given above. In those cases, there is no need to actually construct a value of the mode specified by BOOL. The operators may simply compare their two operands (for instance by some mask instruction) and the result may subsequently be used directly by the if statement.

Furthermore, by carefully arranging these tests, the normal situation can often be detected by one single test. For example, the call 'put(f, "a")' will now lead to the following chain of tests and procedure calls:

```
put(f, "a")
-> status SAYS put char status
   status SAYS line ok
   put char(f, k)
   -> (write char OF f)(f, char)
      c OF cpos OF cover > char bound OF cover.
```

The first check is performed to see whether the file is opened, and whether one is currently writing characters (as opposed to binary transput). This test will usually succeed, unless one is frequently alternating reading and writing, or character and binary transput.

The third line tests whether the current position is "good", i.e., whether another character fits on the same line. This test will fail once in a while, viz., when the line has overflowed, after which one of the ensure routines is called which finds a good position on a subsequent line. The postcondition of this ensure routine is such that it has indeed found a good position.

Finally, after the character has been written to the buffer (which is taken care of by calling the primitive routine 'write char'), it is checked whether the line has now overflowed. If so, the status is updated such that,

e.g., a subsequent test for the line being ok will fail.

The above chain of tests and actions seems to be the least one required to write one single character.

Efficiency is of course most important for those parts of the transput system that are used most heavily. Presumably, numbers will be output more often than boolean values. Care has been taken to make those parts of the ALGOL 68 text as efficient as possible. Also, suggestions to improve the efficiency through in-line expansion are included at crucial places. By implementing the model as given in part II and collecting statistics on how it is being used, more places for suitable optimizations may well be found.

It is to be emphasized that optimizations and the resulting complexity are programming vices. It is much more important for complex systems such as the ALGOL 68 transput to be transparantly simple and self-evidently correct. Optimizations tend to make a system less reliable and harder to maintain; optimizations also obscure [66]. Therefore, where there were conflicts between clarity and efficiency, clarity always took precedence.

3.3. EVALUATION

The design and implementation of the transput model has learned that, despite the complexity of the ALGOL 68 transput, an efficient and easily portable implementation is well possible. The underlying primitives are fairly simple and straightforwardly implementable. The language ALGOL 68 turned out to be well suited to describe a rather complicated piece of software in. The resulting program texts are in general straightforward and well-readable.

While studying the transput sections of the Revised Report and during the design of the implementation model, the question of how to improve the ALGOL 68 transput reared its head from time to time. It is tempting to think that the ALGOL 68 transput can be improved considerably by massaging its troublespots. Though it may be true that the transput becomes more useful and easier to implement when these deficiencies are removed, I do not think this will lead to fundamental improvements.

A much better transput system can only be obtained if it is designed from scratch again. To start with, a thorough study of operating-system facilities and user requirements would be needed. If such is done, I expect the resulting system to be completely different from the present one. For example, some kind of record i/o (just think of the many possible database applications) is likely to form a central part of such a new system. The present operating-system interface may well be too simple for such applications.

It has not been the aim of this treatise to explore possible ways to arrive at such a new transput system, although future research may well profit from the results reported on. The prime goal of this study has been to aid ALGOL 68 implementers, and as such it is of extreme importance to stick to the definition given in the Revised Report as closely as possible. Uniformity in language implementations is a fundamental issue. This is the more true for the transput part, since this has immediate repercussions when programs are to be ported.

4. ABOUT THE IMPLEMENTATION ON THE CYBER

The implementation model given in part II has been thoroughly tested using the CDC ALGOL 68 implementation [56]. This implementation covers almost the complete language. Some small changes to the ALGOL 68 text had to be made in order to cope with the deviations of this implementation (e.g., VOID is not allowed as constituent of a united mode and flexibility is not part of the mode).

In the next sections, two interesting aspects of the implementation are discussed in some detail. First, the basic interface with the operating system NOS/BE [67] is described. This discussion centers around the implementation of the buffer concept. Next, the primitives involving real arithmetic are described.

Using the CDC ALGOL 68 implementation, two mechanisms can be employed to handle non-ALGOL 68 parts:

- operators can be defined by ICF macros and compiled in-line. (ICF stands for Intermediate Code File. This is the intermediate code used in the CDC ALGOL 68 implementation. There does not exist an official definition of ICF.) The definition of such an operator takes the following form:

```
OP <formal part>    <operator> =
PR inline
<ICF macros>
PR SKIP.
```

- operators or procedure names can be defined as external by a declaration such as:

```
PROC <identifier> = <formal part>:
PR xref <naming> PR SKIP.
```

"naming" is the entry point for the given routine. This routine may then for instance be written in COMPASS, the assembly language of the Cyber [68].

Both mechanisms have been used to implement the pseudo-comments of the implementation model. Some of the more special constructs, such as the modes INTYPE and OUTTYPE, are handled in the same way as they are handled in the CDC implementation. The solutions to these problems are very implementation-dependent and are not further dealt with below.

In the sections below, I have tried to concentrate on the general ideas behind the implementation. Readers who are not familiar with the machine characteristics of the Cyber should be able to grasp the essential ideas. No specific sequences of machine-code instructions are given.

In implementing and testing the various primitives, I have benefitted much from the CDC ALGOL 68 implementation. Where possible, the strategies used in the CDC transput implementation were employed. The following sections therefore do not contain many new ideas.

4.1. IMPLEMENTING THE BASIC MODEL

The most central concept of the implementation model is the "buffer". Together with the book and channel, and the various primitives to cope with these, it forms the basic interface with the operating-system environment. The main elements of its implementation under the NOS/BE operating system [67] are outlined below. The discussion below centers around the complexity of this implementation.

The basic interface has been implemented using the "Record Manager" [69]. The Record Manager (RM) is a group of routines that provide an interface between a program and the system routines that read and write files on hardware devices.

The reasons for using the RM are the following:

- most other compilers on the system also use the RM, so that it is easy to manipulate files created by programs written in another language, and vice versa;

- it becomes easier to accommodate with changes at the operating-system level, since the RM is maintained by the manufacturer (see [70] for an

example of the disastrous effects that may result from sudden changes at the operating-system level in the case where one used one's own interface);

- the CDC ALGOL 68 implementation also uses the RM, so that parts of their implementation could profitably be used.

There are also some serious disadvantages to using the RM. It is a very baroque tool, mainly designed to interface FORTRAN and COBOL programs; most of its features do not pertain to the simple ALGOL 68 interface. Partly as a result of this, calls to RM routines are very time-consuming.

In the sequel, I will not dwell on implementation details. Rather, the mapping of the primitives used in the implementation model onto RM features will be discussed in global terms.

In RM terms, the buffer is called "Working Storage Area" (WSA). There also exists a second-level buffer behind the scenes, the so-called "circular buffer". Reading from the circular buffer into the WSA is done by calling the RM routine "get". To write the contents of the WSA to the circular buffer, the RM routine "put" is called. Reading and writing circular buffers is done automatically by the RM.

The RM administration of the circular buffer and the WSA is contained in two tables: the File Information Table (FIT) and the File Environment Table (FET). The FIT contains, for example, the logical file name (which in this implementation equals the identification string), the record size (which is the maximum length of a line), and information on the blocking structure. The current contents of the FIT defines the way the file is processed at any given time. (In this section, the term "file" refers to the external data, and has nothing to do with the ALGOL 68 notion of a file.)

The FET is created and maintained by the RM in response to user statements for input or output. It contains for example information on where to read or write the next WSA to or from the circular buffer. The user need not be concerned with this table, and he does better not to manipulate its fields.

The discussion below will concentrate on files containing Z-type records. To put it simply, those are the files that can be output to a line printer. These files are sequential, and each record constitutes one line. The end of a record is indicated by two zero bytes (hence the name Z-type). The records may contain control characters. Conversion keys are not dealt with below.

On such files, 10 characters are packed into one machine word. When the RM routine "get" is called, successive words are copied from the circular buffer to the WSA until one is found which contains the special encoding indicating the end of the line (= record). The encoding itself is not copied. Instead, the WSA is filled with spaces from the last actual character read until the end. The number of characters read (rounded upwards to a multiple of 10) is stored in the FIT. In this way, the exact number of characters on the line is not known. This quantity can only be determined by inspecting the circular buffer itself. Reading single characters from the WSA is now relatively simple: knowing how many control characters appear at the start of a line and the value of the current position, the next character can be retrieved by simple shift and mask instructions.

On output, the reverse scheme is followed. Outputting control characters is a bit tedious. These control characters have to be saved until the WSA is re-initialized by a subsequent call of 'init buffer'. (They are stored in the book when one of the routines 'newline' or 'newpage' is called and subsequently retrieved by 'init buffer'.) Again, writing single characters is very simple. The contents of the WSA is written to the circular buffer when the RM routine "put" is called. "put" may be given the number of characters to be written as parameter; by default, the complete WSA is output. "put" itself adds the special encoding to indicate the end of the line.

Different kinds of files can be accommodated by providing different channels. To this end, the channel contains part of the FIT, and encodings which determine the number of characters packed into one machine word and the number of control characters that appear at the start of a line.

The complete FIT and FET are contained in the book. They are initialized when the book is searched for ('find book in system', 4.1.2.e) or

constructed ('construct book', 4.1.2.f). The contents of the FIT is completely determined by the actual parameters given to these routines, i.e., part of the FIT is copied from the channel, and the remainder is determined by the other parameters. The FET is subsequently initialized by the RM when the RM routine "openm" is called. This routine opens the file if it already exists, and creates an empty file otherwise. Both 'find book in system' and 'construct book' allocate space for the circular buffer, WSA, FIT and FET.

It follows from the above discussion that implementing the basic model using the RM is relatively easy. The global concepts fit well, though some details are rather difficult to implement correctly. One of the most difficult parts of the implementation has been to decide which additional information is to be incorporated in the book and the channel. The routines 'find book in system' and 'construct book', which perform the initialization of the FIT, FET, and the like, are rather critical. Further implementation of the primitives to read and write single characters or complete buffers turned out to be relatively simple and straightforward.

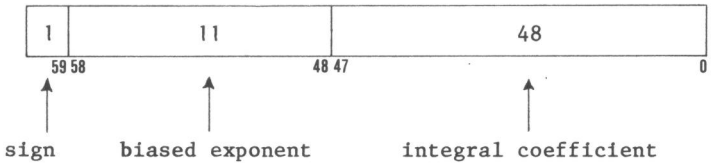
4.2. IMPLEMENTING REAL ARITHMETIC

The main problem encountered when the primitives involving real arithmetic (the conversion of numbers to sequences of characters and vice versa) are to be implemented is to achieve the highest possible accuracy. When the accuracy of numerical algorithms is to be measured, the measurements should not be downgraded by the inaccuracy of the conversion.

When some number (integral or real) is to be output, it must at some stage be converted to a sequence of digits. Outputting an integer as just a sequence of digits is described in ALGOL 68 proper (it does not involve any real arithmetic). Outputting a number in floating-point or fixed-point form (i.e., with or without an exponent part) does involve real arithmetic. It is the routine 'subfixed' (see 8.2.f of the implementation model for a precise description of its semantics) which takes care of the conversion in this case. To put it very simple, 'subfixed' delivers n digits, where n is somehow determined by the parameters of the call. 'subfixed' does not perform any rounding.

At the machine level, the number of real operations needed to extract these digits should be kept minimal so as to ensure the highest precision. If successive digits are extracted using real operations, the last few digits returned may well be meaningless. Therefore, the number is first converted to an integer and successive digits are thereafter extracted from this integer. This will give very accurate results, since integral arithmetic is exact.

On the Cyber 73, single-precision floating-point numbers are represented internally in a 60-bit word consisting of a sign bit, an 11-bit biased exponent and a 48-bit integral coefficient:



The 48-bit integral coefficient corresponds to approximately 14 decimal digits. The 11-bit exponent carries a bias of 2^{10} (2000_8). This bias is always assumed to be implicitly dealt with in the discussion below. If normalized, the highest order position of the coefficient portion of the word (bit 47) equals the sign bit. In this way, the numbers that can be represented are approximately in the range $[10^{-293}, 10^{322}]$. (The decimal point is on the right of the integral coefficient.)

Negative numbers are represented in one's complement notation. In the sequel, I will always assume that the number to be converted is positive, i.e., bit 59 equals 0. The case where the number to be converted is given in single precision will be treated first.

ALGOL 68 integers range up to $2^{48}-1$; the exponent portion of the word is not used. (Although the machine's add and subtract operations operate on 60 bits, the multiply and divide operations only use the lower-order 48 bits.) Conversion from integers to floating-point numbers is relatively simple. It only involves setting the appropriate exponent, followed by a normalizing instruction (which is a left shift in order to place the most significant digit of the coefficient in the highest-order position of the coefficient portion of the word).

Given a positive, single-precision floating-point number V to be converted, the number of digits in front of the decimal point (i.e., the length of the integral part) must be determined. This number, say A, is needed to determine the number of digits to be returned in the case where 'subfixed' is used to convert a value into fixed-point form, or to determine the exponent to be returned if it concerns a conversion into floating-point form.

Obviously, A = entier($\log_{10}$V) + 1. (Throughout, the case V=0 must be treated separately.) The value of A can easily be approximated from the binary exponent of the floating-point representation of V, as is already indicated in 8.2.g of the implementation model. One may just take

$$A' = \text{ENTIER}(\log_{10}2 * (E + 1)) + 1,$$

where E is the, properly scaled, binary exponent (E = ENTIER($\log_2$V)). Note that A' may be one larger than the proper value of A.

Converting a floating-point number X to an integer can be done very easily on the Cyber if X is such that

$$0.1 \leq X < 1.0.$$

Therefore, V is first scaled by multiplying it with a proper power of 10. Since this will cost some precision if done in single precision, one needs a few additional bits. The simplest way to do this is to use the machine's double-precision arithmetic.

Rather than using one huge table of powers of 10, two tables are used. The first table contains powers of 10.0 (10.0^0, ..., 10.0^{15}), the second one contains powers of 10.0^{16}. (This is a nice compromise: 16 is a power of 2, so that the exponent (being A') can be split up using shift and mask operations only, while the table containing the numbers 1.0 up to 10.0^{15} can still be represented exactly in single precision. The second table is represented in double precision.) By using at most two multiplications, V is thus multiplied by $10^{-A'}$. Since A' may be one too large, the result may be less than 0.1. In that case, an extra multiplication with 10.0 is needed, and A' is decreased by 1 so as to get the proper value of A.

As a result of this, a double-precision floating-point number is obtained which looks as follows:

(The decimal point is in between the two parts of the number.) The value of X, being the scaled value of V, now is:

$$2^{EXP1} * M1 + 2^{EXP2} * M2, \text{ where } EXP2 = EXP1 - 48.$$

This double-precision floating-point number X must now be converted to an integer from which the successive digits are extracted. This part is rather interesting and will therefore be elaborated in some detail.

Extracting digits is very easy if the integer is stored as follows:

The 12 higher-order bits are 0 in both words. The value of the integer is $2^{48} * N1 + N2$, so that it relates to the value X through the formula

$$X = 2^{-96} * (2^{48} * N1 + N2).$$

If this integer is multiplied by 10 (by shifting and adding, but now over 60 bits), the result becomes:

The first digit I is in bits 48-50 of the high-order word. It may subsequently be extracted by a right shift over 48 bits of this word. To continue the search for the next digit, this exponent portion must be made 0 again. This is extremely simple: the machine's unpack operation, applied to the high-order word, will do. If implemented this way, no precision will be lost if successive digits are extracted.

As said above, the integer should be such that

$$X = 2^{-96} * (2^{48} * N1 + N2) \{= 2^{-48} * N1 + 2^{-96} * N2\}.$$

After scaling, the value of X is $2^{EXP1} * M1 + 2^{EXP2} * M2$, where EXP2 = EXP1 - 48. If EXP1 were -48, the integer could be obtained simply by unpacking both parts of the double-precision floating-point number (unpacking a floating-point number just sets the exponent portion of the word equal to 0).

Since X is normalized, bit 47 of M1 equals 1, so $2^{47} \leq M1 < 2^{48}$. X is approximated by $2^{EXP1} * M1$, and $0.1 \leq X < 1.0$. From this, we conclude that $-50 \leq EXP1 \leq -48$. Adjusting EXP1 so that it always equals -48 can be done easily by adding a number B to X, such that the integral coefficient of B equals 0, while the exponent equals -48. (The floating-point add operation first adjusts the value with the smaller exponent such that both exponents are the same, and then performs the actual add operation. The result is not normalized again.) As a result of this adjustment (which just amounts to a right shift of the coefficient portion and a proper adjustment of the exponent portion), at most two of the low-order bits of the lower-order word may get lost.

To sum things up, there are at most 4 operations that may lead to a loss of precision: one to scale EXP1 as described in the above paragraph, and at most 3 to scale V by multiplications with powers of 10. These operations will only affect the lower-order word of the double-precision floating-point number, so the results will be very accurate.

In case one has to convert double-precision numbers, one again needs a few extra bits to ensure the maximum possible accuracy. The tables with powers of 10 will get larger and also have to be represented with more precision. The multiplications with those powers of 10 will then be the hardest part. One must take care also that a few extra bits are always needed in the ultimate integer.

When some sequence of digits is to be converted to a real number, the routine 'string to real' (8.2.o of the implementation model) is used. Converting a sequence of digits to an integer is taken care of in ALGOL 68

proper again. Below, only the single-precision case is described. The double-precision case needs some obvious adjustments.

The precondition of 'string to real' is such that at most 'real width + 1' significant digits are supplied. As a first step, this sequence is converted to an integer exactly representing this sequence. Since 'real width' equals 16 on the CDC ALGOL 68 implementation [56], the maximum integer that may result is $10^{18} - 1$, which can still be represented in one 60-bit word (although it may well exceed 'max int', which equals $2^{48} - 1 < 10^{15}$).

Without loss of precision, this integer may be converted to a double-precision floating-point number (the exponent portions just have to be set properly and the number must be normalized). The next step is to multiply this value by 10^{EXP}, where EXP is given as parameter. Similar to the case of 'subfixed', this multiplication is done in at most 2 steps. It does involve some nasty tests on underflow/overflow. The resulting value still has to be rounded (upwards) to get the desired single-precision number (this necessitates another underflow/overflow test). Loss of precision can only be caused by the at most 2 multiplications by powers of 10. Again, these will only affect the lower-order word, so that maximum accuracy is ensured.

REFERENCES

[1] WIJNGAARDEN, A. VAN & B.J. MAILLOUX, A Draft Proposal for the Algorithmic Language ALGOL X, WG 2.1 Working Paper, October 1966.

[2] WIJNGAARDEN, A. VAN, B.J. MAILLOUX & J.E.L. PECK, A Draft Proposal for the Algorithmic Language ALGOL 67, Mathematisch Centrum, Amsterdam, MR88, May 1967.

[3] WIJNGAARDEN, A. VAN, B.J. MAILLOUX & J.E.L. PECK, A Draft Proposal for the Algorithmic Language ALGOL 68, Mathematisch Centrum, Amsterdam, MR92, November 1967.

[4] WIJNGAARDEN, A. VAN (Ed.), B.J. MAILLOUX, J.E.L. PECK & C.H.A. KOSTER, Draft Report on the Algorithmic Language ALGOL 68, Mathematisch Centrum, Amsterdam, MR93, January 1968.

[5] WIJNGAARDEN, A. VAN (Ed.), B.J. MAILLOUX, J.E.L. PECK & C.H.A. KOSTER, Working Document on the Algorithmic Language ALGOL 68, Mathematisch Centrum, Amsterdam, MR95, July 1968.

[6] WIJNGAARDEN, A. VAN (Ed.), B.J. MAILLOUX, J.E.L. PECK & C.H.A. KOSTER, Penultimate Draft Report on the Algorithmic Language ALGOL 68, Mathematisch Centrum, Amsterdam, MR99, October 1968.

[7] WIJNGAARDEN, A. VAN (Ed.), B.J. MAILLOUX, J.E.L. PECK & C.H.A. KOSTER, Final Draft Report on the Algorithmic Language ALGOL 68, Mathematisch Centrum, Amsterdam, MR100, December 1968.

[8] WIJNGAARDEN, A. VAN (Ed.), B.J. MAILLOUX, J.E.L. PECK & C.H.A. KOSTER, Report on the Algorithmic Language ALGOL 68, Mathematisch Centrum, Amsterdam, MR101, February 1969 (also in Numerische Mathematik 14 (1969), pp 79-218).

[9] WIJNGAARDEN, A. VAN, B.J. MAILLOUX, J.E.L. PECK, C.H.A. KOSTER, M. SINTZOFF & C.H. LINDSEY (Eds.), Draft Revised Report on the Algorithmic Language ALGOL 68, WG 2.1 Working Paper, February 1973.

[10] WIJNGAARDEN, A. VAN, B.J. MAILLOUX, J.E.L. PECK, C.H.A. KOSTER, M. SINTZOFF & C.H. LINDSEY (Eds.), with the assistance of L.G.L.T. MEERTENS & R.G. FISKER, Draft Revised Report on the Algorithmic Language ALGOL 68, WG 2.1 Working Paper, July 1973.

[11] WIJNGAARDEN, A. VAN, B.J. MAILLOUX, J.E.L. PECK, C.H.A. KOSTER, M. SINTZOFF, C.H. LINDSEY, L.G.L.T. MEERTENS & R.G. FISKER (Eds.), Revised Report on the Algorithmic Language ALGOL 68, Technical Report TR 74-3, Department of Computing Science, University of Alberta, March 1974 (Supplement to ALGOL Bulletin 36).

[12] WIJNGAARDEN, A. VAN, B.J. MAILLOUX, J.E.L. PECK, C.H.A. KOSTER, M. SINTZOFF, C.H. LINDSEY, L.G.L.T. MEERTENS & R.G. FISKER (Eds.), Revised Report on the Algorithmic Language ALGOL 68, Acta Informatica 5 (1975) pp 1-236. (Also published in SIGPLAN Notices 12 no 5, pp 1-70, May 1977, and as Mathematical Centre Tracts No 50).

[13] POEL, W.L. VAN DER, Some notes on the history of ALGOL, in MC-25 Informatica Symposium, Mathematisch Centrum, Amsterdam, Tract 37 (1971).

[14] SINTZOFF, M., On the Revised ALGOL 68 Report, ALGOL Bulletin 36, pp 28-39, November 1973.

[15] KNUTH, D.E. et. al., A proposal for input-output conventions in ALGOL 60, CACM 7, pp 273-283, May 1964.

[16] IFIP, Report on input-output procedures for ALGOL 60, CACM 7, pp 628-630, October 1964.

[17] GARWICK, J.V., The question of I/O procedures, ALGOL Bulletin 19, pp 39-41, Januari 1965.

[18] WOODGER, M., ALGOL X. Note on the proposed successor to ALGOL 60, ALGOL Bulletin 22, pp 28-33, February 1966.

[19] GARWICK, J.V., J.M. MERNER, P.Z. INGERMAN & M. PAUL, Report of the ALGOL-X-I-O Subcommittee, WG2.1 Working Paper, July 1966.

[20] NAUR, P., Successes and failures of the ALGOL effort, ALGOL Bulletin 28, pp 58-62, July 1968.

[21] LINDSEY, C.H., Proposals for amendment of ALGOL 68, ALGOL Bulletin 28, pp 50-57, July 1968.

[22] HILL, I.D., Some remarks on the draft report, ALGOL Bulletin 28, pp 65-69, July 1968.

[23] PECK, J.E.L. (Ed.), Proceedings of an informal conference on ALGOL 68 implementation, August 29-30, 1969, Department of Computer Science, University of British Columbia, Canada, 1969.

[24] PECK, J.E.L. (Ed.), ALGOL 68 Implementation, Proceedings of the IFIP Working Conference on ALGOL 68 Implementation, Munich, July 20-24, 1970, North Holland Publishing Cy, Amsterdam, 1971.

[25] PAUL, M., IFIP WG2.1 - Activity Report no. 10, ALGOL Bulletin 32, pp 51-52, May 1971.

[26] IFIP Working Group 2.1 - Report of the Subcommittee on Data-processing and Transput, ALGOL Bulletin 32, pp 25-39, May 1971.

[27] WOODWARD, P.M. & S.G. BOND, ALGOL 68-R Users Guide, Royal Radar Establishment, Malvern, England, 1972.

[28] IFIP Working Group 2.1 - Report of the Subcommittee on Data-processing and Transput, ALGOL Bulletin 33, pp 26-42, March 1972.

[29] HUDEC, I.P., Features in ALGOL 68 for handling records, M.Sc. Thesis, University of Manchester, January 1974.

[30] LINDSEY, C.H., LQ216 - Various transput proposals, undated.

[31] IFIP Working Group 2.1 - Proposals for revision of the transput sections of the report (Fontainebleau 10), prepared by the Subcommittee on Data-processing and Transput, ALGOL Bulletin 34, pp 28-40, July 1972.

[32] IFIP Working Group 2.1 - Further report on improvements to ALGOL 68, ALGOL Bulletin 35, pp 5-13, March 1973.

[33] LINDSEY, C.H. & R.G. FISKER, Standard prelude - Transput routines, WG2.1 Working Paper, April 1973.

[34] MEERTENS, L.G.L.T., The transput section, Mathematisch Centrum, Amsterdam, March 1, 1974.

[35] MEERTENS, L.G.L.T., letter to C.H. LINDSEY & R.G. FISKER, December 10, 1973.

[36] GRUNE, D., L.G.L.T. MEERTENS, J.C. VAN VLIET & R. VAN VLIET, letter to the editors, March 18, 1974.

[37] GRUNE, D., L.G.L.T. MEERTENS, J.C. VAN VLIET & R. VAN VLIET, Remarks on the transput section of the revised report, Mathematisch Centrum, Amsterdam, IN7/74, April 1974.

[38] IFIP Working Group 2.1 - Subcommittee on ALGOL 68 Support (reported by P.R. KING), ALGOL Bulletin 37, pp 4-11, July 1974.

[39] FISKER, R.G., The transput section for the revised ALGOL 68 report, M.Sc. thesis, Department of Computer Science, University of Manchester, August 1974 (revised version).

[40] SINTZOFF, M., A brief review of ALGOL 68, ALGOL Bulletin 37, pp 54-62, July 1974.

[41] VLIET, J.C. VAN, Towards a machine-independent transput section, Proc. of the Strathclyde ALGOL 68 Conference, SIGPLAN Notices 12, no 6, pp 71-77, June 1977 (an extended version thereof is registered as IW73/77, Mathematisch Centrum, Amsterdam, January 1977).

[42] VLIET, J.C. VAN, On the ALGOL 68 transput conversion routines, ALGOL Bulletin 41, pp 10-24, July 1977.

[43] VLIET, J.C. VAN, Compilation of problems and errors in section 10.3 of the Revised Report, Mathematisch Centrum, Amsterdam, DWA 11, May 1977.

[44] VLIET, J.C. VAN, An implementation-oriented definition of the ALGOL 68 transput, Part I, II, Mathematisch Centrum, Amsterdam, July 1977.

[45] FISKER, R.G., Comments on "An implementation-oriented definition of the ALGOL 68 transput", University of Manchester, August 1977.

[46] LINDSEY, C.H., Commentary on the bugs reported by J.C. VAN VLIET, University of Manchester, July 1977.

[47] THOMSON, C.M., A description of the FLACC transput system, Chion Corporation, Edmonton, August 1977.

[48] THOMSON, C.M. & C.G. BROUGHTON, A description of a new transput system, Chion Corporation, Edmonton, August 1977.

[49] IFIP WG2.1 - Subcommittee on ALGOL 68 Support, Minutes of the Subcommittee meeting in Kingston, Ontario, August 1977.

[50] IFIP WG2.1 - Subcommittee on ALGOL 68 Support, Commentaries on the Revised Report, ALGOL Bulletin 43, pp 6-18, December 1978.

[51] LINDSEY, C.H., ALGOL 68 and your friendly neighbourhood operating system, ALGOL Bulletin 42, pp 22-35, May 1978.

[52] FISKER, R.G., Manchester University portable ALGOL 68 implementation, University of Manchester, July 1977.

[53] VLIET, J.C. VAN, Towards an implementation-oriented definition of the ALGOL 68 transput, Mathematisch Centrum, Amsterdam, IW90/77, October 1977.

[54] VLIET, J.C. VAN, An implementation model of the ALGOL 68 transput (Draft version), Mathematisch Centrum, Amsterdam, IN15/78, July 1978.

[55] EHLICH, H. & H. WUPPER, Experiences with ALGOL 68 transput and its implementation, Arbeitsberichte des Rechenzentrums der Ruhr-Universitaet Bochum, ISSN 0341-0358, Nr 7901, 1979.

[56] Control Data Corporation, ALGOL 68 Version I Reference Manual, Control Data Services BV, Rijswijk, The Netherlands, 1976.

[57] VLIET, J.C. VAN, An implementation model of the ALGOL 68 transput, Part I, II, Working Document Transput Task Force meeting, Mathematisch Centrum, Amsterdam, December 1978.

[58] VLIET, J.C. VAN, An implementation model of the ALGOL 68 transput, Working Document Subcommittee on ALGOL 68 Support, Mathematisch Centrum, Amsterdam, March 1979.

[59] BOEHM, B.W., Software engineering, IEEE Transactions on Computers, Vol. C25, no 12, December 1976.

[60] PECK, J.E.L. & G.F. SCHRACK, The portability of quality software, in P.G. HIBBARD & S.A. SCHUMAN (Eds.), Constructing Quality Software, North Holland Publishing Cy, 1978.

[61] KERNIGHAN, B.W. & P.J. PLAUGER, Software Tools, Addison Wesley Publishing Cy, 1976.

[62] WULF, W.A., et. al., The Design of an Optimizing Compiler, Programming Languages Series 2, American Elsevier Publishing Cy, 1975.

[63] BRINCH HANSEN, P., The Architecture of Concurrent Programs, Prentice Hall Series in Automatic Computation, Prentice Hall, 1977.

[64] KERNIGHAN, B.W. & P.J. PLAUGER, The Elements of Programming Style, Mc.Graw-Hill Book Cy, 1974.

[65] WASSERMAN, A.I. & L.A. BELADY, Software Engineering: The Turning Point, Computer 11, no 9, pp 30-41, September 1978.

[66] JACKSON, M.A., Principles of Program Design, APIC Studies in Data Processing no 12, Academic Press, 1975.

[67] Control Data Corporation, NOS/BE 1 Reference Manual, Publication No 60493800, 1977.

[68] Control Data Corporation, COMPASS Version 3 Reference Manual, Publication No 60492600, 1977.

[69] Control Data Corporation, Cyber Record Manager Version 1 User's Guide, Publication No 60359600, 1978.

[70] GRUNE, D., Experiences with porting the ALEPH-compiler from SCOPE 3.4.1 to SCOPE 3.4.4, or the software crisis rages, IN12/76, Mathematisch Centrum, Amsterdam, October 1976 (in Dutch).

[71] NICHOLLS, J.E., The Structure and Design of Programming Languages, The Systems Programming Series, Addison-Wesley Publishing Cy, 1975.

[72 CARROLL, J.G., A comparison of the formatted transput of ALGOL 68, PL/I, and FORTRAN, M.Sc. Thesis, Oklahoma State University, Durant, Oklahoma, December 1978.

PART II

AN IMPLEMENTATION MODEL

Table of contents

1. INTRODUCTION

This report aims at a precise description 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 transfer 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 definition 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 FISKER [11] have been working on transput systems of a similar structure.

The present report is written on request of the Task Force on Transput, which has been set up by the Subcommittee on ALGOL 68 Support of Working Group 2.1 of IFIP. A preliminary version appeared in [12]; that version was discussed at the first meeting of the Task Force in Oxford, December 14-15, 1977. The main objections against [12] concerned its efficiency (most notably in string processing and formatted transput). At the next meeting of the Task Force, which took place in Amsterdam, August 23-24, 1978, a second draft of the implementation model, in which the efficiency problems were paid due attention, was discussed [13]. Extensively polished versions of this second draft were discussed at the meetings of the Task Force in Cambridge (UK), December 18-19, 1978 and Summit (New Jersey, USA), April 3-4, 1979 [14, 15]. The present report differs from [15] at a number of minor points.

{A first implementation of the transput as defined in [12] has become available on a TR 440 at the University of Bochum (West-Germany) [16]. This implementation does not support formatted transput. It was finished within 4 months, solely from the description in [12], which places some confidence in its implementability. Also, each piece of program text from the present report has been tested using the CDC ALGOL 68 implementation [3].}

The approach taken is similar to the one in the Revised Report: the transput is described in pseudo-ALGOL 68. The pseudo-ALGOL 68 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 are defined. One advantage of a description in pseudo-ALGOL 68 is that it can largely be tested mechanically. It has been the intention that the ALGOL 68 text, after suitable substitution of the pseudo comments, could be compiled, thereby 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. The full transput is defined, even those features whose value may be questioned (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 'DIFFERENCES'.

At its meetings in December 1977 and August 1978, the Task Force on Transput also discussed a host of reported errors and problems in the area of transput. Most of these are collected in [17]. The Task Force has prepared a document containing solutions and answers to these problems [18]. This document has been formally approved by the Subcommittee on ALGOL 68 Support. The changes caused thereby have been incorporated in this document.

The main differences between the transput model presented here and that from the Revised Report are:

i) Books and channels are considered to form part of the operating-system interface; as such, the modes BOOK and CHANNEL are not specified;

ii) On most 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, such as SKIP, UP gremlins and UP bfileprotect, have been paid due attention;

iv) The number of tests that are 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 and optimizations have been made in the sections on formatted transput.

The differences between the present model and the definition in the Revised Report have been collected in the sections headed 'DIFFERENCES'. The following classes are distinguished:

i) Differences in the descriptive method being used. Only major changes in the descriptive method are listed, such as the change in the internal representation of formats. They are marked {D};

ii) Definition of 'undefined'. Wherever the Revised Report uses 'undefined', the present model prescribes a more precise action, for instance the production of some error message. These are marked {U};

iii) Violation of the semantics as given in the Revised Report. There are a few places where the semantics of the Revised Report are felt to be unreasonable. Care has been taken to keep the number of such changes minimal, since they affect the behavior of user programs. These differences are marked with an {S}, or, if they are a result of [18], with a {C};

iv) Fixing admitted bugs. The most important bugs in the transput section of the Revised Report are listed under entries marked {C}, although the user should consult [18] for a complete list of them;

v) Extensions. At a few places also, extensions to the definition as given in the Revised Report are defined. They are marked {E}.

Differences classified as {C} refer to the Commentaries issued by the Subcommittee on ALGOL 68 Support [18]. They are of the form {Cn}, where n is the number of the corresponding Commentary in [18].

Care has been taken to ensure reasonable efficiency of the whole system. Of course, efficiency is most important for heavily used procedures like 'put' or 'whole'. It might well be worthwhile to tune such routines to a specific operating environment. At places where efficiency is felt to be essential, indications are given that may help the implementer.

The text of this document is available in computer-readable form. People interested should contact the author to obtain more detailed information.

Acknowledgements. The author has benefited much from his correspondence and discussions with the members of the Task Force on Transput, convened by C.J. Cheney from the University of Cambridge (UK). Special mention should be made of H. Wupper, H.Ehlich, and their colleagues from the Ruhr-Universitaet at Bochum (West-Germany) for the feedback provided by their implementation of the model as given in [12]. A.N. Maslov and V.B. Yakovlev of Moscow are acknowledged for their criticism on the formatted-transput sections, and L.G.L.T. Meertens for his advice concerning the ALGOL 68 formalism used in section 11.2.2. C.H. Lindsey and R.G. Fisker from the University of Manchester have provided a fruitful insight into operating system interface problems [19], and an extensive survey on the structure of the transput section [20], respectively. H. de J. Laia Lopes from the Mathematical Centre assisted in testing the implementation model using the CDC ALGOL 68 implementation [3]. The Board of Directors and my colleagues from the Mathematical Centre have provided the pleasant environment in which this research could be performed.

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 (e.g., a binary paper tape). 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, say, the size of a number. There also exist systems where the buffer has a fixed size (typically 512 "words"), regardless of the file being used. 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 environment 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 section 4.4.1.

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 as well (section 4.4.2).

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. However, the hardware may not permit characters to be punched alternately in, say, ASCII and EBCDIC. It is to be expected that on many devices there will be only limited possibilities for changing the conversion key. Perhaps 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 conversions on a line by line basis can be set up with one hardware instruction. The present model is flexible enough to allow most of these very different implementations of conversions: 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.

Although the implementer is given the freedom to do the conversion either per character or per line, the possibilities for using different conversion keys are limited. First of all, it is recommended that there be only one type of buffer. If an implementation uses two very different types of buffers, say one type in which 10 characters are packed into one word, and another type in which the buffer simply corresponds to an ALGOL 68 row-of-character, then the mode BUFFER would probably look like

```
MODE BUFFER = UNION(REF [] WORD, REF [] CHAR).
```

Such a scheme leads to a lot of overhead, since each routine accessing the buffer has to select the appropriate case.

Furthermore, changing the conversion key by 'make conv' is only possible if both the current and the new conversion is done on a character by character basis. If the conversion is done on a line by line basis, it will be linked up with the primitives that write a buffer and read the next buffer. These primitives are part of the set of "buffer primitives", and

they cannot be changed; rather, they are determined once and for all when a book is linked with the file via some channel. If the conversion is done per character, it will be taken care of by the primitives provided for reading and writing single characters from or to the buffer. These conversion primitives in a narrow sense form the "conversion key"; they may possibly be changed at runtime.

With each channel, a standard conversion key is associated. If many different standard conversion keys are supported, this may easily lead to a multitude of channels to be supported as well. Since the conversion key also depends on the book, and thus on the identification string, one may also choose to distinguish between various conversion keys by providing suitable identification strings.

(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 primitive routines 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 sections 4.4.1 and 4.4.2.

It should be emphasized here that the Task Force on Transput strongly dissuades the use of conversion keys (Commentary 22 in [18]). The CONV feature of the transput is one of the less satisfactory features of the Revised Report. It does not interface well with conversion features that are likely to be provided by the hardware or operating system (which would be better invoked by Job Control Language commands or features in identification strings). Its correct implementation is difficult to do efficiently.

Noting that the Revised Report nowhere obliges an implementer to provide any conversion keys beyond one standard conversion key for each channel (and

these are likely to be null operations, or at least very straightforward, in most systems), the Task Force recommends that implementers should not in general provide additional conversion keys unless they have pressing problems with their local character codes.

Further machine-dependent primitives are concerned with routines that may move the current position to another line: 'newline', 'newpage', 'reset' and 'set'. The critical parts of these routines are considered to be primitive and are also discussed in section 4.4.1.

In the present model, all the above machine dependencies are linked up with the channel. A channel determines 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 attach these primitives to the channel. A similar approach is also taken in the ALGOL68S implementation at Carnegie Mellon University [21, 22] and in the implementation at Manchester [11]. (The ALGOL68S implementation at Carnegie Mellon also uses a buffer concept, albeit a slightly different one [22].)

A full list of the primitives that have been attached to the channel is:

- 'write char', 'write bin char', 'read char', 'read bin char';
- 'init buffer', 'write buffer', 'do newline', 'do newpage', 'do reset' and 'do set'.

The first set of routines comprises the "conversion key" (and these may possibly be changed at runtime), the second set is independent of the conversion key (these routines are termed "buffer primitives"). Upon linking a book with a file via some channel, all these primitives are incorporated in the file. Further transput then uses the corresponding fields of the file. This scheme makes it relatively easy to add new conversion keys and/or channels to the library prelude.

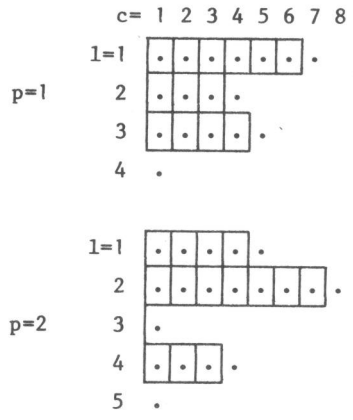
3. POSITIONS

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 at which the next transput operation will (normally) operate.

Before any actual transput operation may take place, the validity of the current position must 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, 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 the above picture shows a text which has 2 pages.}

{If the page has overflowed, the current line is empty (so the line has overflowed too), and, if the book 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 complete 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". It is impossible for the current position to be beyond the logical end. Likewise, it is impossible for the logical end to be beyond the physical end.

{Thus, if "≤" denotes the intuitive 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 section 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 transput is attempted at an overflow position, 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.

{In the Revised Report, 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: when (on output) the physical end of the book is reached, the event routine corresponding to "on physical file end" is supposed to be called as soon as the user attempts to put any character on the line (just beyond the physical end) that is not there. If the physical limits of the book are known to the

runtime system in advance (e.g., they are simply the 'p', 'l' and 'c' parameters of 'establish'), this raises no problem. In practice, however, the limit usually becomes known only when the operating system refuses to accept the contents of the buffer, which will not occur until, having put characters on the line with apparent success, an attempt is made to write the buffer to the book. This event need not occur at the first position on a new page. In the present model, the event routine may therefore also be called from within the routine that writes a buffer to the book. (See also Commentary 23 from [18].)}

The transput system has to administer 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.

Except for 'newpage' and 'set', no routine needs precise knowledge of the logical end, as long as it is not within the current line. Also, it is anticipated that the logical end will in general not be known if it is not within the current line. Therefore, the present model maintains a flag 'lfe in current line', which indicates whether or not the logical end is within the current line, and a variable 'c of lpos' which indicates the position up to which the current line has been filled with information. Questions regarding the logical end are then answered by first inspecting the 'lfe in current line' flag, and, if this flag is raised, by subsequently inspecting the value of 'c of lpos'.

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 being written to. The Revised Report maintains the most important invariant that connects 'cpos' and 'lpos':

I: cpos $\leq$ 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 for more efficient implementation. 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}.

Since 'lpos' is not maintained as such in the present model, the implementation looks slightly different. Of course, 'lpos' > 'cpos' if the logical end is not within the current line. If the logical end is within the current line, then the line is filled with information up to the position indicated by MAX('c OF cpos', 'c of lpos'). (Due to the proper initialization of 'c of lpos' inside the routine 'init buffer' (4.4.1.a), MAX('c OF cpos', 'c of lpos') always indicates the position up to which the line is filled with information. The routine 'write buffer' (4.4.1.b) uses this knowledge.)

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 true logical end must first be stored in case the current position is beyond the logical end as recorded. (The routine 'mind logical pos' (6.2.g) serves this purpose.) The logical end, as recorded in the book,

is also updated if the buffer is ultimately written to the book (see 4.4.1.b). It is expected that a considerable gain in efficiency will result from this change in representation.

4. BOOKS, CHANNELS AND FILES: THE BASIC MODEL

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

4.1. BOOKS

4.1.1. DIFFERENCES

- {D} The various fields of the mode BOOK are not specified. Instead, routines are provided to construct books, or to 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. 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 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 scenes, and so is the semaphore.

- {U} 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 'newpage' and 'set', no routine needs precise knowledge of the logical end of the book, as long as it is not in the current line. The book also has to contain the (maximum) size of the book, in order to be able to give a reasonable interpretation of the routines 'newline' and 'newpage' (7.2.c, d). If a book is "opened" (5.2.c), it may be necessary to provide a default size (the default size of the given channel seems a reasonable choice here).

Books are either searched for in the system, or constructed according to certain user-defined requisites. Books are searched for by the routine 'find book in system' (e). A book is constructed by the routine 'construct book' (f). Note that the scope of objects (books and buffers) generated by 'find book in system' and 'construct book' should not be newer than the scope of the file they will ultimately become part of. This file may be one of the files in the standard prelude (Chapter 13).

a) MODE ? BOOK =
    C Some mode, whose values contain at least the following operating-system-dependent information:
    - the identification string;
    - some reference to the actual information, and some indication where to read or write the next buffer (see also section 4.4.1);
    - the logical end of the book {, which may however be unknown};
    - information that tells whether the book may again be opened for input or output. {Often, a book may be opened only once for output, but may be opened many times for input, simultaneously.};
    - the maximum size of the book, as set by 'construct book' (f) or 'find book in system' (e).

    To avoid excessive copying of information, the implementer is advised to define the mode BOOK as a reference to the necessary data. 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 ? find book in system =
      (STRING idf, CHANNEL chan, REF BOOK book, REF BUFFER buffer) INT:
        C If it is possible to "open" (5.2) another file on the given
          channel at this instant of time, then the pool of available books
          is searched for a book with the following properties:
          - the book may be identified by 'idf' and 'chan';
            (Note that, in general, 'idf' and 'chan' together identify the
            book.)
          - the book may be legitimately accessed through 'chan';
          - opening is not inhibited by other users of the book.
          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 returns 0; the buffer is not initialized. Otherwise,
          the routine returns some positive integer that corresponds to the
          appropriate error code. (For a list of these error codes, see
          section 5.2.) If the maximum size of the book found is not known,
          some default (for example, the default size of a book linked via
          the given channel) should be taken instead. C;
```

```
f) PROC ? construct book =
      (INT p, l, c, STRING idf, CHANNEL chan, REF BOOK book,
       REF BUFFER buffer) INT:
        C 'construct book' starts by checking the validity of its
          parameters:
          - it should be possible to "establish" (5.2) another file on the
            given channel at this instant of time;
          - 'idf' should be acceptable to the implementation as the
            identification of a new book;
          - the size indicated by ('p', 'l', 'c') should not exceed the
            maximum size allowed for a book linked via the given channel.
          If one of these checks fails, the routine returns some positive
          integer that corresponds to the appropriate error code. Otherwise,
          a book that may be written to is constructed and assigned to
          'book'. The book has a text of the size indicated by ('p', 'l',
          'c') and an identification string 'idf'. (Note that the
          identification string stored into the book may not be the whole of
          'idf', since in some systems 'idf' might contain information about
          access rights, record types, etc.) 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 filled with information. The
          logical end of the book is set to (1, 1, 1). Finally, the routine
          returns 0. 'construct book' is called by 'establish' (5.2.a) C;

g) PROC ? default idf = (CHANNEL chan) STRING:
     C A default identification string for files which are opened on the
       given channel via 'create'. Presumably, successive calls of 'default
       idf' should return different strings. C;

h) 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 COVER cover = cover OF f;
       c of lpos OF cover:= c OF cpos OF cover;
       status OF cover ANDAB lfe in current line
     END;
```

{This routine performs the resetting of 'lpos' enforced when a change of reading to writing occurs in the middle of a sequential-access file. To this end, it is called from 'set write mood' (5.2.f). The possibility of expanding the size of the current line and page and of all subsequent lines and pages (e.g., to the sizes with which the book was originally opened) of such a file is dealt with in 'set write mood' as well. The routine is also called by 'write buffer' (4.4.1.b) to administer the logical end of files that are being written to. Note that this routine is never called with an associated file as actual parameter.}

4.2. CHANNELS

4.2.1. DIFFERENCES

- {D} To allow for an efficient implementation of channels, the various fields of the mode CHANNEL are not specified. At places where these fields were previously used, pseudo-comments are placed to indicate the desired action. The boolean procedures which determine the available methods of access to a book linked via the channel are replaced by one routine 'access methods' (1), which determines all those properties and stores them in a value of mode POSSIBLES. When a file is linked to a book via some channel (5.2), this routine is called and the resulting value is assigned to the 'possibles' field of the cover of the file. The environment enquiries for files then inspect this field.

- {E} A property 'backspace possible' is added, which determines whether the routine 'backspace', as defined in section 10.3.1.6 of the Revised Report, is available on the channel. It also controls whether 'choice-patterns' may be used in formatted input. This property is to be determined by the routine 'access methods' (1).

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

- {D} In the present model, the channel also determines six "buffer primitives": 'init buffer', 'write buffer', 'do newline', 'do newpage', 'do reset' and 'do set'. A routine 'buffer primitives' (m) is provided to associate these primitives with a given file.

- {E} The present model is less prescriptive than the Revised Report as regards the access properties of books linked via one of the standard channels. The access properties of such books are not solely determined by the channel; rather, they are obtained by the intersection of the properties of the book and the channel. In this way, the number of channels to be supported can be kept within reasonable bounds.

4.2.2. NEW DEFINITION

A "channel" determines a collection of attributes that defines some particular style of transput. Most of these attributes are concerned with the available methods of access to a book linked via some channel. These methods of access are determined by the routine 'access methods' (1). Since the access methods may well depend on the book as well as on the channel, this routine is given both the book and the channel as parameters. Upon linking a book with a file (5.2), the available methods of access are determined by calling the routine 'access methods' (1) and these access properties are then incorporated in the file. The properties may subsequently be examined by use of the environment enquiries for files (4,3.2). {If these properties are subject to changes at runtime, the 'possibles' field of the file may have to be updated at runtime. This updating then has to take place by some method that is not described here.}

The access properties of the standard channels defined in the Revised Report are prescriptive, i.e., if some book is opened via one of the standard channels, it should be checked that the given book actually allows what is required by the channel. Thus, if a book is not random access, for instance because it resides on magnetic tape, it cannot be opened via 'standback channel'. If one views the additional channels provided in a standard prelude in the same way, a multitude of channels may have to be supported, viz. one for each desirable combination of properties. The present model is less restrictive as regards such channels. The access properties of a book opened via such a channel are obtained by the intersection of the properties of the book and the channel. For example, reading is possible if both the book and the channel allow reading. There seems no reason not to treat the standard channels in the same way. It is however the intention that the properties of a book successfully opened on a given channel be those of the channel, unless some are explicitly prohibited by the book (e.g., by some encoding in the identification string). If implemented this way, the standard files that are opened in the standard prelude (Chapter 13) are expected to have the same properties as those defined in the Revised Report. Also, identification strings as determined by the routine 'default idf' (4.1.2.g) should not prohibit any access property of the given channel.

Similar to the status of the file (6.2), the access properties are collected in a value of the mode BITS. For the sake of clarity, the mode is called POSSIBLES if it concerns these access properties (see i, j and k below).

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".

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 four routines to transput characters to and from the buffer.

The implementation may provide additional conversion keys in its library-prelude. In addition to (or instead of) the character by character conversion controlled by the conversion key, there may be a line by line conversion controlled by the buffer primitives. See also Chapter 2 for a discussion on the use of conversion keys.

The conversion key is understood to be one part of the interface between the model and the operating-system environment. The routines that the conversion key consists of, the so-called "conversion primitives", are sketched in section 4.4.2. The other part of this interface consists of six "buffer primitives": 'init buffer', 'write buffer', 'do newline', 'do newpage', 'do reset' and 'do set', which are dealt with in section 4.4.1. A routine 'buffer primitives' (m) is provided to associate these primitives with a given file. After linking a book with a file via some channel, the buffer primitives as determined by the channel and the book are copied to the corresponding fields of the file, except for associated files, where they are constructed inside the routine 'associate' (5.2.d).

```
a) MODE CHANNEL =
     C Some mode, whose values determine at least the following information:
       - the standard conversion key of the channel (d);
       - the buffer primitives of the channel (m);
       - the access properties of the channel (l);
       - whether or not another file may be "established" (5.2.a) on the
         channel (c);
       - the (next) default identification string of a book linked via this
         channel (4.1.2.g);
       - the default size of a book linked via this channel (n);
       - the maximum size of a book linked via this channel (4.1.2.f). C;

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

   {A conversion key may also contain other, implementation-dependent,
    fields.}

c) PROC estab possible = (CHANNEL chan) BOOL:
     C a routine which returns true if another file may be "established"
       (5.2.a) on the channel 'chan' C;

d) PROC standconv = (CHANNEL chan) PROC (BOOK) CONV:
     C a routine which may be used to obtain the default "conversion key";
       note that the actual primitives are not obtained until some specific
       book is provided also C;

e) CHANNEL stand in channel =
     C a channel value such that for each file successfully opened on this
       channel, 'get possible' and 'backspace possible' always return true
       (unless the book linked with the file explicitly indicates
       otherwise), while the other environment enquiries for files return
       some suitable values C;
```

f) CHANNEL stand out channel =

C a channel value such that for each file successfully opened on this channel, 'put possible' and 'backspace possible' always return true (unless the book linked with the file explicitly indicates otherwise), while the other environment enquiries for files return some suitable values C;

g) CHANNEL stand back channel =

C a channel value such that for each file successfully opened on this channel, 'set possible', 'reset possible', 'get possible', 'put possible', 'bin possible' and 'backspace possible' always return true (unless the book linked with the file explicitly indicates otherwise), while the other environment enquiries for files return some suitable values C;

h) CHANNEL ? associate channel =

C Some channel value which is distinct from all other channel values. It is used by the routine 'associate' (5.2.d) to arrange for a suitable value for the 'channel' field of the file. Note that the various routines that are given a channel value as parameter will have to be able to handle this case also. C;

i) MODE ? POSSIBLES = BITS;

j) OP SAYS = (POSSIBLES s, t) BOOL: s $\geq$ t;

{The above operator definition should only be included if the modes 'POSSIBLES' and 'STATUS' are not "related" (see section 7.1.1 of the Revised Report for a definition of this term.) The corresponding priority definition is given in 6.2.e.}

```
k) # Some constant declarations. #
   POSSIBLES ? reset poss       = 2r 100000000,
             ? set poss         = 2r 010000000,
             ? get poss         = 2r 001000000,
             ? put poss         = 2r 000100000,
             ? bin poss         = 2r 000010000,
             ? compress         = 2r 000001000,
             ? reidf poss       = 2r 000000100,
             ? backspace poss   = 2r 000000010,

             ? system get poss = 2r 000000001,
                  # see section 4.4.1 for the use of this property #

             ? associate poss   = 2r 111100011;

l) PROC ? access methods = (BOOK book, CHANNEL chan) POSSIBLES:
     C a routine which returns a value of the mode POSSIBLES from which the
       available methods of access (immediately after opening) to the book
       'book' linked via the channel 'chan' may be determined C;

m) PROC ? buffer primitives = (REF FILE f) VOID:
     C The buffer primitives as determined by 'chan OF f' are assigned to
       the 'init buffer', 'write buffer', 'do newline', 'do newpage', 'do
       reset' and 'do set' fields of the cover of 'f'. Note that this
       routine also has access to the book linked with the file, and thus to
       the identification string. These may both affect the actual
       primitives to be incorporated in the file. C;

n) PROC ? default size = (CHANNEL chan) POS:
     C the default size of a book linked via 'chan' C;

   {If a file is "created" (5.2.b), 'default size' is used to determine the
    size of the book to be created.}
```

4.3. FILES

4.3.1. DIFFERENCES

- {C5} Here, as well as in the Revised Report, the mode FILE is defined to be a very specific structure (albeit with hidden field selectors). Nevertheless, the user may in theory (but without practical use) write

```
FILE f:= (SKIP, SKIP, SKIP, SKIP, ...).
```

  Similarly, the user may write

```
standconv(stand in channel)(SKIP).
```

  Implementers should feel no obligation to implement 'standconv' and 'FILE' with these particular modes (provided, of course, that 'standconv' has one CHANNEL parameter) nor to accept constructs such as the above which rely upon them. (Since it is impossible to write structure-displays having one element, a very easy way to make phrases such as the above illegal is to define

```
MODE FILE = STRUCT(FFILE ? f),
```

  and to define FFILE as the structure given below (a). This has not been done in this document for the sake of clarity.)

- {D} The fields of a file are divided into two categories: those that may differ between copies of a file, and those that may not. The latter are collected in the "cover" field of the file. Thus, after an assignment 'f:= g', transput using the file 'f' and transput using the file 'g' may differ in the following respects:
  i) they may use different formats,
  ii) they may use different terminator strings,
  iii) they may use different conversion keys, and
  iv) they may use different event routines.

- {D} The cover includes a field 'buffer' which is 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 (but see section 4.4.3 for the case where the buffer does not correspond to one line of the text).

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

- {D} The cover includes a field 'possibles' from which the available methods of access to the book of the file may be obtained. This field is inspected by the environment enquiries provided for files.

- {D} A field 'c of lpos' is included in the cover. This field indicates the position up to which the current line is filled with information. If the logical end is in the current line (which can be derived from the status information), it indicates the position of the logical file end, except possibly when the file is currently being written to. See also Chapter 3, where the use of this field is discussed in detail.

- {D} The cover includes fields 'init buffer', 'write buffer', 'do newline', 'do newpage', 'do reset' and 'do set', which are the buffer primitives as determined by the channel via which the book is linked. These fields are (except for associated files) initialized by the routine 'buffer primitives' (4.2.2.m) which is called when the file is opened (5.2).

- {E} An enquiry 'backspace possible' is provided, which returns true if the file may be "backspaced" (7.2). Provided backspacing is always possible on the three standard channels, this can be regarded as an extension, rather than a deviation from the Revised Report.

- {S} In the Revised Report, a call of 'reidf' has no effect whatsoever if either of the tests 'reidf possible' and 'idf ok' fails. In the present model, an error message is given in those cases, and the program is aborted.

- {C24} The routine 'reidf' can only be called when the book to be renamed has already been opened via some file. However, not all operating systems are able to perform their renaming function on an opened book. For implementers faced with such systems, one of the following courses of action is recommended:

  i) Do not provide the "reidf" facility at all (the Revised Report nowhere requires that 'reidf possible' should ever return true);

  ii) 'reidf' causes no immediate action, but the revised identification string is remembered and the book is renamed when the file is eventually closed (or locked).

  It is to be noted that, where 'reidf' is provided, the set of strings acceptable as its 'idf' parameter might be only a subset of the identification strings acceptable to 'open' and/or 'establish', especially if the latter are permitted to contain information concerning the disposal of, or the access rights to, the book.

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 "cover". The cover contains all information common between copies of a file. It includes a reference to the text, which is used for associated files (5.2), and the buffer, which is used for non-associated files. The cover also contains information necessary for the transput routines to work with the book, including its current position "cpos" in the text, its current "status" 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 buffer must be written back to the book;
- 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 has ended. (Note that the logical file is never ended if the file is in write mood.);
- 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, except possibly if the file is in write mood.);
- 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 following obvious assertions can be made about the information in the status (obviously, a IMPLIES b = NOT b IMPLIES NOT a):

NOT (read mood AND write mood),
NOT (char mood AND bin mood),

write back IMPLIES buffer initialized,
write back IMPLIES NOT page overflow,
write back IMPLIES NOT file overflow,
    {these are preconditions of 'write buffer' (4.4.1.b)}

```
file overflow IMPLIES page overflow,
page overflow IMPLIES line overflow,
logical file ended IMPLIES logical file end in current line,
write mood IMPLIES NOT logical file ended,
    {see Chapter 3 for a detailed discussion of this last set of
     invariants}
```

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 'cover' 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 characters to and from the buffer. After opening a file, the standard conversion key determined by the channel on which it is opened is provided by default (the actual primitives incorporated in the file may also depend on the given book and its identification string). 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 factors not defined by this model.}

The routine 'make term' is used to associate a string with a file. This string is used when reading 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. A similar feature can profitably be used in 'get' (10.2.c) and 'getf' (11.2.3.p). To this end, the

mode CHARBAG is introduced, together with an operator STRINGTOBAG and a routine 'char in bag' (10.2.f).}

The available methods of access to a book which has been opened on a file are collected in the "possibles" field of the cover of the file. This field is inspected by the following environment enquiries provided for files (these routines are provided for the user to determine the available methods to the book of a file at a given instant of time):

- '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.

{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'.}

There is one other environment enquiry for files: '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).

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 supposed that the condition has been mended in some way, and, if possible, transput is re-attempted, but if it returns false, then the system continues with its default action.

{There is a very minor, if not obscure difference between the default action of the Revised Report and the one given below for the 'on line end' event (Commentary 27 from [18]). A precise description of this difference is given in section 7.1.}

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. The default action is to
  i) stop reading in the case of unformatted input of strings;
  ii) give a suitable error message and to abort the program otherwise.

- '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 output is attempted (see 7.2). The default action is to give a suitable error message and to abort the program.

- '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). The default action is to call 'newpage' (7.2.d).

- '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). The default action is to
  i) give a suitable error message and to abort the program in the case of formatted transput;
  ii) stop reading in the case of unformatted input of strings;
  iii) call 'newline' (7.2.c) otherwise.

- '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). Since, for efficiency reasons, the present model does not deal with the possibility of conversion errors on output, this event routine is never called during output. 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 by the user and after checking for suitability) 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, an error message is given and the program is aborted (see Chapter 5 for the use of the routine 'error' and the label 'abort');

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, except possibly to the real part of a complex number); 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; if the routine returns false, the current format is repeated.

```
a) MODE FILE = STRUCT(
     REF COVER ? cover,
     REF REF FORMATLIST ? piece,
     CHARBAG ? term,
     PROC (REF FILE, CHAR) VOID ? write char,
     PROC (REF FILE, BINCHAR) VOID ? write bin char,
     PROC (REF FILE, REF CHAR) BOOL ? read char,
     PROC (REF FILE, REF BINCHAR) VOID ? read bin char,
     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);

   {A file may also include other, implementation dependent, fields.}

b) MODE ? COVER = STRUCT(
     BOOK book,
     CHANNEL chan,
     BUFFER buffer,
     REF [] [] [] CHAR text, # for associated files only #
     POS cpos,
     INT c of lpos,
     STATUS status,
     INT char bound,
     POSSIBLES possibles,
     PROC (REF FILE) VOID init buffer, write buffer, do newline,
          do newpage, do reset,
     PROC (REF FILE, INT, INT, INT) VOID do set);

c) MODE ? BUFFER =
     C Some mode, which probably starts with 'reference to'. A buffer should
       be able to hold up to 'char bound' characters (possibly, depending on
       the implementation of conversion keys, in their external (converted)
       form). C;
```

```
d) PROC get possible = (REF FILE f) BOOL:
     IF REF COVER cover = cover OF f;
       status OF cover SAYS opened
     THEN possibles OF cover SAYS get poss
     ELSE error(notopen); abort
     FI;

e) PROC put possible = (REF FILE f) BOOL:
     IF REF COVER cover = cover OF f;
       status OF cover SAYS opened
     THEN possibles OF cover SAYS put poss
     ELSE error(notopen); abort
     FI;

f) PROC bin possible = (REF FILE f) BOOL:
     IF REF COVER cover = cover OF f;
       status OF cover SAYS opened
     THEN possibles OF cover SAYS bin poss
     ELSE error(notopen); abort
     FI;

g) PROC compressible = (REF FILE f) BOOL:
     IF REF COVER cover = cover OF f;
       status OF cover SAYS opened
     THEN possibles OF cover SAYS compress
     ELSE error(notopen); abort
     FI;

h) PROC reset possible = (REF FILE f) BOOL:
     IF REF COVER cover = cover OF f;
       status OF cover SAYS opened
     THEN possibles OF cover SAYS reset poss
     ELSE error(notopen); abort
     FI;
```

```
i) PROC set possible = (REF FILE f) BOOL:
     IF REF COVER cover = cover OF f;
       status OF cover SAYS opened
     THEN possibles OF cover SAYS set poss
     ELSE error(notopen); abort
     FI;

j) PROC backspace possible = (REF FILE f) BOOL:
     IF REF COVER cover = cover OF f;
       status OF cover SAYS opened
     THEN possibles OF cover SAYS backspace poss
     ELSE error(notopen); abort
     FI;

k) PROC reidf possible = (REF FILE f) BOOL:
     IF REF COVER cover = cover OF f;
       status OF cover SAYS opened
     THEN possibles OF cover SAYS reidf poss
     ELSE error(notopen); abort
     FI;

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

```
m) PROC make conv = (REF FILE f, PROC (BOOK) CONV c) VOID:
     IF status OF cover OF f SAYS opened
     THEN
       C If possible, the conversion key of 'f' is made to be the result of
         calling 'c(book OF cover OF f)'. 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 the conversion key of an associated file (5.2.d) is
         not changed. See also Chapter 2 for a discussion of conversion keys
         and their use. C
     ELSE error(notopen); abort
     FI;

n) PROC make term = (REF FILE f, STRING t) VOID:
     term OF f:= STRINGTOBAG 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 INT er = C If the file is opened and the 'idf' field of the book of
                   the cover of the file may be changed, and the 'idf'
                   parameter of the call may be used as the identification
                   of a new book, then the value 0 is returned. Otherwise,
                   some positive integer that corresponds to the appropriate
                   error code is returned. C;
       er = 0
     THEN
       C the identification string of the book of the cover of 'f' is made
         to be 'idf' (or some function thereof, in case it contains other
         auxiliary information which may, for example, affect the
         'possibles' field of the cover) C
     ELSE error(er); abort
     FI;
```

4.4. MAIN OPERATING SYSTEM INTERFACE

{"I ca'n't go no lower," said the Hatter:
"I'm on the floor, as it is."
Alice's Adventures in Wonderland,
Lewis Carroll.}

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 or move buffers are given.

These routines are meant to be models of what each implementer must provide. For each channel, both the "buffer primitives" (4.4.1) and the "conversion primitives" (4.4.2) are needed. For each additional conversion key, the corresponding set of conversion primitives must be provided.

If the buffer corresponds to one line of text, the semantics are somewhat easier to describe than in the case where the buffer corresponds to a smaller unit of information. Therefore, this case is treated first (sections 4.4.1 and 4.4.2). Section 4.4.3 contains some hints as to which changes are needed when the buffer corresponds to a smaller unit of information.

4.4.1. BUFFER PRIMITIVES

The buffer primitives are incorporated in the cover of the file by means of a call of the routine 'buffer primitives' (4.2.2.m) when the file is "opened", "created" or "established". In 'associate' (5.2.d), the necessary routines are built and assigned to the corresponding fields of the cover directly.

The routines 'init buffer' and 'write buffer' form the main interface with the operating system environment as far as the exchange of actual data is concerned. The routine 'write buffer' writes the contents of the buffer to the book; 'init buffer' initializes the next buffer. There is no need for an explicit primitive 'read buffer', since, if necessary, 'init buffer' already reads in the next buffer.

If a file is 'opened' or 'reset', the buffer is not initialized directly. The reason for this is that alternating reading and writing on a sequential-access file is not always possible. Whether it is the intention to read or write can, in general, only be determined at the first actual transput operation. Thus, the buffer cannot be initialized until that moment. {This is more important for files opened on some interactive device.} As soon however as a single character is to be transput or a layout routine is to be called, the buffer must be properly initialized. This is ensured by calling 'init buffer' from the routine 'ensure physical file'. Whether or not the buffer has been initialized can be determined from the status of the file (6.2).

After having read a buffer, its contents may or may not get changed. If the contents do not get changed, there is no need to write them back to the book prior to the initialization of the next buffer. Whether or not the buffer must be written back to the book is determined by the flag 'write back' in the status of the file and this flag is inspected prior to the calling of 'write buffer'. (Note that it is not sufficient to raise this flag in 'write char' and 'write bin char' only. For sequential-access files, a mere change to write mood necessitates the buffer ultimately being written back to the book (see also 5.2.f).)

When one of the primitive routines is called, certain events may occur, such as page end or physical file end. Several philosophies are possible as regards the exact moment at which these events may occur. The two extreme possibilities are:

i) events cannot be foreseen; e.g., the page-end event is only detected when one tries to initialize the buffer which corresponds to the first nonexistent line of the page;
ii) events can be foreseen, i.e., the end of a page is detected by just looking at the current line number.

For most implementations, the first scheme seems the most appropriate. It is to be expected that the logical end will in general not be detected until one tries to read beyond it. (In some cases, it may be possible to foresee the end of a page.)

Following the approach where events cannot be foreseen, the various primitive routines take the following form:

```
try something;
IF it did not work
THEN raise the appropriate flags
FI.
```

The other approach leads to the scheme:

```
IF special condition
THEN raise the appropriate flags
ELSE do something
FI.
```

In the present model, the first approach is taken.

The routine 'init buffer' is used to initialize the next buffer. An attempt is made to read in a buffer if the following two conditions are fulfilled:

i) it is possible to read from the book, i.e., the operating system allows reading from the book. (Note that if the operating system allows reading, the channel may still prevent the user from doing so. Therefore, two different flags 'system get poss' and 'get poss' are incorporated in the 'possibles' field of the cover of the file. In 'init buffer', it is the 'system get poss' flag that is inspected. Obviously, 'get poss' implies 'system get poss'.)

ii) the file that is being written to is not sequential-access (see also Chapter 3).

{It should be emphasized that, for random-access files, the buffer must always be read in (provided one is not writing at the logical end). Even if the user is currently writing to the book, he need not overwrite the whole buffer. In such a case, part of the old contents must be preserved. Using the present model, it is therefore not possible to implement random-access files that cannot be read by the underlying system.}

One possible result of a call of 'init buffer' is that one or both of the logical-file-ended and page-ended flags in the status of the file are raised, indicating the corresponding event(s), without actually reading in the next buffer.

```
a) PROC init buffer = (REF FILE f) VOID:
     # The precondition of 'init buffer' is:
         . opened,
         . NOT physical file ended,
         . NOT page ended,
         . c OF cpos OF cover OF f = 1. #
     BEGIN REF COVER cover = cover OF f;
       REF STATUS status = status OF cover;
       status ORAB buffer initialized;
       IF possibles OF cover SAYS system get poss AND
           NOT (status SAYS write sequential OR
                status SUGGESTS lfe in current line)
       THEN status ANDAB NOT write back;
         buffer OF cover:= C The next line from the book of 'f', possibly
                           after conversion (in which case the conversion
                           is done on a line by line basis). Note that a
                           page end may prevent this line from being read.
                           C;
         charbound OF cover:= C the maximum length of the line just (partly)
                               read C;
             # This length may be zero (e.g., at page end), and it may be
               greater than the number of characters read if it concerns the
               last logical line of the file. #
         c of lpos OF cover:= C the position up to which the buffer has been
                               filled by the above read operation C + 1;
         IF C the logical end is in the buffer just read C
         THEN status ANDAB lfe in current line;
           IF status SAYS read mood AND c of lpos OF cover = 1
           THEN status ANDAB logical file ended
           FI
         FI
```

```
      ELSE # we are writing at the logical end #
        char bound OF cover:= C the maximum length of a line (as recorded
                               in the book), or 0 if the page has
                               overflowed C;
        c of lpos OF cover:= 1;
        status ORAB write back
      FI;
      IF C the page of the book has overflowed C
            # i.e., some marker indicating a page end has been read, or
              the current line number exceeds the maximum as recorded in
              the book #
      THEN status ANDAB page end
      ELIF char bound OF cover = 0
      THEN status ANDAB line end
      ELSE status ORAB not line end
      FI
    END # init buffer #;
```

The routine 'write buffer' is used to write the contents of the buffer to the book. The only unexpected event that may occur is that the physical end of the book is reached while writing this buffer. In that case, the event routine corresponding to 'on physical file end' is called. If the user fails to mend the situation, an error message is given and the program is aborted. Otherwise, the test is repeated until a situation occurs where the physical file is no longer ended. (This whole process occurs inside the routine 'ensure physical file'.) After that, another attempt is made to write the buffer to the file. It is not explicitly defined by this model how much of the buffer has been written to the book at the time the physical end is reached. {As a result of this call of 'ensure physical file', all fields of the file may have been changed. Therefore, they must be selected anew after a call of 'write buffer'.}

{One of the few sensible possibilities for a user to react to this event is to re-open the file on some other book and to continue writing to this new book. Since opening a file is merely a special kind of assignment, the old buffer gets lost. Therefore, it must be saved. An attempt to write this same buffer again cannot be expressed in ALGOL 68 so simply, since the two books may well be incompatible. For instance, the two buffers may differ in

size.}

Two cases must be distinguished when the routine 'write buffer' is called:

i) The buffer has first been (partially) read by 'init buffer';
ii) The buffer has not been read in (in which case one is writing at the logical end).

For case i), the buffer should obviously be written back at the same (physical) position where it has previously been read in. This may involve some kind of backspacing. The easiest way to trap this case is to introduce a flag in the status that indicates whether or not the buffer has been read in. This may however not suffice if lines are of variable length, since then the amount of backspacing may be unknown. The alternative is to maintain two pointers in the cover of the file, one indicating from which position onwards the next buffer should be read ('rp'), and one indicating from which position onwards the next buffer should be written ('wp'). Upon opening a file, 'rp' must be initialized properly. The routine 'init buffer' now starts with the assignment 'wp:= rp', and 'rp' must be updated if (part of) the buffer is read in. 'write buffer' then always starts writing at the position indicated by 'wp'.

```
b) PROC write buffer = (REF FILE f) VOID:
     # The precondition of 'write buffer' is:
          . opened,
          . either write mood or read mood,
          . the buffer is initialized,
          . the 'write back' flag is raised,
          . NOT physical file ended,
          . NOT page ended. #
     BEGIN REF COVER cover = cover OF f;
       REF STATUS status = status OF cover;
       status ANDAB NOT write back; # to prevent recursion #
       c OF cpos OF cover:= c OF cpos OF cover MAX c of lpos OF cover;
       IF status SUGGESTS lfe in current line AND
            NOT (possibles OF cover SAYS compress)
       THEN C Fill the buffer from the position indicated by 'c OF cpos OF
              cover' up to and including the position indicated by
              'charbound OF cover' with spaces (or, in bin mood, some
              undefined character). C;
         c OF cpos OF cover:= char bound OF cover + 1
       FI;
       C The contents of the buffer of the file (up to the position
         indicated by 'c OF cpos') is, possibly after conversion (in which
         case the conversion is on a line by line basis), written to the
         book of the cover of the file. C;
       IF C this fails (i.e., the physical end of the book is reached while
            writing) C
       THEN status ANDAB physical file end;
         BUFFER b = buffer OF cover;
            # or something else to ensure that the contents of this buffer
              will not get lost #
         ensure physical file(f);
            # observe that the user's event routine will find the current
              position at the end of the line he is trying to write #
         C try to write 'b' to the (new) book of the cover of 'f' C
       FI;
```

```
    IF status OF cover OF f SUGGESTS lfe in current line
    THEN set logical pos(f)
    FI
  END # write buffer #;
```

The routines 'do newline' and 'do newpage' will move the current position to another line. They do not initialize the next buffer on reading; this is especially important when inputting from an intercative device. 'do reset' and 'do set' do not necessarily move the current position to another line, although they probably will in most cases. Moreover, a separate routine 'set char number' is provided to move the current position about in the current line.

The routine 'do newline' is used actually to move the current position to the beginning of the next line. If the newline operation would pass the logical end while reading, the current position is set to the logical end and 'newline' is called recursively (7.2.c). Therefore, 'do newline' can itself operate in a straightforward way.

```
c) PROC do newline = (REF FILE f) VOID:
     # The precondition of 'do newline' is:
           . page ok (see 7.2),
           . read mood => NOT lfe in current line. #
     BEGIN
       IF status OF cover OF f SAYS write back
       THEN (write buffer OF cover OF f)(f)
       FI;
       REF COVER cover = cover OF f;
       l OF cpos OF cover +:= l; c OF cpos OF cover:= l;
       IF status OF cover SAYS write mood
       THEN (init buffer OF cover)(f)
       ELSE status OF cover ANDAB NOT buffer initialized
       FI
     END # do newline #;
```

The routine 'do newpage' moves the current position to the beginning of the next page. Its semantics are hard to describe formally, since different operating systems (and possibly even different channels on the same operating system) are likely to differ considerably here.

```
d) PROC do newpage = (REF FILE f) VOID:
     # The precondition of 'do newpage' is:
          . physical file ok (see 7.2). #
     BEGIN
       IF status OF cover OF f SAYS write back
       THEN (write buffer OF cover OF f)(f)
       FI;
       REF COVER cover = cover OF f;
       C Move to beginning of next page. This may result in compressing the
         current line and page, or in writing lines full of spaces. C;
       IF C this fails to succeed C
              # i.e., on reading, the logical end is reached while searching
                for the beginning of the next page; it is assumed that the
                buffer containing the logical end has been initialized
                properly #
       THEN c OF cpos OF cover:= c of lpos OF cover;
         status OF cover ANDAB logical file ended;
         newpage(f)
           # which will immediately lead to a call of the event routine
             corresponding to 'on logical file end' #
       ELSE cpos OF cover:= (p OF cpos OF cover + 1, 1, 1);
         IF C this causes the physical end to be reached C
           # I.e., the current page number exceeds the maximum as recorded
             in the book. This event can occur only when the file is in
             write mood. Also, the operating system may be unable to detect
             it, in which case some subsequent call of 'write buffer' will
             do so. #
         THEN status OF cover ANDAB physical file end
         ELSE status OF cover ORAB not page end;
           IF status OF cover SAYS write mood
           THEN (init buffer OF cover)(f)
           ELSE status OF cover ANDAB NOT buffer initialized
           FI
```

```
        FI
      FI
    END # do newpage #;
```

The routine 'do reset' is used to reset a book to the initial position (1, 1, 1). Note that a file is then left in a status similar to that after a call of 'open', i.e., the buffer is not initialized.

```
e) PROC do reset = (REF FILE f) VOID:
     # The precondition of 'do reset' is:
           . opened,
           . reset possible. #
     BEGIN
       IF status OF cover OF f SAYS write back
       THEN (write buffer OF cover OF f)(f)
       FI;
       REF COVER cover = cover OF f;
       cpos OF cover:= (1, 1, 1);
       C The book is physically reset. C
     END # do reset #;
```

The routine 'do set' is used to position a random-access file. It is assumed that, in general, it will move the current position to another line. 'do set' starts by searching for the line indicated by its 'p' and 'l' parameters, in the mean time updating the current position. Searching may stop at the following positions:

- at the first position of the last (logical) line if the position indicated by 'p', 'l' and 'c' is beyond that position;
- just beyond the page indicated by 'p' if 'l' exceeds the number of lines in that page;
- at the first position of the line indicated by 'p' and 'l' otherwise.

```
f) PROC do set = (REF FILE f, INT p, l, c) VOID:
     # The precondition of 'do set' is:
          . opened,
          . set possible. #
     IF
       IF status OF cover OF f SAYS write back
       THEN (write buffer OF cover OF f)(f)
       FI;
       POS(1, 1, 1) EXCEEDS POS(p, l, c)
     THEN error(posmin); abort
     ELIF REF COVER cover = cover OF f;
       REF STATUS status = status OF cover;
       status ORAB open status;
       C Search for the line indicated by 'p' and 'l'. C;
       (init buffer OF cover)(f);
       REF INT ccpos = c OF cpos OF cover,
       INT char bound = char bound OF cover;
       status SUGGESTS lfe in current line
     THEN INT clpos = c of lpos OF cover;
       IF BOOL beyond lpos =
              POS(p, l, 1) BEYOND cpos OF cover OR c > clpos;
         beyond lpos OR c = clpos
       THEN STATUS reading = state(f);
         ccpos:= clpos;
         IF status SAYS read mood
         THEN status ANDAB logical file ended
         ELIF ccpos > char bound
         THEN status ANDAB line end
         FI;
         IF beyond lpos
         THEN BOOL mended = (logical file mended OF f)(f);
           ensure state(f, reading);
           (NOT mended | error(wrongset); abort)
         FI
       ELSE ccpos:= c
       FI
```

```
    ELIF POS(p, 1, 1) EXCEEDS cpos OF cover OR
           c > char bound + 1
    THEN error(posmax); abort
    ELSE ccpos:= c;
      (ccpos > char bound | status ANDAB line end)
    FI # do set #;
```

4.4.2. CONVERSION PRIMITIVES

This section describes the routines that comprise the conversion key. Conversion keys and their use are described in detail in Chapter 2.

It is to be noted that the present model does not deal with the possibility of conversion errors on output. The possibility of such errors would seriously affect the efficiency of output, especially as regards unformatted output of numbers and strings. Moreover, the Task Force on Transput strongly dissuades the use of conversion keys (Commentary 22 in [18]).

If conversion keys are such that at least the digits and the other characters required for the transput of arithmetic values are always convertible (as suggested in Commentary 22 of [18]), an intermediate form can be used which does deal with conversion errors on output, and still outputs numbers in an efficient way. In that case, the routine 'write char' takes the following form:

```
PROC write char = (REF FILE f, CHAR char) BOOL:
  IF EXTCHAR c = C the external form of the character 'char' C;
    C if the conversion succeeds C
  THEN REF COVER cover = cover OF f;
    C the (external) character 'c' is written to the buffer of 'cover' at
      the position indicated by 'c OF cpos OF cover' C;
    c OF cpos OF cover +:= 1;
    status OF cover ORAB write back;
    TRUE
  ELSE FALSE
  FI # write char #;
```

{There is no reason to expect that internal characters still look like characters after conversion; therefore, the mode 'EXTCHAR' is used.}

The routine 'put char' (10.2.b) should then be rewritten as follows:

```
PROC ? put char = (REF FILE f, CHAR char) VOID:
  IF (write char OF f)(f, char)
  THEN REF COVER cover = cover OF f;
    IF c OF cpos OF cover > char bound OF cover
    THEN status OF cover ANDAB line end
    FI # test line end #
  ELSE CHAR k:= ".";
    BOOL mended = (char error mended OF f)(f, k);
    ensure state(f, put char status);
    IF NOT mended
    THEN error(wrongchar); k:= "."
    FI;
    IF check pos(f)
    THEN put char(f, k)
    ELSE error(nocharpos); abort
    FI
  FI # put char #;
```

The optimization used in 'put' (10.2.a) to output rows of characters efficiently cannot be used in this case. Various other obvious changes, caused by the change in the mode of 'write char' should then also be made throughout the text.

This scheme has not been used in the present model.

```
a) PROC write char = (REF FILE f, CHAR char) VOID:
     # The precondition of 'write char' is:
           . line ok (see 7.2). #
     BEGIN REF COVER cover = cover OF f;
       C The character 'char' is (possibly after conversion) written to the
         buffer of 'cover' at the position indicated by 'c OF cpos OF
         cover'. C;
       c OF cpos OF cover +:= 1;
       status OF cover ORAB write back
     END # write char #;

b) PROC write bin char = (REF FILE f, BINCHAR char) VOID:
     # The precondition of 'write bin char' is:
           . line ok. #
     BEGIN REF COVER cover = cover OF f;
       C The binary character 'char' is written to the buffer of 'cover' at
         the position indicated by 'c OF cpos OF cover'. (In binary transput
         there is no conversion.) C;
       c OF cpos OF cover +:= 1;
       status OF cover ORAB write back
     END # write bin char #;

c) PROC read char = (REF FILE f, REF CHAR char) BOOL:
     # The precondition of 'read char' is:
           . line ok (see 7.2). #
     BEGIN CHAR c = C The character read (and possibly converted) from the
                      buffer of the cover of 'f' at the position indicated
                      by 'c OF cpos OF cover OF f'. C;
       c OF cpos OF cover OF f +:= 1;
       IF C the conversion succeeds, or no conversion takes place C
       THEN char:= c; TRUE
       ELSE FALSE
       FI
     END # read char #;
```

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

4.4.3. BOOKS WITH A BUFFER SIZE LESS THAN ONE LINE

If the buffer does not correspond to one line of the text, the routines in sections 4.4.1 and 4.4.2 essentially remain the same. The routine 'init buffer' (4.4.1.a) may now be used to initialize the first buffer of a line. If the initialization pertains to a buffer which is not the first one of a line, the same routine may be used, except for the assignments to the 'char bound' field and the test for the page-end event, which should both be omitted. When reading, it may be impossible to determine the length of the line until the last buffer of that line is being read; one may then temporarily assign some large value to the 'char bound' field. Similarly, it may be impossible to determine whether or not the logical end is in the current line until the initialization of the last buffer of the last line. It seems sensible to interpret 'lfe in current line' as 'lfe in current buffer' for such files. This leads to a very minor deviation from the Revised Report in the case where the user starts to write to a sequential-access file in the last line, but not in the last buffer of that line. In that case, the logical end will be moved to the current position, which is not in accordance with the semantics of the Revised Report.

In the routine 'write buffer' (4.4.1.b), the buffer must now be filled with spaces to the end, or, in the case of the last buffer of the line, to the position indicated by the 'char bound' field.

Normal transput which takes place via 'write char OF f', 'read char OF f' and the like now assumes that the next buffer is automatically initiated as soon as the current buffer has overflowed. (This may cause some problems for files corresponding to interactive devices. For example, some interrupt is probably needed to start initialization of the next buffer. This is felt

to be very operating-system-dependent and therefore falls beyond the scope of the present model.)

The value of the current position will now, in general, not directly lead to a position in the buffer, but some offset is needed, which must be properly updated if a next buffer is initiated.

The other buffer primitives from section 4.4.1 need some obvious changes as well. In 'do newline' (4.4.1.c), one may have to write a series of buffers, rather than just one. Also, the beginning of the next line may have to be sought for before 'init buffer' can be called. The descriptions of 'do newpage' and 'do reset' remain the same. Obviously, 'do set' now has to search for the appropriate buffer, rather than the appropriate line.

If the buffer has no relation whatsoever with a line (for instance on a system where the buffer has a fixed size, say, of 512 "words"), similar changes should be made to the primitives.

5. OPENING AND CLOSING FILES

5.1. DIFFERENCES

- {U} The numerous calls of 'undefined' in the Revised Report have been assigned meanings. Due attention has been paid to hidden kinds of undefined actions like SKIP, UP gremlins and UP bfileprotect. At places where 'undefined' was called in the Revised Report, the present system issues an error message. In cases where 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; rather, opening a file is treated as a special kind of assignment.

- {U} If opening is not successful, a non-zero error code is returned, and 'undefined' is not called.

- {C6} 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 to be essentially primitive actions.

- {C7} Contrary to what is stated in the Revised Report, this model allows all upper bounds of the multiple value in 'associate' to be less than 1.

- {E} In the present model, both a default size and a maximum size is associated with each channel. The default size is used when files are "created", the maximum size is used to check the parameters of 'establish' (which is done by 'construct book'). In the Revised Report, there is one size which is used in both tests. (See also 4.2.2.n, 4.1.2.f.)

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' (j), 'lock' (k) or 'scratch' (l).

When a file is "established" on a channel, a book is constructed (4.1.2.f), with a 'text' of the given size that may be written to, and the given identification string. The logical end of the book is set to (1, 1, 1). {An implementation may require (4.1.2.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. This string may contain other auxiliary information defining properties of the book, and these may subsequently affect the conversion and buffer primitives provided, the available methods of access, etc.} 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, a file is established with a book whose text has the default size (4.2.2.n) for the channel, and whose identification string is a default identification string (4.1.2.g) for the channel.

When a file is "opened", the pool of available books is searched for a book with the following properties:

- The book may be identified by the given identification string. Again, the identification string may contain other auxiliary information;
- 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, some non-zero integer is returned which indicates why the file was not opened successfully. 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. Note that the scope of the multiple value must not be newer than the scope of the given file 'f', and that the associated multiple value will always be rectangular (i.e., all lines and pages are of the same length).

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 cover of the file. The various primitives that are associated with 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:

- 'cover'.
  A special generator is needed to create the reference to the cover of an associated file: the newly created name must have a scope which is not older than the scope of the value to which it is made to refer, nor newer than that of the file it is to become a field of. The latter check is performed syntactically. As for the first one, the scope of the cover is determined by its 'text' field, whose value is supplied by the user. Therefore, the scope of the generator must be that of the multiple value provided by the user. (For files that are "opened", "created" or "established", this scope is determined by the 'buffer' and 'book' field, which in turn are determined by the routines 'find book in system' and 'construct book' (4.1.2.e, f). Both the buffer and the book must have a global scope, since they may have to be incorporated in one of the standard files; therefore, a heap-generator can be used in these cases.)
  The fields of the cover 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', the field is left undefined, since it is never used for associated files;

- 'chan'.
  In 'establish', 'create' and 'open', the channel is provided by the user. In 'associate', the 'associate channel' (4.2.2.h) is used;

- 'buffer'.
  As a result of calling 'establish' or 'create', a buffer is obtained by the routine that constructs the book; this buffer is initialized for writing. When 'open' is called, the buffer is obtained by 'find book in system' along with the book. The buffer is not initialized; initialization has to await the first transput operation. The field 'buffer' is left undefined (and is not used) for associated files (but it is said to be not initialized);

- 'text'.
  The field 'text' is used only 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 with the logical end of the book. 'c of lpos' indicates the position up to which the current line has been filled with information. If the logical end is in the current (i.e., 1st) line, then this position coincides with the logical end. 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, and 'c of lpos' always equals 'char bound + 1'. (Note that, for associated files, this field must be maintained properly, since it is used by the routine 'set char number' (7.2.o).);

- '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. If 'open' is called, then setting these flags has to await the first transput operation. If the buffer is not initialized, both the 'buffer initialized' and 'write back' flag are not raised. As a result of calling 'establish' or 'create', both the book and the buffer are initially empty, and so both these flags are raised. 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' if 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. This is incorporated in the status information to simplify some of the tests;

- 'char bound'.

  The value of the 'char bound' field differs from case to case:
  - it is given if 'establish' is called (the parameter 'c'),
  - it is the default line length of the given channel if 'create' is called;
  - it is left undefined if 'open' is called (but becomes specified when the buffer is initialized);
  - it depends on the associated multiple value if 'associate' is called (then it is equal to 'UPB sss[1][1]', if this element exists, and 0 otherwise);

  Note that, for associated files, this field must be maintained properly, since it is used by the routine 'set char number' (7.2.o);

- 'possibles'.
  In 'establish', 'create' and 'open', the 'possibles' field is determined by the channel provided and the book that is constructed or found. In 'associate', it is such that 'reset possible', 'set possible', 'get possible', 'put possible' and 'backspace possible' always return true, while the other environment enquiries provided for files always return false ('associate poss', see 4.2.2.k);

- 'init buffer', 'write buffer', 'do newline', 'do newpage', 'do reset' and 'do set'.
  For associated files, these routines are built by the routine 'associate'. They do not make use of an intermediate buffer. For the other opening routines, they are determined by the channel provided and the book that is constructed or found, and the corresponding fields are initialized by means of a call of the routine 'buffer primitives' (4.2.2.m);

- 'piece'.
  The field selected by 'piece' is assigned a nil name;

- 'term'.
  Initially, the terminator string is the empty string;

- 'write char', 'write bin char', 'read char', 'read 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 determined by the given channel and the given book;

- 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 need not be 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 may be 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-file condition and the like. The status information is updated such that the file is no longer said to be opened.

Note that it is well-defined to open a file which is already opened, since opening a file is merely treated as a special kind of assignment. Similarly, closing a file which is not opened is a dummy operation. If a file is closed, one may want to free certain fields of the cover for optimization purposes. For example, it is recommended that the 'text' field of the cover be freed upon closing an associated file.

If opening is not successful, a non-zero error code is returned. Below, these are numbered from 2 up to 7. Error code 1 is reserved because of possible restrictions on such codes imposed by the operating environment. The error codes may obviously be refined in a specific implementation (preferably, the refinement is such that the general error codes given below can be easily extracted). The error codes are also used as parameters to the routine 'error'. This routine is not further specified (except that it should not abort the program). The error codes have been given the following symbolic names (obviously, these identifiers are not visible to the user):

(2) 'badidf' - The string argument to 'open' or 'establish' is wrong; in 'establish', the given string cannot be used as the identification string of a new book, in 'open', no matching book is found.

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

(4) 'notavail' - At this instance of time, another file may not be "opened", "created" or "established" on the given channel. {This may be a temporary problem which can be handled by retrying the calling of 'open', 'create' or 'establish' - for instance, there is no room in the directory system for a new file.}

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

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

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

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

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

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

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

(12) 'nobackspace' - An attempt is made to backspace on, or to input using a 'choice-pattern' from, a file for which 'backspace possible' returns false.

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

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

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

(16) 'nomood' - One of the layout routines is called while no mood has yet been set.

(17) 'wrongmult' - 'associate' is called with a reference to a multiple value as parameter whose bounds are incorrect.

(18) 'wrongset' - 'set' is called with a position as parameter which is beyond the logical end, and this situation is not mended by the user.

(19) 'nocharpos' - No "good" character position can be ensured.

(20) 'noline' - No "good" line can be ensured.

(21) 'nopage' - No "good" page can be ensured.

(22) 'wrongpos' - 'set char number' is called with a wrong character position as parameter.

(23) '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.

(24) '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.

(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 or the replicator of a 'collitem' or 'format pattern', 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.

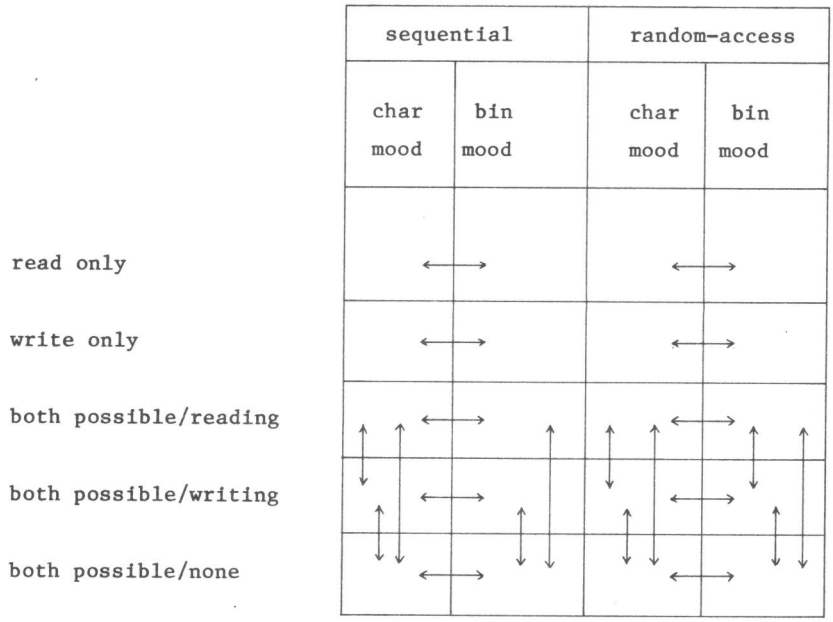
(30) 'noreidf' - An attempt is made to change the identification string of a file which cannot be re-identified.

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-access;
- 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. Implementers should feel free to omit this peculiar restriction if their operating system allows so. To this effect, the lines

```
ELIF status SAYS read to write not possible
THEN error(noalter); abort
```

should be removed from 'set write mood' (f). A similar change applies to

'set read mood' (g) in that case. On the other hand, a specific operating system may also force one to put in additional restrictions. These will then lead to some obvious extra tests to be made in the routines that set the mood (f, g, h and i below).}

{If the mood of a sequential-access file is set to write mood while the logical end is not within the current line, then the logical end is immediately moved to the current position (i.e., all information beyond the current position gets lost). If a sequential-access file is in write mood, then the logical end is always in the current line, so there will be limited possibilities for reading.}

{Note that arrows going downwards into the category "both possible/none" correspond to transitions caused by a call of 'reset'. Similarly, transitions from 'char mood' to 'bin mood' on a sequential-access file are only possible after a call of 'reset'.}

```
a) PROC establish =
      (REF FILE f, STRING idf, CHANNEL chan, INT p, l, c) INT:
    IF NOT estab possible(chan) THEN noestab
    ELIF POS(l, l, l) EXCEEDS POS(p, l, c) THEN posmin
    ELIF BUFFER buffer, BOOK book;
      INT er =
          construct book(p, l, c, idf, chan, book, buffer);
      er ≠ 0
    THEN er
    ELIF POSSIBLES possibles = access methods(book, chan);
      NOT (possibles SAYS put poss)
    THEN nowrite
    ELSE
      CONV cc = standconv(chan)(book);
      STATUS st:= establish status;
      (NOT (possibles SAYS bin poss) | st ORAB char mood);
      (NOT (possibles SAYS set poss) | st ORAB not set poss);

      f:=
        (HEAP COVER:=
            (book, chan, buffer,
             SKIP, # since the 'text' field is never used #
             POS(l, l, l),  # the current position 'cpos' #
             l,             # c of lpos #
             st,            # status #
             c,             # char bound #
             possibles,
             SKIP, SKIP, SKIP, SKIP, SKIP, SKIP
                # the buffer primitives will be assigned later #
            ),

          REF REF FORMATLIST(NIL),  # the current format #
          STRINGTOBAG "",           # terminator string #

          write char OF cc, write bin char OF cc,
          read char OF cc, read bin char OF cc,
             # the standard conversion key #
```

```
            # the "on routines" all return false initially: #
          false, false, false, false, false, false,
          (REF FILE f, REF CHAR c) BOOL: FALSE);
        buffer primitives(f);
        0
    FI # establish #;

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

c) PROC open = (REF FILE f, STRING idf, CHANNEL chan) INT:
    IF BOOK book, BUFFER buffer;
      INT er = find book in system(idf, chan, book, buffer);
      er ≠ 0
    THEN er
    ELSE CONV c = standconv(chan)(book);
      STATUS st:= open status;
      POSSIBLES possibles = access methods(book, chan);
      (NOT (possibles SAYS put poss) | st ORAB read mood);
      (NOT (possibles SAYS bin poss) | st ORAB char mood);
      (NOT (possibles SAYS set poss) | st ORAB not set poss);

      f:=
        (HEAP COVER:=
            (book, chan, buffer,
             SKIP,          # the 'text' field is never used #
             POS(1, 1, 1), # the current position 'cpos' #
             SKIP,          # 'c of lpos' will be initialized by
                              'init buffer' #
             st,            # status #
             SKIP,          # 'char bound' will be initialized by
                              'init buffer' #
             possibles,
```

```
            SKIP, SKIP, SKIP, SKIP, SKIP, SKIP
               # the "buffer primitives" will be assigned later #
            ),

          REF REF FORMATLIST(NIL),  # the current format #
          STRINGTOBAG "",           # terminator string #

          write char OF c, write bin char OF c,
          read char OF c, read bin char OF c,
            # the standard conversion key #

            # on routines: #
          false, false, false, false, false, false,
          (REF FILE f, REF CHAR c) BOOL: FALSE);
        (NOT (possibles SAYS get poss) | set write mood(f));
        buffer primitives(f);
        0
      FI # open #;

d) PROC associate = (REF FILE file, REF [][][] CHAR sss) VOID:
      IF INT lp = LWB sss, up = UPB sss;
         INT ll = (up ≥ lp | LWB sss[lp] | 1),
             ul = (up ≥ lp | UPB sss[lp] | 0);
         INT lc = (ul ≥ ll | LWB sss[lp][ll] | 1),
             uc = (ul ≥ ll | UPB sss[lp][ll] | 0);
        lp ≠ 1 OR ll ≠ 1 OR lc ≠ 1
      THEN error(wrongmult); abort
      ELSE
        file:=
         (C a newly created name which is made to refer to the yield of an
            actual-cover-declarer and whose scope is equal to the scope of
            'sss' C
```

```
:= (SKIP,              # the book is not needed #
    associate channel,# see 4.2.2.h #
    SKIP,              # the buffer is not used either #
    sss,               # the text, directly accessed by the
                         buffer and conversion primitives #
    POS(1, 1, 1),      # the current position 'cpos' #
    uc + 1,            # 'c of lpos': this field has to
                         be maintained properly #
    associate status,  # see 6.2.f; note that the buffer is
                         said to be not initialized #
    uc,                # 'char bound': this field has to be
                         maintained properly #
    associate poss,    # see 4.2.2.k #

    # init buffer #
       (REF FILE f) VOID:
         IF REF COVER cover = cover OF f;
           REF STATUS status = status OF cover;
           status ORAB buffer initialized;
           REF INT char bound = char bound OF cover,
                   clpos = c of lpos OF cover;
           p OF cpos OF cover > UPB text OF cover
         THEN # file ended #
           char bound:= 0; clpos:= 1;
           status ANDAB
                 (status SAYS read mood | logical file ended
                                        | physical file end)
         ELIF 1 OF cpos OF cover > UPB (text OF cover)[1]
         THEN # page ended #
           char bound:= 0; clpos:= 1;
           status ANDAB page end
         ELIF char bound:= UPB (text OF cover)[1][1];
              (clpos:= char bound + 1) = 1
         THEN status ANDAB line end
         ELSE status ORAB not line end
         FI,

       # write buffer # SKIP,
```

```
# do newline #
(REF FILE f) VOID:
  BEGIN REF COVER cover = cover OF f;
    REF POS cpos = cpos OF cover;
    l OF cpos +:= l; c OF cpos:= l;
    (init buffer OF cover)(f)
  END,

# do newpage #
(REF FILE f) VOID:
  BEGIN REF COVER cover = cover OF f;
    REF POS cpos = cpos OF cover;
    cpos:= (p OF cpos + l, l, l);
    (init buffer OF cover)(f)
  END,

# do reset #
(REF FILE f) VOID:
  BEGIN REF COVER cover = cover OF f;
    cpos OF cover:= (l, l, l);
    status OF cover:= associate status
  END,
```

```
# do set #
(REF FILE f, INT p, l, c) VOID:
  IF POS(l, l, l) EXCEEDS POS(p, l, c)
  THEN error(posmin); abort
  ELIF REF COVER cover = cover OF f;
    REF STATUS status = status OF cover;
    INT up = UPB text OF cover + l;
    POS lpos = (up + l, l, l),
    REF POS cpos = cpos OF cover,
    REF INT char bound = char bound OF cover,
            clpos = c of lpos OF cover;
    status:= associate status OR buffer initialized OR
         (status AND read or write mood);
    NOT (lpos BEYOND POS(p, l, c))
                # i.e., cpos ≥ lpos #
  THEN STATUS reading = state(f);
    cpos:= lpos; char bound:= 0; clpos:= l;
    status ANDAB
       (status SAYS read mood | logical file ended
                              | physical file end);
    IF POS(p, l, c) BEYOND lpos
    THEN BOOL mended = (logical file mended OF f)(f);
      ensure state(f, reading);
      (NOT mended | error(wrongset); abort)
    FI
  ELIF # 0 < p ≤ UPB text OF cover #
    INT ul = UPB (text OF cover)[l] + l;
    INT uc = (ul > l | UPB (text OF cover)[l][l] + l | l);
    l > ul OR c > uc
  THEN error(posmax); abort
  ELIF cpos:= (p, l, c); char bound:= uc - l; clpos:= uc;
    l = ul
  THEN status ANDAB page end
  ELIF c = uc
  THEN status ANDAB line end
  FI),   # end of cover initialization #
```

```
        REF REF FORMATLIST(NIL),  # the current format #
        STRINGTOBAG "",           # terminator string #

        # write char #
           (REF FILE f, CHAR char) VOID:
             BEGIN REF COVER cover = cover OF f;
               REF POS cpos = cpos OF cover;
               (text OF cover)[p OF cpos][1 OF cpos][c OF cpos]
                    := char;
               c OF cpos +:= 1
             END,

        # write bin char # SKIP,

        # read char #
           (REF FILE f, REF CHAR char) BOOL:
             BEGIN REF COVER cover = cover OF f;
               REF POS cpos = cpos OF cover;
               char:= (text OF cover)[p OF cpos][1 OF cpos]
                                    [c OF cpos];
               c OF cpos +:= 1; TRUE
             END,

        # read bin char # SKIP,

           # on routines: #
        false, false, false, false, false, false,
        (REF FILE f, REF CHAR c) BOOL: FALSE)
    FI # associate #;

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

```
f) PROC ? set write mood = (REF FILE f) VOID:
     # opened and (in general) NOT write mood #
     IF REF COVER cover = cover OF f;
       REF STATUS status = status OF cover;
       NOT (possibles OF cover SAYS put poss)
     THEN error(nowrite); abort
     ELIF status SAYS read to write not possible
     THEN error(noalter); abort
     ELSE
       IF NOT (status SAYS buffer initialized)
       THEN (init buffer OF cover)(f)
       FI;
       status ANDAB NOT read mood ORAB write mood ORAB
                 logical pos ok;
       IF status SAYS (write sequential OR not lfe in current line)
       THEN set logical pos(f);
         C the size of the current line and page and of all subsequent lines
           and pages may be expanded (e.g., to the sizes with which the book
           was originally opened) C;
         # this pseudo-comment replaces the one in the routine 'put char' of
           the Revised Report #
         IF status SAYS line ok
         THEN status ORAB write back
               # since the rest of the line gets lost #
         FI
       FI
     FI # opened & write mood & logical pos ok #;
```

```
g) PROC ? set read mood = (REF FILE f) VOID:
     # opened and (in general) NOT read mood #
     IF REF COVER cover = cover OF f;
       REF STATUS status = status OF cover;
       NOT (possibles OF cover SAYS get poss)
     THEN error(noread); abort
     ELIF status SAYS write to read not possible
     THEN error(noalter); abort
     ELSE status ORAB read mood ANDAB NOT write mood;
       IF status SUGGESTS lfe in current line
       THEN mind logical pos(f)
         # to trap the case where 'c of lpos ≤ c OF cpos' #
       FI
     FI  # opened & read mood #;

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

i) PROC ? set bin mood = (REF FILE f) VOID:
     # opened and (in general) NOT bin mood  #
     IF REF COVER cover = cover OF f;
       REF STATUS status = status OF cover;
       NOT (possibles OF cover SAYS bin poss)
     THEN error(nobin); abort
     ELIF status SAYS char to bin not possible
     THEN error(noshift); abort
     ELSE status ORAB bin mood ANDAB NOT char mood
     FI  # opened & bin mood #;
```

```
j) PROC close = (REF FILE f) VOID:
     IF REF COVER cover = cover OF f;
       status OF cover SAYS opened
     THEN
       IF status OF cover SAYS write back
       THEN (write buffer OF cover)(f)
       FI;
        # note that this may cause a physical file end event #
       status OF cover OF f ANDAB closed;
       C The information (in the book) on the number of users is updated.
         Some system-task may be activated to actually close the book; in
         this case, the book may be re-opened. C
     FI;

k) PROC lock = (REF FILE f) VOID:
     IF REF COVER cover = cover OF f;
       status OF cover SAYS opened
     THEN
       IF status OF cover SAYS write back
       THEN (write buffer OF cover)(f)
       FI;
       status OF cover OF f ANDAB closed;
       C The information on the number of users is updated. Some system-task
         may be activated to actually lock the book; in this case, it is not
         possible to re-open the book. C
     FI;

l) PROC scratch = (REF FILE f) VOID:
     IF REF COVER cover = cover OF f;
       status OF cover SAYS opened
     THEN
       IF status OF cover SAYS write back
       THEN (write buffer OF cover)(f)
       FI;
       status OF cover 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
    FI;
```

6. POSITION ENQUIRIES

6.1. DIFFERENCES

- {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 the cover 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 buffer should ultimately be written to the book;
- 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 {which is never true if the file is in write mood};
- 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 13, some trivial modifications have to be made in the definitions given below (d, e and f).) An implementer may also wish to incorporate other (machine-dependent) information in the status.

There are a number of invariants that hold between the various bits of the status; these invariants are listed in section 4.3.2.

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 3, the current position 'cpos' and the logical position 'lpos' are related through

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

where 'cpos1' and 'lpos1' are the (implicit) variables maintained by the system described in this report. If one of the above mentioned routines is called, relation (*) may no longer be valid if lpos1 < cpos1. For this purpose, the routine 'mind logical pos' (g) has been provided. Since the

logical position is of importance also when reading and writing is alternated, 'mind logical pos' is also called from within 'set read mood' (5.2.g).

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) if 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. The 'line end' and 'logical file end' information is updated in the text. Updating 'buffer initialized', 'page end', 'physical file end' and 'logical file end in current line' information obviously is one of the tasks of the primitive routines 'init buffer', 'write buffer' and the like.

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

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

c) PROC page number = (REF FILE f) INT:
     IF REF COVER cover = cover OF f;
       status OF cover SAYS opened
     THEN p OF cpos OF cover
     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, where the numbers refer to
       the columns in (f) below):

       bit  1 = 1 <=> the file is opened;

       bit  2 = 1 <=> the buffer is initialized;
       bit  3 = 1 <=> the buffer should ultimately be written back;

       bit  4 = 1 <=> NOT line ended;
       bit  5 = 1 <=> NOT page ended;
       bit  6 = 1 <=> NOT physical file ended;

       bit  7 = 1 <=> NOT logical file ended;
       bit  8 = 1 <=> NOT lfe in current line;

       bit  9 = 1 <=> read mood;
       bit 10 = 1 <=> write mood;
       bit 11 = 1 <=> char mood;
       bit 12 = 1 <=> bin mood;

       bit 13 = 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 = 5,
   OP SAYS = (STATUS s, t) BOOL: s ≥ t;

   PRIO ? SUGGESTS = 5,
   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).}
```

```
   {Each of the constants given below is either always used in combination
with 'SAYS' and 'ORAB', or it is always used in combination with 'SUGGESTS'
and 'ANDAB'. The latter are marked with a (*).}

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

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

          ? logical pos ok             = 2r 1 00 000 10 0000 0,
          ? logical file ended         = 2r 1 11 111 00 1111 1,  (*)

          ? opened                     = 2r 1 00 000 00 0000 0,
          ? closed                     = 2r 0 00 000 00 0000 0,  (*)

          ? buffer initialized         = 2r 0 10 000 00 0000 0,
          ? write back                 = 2r 0 01 000 00 0000 0,

          ? not lfe in current line    = 2r 0 00 000 01 0000 0,
          ? lfe in current line        = 2r 1 11 111 10 1111 1,  (*)

          ? line end                   = 2r 1 11 011 11 1111 1,  (*)
          ? not line end               = 2r 0 10 111 00 0000 0,
          ? page end                   = 2r 1 10 001 11 1111 1,  (*)
          ? not page end               = 2r 0 00 011 00 0000 0,
          ? physical file end          = 2r 1 10 000 11 1111 1,  (*)

          ? not set poss               = 2r 0 00 000 00 0000 1,
          ? write sequential           = 2r 0 00 000 00 0100 1,

          ? establish status           = 2r 1 11 111 10 0100 0,
          ? open status                = 2r 1 00 111 11 0000 0,
          ? associate status           = 2r 1 00 111 11 0010 0,
```

```
        ? read mood                    = 2r 0 00 000 00 1000 0,
        ? write mood                   = 2r 0 00 000 00 0100 0,
        ? char mood                    = 2r 0 00 000 00 0010 0,
        ? bin mood                     = 2r 0 00 000 00 0001 0,
        ? read or write mood           = 2r 0 00 000 00 1100 0,
        ? not read or write mood       = 2r 1 11 111 11 0011 1,  (*)

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

g) PROC ? mind logical pos = (REF FILE f) VOID:
     # opened #
     IF REF COVER cover = cover OF f;
       REF STATUS status = status OF cover;
       status SAYS not lfe in current line
     THEN SKIP
     ELIF c OF cpos OF cover ≥ c of lpos OF cover
     THEN c of lpos OF cover:= c OF cpos OF cover;
       IF status SAYS read mood
       THEN status ANDAB logical file ended
       FI
     FI # lpos ≥ cpos #;
```

6.3. EFFICIENCY

The efficiency of the transput system critically depends on the way the operators SAYS and SUGGESTS are implemented. For instance, it will very often not be necessary for these operators actually to yield a value of the mode specified by BOOL. In an enquiry-clause, the result of comparing two status values might be kept in a condition register, subsequently to be tested.

If the compiler is not able to detect cases of the form

```
IF NOT (status SAYS ... ),
```

an implementer may wish to introduce operators 'NOTSAYS' and 'NOTSUGGESTS', so that the above statement might be written as

```
IF status NOTSAYS ...,
```

in which case the same optimization techniques can be applied.

7. LAYOUT ROUTINES

7.1. DIFFERENCES

- {C31} On a sequential-access file, output immediately causes the logical end to be moved to the current position, unless both are in the same line. The present model behaves slightly differently from the Revised Report in this respect. In the Revised Report, the layout routines 'space', 'newline' and 'newpage' simply advance the current position in this case, thus skipping over the information already present in the book. This requirement is felt too difficult to fulfill on write-only sequential-access files. Therefore, on writing, a call of one of the above layout routines with a sequential-access file as parameter also leads to moving the logical end to the current position in the present model. If reading is possible, the skip effect may still be obtained by calling 'get(f, space)', 'get(f, newline)' and 'get(f, newpage)', respectively.

  Furthermore, calls like 'put(f, ())' or 'put(f, "")' are treated as dummy operations in this respect by the Revised Report, i.e., the logical end is not moved to the current position by one of these calls.

  In the present model, each operation which will leave the file in write mood is considered to be an output operation. Consequently, each such operation causes the logical end to be moved to the current position, provided it is not within the current line already. To this effect, the routine 'set logical pos' (4.1.2.h), which performs the desired action, is called from within the routine 'set write mood' (5.2.f), rather than from within 'put char', as is the case in the Revised Report. On random-access files, the layout routines still skip over the information already present in the book since the logical end is not moved to the current position on such files.

- {E} '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, calling '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.}

- {D} The routine 'get good file' in the Revised Report serves a twofold purpose: on reading, it tests 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'.

- {C27} It is generally the case that, in each situation where an event routine is called, a default action is provided and is taken if the routine returns false (even if the user has actually mended the situation). In the Revised Report, however, there are three exceptions to this, brought about when the position is not "good": viz., when a string is being input with 'get', or any value is being input with 'getf', or any value is being output with 'putf'. In these cases, the default action is only taken if the position remains "bad", and the value returned by the event routine makes no difference (except insofar as it may prevent any further attempts to fix the situation).

  To obtain a more consistent treatment of event routines, the default action when an event routine returns false is <u>always</u> taken in the present model.

  To accomodate this, a different routine 'check pos' (1) has been provided. In the present model, 'check pos' returns a boolean which indicates whether or not the user has mended the situation in an appropriate way. At each place where 'check pos' is called, this boolean is inspected to see whether or not the system should continue with its default action. (As a consequence of this change, the tests for the line and/or logical file being ended which are made at the

start of the routines 'put char' and 'get char' in the Revised Report have become obsolete.)

{From the user's point of view, the difference is very minor: it only becomes apparent if the event routine corresponding to the 'on line end' event causes the current position to be changed, while it nevertheless returns false. In the present model, the system continues with its default action since the event routine returned false. In the Revised Report, another attempt is made to fix a page overflow, if any; after that, the default action is only taken if the situation remains "bad".}

- {S} In the Revised Report, the mood must be set properly prior to a call of 'set'. In the present model this requirement is only made if the user attempts to set the file to a position which is at or beyond the logical end of the file. This change makes it in general possible to set a file after a call of 'open' without first having to call 'put' or 'get' in order to properly set the mood.

7.2. NEW DEFINITION

A book input from an external medium by some system-task may contain lines and pages not all of which are 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 blank is written (similar action being taken in the case of 'newline' and 'newpage' if the book is not compressible). Note that, when outputting to a sequential-access file, the logical end is always in the current line. In that case, 'space' is only treated as a skip operation if the logical end is beyond the current position. 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, since in both cases the logical end is first moved to the current position, provided it is not within the current line already. (The first call of 'space' will skip over the character "a" just written, while the second one will write a blank; 'print(blank)' writes a blank in both cases.)

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

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 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 (except for associated files).

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 (except possibly in the case of a call of 'set'; see 7.1), an error message is given and the program is aborted. On exit, the read/write mood present on entry is restored.

The layout routines 'newline', 'newpage', 'reset' and 'set' all (potentially) move the current position to another line (and hence to another buffer). Therefore, part of these routines is considered primitive; the semantics of these primitives are further discussed in section 4.4.1.

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 'backspace', '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 and unformatted input of strings (which make 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 cannot 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.

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 defined (i.e., the file is either in read mood or in write mood).
- physical file ok: the file has been opened, the read/write mood is correct (i.e., as on entry) 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 invariants may be observed:

line ok IMPLIES page ok,
page ok IMPLIES physical file ok,
physical file ok IMPLIES mood ok,
mood ok IMPLIES opened,

and

logical file ok IMPLIES physical file ok.
}

In terms of the above conditions, the pre- and post-conditions (if the routine returns true) 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
       IF state(f) SAYS line ok
       THEN TRUE
       ELSE ensure line(f)
       FI
     THEN  # line ok #
       REF COVER cover = cover OF f;
       REF STATUS status = status OF cover;
       REF INT c = c OF cpos OF cover;
       IF c < c of lpos OF cover
       THEN (read char OF f)(f, LOC CHAR);
         IF c = c of lpos OF cover
         THEN  # line end or logical file end #
           IF status SAYS not lfe in current line
           THEN status ANDAB line end
           ELIF status SAYS read mood
           THEN status ANDAB logical file ended
           ELIF c > char bound OF cover
           THEN status ANDAB line end
           FI
         FI
       ELSE # write at logical end #
         IF status SAYS bin mood
         THEN (write bin char OF f)(f, SKIP)
            # or something more suitable #
         ELSE status ORAB char mood;
               # default mood = char mood #
           (write char OF f)(f, ".")
         FI;
         (c > char bound OF cover | status ANDAB line end)
       FI
     ELSE error(nocharpos); abort
     FI;
```

```
b) PROC backspace = (REF FILE f) VOID:
     IF state(f); # to ensure that the mood has been set #
       REF COVER cover = cover OF f;
       NOT (possibles OF cover SAYS backspace poss)
     THEN error(nobackspace); abort
     ELSE mind logical pos(f);  # lpos ≥ cpos #
       REF INT c = c OF cpos OF cover;
       (c > 1 | c -:= 1 | error(wrongbacksp); abort);
       status OF cover ORAB line ok
     FI  # logical file ok & NOT line ended #;

c) PROC newline = (REF FILE f) VOID:
     IF
       IF state(f) SAYS page ok
       THEN TRUE
       ELSE ensure page(f)
       FI
     THEN  # page ok #
       REF COVER cover = cover OF f;
       REF STATUS status = status OF cover;
       IF status SUGGESTS lfe in current line AND
              status SAYS read mood
       THEN c OF cpos OF cover:= c of lpos OF cover;
         status ANDAB logical file ended;
         newline(f)
           # which will immediately lead to a call of the event routine
             corresponding to 'on logical file end' #
       ELSE (do newline OF cover)(f)
       FI
     ELSE error(noline); abort
     FI;
```

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

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

f) PROC ? ensure state = (REF FILE f, STATUS reading) VOID:
     IF NOT (status OF cover 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) BOOL:
     # logical file ended & mood ok #
     BEGIN STATUS old = status OF cover OF f;
       BOOL mended = (logical file mended OF f)(f);
       ensure state(f, old);
       REF STATUS status = status OF cover OF f;
       IF NOT (status SAYS buffer initialized)
       THEN (init buffer OF cover OF f)(f)
       FI;
         # buffer initialized #
       IF status SAYS logical pos ok
       THEN TRUE
       ELIF mended
       THEN ensure logical file(f)
       ELSE FALSE
       FI
     END;
```

```
h) PROC ? ensure physical file = (REF FILE f) BOOL:
     # the mood is correct, the file generally not #
     IF REF STATUS status = status OF cover OF f;
       IF NOT (status SAYS buffer initialized)
       THEN (init buffer OF cover OF f)(f)
       FI;
         # buffer initialized #
       IF status SAYS logical pos ok
       THEN TRUE
       ELSE ensure logical file(f)
       FI
     THEN  # logical file ok #
       STATUS old = status OF cover OF f;
       IF old SAYS physical file ok
       THEN TRUE
       ELSE  # physical file ended #
         BOOL mended = (physical file mended OF f)(f);
         ensure state(f, old);
         IF mended
         THEN ensure physical file(f)
         ELSE error(nopage); abort
         FI
       FI
     ELSE FALSE
     FI;
```

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

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

```
k) PROC ? next pos = (REF FILE f) VOID:
     # the mood is ok, the line is not #
     IF NOT ensure line(f)
     THEN error(nocharpos); abort
     FI  # line ok #;

l) PROC ? check pos = (REF FILE f) BOOL:
     # the mood is ok, the line generally not #
     IF
       IF status OF cover OF f SAYS page ok
       THEN TRUE
       ELSE ensure page(f)
       FI
     THEN  # page ok #
       STATUS old = status OF cover OF f;
       IF old SAYS line ok
       THEN TRUE
       ELSE  # line ended #
         BOOL mended = (line mended OF f)(f);
         ensure state(f, old);
         (mended | check pos(f) | FALSE)
       FI
     ELSE FALSE
     FI;

m) PROC set = (REF FILE f, INT p, l, c) VOID:
     IF NOT set possible(f)
     THEN error(noset); abort
     ELSE mind logical pos(f);  # lpos ≥ cpos #
       (do set OF cover 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);  # lpos ≥ cpos #
       (do reset OF cover OF f)(f);
       REF COVER cover = cover OF f;
       POSSIBLES possibles = possibles OF cover,
       REF STATUS st = status OF cover:= open status;
       (NOT (possibles SAYS put poss) | st ORAB read mood);
       (NOT (possibles SAYS bin poss) | st ORAB char mood);
       (NOT (possibles SAYS set poss) | st ORAB not set poss);
       (NOT (possibles SAYS get poss) | set write mood(f))
     FI;
```

```
o) PROC set char number = (REF FILE f, INT c) VOID:
     IF REF COVER cover = cover OF f;
       IF NOT (state(f) SAYS buffer initialized)
       THEN (init buffer OF cover)(f)
       ELSE mind logical pos(f)
       FI;
       c ≥ 1 AND c < c of lpos OF cover
     THEN # handle simple case #
       IF c < c OF cpos OF cover AND
            NOT (possibles OF cover SAYS backspace poss)
       THEN error(nobackspace); abort
       FI;
       c OF cpos OF cover:= c;
       status OF cover ORAB line ok
     ELSE
       WHILE INT ccpos = c OF cpos OF cover OF f; ccpos ≠ c
       DO
         IF c < 1 OR c > char bound OF cover OF f + 1
         THEN error(wrongpos); abort
         ELIF c > ccpos
         THEN space(f)
         ELSE backspace(f)
         FI
       OD
     FI;
```

7.3. EFFICIENCY

Before any transput operation may take place, the status of the file is checked for the following:

i) the file must be opened,
ii) the read/write mood and char/bin mood must be ensured, and
iii) the current position must be "good".

Normally, the checks for the current position are performed by one of the ensure routines (g, h, i and j), or by 'next pos' (k) or 'check pos' (l). In the present model, one overall test is made to detect the "normal"

situation prior to calling one of these routines. Thus the precondition of all these calls is that the current position is not "good". This overall test, however, is not made in case one of these routines is called recursively, since this will probably not occur very often.

For most transput operations, the line must be ensured. This is generally done by calling either 'next pos' (k) or 'check pos' (1). Except for the recursive call of 'check pos', the precondition of both these routines is: NOT line ok, i.e., the line has overflowed. Most often, this will correspond to a normal line overflow, where the page and/or file have not overflowed. If the user has not provided an event routine corresponding to the 'on line end' event, the default action will then be to call 'newline' in case of a call of 'next pos', while 'check pos' would return false. Suppose the status includes a flag "no line mended routine provided", to be raised in case the user has provided no such routine. After opening the file there would be no routine provided and the flag must be lowered as soon as the user provides one by means of a call of 'on line end' (4.3.2.r). If we introduce a constant 'default line mended' of mode 'STATUS' such that it indicates

no line mended routine provided & not page end,

then 'next pos' could be written as:

```
PROC ? next pos = (REF FILE f) VOID:
  IF status OF cover OF f SAYS default line mended
  THEN newline(f); next pos(f)
  ELIF NOT ensure line(f)
  THEN error(nocharpos); abort
  FI.
```

'check pos' could then start as follows:

```
  IF status OF cover OF f SAYS default line mended
  THEN FALSE
  ELSE ... ,
```

or, since each call of this routine (except for the recursive one) is of the

form

```
    IF status OF cover OF f SAYS line ok
    THEN TRUE
    ELSE check pos(f)
    FI,
```

one could rewrite these statements as

```
    IF status OF cover OF f SAYS line ok
    THEN TRUE
    ELIF status OF cover OF f SAYS default line mended
    THEN FALSE
    ELSE check pos(f)
    FI,
```

and leave 'check pos' as it is.

The checks for the file being opened and the mood being set appropriately are, in general, made by calling 'ensure state' (f). This routine is called at the start of 'put', 'get' and the like, and after each call of a routine which is (or may be) provided by the user. Most such calls occur after the calling of some event routine. Since these cases are expected to occur rather seldom, whether it is necessary to call 'ensure state' at all in these cases is not tested.

8. CONVERSION ROUTINES

8.1. DIFFERENCES

- {C9} In the set of conversion routines given in the Revised Report, it is not always clear exactly when 'undefined' is called. It was the intention to call 'undefined' only when, whatever the value of the first parameter, no string can be delivered satisfying the constraints set by the other parameters. This intention has been observed in the present model. Moreover, 'undefined' is never called; rather, a string filled with 'errorchar' is just returned.

  If 'width' = 0, it is intended that the shortest possible string containing the correct result should be returned (except in 'float' where the 'width' parameter should never be zero). Following this intention, if a string of 'errorchar's is to be returned, a string of length 1 is returned in the present model, rather than the empty string as prescribed in the Revised Report.

- {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 routines 'subfixed' and 'log10', the operator 'EXPLENGTH', and a few lines in 'string to 1 real' use real arithmetic.

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

- {C25} The environment enquiries 'L int width', 'L real width' and 'L exp width' are primarily provided in order to fix the numbers of digits to be allowed for when numbers are output using 'put'. In the case of real (and hence also complex) numbers, a sensible choice of 'L real width' will depend upon how accurately the implementation actually performs its conversion in 'put'. (A numerical analyst would expect, in an ideal world, that conversions would produce sufficient digits to ensure that different real numbers are always converted to different strings. In actual implementations this may not be practicable.) 'L exp width' should reflect the manner in which 'L max real' is actually converted. In the present model, 'L real width' and

'L exp width' are therefore defined by means of pseudo-comments which describe the intended meaning.

8.2. NEW DEFINITION

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', 'log10' and 'EXPLENGTH'. Unless the exponent in 'float' may be of the order of magnitude of 'max int', the integral arithmetic presents no trouble;

- Arithmetic values 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 values like 'L max real' may be converted using 'fixed' or 'float';

- The routines are written non-recursively;

- The routines do not distinguish between various lengths of numbers.

- Care has been taken to avoid string operations. The routines start by establishing a reasonable upper bound on the width of the converted value, and a reference-to-row-of-character of that length is used thereafter.

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 to be 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).

For an exhaustive list of examples of the use of these routines, see Commentary 18 from [18].

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 value being converted, digits after the decimal point are sacrificed. If the number of digits after the decimal point is reduced to zero and the value still does not fit, a string filled with 'errorchar' is returned. The routine uses 'subfixed' for the actual conversion, and 'log10' to estimate the width of the integral part of the value submitted.

Examples:
'fixed(x, -6, 3)' might yield ".2.718", "27.183", "271.83" (in which one place after the decimal point has been sacrificed in order to fit the number in), "2718.3", ".27183" or "271828" (in the last two examples, all positions after the decimal point have been 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. 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 before the exponent. If the value of the 'exp' parameter is zero, 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 blank 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.) At each step during this process, a check must be made to ensure that some positions for digits remain; if this is not the case, a string filled with 'errorchar' is returned. The routine uses 'subfixed' for the actual conversion, and the operator 'EXPLENGTH' to determine the width needed for the exponent.

Examples:

'float(x, 9, 3, 2)' might yield "-2.718e+0", "+2.72e+11" (in which 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" (when 'x' has the value 0.996e-9).

The routine 'subfixed' performs the actual conversion from numbers to strings. The parameter 's' receives the representation of the value submitted; the digits are placed from position 1 onwards. Position 0 is always filled with the character "0", while the positions numbered 'LWB s' up to 0 are filled with spaces. When called from 'fixed', the routine delivers all digits from the integral part, followed by ".", followed by 'after+1' digits from the fractional part. When called from 'float', the

first 'i+after+1' significant digits are delivered and the decimal point is placed after the i-th such digit, where i is the initial value of the parameter 'p'. 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.) The parameter 'p' receives the position of the decimal point of the value submitted, where the position to the right of the first digit is counted positive and to the left zero or negative. The routine returns an indication of the sign of the value submitted (true iff negative).

The routine 'subfixed' should adhere to the best possible principles of numerical analysis. Rather than an ALGOL 68 routine, a semantic definition is given below.

The (hidden) routines 'power10' and 'round' are used for rounding. For 'power10', the parameter 's' refers to the string to be rounded, the parameter 'last' is the position where the rounding process starts, and the parameter 'rp' will obtain as value the index P in 's' such that 's[P+1]' ... 's[last]' are all equal to "9" (or ".") if 's[last+1]' $\geq$ 5, and, otherwise, 'rp=last+1'. The routine delivers true if the rounding causes a carry out of the leftmost digit of the number in 's'. At a subsequent stage, 'round' is called with the same parameters 's', 'rp' and 'last'. This routine performs the actual rounding. If 'rp' > 'last', no action is performed since the number may just be truncated at the position indicated by 'last'. Otherwise, 's[rp]' gets as value the character representing the next higher digit, and 's[rp+1]' ... 's[last]' all get the value "0" (except at the position of the decimal point). The rounding is done in two steps, since 'float' may necessitate several calls of 'power10' before the actual rounding may take place.

```
a) MODE ? NUMBER = UNION({L REAL}, {L INT});

b) PROC whole = (NUMBER v, INT width) STRING:
     CASE v IN
       (UNION({L INT}) x):
         (INT abs width = ABS width;
          INT upb = abs width MAX INTWIDTH x;
          INT lwb:= upb - abs width + 1;
          INT first:= lwb, [0 : upb] CHAR s;
          IF BOOL neg = subwhole(x, first, upb, s); neg OR width > 0
          THEN s[first -:= 1]:= (neg | "-" | "+")
          FI;
            # the converted number (including a possible sign) is stored in
              's[first : ]', while the elements 's[lwb MAX 0]' ...
              's[first-1]' contain spaces #
          (width = 0 | lwb:= first);  # no leading spaces needed #
          IF first ≥ lwb
          THEN s[lwb : ]
          ELSE abs width * errorchar
          FI)
     OUT fixed(v, width, 0)
     ESAC;

c) PROC fixed = (NUMBER v, INT width, after) STRING:
     IF INT abs width = ABS width,
        BOOL poswidth = width > 0, zerowidth = width = 0;
        INT point:= log10(v) - 1, length:= abs width - ABS poswidth;
          # there will be either 'point' or 'point+1' digits before the
            decimal point #
       after < 0 OR
       NOT zerowidth AND (after ≥ length OR point > length)
          # a partial test for the correctness of the parameters #
     THEN 1 MAX abs width * errorchar
```

```
ELIF INT aft = (zerowidth | after | after MIN (length - point));
  # now aft equals the maximum number of digits to be delivered after
    the decimal point #
  INT upb = point + aft + 3;
  [0 MIN (upb - abs width) - 2 : upb] CHAR s;
  BOOL neg = subfixed(v, aft, point, s, FALSE);
    # 's[1 : point + aft + 1]' contains the relevant digits,
      's[point]' = ".", s[0] = "0" #
  INT last:= point + aft, first:= 1, rp;
  IF zerowidth THEN length:= last
  ELSE length:= abs width - ABS(neg OR poswidth);
    last:= last MIN length
  FI;
    # 'last' is the index of the last character that can be returned;
      'length' is the total space available for digits and decimal
      point #
  IF power10(s, rp, last)
  THEN first:= 0;
    (NOT zerowidth AND last = length | last -:= 1)
      # decrement 'last' because of rounding #
  FI;
  (last = point | last -:= 1);
   # the result should not end with "." #
  (point = 1 AND last < length | first:= 0);
   # deliver "0" or "0.xxx" #
  point > last + 1 OR last < first
    # definitive test for the correctness of the parameters #
THEN abs width * errorchar
ELSE s[first - 1]:= (neg | "-" |: poswidth | "+" | ".");
  round(s, rp, last);
  s[(zerowidth | first - ABS neg | last - abs width + 1) : last]
FI;
```

```
d) PROC float = (NUMBER v, INT width, after, exp) STRING:
     IF INT abs width = ABS width, sign after = SIGN after,
       INT exp places:= ABS exp;
       INT last:= abs width - exp places - 2;
       INT before:= last - after - sign after;
       SIGN before + sign after ≤ 0
         # partial test for correctness of parameters #
     THEN ! MAX abs width * errorchar
     ELIF INT first:= 1, exponent:= before, rp:= last + 1;
       [-1 : abs width - sign after] CHAR s, BOOL exp sign = exp > 0;
       BOOL neg:= subfixed(v, after, exponent, s, TRUE);
         # 's[1 : before + after + 2]' contains the relevant digits,
           's[before + 1]' = "." #
       exponent -:= before;
         # now 'exponent' is the real exponent #
       WHILE
         IF rp > last
         THEN
           IF power10(s, rp, last)
           THEN first:= 0; s[before]:= ".";
             s[before + 1]:= (before = 0 | rp:= 1; "0" | "9");
             before -:= 1; last -:= 1; exponent +:= 1
           FI
             # move the decimal point one place to the left and adjust the
               various parameters #
         FI;
         exp sign EXPLENGTH exponent > exp places AND last ≥ first
           # the exponent does not fit and it is still possible to sacrifice
             digits #
```

```
    DO last -:= 1; exp places +:= 1;
      CASE SIGN(last - before - 1) + 2
      IN
        (before -:= 1; exponent +:= 1),
          # "after" = 0, so decrement 'before' #
        (before +:= 1; exponent -:= 1;
         REF CHAR sb1 = s[before + 1]; s[before]:= sb1; sb1:= ".")
          # "after":= 0, so the decimal point also disappears; as a
            consequence, 'before' can be incremented #
    # OUT "after" is decremented, but still greater than 0 #
      ESAC
    OD;
    last < first   # no digits left in mantissa #
  THEN abs width * errorchar
  ELSE round(s, rp, last);
    INT p:= first;
    WHILE s[p] = "0" AND s[p + 1] ≠ "." AND p < last
    DO s[p]:= "."; p +:= 1 OD;
      # change "000.00" to "..0.00" #
    s[p - 1]:= (neg | "-" |: width > 0 | "+" | ".");
    s[last +:= 1]:= "e"; INT l = last + exp places;
      # convert exponent into 's[last + 1 : l]': #
    neg:= subwhole(exponent, last +:= 1, l, s);
    (neg OR exp sign | s[last - 1]:= (neg | "-" | "+"));
      # place sign of exponent #
    s[first - 1 : l]
  FI;
```

```
e) PROC ? subwhole =
     (UNION(⦃L INT⦄) x, REF INT first, INT upb, REF [] CHAR s) BOOL:
       # The digits of 'x' are placed in 's' (right justified to the
         position with index 'upb'); the result will be a boolean indicating
         the sign of 'x'. As a result, 'first' will point to the first digit
         of 'x' in 's'. Leading spaces will be placed from position F
         onwards, where F is the initial value of 'first'. #
     CASE x IN
       ⦃(L INT x):
         BEGIN L INT n:= ABS x, INT f = first; first:= upb + 1;
           WHILE s[first -:= 1]:= dig char(S(n MOD L 10));
             n OVERAB L 10; n ≠ L 0
           DO SKIP OD;
           FOR i FROM f TO first - 1 DO s[i]:= "." OD;
           x < L 0
         END⦄
     ESAC;
```

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

- B is true if V is negative, and false otherwise;
- For all i from LWB S to -1, S[i] = "_.";
- S[0] = "0";
- Case A: FLOATING is false:
  - it maximizes

$$M = \sum_{i=1}^{P-1} c_i * 10^{P-1-i} + \sum_{i=P+1}^{U} c_i * 10^{P-i}$$

under the following constraints:

- If $|V| \geq 1.0$, then P = ENTIER($\log_{10}(|V|)$) + 2, and, otherwise, P = 1;
- U = P + AFTER + 1;
- $M \leq |V|$;
- S[P] = ".";
- For all i from 1 to U, if $i \neq P$, then $0 \leq c_i \leq 9$, and S[i] = 'dig char(c_i)'.

{This simply amounts to the following: S contains all digits from the integral part of V at positions 1 to P-1, a decimal point at position P, and 'after+1' digits from the fractional part of V at positions P+1 to U. Furthermore, if we convert the contents of S back to some value M, this M should be "close" (in the sense of numerical analysis) to the original value |V|.}

Case B: FLOATING is true:

- it maximizes

$$M = \sum_{i=1}^{Q} c_i * 10^{Q-i} + \sum_{i=Q+2}^{U} c_i * 10^{Q+1-i}$$

under the following constraints:

- Q is the initial value of 'p' {i.e., the number of digits desired before the decimal point};
- If V = 0, then P = 1, and, otherwise, P = ENTIER($\log_{10}$(|V|))+1 {i.e., |V| = (appr.) 10^P*x, where x is normalized such that $0.1 \leq x < 1.0$};
- U = Q + after + 2;
- $M \leq |V| * 10^{Q-P}$;
- S[Q + 1] = ".";
- For all i from 1 to U, if $i \neq Q + 1$, then $0 \leq c_i \leq 9$, and S[i] = 'dig char(c_i)'.

{Thus, S contains the first Q significant digits of V, followed by a decimal point, followed by the next 'after+1' digits of V. If the contents of S is converted back to some value M, this M should be close to the (scaled) original value |V|.}

C;

g) PROC ? log10 = (NUMBER v) INT:
C A number P such that, given a value V, P = 0 if |V| < 1.0, and, if |V| ≥ 1.0, then P is such that

$$\text{ENTIER}(\log_{10}(|V|)) + 1 \leq P \leq \text{ENTIER}(\log_{10}(|V|)) + 2.$$

Thus, P is an estimate of the number of digits in the integral part of V; this estimate may be at most 1 too large. This definition should allow efficient computation of P; e.g., using the normal floating point representation of V with a mantissa M and base 2 exponent E (E = ENTIER($\log_2$|V|)), P may be given the value (ENTIER($\log_{10}2$ * (E+1))+1) MAX 0. Obviously also, 'log10' is a typical candidate for inline expansion. C;

```
h) PRIO ? EXPLENGTH = 9;
   OP EXPLENGTH = (BOOL sign, INT exp) INT:
     C The smallest E such that 'whole(exp, ABS sign * E)' succeeds. This
       operator is used to estimate the length needed to convert the
       exponent in 'float'. This is probably easier and faster than actually
       converting the exponent and subsequently testing its width, since on
       most implementations the exponent will be a relatively small integer
       (≤ 322, say). C;

i) PROC ? power10 = (REF [] CHAR s, REF INT rp, INT last) BOOL:
     IF rp:= last + 1; CHAR c:= s[rp];
       char dig((c = "." | s[rp + 1] | c)) ≥ 5
     THEN
       WHILE c:= s[rp -:= 1]; c = "9" OR c = "." DO SKIP OD;
       rp = 0
     ELSE FALSE
     FI;

j) PROC ? round = (REF [] CHAR s, INT rp, last) VOID:
     IF rp ≤ last
     THEN REF CHAR srp = s[rp]; srp:= dig char(char dig(srp)+1);
       FOR i FROM rp + 1 TO last
       DO REF CHAR si = s[i]; (si ≠ "." | si:= "0") OD
     FI;
```

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

1) PRIO ? MAX = 9;
   OP MAX = (INT a, b) INT: (a > b | a | b);

m) PRIO ? MIN = 9;
   OP MIN = (INT a, b) INT: (a < b | a | b);
```

Strings are converted to arithmetic values by the operator 'ADD' and the routine 'string to 1 real'. 'ADD' is used to construct an integer value. Since getting an integer is relatively simple, no intermediate string is constructed; rather, each digit read is directly appended to the partially constructed number. For real values, a string S is first built which contains the (at most) 'L real width + 1' most significant digits. Subsequently, an integer exponent EXP is computed such that, if d_1, ..., d_n are the digits corresponding to the elements of S, then the number

$$d_1 d_2 \cdots d_n * 10^{EXP}$$

is a close approximation of (the absolute value of) the number read. This construction is performed by 'string to 1 real', which also gets the sign of the result as parameter.

```
n) PRIO ? ADD = 1;
   OP ADD = (REF L INT a, INT d) BOOL:
     IF L INT amax = L max int OVER L 10,
              dmax = L max int MOD L 10;
       a > amax OR a = amax AND K d > dmax
     THEN FALSE
     ELSE a:= L 10 * a + K d; TRUE
     FI;
```

```
o) PROC ? string to l real =
     (REF [] CHAR s, INT exp, BOOL neg, UNION(¢REF L REAL¢) x) BOOL:
   C A unit which, given values S, EXP and NEG, yields a value B such
     that:

     • Let M be equal to
```

$$\left\{ \sum_{i=LWB\ S}^{UPB\ S} c_i * 10^{UPB\ S - i} \right\} * 10^{EXP},$$

```
       where, for all i from LWB S to UPB S, c[i] = 'char dig(S[i])';
     • IF M ≤ L max real, then:
       • 'x' is made to refer to a value X, where X is "close" to
         M * (NEG | -1 | 1);
       • B is true;
     Otherwise,
       • B is false {, and 'x' is left unchanged}.
   C;

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

q) 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);

r) 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);
```

```
s) 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;

t) 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;

u) OP ? INTWIDTH = (UNION(⦃L INT⦄) x) INT:
     CASE x IN ⦃(L INT): L int width⦄ ESAC;

   OP ? REALWIDTH = (UNION(⦃L REAL⦄) x) INT:
     CASE x IN ⦃(L REAL): L real width⦄ ESAC;

   OP ? RREALWIDTH = (UNION(⦃REF L REAL⦄) x) INT:
     CASE x IN ⦃(REF L REAL): L real width⦄ ESAC;

   OP ? EXPWIDTH = (UNION(⦃L REAL⦄) x) INT:
     CASE x IN ⦃(L REAL): L exp width⦄ ESAC;

   {These operators are introduced to restrict the amount of code needed for
    various routines in Chapters 8, 10 and 11. It is recommended that in-
    line code be generated for them. In 'whole' (b), for instance, there is
    now one piece of code which can be used to convert integers of any
    length.}
```

8.3. EFFICIENCY

The conversion routines are likely to be used quite heavily. Therefore, the efficiency of their implementation is of crucial importance. Care has already been taken to make the ALGOL 68 text as efficient as possible. Most notably, all string processing has been avoided. (Building up a string of n characters one by one using the standard operations PLUSTO and PLUSAB will probably result in allocation of about (n**2)/2 storage cells, most of which are garbage.)

The following machine-dependent optimizations that will speed up the code considerably are recommended (although it may in practice turn out to be advantageous wholly to rewrite the section on conversion routines in machine-code):

i) Since the character arrays that are used are compact one-dimensional arrays, an efficient and simple array-subscripting mechanism can be used on most machines.

ii) The routines 'dig char' and 'char dig' can often be implemented much more efficiently using knowledge about the internal ordering of the characters.

Note that the routines 'power10' and 'round' are rather expensive if the number to be rounded is of the form 'xxx99996', say. Such cases are assumed to occur rarely.

It is recommended that in-line code be generated for the operators 'MIN' and 'MAX'. These operators are used quite frequently throughout the model and a much more efficient implementation can easily be obtained. Preferably, in-line code should also be generated for the operators 'INTWIDTH', 'REALWIDTH', 'RREALWIDTH' and 'EXPWIDTH'. Possibly also, the overflow check in 'ADD' can be made in a much simpler way using knowledge of the arithmetic of the actual machine.

9. TRANSPUT MODES AND STRAIGHTENING

9.1. DIFFERENCES

- {C26} Because of the way INTYPE is defined in the Revised Report, 'flexible row of character' may only occur immediately after 'reference to' and may not be stowed. Thus, input to names of modes such as that specified by REF STRUCT(BOOL b, STRING s) is not possible. This restriction is contrary to the way in which stowing is normally handled; also, straightening is well capable of handling this task. This unintended restriction has been removed.

9.2. NEW DEFINITION

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

b) {Here, upper case 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) {Here, upper case stands for metanotions.}
INTYPE:: union OF INTYPERS mode.
INTYPERS:: reference to INTYPER; reference to 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 sections 10.3.2.3.c and d.}

10. FORMATLESS TRANSPUT

10.1. DIFFERENCES

- {C31} 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, the mood is also ensured in the case of a call 'put(f, ())'. Note that this may affect the contents of the file in the case of sequential-access; see also the first item of section 7.1. {In contrast, the mood is not ensured each time around the main loop of 'put' and 'get', but only after calling a PROC (REF FILE) VOID item encountered in the data list (for the other cases, the check is superfluous). The check is not performed the last time around the loop, so that it remains for instance possible to write 'put(f, (x, close))'.}

- {C10} 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, however, a good page will be found.

- {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'. This has mainly been done to enable efficient output of numbers and strings (see also section 4.4.2).

- {S,E} 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. It is never required to move back over more than one position.

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 reference to a file as its 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.

Generally speaking, if a value of mode A is successfully output using 'put' at a given position (p, l, c), it may be re-input into a REF A name by a call of 'get' at that same position. Moreover, the current position after the call of 'get' will be the same as it was after the call of 'put'. The principal exceptions to this are listed below. It is assumed that the same conversion keys are used on both occasions and that no event routines have been provided.

i) If a string that is output includes characters from the current terminator string of the file, the re-input string will be truncated to just before the first such character.

ii) If a string that is output does not completely fill the line, and if the file is not compressible or another string is subsequently put on

the same line, the re-input string will extend to the end of the line (or to an earlier terminating character). As a special case of this, putting an empty string will, in general, not lead to re-inputting an empty string.

iii) If a string that is output is split over the end of a line (the default action when no 'on line end' event routine is provided), the re-input string will be truncated to the line end. As a special case of this, if a nonempty string is put when the line has overflowed, an empty string will be re-input at that overflow position.

iv) If the mapping performed by the conversion key is not the same in both directions (i.e., the mapping is not bijective), then differences may arise. {For example, it is quite possible that "A" and "a" both map into the external character "A", which would presumably be re-input as "A".}

v) If, when putting, the current position is before the logical end of the file (which can happen anywhere in a random-access file, but only in the last logical line of a sequential-access file), the characters already present between the current position and the logical end can affect what is read back. {For example, supposing that 'int width' = 6:

```
put(f, (newline, ".......789."));
set char number(f, 1);
put(f, 123)
```

will result in the line "...+123789.", with obvious disastrous consequences upon a subsequent call of 'get'.}

vi) A related problem arises if there is insufficient room for an arithmetic value on the remainder of the current line. {Suppose the current line contains "...123456", the current position is at position 4 and some real number is output that does not fit on the current line. This number will then be put on the next line so that a subsequent call of 'get', at the position of the original call of 'put', will obtain the number '+123456.0'.}

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 blank is output 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, if not at the beginning of a line, a blank is given and, next, V is output as follows:

- if the mode of V is specified by L INT, then a string of length 'L int width + 1' is output, consisting of zero or more blanks, the sign of V, and the digits of V, in this order;

- if the mode of V is specified by L REAL, then V is output in floating-point form. The mantissa has length 'L real width + 2', and consists of the sign of V, followed by the first 'L real width' significant digits of V, with a decimal point after the first such digit. The exponent part has length 'L exp width + 2', and consists of an "e", zero or more spaces, the sign of the exponent, and the digits of the exponent, in this order;

- if the mode of V is specified by L COMPL, then the real and imaginary part are output in floating-point form as above, seperated by "i".

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 (pages) 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 (pages) being taken as required).

```
a) PROC put = (REF FILE f, [] UNION(OUTTYPE, PROC (REF FILE) VOID) x) VOID:
     BEGIN

      IF NOT (status OF cover OF f SAYS put char status)
      THEN ensure state(f, put char status)
      FI;

      FOR i TO UPB x
      DO
```

```
CASE x[i] IN

  (PROC (REF FILE) VOID pf):
    (pf(f);
     IF i < UPB x
     THEN
       IF NOT (status OF cover OF f SAYS put char status)
       THEN ensure state(f, put char status)
       FI
     FI),

  (OUTTYPE ot):
    BEGIN [] SIMPLOUT y = STRAIGHTOUT ot;

      PROC l real conv =
           (REF [] CHAR s, UNION(⦃L REAL⦄) x) VOID:
        # this routine converts 'x' into
          s[0 : L real width + L exp width + 3] #
        BEGIN INT exponent:= 1, rp, last:= REALWIDTH x + 1;
          BOOL neg =
             subfixed(x, last - 2, exponent, s, TRUE);
            # now 's' = "0x.xxxx..." #
          exponent -:= 1;
          IF power10(s, rp, last)
          THEN s[1]:= "1"; s[2]:= ".";
            FOR i FROM 3 TO last DO s[i]:= "0" OD;
            exponent +:= 1
          ELSE round(s, rp, last)
          FI;
          s[0]:= (neg | "-" | "+"); s[last +:= 1]:= "e"; last +:= 1;
          BOOL expneg =
             subwhole(exponent, last, last + EXPWIDTH x, s);
            # convert exponent into tail of 's' #
          s[last - 1]:= (expneg | "-" | "+")
        END;

      FOR j TO UPB y
      DO
```

```
CASE y[j] IN

  (UNION(NUMBER, {L COMPL}) nc):
    BEGIN
      INT upb =
        CASE nc IN
          {(L INT): L int width},
          {(L REAL): L real width + L exp width + 3},
          {(L COMPL): 2 * L real width +
                      2 * L exp width + 9}
        ESAC;
      [0 : upb] CHAR s;
      WHILE
        IF NOT (status OF cover OF f SAYS line ok)
        THEN next pos(f)
        FI;
        REF COVER cover = cover OF f;
        IF upb ≥ char bound OF cover
        THEN error(smallline); abort
        FI;
          # the number would not fit on the line, even if
            it were empty #
        c OF cpos OF cover + upb
              + SIGN(c OF cpos OF cover - 1) >
            char bound OF cover
          # the number does not fit on the remainder of
            the line #
      DO BOOL mended = (line mended OF f)(f);
        ensure state(f, put char status);
        (NOT mended | newline(f))
      OD;
      IF c OF cpos OF cover OF f ≠ 1
      THEN (write char OF f)(f, ".")
      FI;
        # a number is preceded by one space if not at
          the start of a line #

      CASE nc IN
```

```
      (UNION(⦃L INT⦄) k):
        BEGIN INT first:= 0;
          BOOL neg = subwhole(k, first, upb, s);
          s[first - 1]:= (neg | "-" | "+")
        END,
      (UNION(⦃L REAL⦄) r): 1 real conv(s, r),
      ⦃(L COMPL z):
        BEGIN 1 real conv(s, re OF z);
          INT istart = upb OVER 2;
          s[istart]:= ".."; s[istart + 1]:= "i";
          1 real conv(s[istart + 2 : @ 0], im OF z)
        END⦄
    ESAC;

    FOR k FROM 0 TO upb
    DO (write char OF f)(f, s[k]) OD;
    REF COVER cover = cover OF f;
    IF c OF cpos OF cover > char bound OF cover
    THEN status OF cover ANDAB line end
    FI  # test line end #
  END # numeric #,

(BOOL b):
  BEGIN
    IF NOT (status OF cover OF f SAYS line ok)
    THEN next pos(f)
    FI;
    put char(f, (b | flip | flop))
  END,

⦃(L BITS lb):
  FOR k TO L bits width
  DO
    IF NOT (status OF cover OF f SAYS line ok)
    THEN next pos(f)
    FI;
    put char(f, (k ELEM lb | flip | flop))
  OD⦄,
```

```
                (CHAR k):
                  BEGIN
                    IF NOT (status OF cover OF f SAYS line ok)
                    THEN next pos(f)
                    FI;
                    put char(f, k)
                  END,

                ([] CHAR ss):
                  BEGIN
                    IF NOT (status OF cover OF f SAYS page ok)
                    THEN ensure page(f)
                    FI;
                    INT from:= LWB ss, to;
                    WHILE from ≤ UPB ss
                    DO
                      IF NOT (status OF cover OF f SAYS line ok)
                      THEN next pos(f)
                      FI;
                      REF COVER cover = cover OF f;
                      FOR k FROM from TO to:=
                        (from + char bound OF cover - c OF cpos OF cover)
                              MIN UPB ss
                      DO (write char OF f)(f, ss[k]) OD;
                      from:= to + 1;
                      IF c OF cpos OF cover > char bound OF cover
                      THEN status OF cover ANDAB line end
                      FI  # test line end #
                    OD
                  END

            ESAC
          OD
        END
      ESAC
    OD
  END;
```

```
b) PROC ? put char = (REF FILE f, CHAR char) VOID:
      # the precondition of 'put char' is:
              . line ok (see 7.2) #
     BEGIN (write char OF f)(f, char);
       REF COVER cover = cover OF f;
       IF c OF cpos OF cover > char bound OF cover
       THEN status OF cover ANDAB line end
       FI  # test line end #
     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 from the current position for the first character that is not a space (finding good positions on subsequent lines as necessary); next, the largest sequence of characters 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 sequence of characters 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 from the current position for the first character that is not a space (finding good positions on subsequent lines as necessary); next, the largest sequence of characters 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) and assigned to the first subname of N; next, the book is searched from the current position for the first character that is not a

space; next, a character is input and, if it is not "⊥", "i" or "I", then the event routine corresponding to 'on char error' is called, the suggestion being "⊥"; finally, a second real number is input and assigned to the second subname of N.

{Numbers are input using a finite-state machine that largely follows the syntax of INTREAL-denotations as given in sections 8.1.1 and 8.1.2 of the Revised Report. However, spaces within numbers are only 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 from the current position 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 (pages) 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 (pages) 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 and returns true, 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 (see also the last item from section 7.1).

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; however, if necessary, 'newpage' will be called before an attempt is made to read any character).

```
c) PROC get = (REF FILE f, [] UNION(INTYPE, PROC (REF FILE) VOID) x) VOID:
     BEGIN

      IF NOT (status OF cover OF f SAYS get char status)
      THEN ensure state(f, get char status)
      FI;

      FOR i TO UPB x
      DO

       CASE x[i] IN

         (PROC (REF FILE) VOID pf):
           (pf(f);
            IF i < UPB x
            THEN
              IF NOT (status OF cover OF f SAYS get char status)
              THEN ensure state(f, get char status)
              FI
            FI),
```

```
(INTYPE it):
  BEGIN [] SIMPLIN y = STRAIGHTIN it, CHAR k;

    PROC mend char = (CHARBAG s, CHAR sugg) VOID:
      # the character read is not in 's'; therefore, the event
        routine corresponding to 'on char error' is called with
        with the suggestion 'sugg' #
      IF k:= sugg;
        BOOL ok =
          IF (char error mended OF f)(f, k)
          THEN char in bag(k, s)
          ELSE FALSE
          FI;
        ensure state(f, get char status);
        IF NOT (status OF cover OF f SAYS buffer initialized)
        THEN (init buffer OF cover OF f)(f)
        FI;
        NOT ok
      THEN error(wrongchar); k:= sugg
      FI;

    PROC skip initial spaces = VOID:
      # skip spaces, in the meantime passing to a next line and/or
        page, if necessary #
      WHILE
        IF NOT(status OF cover OF f SAYS line ok)
        THEN next pos(f)
        FI;
        get char(f, k); k = "."
      DO SKIP OD;
```

```
PROC skip spaces = VOID:
  # only spaces on the current line are skipped #
  WHILE
    IF status OF cover OF f SAYS line ok
    THEN get char(f, k); k = "."
    ELSE (check pos(f) | get char(f, k)
                       | error(nocharpos); abort);
      FALSE
    FI
  DO SKIP OD;

OP NODIGIT = (CHAR c) BOOL:
  NOT char in bag(c, radix10digit);   # in-line code #

¢PROC read L integer = (REF L INT i) BOOL:
  # if an integer is read successfully, it is assigned to 'i',
    and the routine returns true; otherwise, it returns false #
  BEGIN BOOL ok:= TRUE, BOOL neg = k = "-";
    (neg OR k = "+" | skip spaces);
    (NODIGIT k | mend char(radix10digit, "0"));
    L INT j:= K chardig(k);
    WHILE
      IF status OF cover OF f SAYS line ok
      THEN get char(f, k);
        IF NODIGIT k
        THEN backchar(f); FALSE
        ELSE (ok | ok:= j ADD chardig(k)); TRUE
        FI
      ELSE FALSE
      FI
    DO SKIP OD;
    IF ok THEN i:= (neg | -j | j) FI;
    ok
  END¢;
```

```
PROC read l real = (UNION(⫲REF L REAL⫳) r) BOOL:
  # similar to 'read L integer', for real numbers #
  BEGIN BOOL ok:= TRUE, BOOL neg = k = "-",
    INT lrw = RREALWIDTH r;
    (neg OR k = "+" | skip spaces);
    [0 : lrw] CHAR s, INT index:= -1, exp:= 0,
    BOOL sig:= FALSE;
      # and remains false until the first significant digit is
        found #

    PROC read digits = (BOOL after) VOID:
      # Read a sequence of digits (from either the integral or
        the fractional part, depending on the value of the
        'after' parameter). At most 'L real width+1' digits are
        stored in 's'. 'exp' counts the distance from the
        decimal point to the first significant digit. If
        'after' is false, then 'i' = 0, so 'exp' is just
        incremented for each significant digit read. If 'after'
        is true, then 'i' = -1, and 'exp' is decremented as
        long as non-significant digits are read after the
        decimal point. #
      BEGIN INT i = ABS after;
        (NODIGIT k | mend char(radix10digit, "0"));
        WHILE
          IF NODIGIT k
          THEN FALSE
          ELIF
            IF sig:= sig OR k ≠ "0"
            THEN (index < lrw | s[index +:= 1]:= k
                              | exp +:= 1)
            FI;
            exp -:= i; status OF cover OF f SAYS line ok
          THEN get char(f, k); TRUE
          ELSE FALSE
          FI
        DO SKIP OD
      END;
```

```
  IF k ≠ "." THEN read digits(FALSE) FI;
    # if this call of 'read digits' has exhausted the line,
      then 'k' is some digit at this point, so the next test
      fails #

  IF k = "."
  THEN
    IF
      IF status OF cover OF f SAYS line ok
      THEN TRUE
      ELSE check pos(f)
      FI
    THEN get char(f, k)
    ELSE error(nocharpos); abort
    FI;
    read digits(TRUE)
  FI;
    # again, if the line is exhausted, then 'k' is some digit
      at this point #

  IF char in bag(k, times ten to the power)
  THEN INT e; skip spaces; ok:= read integer(e);
    IF ok:= ok AND
        (SIGN e ≠ SIGN exp OR ABS e ≤ max int - ABS exp)
          # test for integral overflow of exponent #
    THEN exp +:= e
    FI
  ELSE back char(f)
  FI;

  IF NOT ok THEN FALSE
  ELSE string to l real(s[ : index], exp, neg, r)
  FI
END;
```

```
FOR j TO UPB y
DO

  CASE y[j] IN

    (UNION({REF L INT}, {REF L REAL}, {REF L COMPL}) irc):
      BEGIN skip initial spaces;
        IF NOT
         CASE irc IN
           {(REF L INT ii): read L integer(ii)},
           (UNION({REF L REAL}) rr): read l real(rr),
           {(REF L COMPL zz):
             BEGIN
               BOOL ok = read l real(re OF zz); skip spaces;
               IF NOT char in bag(k, plus i times)
               THEN mend char(plus i times, "i")
               FI;
               skip spaces;
               ok AND read l real(im OF zz)
             END}
         ESAC
        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):
      BEGIN skip initial spaces;
        IF NOT char in bag(k, flipflop)
        THEN mend char(flipflop, flop)
        FI;
        bb:= k = flip
      END,
```

```
{(REF L BITS lb):
  BEGIN [1 : L bits width] BOOL b;
    FOR i TO L bits width
    DO skip initial spaces;
      IF NOT char in bag(k, flipflop)
      THEN mend char(flipflop, flop)
      FI;
      b[i]:= k = flip
    OD;
    lb:= L bits pack(b)
  END},

(REF CHAR cc):
  BEGIN
    IF NOT (status OF cover OF f SAYS line ok)
    THEN next pos(f)
    FI;
    get char(f, cc)
  END,

(REF [] CHAR ss):
  BEGIN
    IF NOT (status OF cover OF f SAYS page ok)
    THEN ensure page(f)
    FI;
    FOR i FROM LWB ss TO UPB ss
    DO
      IF NOT (status OF cover OF f SAYS line ok)
      THEN next pos(f)
      FI;
      get char(f, ss[i])
    OD
  END,
```

```
        (REF STRING ss):
          BEGIN
            IF NOT (status OF cover OF f SAYS buffer initialized)
            THEN (init buffer OF cover OF f)(f)
            FI;
            INT index:= 0,
            upbs:= char bound OF cover OF f -
                     c OF cpos OF cover OF f + 1;
            STRING s:= LOC [1 : upbs] CHAR;
            WHILE
              IF
                IF status OF cover OF f SAYS line ok
                THEN TRUE
                ELSE check pos(f)
                FI
              THEN get char(f, k);
                IF char in bag(k, term OF f)
                THEN back char(f); FALSE
                ELSE TRUE
                FI
              ELSE FALSE
              FI
            DO
              IF index = upbs
              THEN
                INT u = char bound OF cover OF f -
                          c OF cpos OF cover OF f + 2;
                  # the number of characters from the current
                    position onwards; note that 'u' > 0, since
                    'c OF cpos' ≤ 'char bound + 1' #
                upbs +:= u; s +:= LOC [1 : u] CHAR
              FI;  # estimate new 's' #
              s[index +:= 1]:= k
            OD;
            ss:= s[: index]
          END
      ESAC
    OD
```

```
        END
      ESAC
    OD
  END;
```

d) PROC ? get char = (REF FILE f, REF CHAR char) VOID:

```
  char:=
    IF CHAR k; BOOL conv ok = (read char OF f)(f, k);
      REF COVER cover = cover OF f;
      REF STATUS status = status OF cover;
      IF c OF cpos OF cover = c of lpos OF cover
      THEN # line and/or logical file ended #
        status ANDAB
            IF status SUGGESTS lfe in current line
            THEN logical file ended
            ELSE line end
            FI
      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;
```

e) 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, returning a character from the terminator string of 'f'. This case is assumed to be exceedingly rare.) Note that no more than one call of 'back char' can occur for each item read. So even if 'backspace possible' returns false, 'back char' still works. C;

f) MODE ? CHARBAG =

```
    C Some mode which allows efficient retrieval of information as to the
      presence or absence of a given character in a given set (for example,
      a bit string). C;

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

  PROC ? char in bag = (CHAR k, CHARBAG s) BOOL:
    C This routine returns true if the character 'k' is contained in 's',
      and false otherwise. C;

  OP + = (CHARBAG s, t) CHARBAG:
    C A value K of the mode CHARBAG is delivered, such that the set of
      characters in K is the union of the sets of characters in 's' and
      't'. C;

  CHARBAG ? radix10digit = STRINGTOBAG "0123456789",
          ? radix 2digit = STRINGTOBAG "01",
          ? radix 4digit = STRINGTOBAG "0123",
          ? radix 8digit = STRINGTOBAG "01234567",
          ? radix16digit = STRINGTOBAG "0123456789abcedf",
          ? times ten to the power = STRINGTOBAG "\eE",
          ? flipflop               = STRINGTOBAG (flip + flop),
          ? plus i times           = STRINGTOBAG "⊥iI",
          ? signspace              = STRINGTOBAG "+-.",
          ? plusminus              = STRINGTOBAG "+-",
          ? point                  = STRINGTOBAG STRING("."),
          ? xylpkq                 = STRINGTOBAG "xylpkq";
```

{Depending on the character set available in a specific implementation, some of the above declarations may have to be adjusted.}

10.3. EFFICIENCY

The efficiency of formatless transput critically depends on the efficiency of the routines 'put char' and 'get char'. It is recommended that these be not straightforwardly implemented as procedures; rather, inline code should be generated here. When outputting numbers, there is no need to test for the line being ended after each character. Consequently, the corresponding piece of code in 'put' has been optimized. A similar optimization has been applied for the case where strings are output. If an implementation offers a fast way to output sequences of characters, this may most profitably be applied at these spots.

Various small operators and routines that are used in 'get', such as 'NODIGIT' and 'char in bag', also offer possibilities for optimization. The mode 'CHARBAG' is incorporated just to encourage efficient implementation; if efficiency is not important, the mode 'STRING' may be substituted instead. In that case, 'STRINGTOBAG' becomes a dummy operation, while 'char in bag' may be replaced by 'char in string'. {Note that the routine 'char in bag' is also heavily used in formatted transput ('indit string', 11.2.3.o.).} Implementation in the form of a bit string (provided it can be indexed) or a lookup table seem natural ways to achieve high efficiency here.

When reading a string using 'get', a temporary string is used, whose length is equal to the length of the remainder of the current buffer. In general, this will suffice; only if the event routine corresponding to the 'on line end' or 'on logical file end' event moves the current position and returns true may more space be needed.

If the implementation provides some cunning optimization for expanding strings, this may be applied here. If the implementation permits exceptionally long buffers, it may be profitable to use a smaller temporary string. To this end, the assignment to 'upbs' should be changed, for example to

```
upbs:= 80 MIN (char bound OF cover OF f -
                    c OF cpos OF cover OF f + 1).
```

A similar change then applies to the assignment to 'u'.

In both 'put' and 'get', the treatment of numbers is largely independent of their length (consider for instance the routine 'read 1 real' inside 'get'). Using knowledge about the internal representation of such values, the small conditional clauses involving numbers of different lengths can probably be optimized. This same optimization technique applies to the routines 'putf' and 'getf' (11.2.3.n, p).

11. FORMATTED TRANSPUT

11.1. DIFFERENCES

- {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 (which does not change), and it contains some administration on how this format is currently being used (which does change). These two parts have been separated in the present model. Here, the mode specified by FORMAT only contains a description of the format; thus, formats need no longer be copied upon assignment. The dynamic information is collected in the 'piece' field of the file with which the format is associated. This change results in remarkable simplifications in the routines 'get next picture', 'do fpattern' (called 'do fpict' in this report) and 'associate format' (11.2.3.b, k, l).

- {D} Formatted transput is rather complicated, and slowed down, by the possibility of using "dynamic" replicators, i.e., replicators whose value is not known until runtime. In the Revised Report, both dynamic replicators and non-dynamic replicators (i.e., replicators that are plain integers) are treated in a uniform way; they both give rise to a routine returning an integer. For example, the replicator '10' gives rise to a routine composed from 'INT: 10'; the dynamic replicator 'n(lim-1)' gives rise to 'INT: (lim-1)'. For output, as a consequence, the frames that a pattern is composed of are traversed three times:

  - First, the frames are traversed to evaluate all replicators (i.e., each PROC INT is turned into an INT);

  - Second, the frames are traversed to extract the necessary information to structure the string to be output (the position of the decimal point, the width of the exponent, etc., are determined);

- Third, after building the string, this string is matched against the frames character by character, to determine the actual output (insertions may have to be placed in between, zeroes may be replaced by spaces, etc.).

For input, only the first and (the reverse of) the third kind of traversal is needed.

In the case where a pattern contains only non-dynamic replicators (which is probably the common case), a much simpler scheme is possible: in this case, there is no need to construct routines returning an integer; rather, the integers themselves may be incorporated in the pattern. Also, the compiler is quite capable of extracting the information needed to structure the string in that case. (Alternatively, this may be done upon associating the format with the file.) In this way, the frames need only be traversed once.

In the present model, two kinds of pictures are distinguished: simple, static pictures (whose mode is SPICT), and dynamic pictures. A static picture may only arise from an 'integral-', 'real-', 'boolean-', 'complex-', 'string-' or 'bits-pattern'. The reasons therefore are twofold:

- These kinds of pictures will probably occur most often;

- The treatment of choice-patterns and format-patterns is complicated by the fact that their insertions are evaluated at moments which are different from those of other patterns.

Outputting a value W using a picture P now proceeds as follows:

- If the mode of P is DPICT, then P is staticized, thereby yielding a static picture;

- Subsequently, the picture is either static, or it is a dynamic picture whose pattern is a 'choice-', 'format-', 'general-' or 'void-pattern'. Dynamic pictures are handled in the same way as they are in the Revised Report. Static pictures can now be handled

quite efficiently: the picture contains all information needed to build the string to be output. After having done this, 'edit string' may be called.

- {D} There is still one further optimization possible. In the Revised Report, a format contains quite a lot of redundant information in the form of default values for insertions, replicators and the like. In the definition given below, these default values are not explicitly stored in the data structures that are used. Thus, a "frame" is either a routine (returning an integer), a character, or a fixed name referring to a string. This will save both space and time.

- {C12} If a collection list is replicated zero times, its pictures should not be selected (section 10.3.4.1.1.gg of the Revised Report), although the insertions immediately preceding and following it should be performed. The routine 'get next picture', as formulated in the Revised Report, erroneously selects the pictures of the collection list once.

  In the Revised Report, there is a test in 'get next picture' to ensure that 'undefined' is called if the staticizing of the insertion or the elaboration of the replicator should cause a different format to be associated with the file (for there is no sensible continuation in such a case). In the present model, this test is also made after the getting or putting of the insertion, since this may also cause a new format to be associated with the file.

  Likewise, to ensure uniformity of treatment of format patterns and collections, similar tests are made in 'do fpict' (11.2.3.k).

- {C28} Normally, when a picture is encountered during formatted transput, it is "staticized", i.e., its dynamic replicators are all elaborated (collaterally) before transput using that picture commences. Moreover, if the transput is terminated by a jump at this stage, any subsequent transput will start with the next following picture. Clearly, it is not appropriate to staticize the whole of a collection list all at once, and the Revised Report therefore prescribes that its pictures be staticized one by one as they are

required for transput. Finally, at the end of the collection list, there may be a final insertion to be performed when the collection list has been repeated as many times as required by its replicator. The Revised Report defines that the replicator of this final insertion be elaborated collaterally with the staticizing of the last picture on its last time round; i.e., before any possible side effects of the transput using that picture.

In the present model, these insertions are not concatenated. The final insertion of a collection list is elaborated after the transput of the final picture. Note that, if transput is terminated at this point, the collection list should be deemed to have been completed and future transput must start with the next complete picture or collection list. The routine 'update cp' (11.2.3.c) serves both these purposes.

The final insertion of a format pattern is treated similarly.

- {C29} In formatted transput, a general pattern can be used to obtain the effect of 'put', and, if it is provided with parameters, of 'whole', 'fixed' or 'float' also. Thus, 'printf(($g(6, 3)$, x))' is exactly equivalent to 'print(fixed(x, 6, 3))'. On input, a general pattern can give the effect of 'get' but, since 'whole', 'fixed' and 'float' cannot be used with 'get', there is no meaning for any parameters, and none should therefore be present.

  The Revised Report permits parameterized general patterns on input, but directs that their parameters be ignored. In the present model, these parameterized general patterns on input are regarded as not being "input compatible" (section 10.3.4.1.1.ii of the Revised Report) with their data list values. Therefore, the event routine corresponding to 'on value error' is called in this case (the default action being an error message and abortion of the program).

- {S} On files for which 'backspace possible' returns false, 'choice-patterns' cannot be used on input since, if the first literal does not match, it is impossible to backspace over the characters read in order to try the next literal. The present model issues an error message and aborts the program in such cases.

11.2. NEW DEFINITION

In formatted transput, each straightened value from a data list (cf section 10.2) is matched against a constituent 'picture' of a 'format-text' provided by the user. A 'picture' specifies how a value is to be converted to or from a sequence of characters and prescribes the layout of those characters in the book. Features which may be specified include the number of digits, the position of the decimal point and of the sign, if any, suppression of zeroes and the insertion of arbitrary strings. For example, using the 'picture' '-d.3d "." 3d "." e z+d', the value '1234.567' would be transput as the string "-1.234.567.\.+3".

A "format" is a structured value (i.e., an internal object) of mode 'FORMAT' which mirrors the structure of a 'format-text' (which is an external object). For a description of the syntax of 'format-texts', the reader is referred to sections 10.3.4.1.1 (about 'collections' and 'pictures') and 10.3.4.2 - 10.3.4.10 (about 'patterns') of the Revised Report. Below, only the semantics are described for obtaining the corresponding format from a 'format-text'. This is necessary since the internal representation of a 'format-text' in this model is rather different from the representation used in the Revised Report. The only deviation in the semantics is that, whereas this model prescribes sequential elaboration of replicators, the Revised Report uses collateral elaboration.

A "piece" is brought into being by associating a format with a file. A piece is a structured value composed of a reference to the collection list currently in use, a 'count' field which tells how many times this collection list should be used, the index of the current collection, and a reference to a piece to be used after the current collection list is exhausted. Upon "associating" (11.2.2.q) a format with a file, the piece will contain a reference to the collection list of the format. Upon selection of a picture from this collection list that is itself a replicated "collection", the current collection list is "suspended", and this new collection list is made to be the current collection list. Subsequent transput now uses this collection list until it has been exhausted, after which the suspended collection list is again made to be the current one. Something similar happens when other formats are invoked by means of 'format-patterns' (section 10.3.4.9.1 of the Revised Report).

Although a 'format-text' may contain 'ENCLOSED-clauses' (in 'replicators' and 'format-patterns') or 'units' (in 'general-patterns'), these are not elaborated at this stage, but are turned into routines for subsequent calling as and when they are encountered during formatted transput. Note however that, if a data picture contains only non-dynamic replicators, i.e., replicators that are plain integers, these replicators are elaborated directly, thus giving rise to a so-called "static picture". {The term "data picture" is used as an abbreviation for a picture whose pattern is an 'integral-', 'real-', 'boolean-', 'complex-', 'string-' or 'bits-pattern'.} It still remains true that the elaboration of a 'format-text' does not result in any actions of any significance to the user.

If, at runtime, a data picture is encountered that does contain dynamic replicators, then this data picture is "staticized", thereby yielding a static picture which is used when actually transputting values. Note that the compiler is not forced to construct any static pictures; if it does not, the result will still be the same. An implementer may also choose to staticize non-dynamic pictures upon associating the format with the file.

For ease of description, the semantics given below are such that a static picture is the result of staticizing a dynamic data picture; this intermediate dynamic data picture is of course not necessarily constructed in an actual implementation.

11.2.1. MODE DECLARATIONS

```
a) MODE FORMAT = STRUCT(REF [] COLLECTION ? c);

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

c) MODE ? COLLITEM =
     STRUCT(INSERTION i1,
            PROC INT rep, # replicator #
            REF [] COLLECTION p,
            INSERTION i2);

d) MODE ? PICTURE =
     UNION(SPICT, DPICT, CPICT, FPICT, GPICT, VOIDPICT);

e) MODE ? SPICT =
     STRUCT(UNION(INTPATTERN, REALPATTERN, BOOLPATTERN, COMPLPATTERN,
                  STRINGPATTERN, BITSPATTERN) p,
            REF [] SFRAME sframes);

f) MODE ? DPICT =
     STRUCT(INT type,
            REF [] DFRAME frames);

g) MODE ? CPICT =
     STRUCT(INSERTION i1,
            INT type,
            REF [] INSERTION c,
            INSERTION i2);

h) MODE ? FPICT =
     STRUCT(INSERTION i1,
            PROC FORMAT pf,
            INSERTION i2);
```

```
i) MODE ? GPICT =
     STRUCT(INSERTION i1,
            FLEX [1:0] PROC INT spec,
            INSERTION i2);

j) MODE ? VOIDPICT = INSERTION;

k) MODE ? INSERTION = REF [] DFRAME;

l) MODE ? DFRAME =
     UNION(PROC INT, REF [] CHAR, CHAR);

m) MODE ? SFRAME =
     UNION(INT, REF [] CHAR, CHAR);

n) MODE ? INTPATTERN = REF STRUCT(INT width, sign);
     # 'width': The length of the string controlled by the integral pattern,
                including the possible sign-frame;
       'sign' : The absolute value of 'sign' is the length of the string
                controlled by the sign mould of the pattern. If 'sign' < 0
                (> 0), then the sign mould contains a descendent minus-
                symbol (plus-symbol). If 'sign' = 0, then the pattern
                contains no sign mould. Note that because of this way of
                coding there is no need for u- or v-frames, since the 'sign'
                field contains the necessary information. #
```

```
o) MODE ? REALPATTERN = REF STRUCT(INT b, s1, a, e, s2, point);
     # 'b'     : The length of the string controlled by the first integral
                 mould of the stagnant part of the pattern, including the
                 possible sign-frame;
       'a'     : The length of the string controlled by the second integral
                 mould of the stagnant part;
       'e'     : The length of the string controlled by the integral pattern
                 of the exponent part, including the possible sign-frame;
       's1'    : The length of the string controlled by the sign mould of the
                 stagnant part, coded in the same way as the 'sign' field of
                 the integral pattern;
       's2'    : Idem for the sign mould of the exponent;
       'point': 'point' = 1 if the pattern contains a point frame, and 0
                 otherwise. #

p) MODE ? COMPLPATTERN = REF STRUCT(REALPATTERN re, im);

q) MODE ? BOOLPATTERN = VOID;

r) MODE ? STRINGPATTERN = INT;
     # The length of the string controlled by the pattern. #

s) MODE ? BITSPATTERN = REF STRUCT(INT width, radix);
     # 'width': The length of the string controlled by the pattern;
       'radix': The radix of the radix frame. #
```

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

11.2.2. SEMANTICS

{This section replaces sections 10.3.4.1.2, 10.3.4.8.2, 10.3.4.9.2 and 10.3.4.10.2 of the Revised Report.}

{The yield N, in an environ E, of a 'format-text' F is a structured value W, whose mode is FORMAT, and whose only field is the yield of the elaboration of the constituent 'collection-list' of F in E.}

a) 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, let Cj be the j-th constituent 'collection' of C.
  Case A: The direct descendents of Cj include a 'picture' P:
  - P is elaborated in E;
  - the j-th element of V is the yield 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.

b) The yield N, in an environ E, of a 'picture' PICT is a value W, whose mode is acceptable to 'PICTURE', determined as follows:

- Let V be the yield of the elaboration of the constituent 'pattern' P together with the constituent 'insertion' I of PICT in E (c, d, e, f);
- If the constituent 'pattern' of PICT is an 'integral-', 'real-', 'boolean-', 'complex-', 'string-' or 'bits-pattern' which does not contain any dynamic replicators, then
  - W is the result of "staticizing" (k) V in E;

  Otherwise,
  - W is V.

c) The yield, in an environ E, of an 'integral-', 'real-', 'boolean-', 'complex-', 'string-' or 'bits-pattern' P {sections 10.3.4.2.1.a, 10.3.4.3.1.a,..., 10.3.4.7.1.a of the Revised Report}, together with an 'insertion' I, is a structured value W whose mode is 'DPICT', whose fields, taken in order, are:

- {type} 1 (2, 3, 4, 5) if P is an 'integral-' ('real-', 'boolean-', 'complex-', 'string-') '-pattern' and 6 (8, 12, 20) if P is a 'bits-pattern' whose constituent 'RADIX' is a 'radix-two' ('-four', '-eight', '-sixteen');
- {frames} a newly created name, equal in scope to the environ necessary for P in E, which is made to refer to a value F, whose mode is 'row of FRAME', having a descriptor ((1, n)) and n elements determined as follows:
  - a counter i is set to 0;
  - the constituent frames of P, in order, are "transformed" (h) in E into F, using i;
  - I is "transformed" (i) in E into F.

d) The yield, in an environ E, of a 'choice-pattern' P, together with an 'insertion' I, is a structured value W whose mode is 'CPICT', determined as follows:

- let n be the number of constituent 'NEST-literals' of the 'praglit-list-pack' of P;
- let Si, i = 1, ..., n, be a 'NEST-insertion' akin to the i-th of those constituent 'NEST-literals';
- the insertion I1 of P, all of S1, ..., Sn, and the insertion I are elaborated in E;
- the fields of W, in order, are:
  - {i1} the yield of I1;
  - {type} 1 (2) if P is a 'boolean-' ('integral-') '-choice-pattern';
  - {c} a newly created name, equal in scope to the environ necessary for P in E, which is made to refer to a value F whose mode is 'row-of-INSERTION', having a descriptor ((1, n)), and n elements, that selected by (i), i = 1, ..., n, being the yield of Si;
  - {i2} the yield of I.

e) The yield, in an environ E, of a 'NEST-format-pattern' P, together with an 'insertion' I, is a structured value whose mode is 'FPICT' and whose fields, taken in order, are:

- {i1} the yield of the insertion of P;
- {pf} a routine whose mode is 'procedure yielding FORMAT', composed of a 'procedure-yielding-FORMAT-NEST-routine-text' whose unit U is a new unit akin to the 'meek-FORMAT-ENCLOSED-clause' of P, together with the environ necessary for U in E;
- {i2} the yield of I.

f) The yield, in an environ E, of a 'NEST-general-pattern' P, together with an 'insertion' I, is a structured value whose mode is 'GPICT' and whose fields, taken in order, are:

- {i1} the yield of the insertion of P;
- {spec} a multiple value W whose mode is 'row of procedure yielding integral', having a descriptor ((1, n)), where n is the number of constituent 'meek-integral-units' of the 'width-specification-option' of P, and n elements determined as follows:
  For i = 1, ..., n,
  - the i-th element of W is a routine, whose mode is 'procedure yielding integral', composed of a 'procedure-yielding-integral-NEST-routine-text' whose unit U is a new unit akin to the i-th of those 'meek-integral-units', together with the environ necessary for U in E;
- {i2} the yield of I.

g) The yield N, in an environ E, of an 'insertion' I {section 10.3.4.1.1.d of the Revised Report} is determined as follows:

- N is a newly created name {whose mode is INSERTION};
- N is equal in scope to the environ necessary for I in E;
- N is made to refer to a value W whose mode is 'row of FRAME', having a descriptor ((1, n)) and n elements, determined as follows:
  - a counter i is set to 0;
  - I is "transformed" (i) in E into W, using i.

h) A 'frame' V is "transformed" in an environ E into a multiple value F whose mode is 'row of FRAME', using a counter i, as follows:

- the constituent 'insertion' of V is "transformed" (i) in E into F, using i;
- the constituent 'replicator' of V is "transformed" (j) in E into F, using i;
- if the constituent 'UNSUPPRESSETY-suppression' of V contains a 'letter-s-symbol', then
  - i is increased by 1;
  - the element of F selected by (i) is a 'letter-s-symbol';
- i is increased by 1;
- the element of F selected by (i) is the intrinsic value of the constituent 'symbol' of the marker of V.

i) An 'insertion' I is "transformed" in an environ E into a multiple value F whose mode is 'row of FRAME', using a counter i, as follows:

- Let U1, ..., Un, be the constituent 'UNSUPPRESSETY-replicators' of I, and let Aj, j = 1, ..., n be the 'denoter-coercee' or 'alignment-code' {immediately} following Uj;
- For j = 1, ..., n:
  - Uj is "transformed" (j) in E into F, using i;
  - i is increased by 1;
  - the element of F selected by (i) is determined as follows:

    Case A: Aj is an 'alignment-code':
    - it is the {character which is the} intrinsic value of the 'LETTER-symbol' of Aj;

    Case B : Aj is a 'denoter-coercee':
    - it is a newly created name, equal in scope to the environ necessary for F in E, which is made to refer to the yield of Aj.

j) A 'NEST-UNSUPPRESSETY-replicator' R is "transformed" in an environ E into a multiple value F whose mode is 'row of FRAME', using a counter i, as follows:

If R is not invisible, then:

- i is increased by 1;
- the element of F selected by (i) is a routine whose mode is 'procedure yielding integral', composed of a 'procedure-yielding-integral-NEST-routine-text' whose 'unit' is U, together with the environ necessary for U in E, where U is determined as follows:

  Case A: R contains a 'meek-integral-ENCLOSED-clause' C:
  - U is a new 'unit' akin to C;

  Case B: R contains a 'fixed-point-numeral' D, but no 'ENCLOSED-clause':
  - U is a new 'unit' akin to D;

Otherwise,

- no action is taken.

k) A 'data picture' P is "staticized" in an environ E, yielding a value V of mode 'SPICT', as follows:

- Let F be the value referred to by the 'frames' field of P;
- Let Vi, i = 1, ..., n, be the yield of "staticizing" (l) the i-th element of F, where ((1, n)) is the descriptor of F;
- Let X be the "structure description" (m) determined by the type of P and V1,...,Vn;
- The fields of V, taken in order, are:
  - {p} X;
  - {sframes} a newly created name, equal in scope to the primal environ, which is made to refer to a value F, whose mode is 'row of SFRAME', having a descriptor ((1, n)) and n elements, that selected by (i), i = 1, ..., n, being Vi.

l) A 'frame' V is "staticized" in an environ E, yielding a value S of mode 'SFRAME', as follows:

- If V is a routine, then
  - S is the yield of the calling of V in E;

  Otherwise,
  - S is V.

m) The "structure description" determined by a type I and sframes V1, ..., Vn is a value X, determined as follows:

Case A: I equals 1:

- Let i be the "position" (n) of the "+" or "-" in V1, ..., Vn, if any, and 0 otherwise;
- Let j be the length controlled by V1,...,Vi;
- X is a newly created name, equal in scope to the primal environ, and is made to refer to a value of the mode 'STRUCT(INT width, sign)', whose fields, taken in order, are:
  - {width} the length "controlled" (o) by V1, ..., Vn;
  - {sign} j if i = 0 or Vi = "+", and -j otherwise.

Case B: I equals 2:

- Let e be the position of the "e" in V1, ..., Vn, if any, and n+1 otherwise;
- Let p be the position of the "." in V1, ..., Vn, if any, and e otherwise;
- Let x be the position of the "+" or "-" in V1, ..., Vp, if any, and 0 otherwise;
- Let s1 be the length controlled by V1,...,Vx;
- Let u be the position of the "+" or "-" in V(e+1), ..., Vn, if any, and 0 otherwise;
- Let s2 be the length controlled by V(e+1),...,Vu;
- X is a newly created name, equal in scope to the primal environ, and is made to refer to a value of the mode 'STRUCT(INT b, s1, a, e, s2, point)', whose fields, taken in order, are:
  - {b} the length controlled by V1, ..., V(p-1);
  - {s1} s1 if x = 0 or Vx = "+", and -s1 otherwise;
  - {a} the length controlled by V(p+1), ..., V(e-1);
  - {e} the length controlled by V(e+1), ..., Vn;
  - {s2} s2 if u = 0 or Vu = "+", and -s2 otherwise;
  - {point} 1 if p > 0, and 0 otherwise.

Case C: I equals 3:

- X is a value of mode 'BOOLPATTERN' {and is equal to empty}.

Case D: I equals 4:

- Let i be the position of the "i" in V1, ..., Vn;
- X is a newly created name, equal in scope to the primal environ, and is made to refer to a value of the mode 'STRUCT(REALPATTERN re, im)', whose fields, taken in order, are:
  - {re} the structure description determined by 2 and V1, ..., V(i-1);
  - {im} the structure description determined by 2 and V(i+1), ..., Vn.

Case E: I equals 5:

- X is a value of mode 'STRINGPATTERN' and is equal to the length controlled by V1, ..., Vn.

Case F: I equals 6, 8, 12 or 20:

- X is a newly created name, equal in scope to the primal environ, and is made to refer to a value of the mode 'STRUCT(INT width, radix)', whose fields, taken in order, are
  - {width} the length controlled by V1, ..., Vn;
  - {radix} I-4.

n) The "position" of a character C in sframes V1, ..., Vn is an integer i such that Vi is C {which, if it exists at all, is unique}.

o) The length "controlled" by sframes V1, ..., Vn is an integer I, determined as follows:

- counters i and j are set to 0 and 1, respectively;
- while j $\leq$ m:
  - if Vj is an integer, then rep is set to the maximum of 0 and Vj, and j is increased by 1, otherwise rep is set to 1;
  - if Vj is an "s", then j is increased by 1;
  - if Vj is a ".", "+", "-", "e", "i", "z", "d" or "a", then i is increased by rep;
  - j is increased by 1;
- I is i.

p) During formatted transput, values are transput using the current format of the file. This current format, together with its administration, is incorporated in the 'piece' field of the file. The mode of that field is 'reference to reference to FORMATLIST', and it refers (indirectly) 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.

q) 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) the collection list of the given format;
- (next) a nil name.

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

r) 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, and whose fields are:

- (count) r;
- (cp) 1;
- (p) the yield of c;
- (next) the {old} 'piece' field of 'f'.

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

s) 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 constituent collection list of the value {a format} returned by the 'pf' field of 'fp';
- (next) the {old} 'piece' field of 'f'.

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

In all three cases, a special generator is needed to create W: the newly created name must have a scope which is not older than the scope of the value to which it is made to refer, nor newer than that of the file with which the format is associated. {In the present report, the scope is taken equal to that of the file. Even if another solution is adopted, once a scope has been adopted upon associating, the scope of the 'piece' field will not change with the other manipulations.}

{Note that, strictly speaking, objects of the mode specified by FORMAT cannot be constructed at compile time, since these objects contain procedures. With each procedure, an environ is associated, which is not known at compile time. In a straightforward implementation, these procedures may be implemented as a pair consisting of a pointer to the code of the procedure and a pointer to the necessary environ. It is only necessary to keep one environ for all procedures in the format, viz., the newest of all the necessary environs. (According to the Revised Report, this is the

necessary environ of the format.) The implementer may then choose to incorporate only the code pointers in the format, and to keep the environ pointer in the 'piece' field of the file. In this way, objects of the mode specified by FORMAT may be constructed once and for all at compile time. One does need a special procedure-call mechanism to be able to cope with this.}

11.2.3. FORMATTED TRANSPUT ROUTINES

```
a) MODE ? FORMATLIST =
     STRUCT(INT count,  # number of times current piece is to be
                          repeated #
                 cp,     # pointer to current collection #
            REF [] COLLECTION p,  # current collection list #
            REF FORMATLIST next   # pointer to next piece #
           );

b) PROC ? get next picture = (REF FILE f, REF PICTURE picture) VOID:
     IF piece OF f :=: REF REF FORMATLIST(NIL)
     THEN error(noformat); abort
       # no format provided #
     ELSE BOOL picture found:= FALSE,
       STATUS reading = status OF cover OF f;

       IF cp OF piece OF f > UPB p OF piece OF f
       THEN update cp(f, FALSE, SKIP)
       FI;
         # search for the next complete picture or collection list #

       WHILE NOT picture found
       DO

         IF cp OF piece OF f = 0      # format ended #

         THEN BOOL mended = (format mended OF f)(f);
           ensure state(f, reading);
           IF NOT mended
           THEN cp OF piece OF f:= count OF piece OF f:= 1
             # re-iterate the format #
           ELIF cp OF piece OF f = 0
             # no appropriate mending #
           THEN error(noformat); abort
           FI
```

```
ELSE REF REF FORMATLIST piece = piece OF f;

  CASE (p OF piece)[cp OF piece] IN

    (COLLITEM c1):
      BEGIN
        REF FORMATLIST p1 = piece; piece:= NIL;
          # temporarily #
        [1 : UPB i1 OF c1] SFRAME si;

        staticize frames(i1 OF c1, si);
        IF REF FORMATLIST(piece) ISNT NIL
        THEN error(wrongformat); abort
        FI;

        INT count = rep OF c1;
        IF REF FORMATLIST(piece) ISNT NIL
        THEN error(wrongformat); abort
        FI;

        (reading SAYS read mood | get insertion | put insertion)
                          (f, si);
        IF REF FORMATLIST(piece) ISNT NIL
        THEN error(wrongformat); abort
        FI;
          # make sure that no other format has been associated
            with the file #

        piece:= 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, 1, p OF c1, p1);
```

```
            IF count ≤ 0  # repeat zero times #
            THEN picture found:= TRUE;
              picture:= VOIDPICT(HEAP [1:0] DFRAME:= ());
              cp OF piece:= UPB p OF piece + 1
              # This forces the yielding of a void picture.
                Subsequently, the second insertion of the collitem 'cl'
                will be performed. #
            FI
          END,

        (PICTURE pict):
          (picture found:= TRUE; picture:= pict; cp OF piece +:= 1)

      ESAC
    FI
  OD
FI;
```

```
c) PROC ? update cp =
       (REF FILE f, BOOL perform insertions, STATUS reading) VOID:
     BEGIN REF REF FORMATLIST piece = piece OF f;

       WHILE cp OF piece > UPB p OF piece  # piece ended #
       DO

         IF (count OF piece -:= 1) > 0
         THEN cp OF piece:= 1    # repeat this piece #

         ELIF REF FORMATLIST next = next OF piece;
           next :=: REF FORMATLIST(NIL)
         THEN cp OF piece:= 0    # format ended #

         ELSE piece:= next; cp OF piece +:= 1;
           IF perform insertions
           THEN INSERTION extra =
                  CASE (p OF piece)[cp OF piece] IN
                    (COLLITEM c1): i2 OF c1,
                    (FPICT fp): i2 OF fp
                  ESAC;
             [1 : UPB extra] SFRAME sinsert;
             staticize frames(extra, sinsert);
             (reading SAYS read mood | get insertion | put insertion)
                                 (f, sinsert)
           FI
           # handle second insertion of a collection list or format
             pattern #
         FI
       OD
     END;
```

```
d) PROC ? staticize frames =
      (REF [] DFRAME frames, REF [] SFRAME sframes) VOID:
    FOR i TO UPB frames
    DO sframes[i]:=
         CASE frames[i] IN
           (PROC INT n): n,
           (REF [] CHAR s): s,
           (CHAR a): a
         ESAC
    OD;
```

```
e) PROC ? staticize picture = (DPICT p) SPICT:
       # 'staticize picture' turns a picture containing dynamic replicators
         into one containing only simple (integer) replicators. It also
         extracts information needed to build up the character string to be
         output from the frames of the picture. This information is
         collected in the 'p' field of the static picture that is delivered.
         #
     BEGIN HEAP [1 : UPB frames OF p] SFRAME sf;
       staticize frames(frames OF p, sf);
       [1 : CASE type OF p IN 2, 6, 0, 12, 1 OUT 1 ESAC] INT t,
       INT count:= 0, rep:= 1, info:= 1, point:= 6, sign:= 2;

       FOR i TO UPB t DO t[i]:= 0 OD;

       FOR i TO UPB sf
       DO
         CASE sf[i] IN
           (INT n): rep:= 0 MAX n,
           (CHAR a):
             (  IF a = "a" OR a = "d" OR a = "z" THEN count +:= rep
              ELIF a = "+" THEN count +:= 1; t[sign]:=  count
              ELIF a = "-" THEN count +:= 1; t[sign]:= -count
              ELIF a = "." THEN
                t[info]:= count; count:= 0; info +:= 2; t[point]:= 1
              ELIF a = "e" THEN
                t[info]:= count; count:= 0; info:= point - 2;
                sign:= point - 1
              ELIF a = "i" THEN
                t[info]:= count; count:= 0; info:= 7; sign:= 8;
                point:= 12
              ELSE SKIP
              FI;
              rep:= 1)
         OUT rep:= 1
         ESAC
       OD;
```

```
    (UPB t > 0 | t[info]:= count);

    (CASE type OF p IN

      # integral # HEAP STRUCT(INT width, sign):= (t[1], t[2]),

      # real     # HEAP STRUCT(INT b, s1, a, e, s2, point):=
                     (t[1], t[2], t[3], t[4], t[5], t[6]),

      # boolean  # EMPTY,

      # complex  # HEAP STRUCT(REALPATTERN re, im):=
           ((HEAP STRUCT(INT b, s1, a, e, s2, point):=
              (t[1], t[2], t[3], t[4], t[5], t[6]),
            HEAP STRUCT(INT b, s1, a, e, s2, point):=
              (t[7], t[8], t[9], t[10], t[11], t[12]))),

      # string   # t[1]

    OUT

      # bits     # HEAP STRUCT(INT width, radix):=
                     (t[1], type OF p - 4)

    ESAC, sf)
END;
```

```
f) PROC ? put insertion = (REF FILE f, REF [] SFRAME sf) VOID:
     BEGIN INT rep:= 1;
       IF NOT (status OF cover OF f SAYS put char status)
       THEN ensure state(f, put char status)
       FI;
       FOR sfp FROM LWB sf TO UPB sf
       DO
         CASE sf[sfp] IN
           (INT count): rep:= count,
           (REF [] CHAR s): (put insert string(f, rep, s); rep:= 1),
           (CHAR a): (alignment(f, rep, a); rep:= 1)
         ESAC
       OD
     END;

g) PROC ? put insert string = (REF FILE f, INT rep, REF [] CHAR s) VOID:
     # mood ok #
     TO rep
     DO
       FOR i TO UPB s
       DO
         IF
           IF status OF cover OF f SAYS line ok
           THEN TRUE
           ELSE check pos(f)
           FI
         THEN put char(f, s[i])
         ELSE error(nocharpos); abort
         FI
       OD
     OD;
```

```
h) PROC ? get insertion = (REF FILE f, REF [] SFRAME sf) VOID:
     BEGIN INT rep:= 1;
       IF NOT (status OF cover OF f SAYS get char status)
       THEN ensure state(f, get char status)
       FI;
       FOR sfp FROM LWB sf TO UPB sf
       DO
         CASE sf[sfp] IN
           (INT count): rep:= count,
           (REF [] CHAR s): (get insert string(f, rep, s); rep:= 1),
           (CHAR a): (alignment(f, rep, a); rep:= 1)
         ESAC
       OD
     END;

i) PROC ? get insert string = (REF FILE f, INT rep, REF [] CHAR s) VOID:
     # mood ok #
     BEGIN CHAR c, si;
       TO rep
       DO
         FOR i TO UPB s
         DO
           IF
             IF status OF cover OF f SAYS line ok
             THEN TRUE
             ELSE check pos(f)
             FI
           THEN get char(f, c)
           ELSE error(nocharpos); abort
           FI;
           IF c ≠ (si:= s[i])
           THEN BOOL mended = (char error mended OF f)(f, si);
             ensure state(f, get char status);
             (NOT mended | error(wrongchar); abort)
           FI
         OD
       OD
     END;
```

```
j) 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
       IF status OF cover OF f SAYS read mood
       THEN get insert string
       ELSE put insert string
       FI (f, r, LOC [1 : 1] CHAR:= blank)
     FI;
```

```
k) PROC ? do fpict = (REF FILE f, FPICT fpict) VOID:
     BEGIN [1 : UPB i OF fpict] SFRAME si;
       REF FORMATLIST pl = piece OF f;
       REF REF FORMATLIST(piece OF f):= NIL;   # temporarily #
       STATUS reading = status OF cover OF f;

       staticize frames(il OF fpict, si);
       IF REF FORMATLIST(piece OF f) ISNT NIL
       THEN error(wrongformat); abort
       FI;

       FORMAT pf = pf OF fpict;
       IF REF FORMATLIST(piece OF f) ISNT NIL
       THEN error(wrongformat); abort
       FI;

       (reading SAYS read mood | get insertion | put insertion)(f, si);
       IF REF FORMATLIST(piece OF f) ISNT NIL
       THEN error(wrongformat); abort
       FI;
         # make sure that no other format has been associated with
           the file #

       cp OF pl -:= 1;
         # to ensure that the second insertion will eventually be handled #

       REF REF FORMATLIST(piece 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, c OF pf, pl)
     END;
```

```
1) PROC ? associate format = (REF FILE f, FORMAT format) VOID:
     piece 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, c OF format, NIL);

m) PROC ? edit string =
       (REF FILE f, REF [] SFRAME sf, REF INT sfp, REF [] CHAR s,
        BOOL end) VOID:
     # All characters in 's' are mapped one by one against the frames in
       'sf' from the position indicated by 'sfp' onwards. At the end, 'sfp'
       will point just beyond the last frame in 'sf' that has been used. If
       'end' is true, the possible trailing frames in 'sf' (which in that
       case correspond to some final insertion) are handled also. #
     BEGIN INT rep:= 1, j:= LWB s - 1, CHAR k,
       BOOL supp:= FALSE, zs:= TRUE;

       PROC copy = (CHAR c) VOID:
         IF
           IF status OF cover OF f SAYS line ok
           THEN TRUE
           ELSE check pos(f)
           FI
         THEN put char(f, c)
         ELSE error(nocharpos); abort
         FI;
```

```
WHILE j < UPB s OR (end AND sfp ≤ UPB sf)
DO

  CASE sf[sfp] IN

    (INT count): rep:= count,

    (REF [] CHAR s): (put insert string(f, rep, s); rep:= 1),

    (CHAR a):

      IF a = "s" THEN supp:= TRUE
      ELSE

        IF a = "d" THEN zs:= TRUE;
          IF supp THEN j +:= rep
          ELSE
            TO rep
            DO k:= s[j +:= 1]; copy((k = "." | "0" | k)) OD
          FI

        ELIF a = "z" THEN
          TO rep
          DO k:= s[j +:= 1];
            (zs | (k = "0" | k:= "." |: k ≠ "." | zs:= FALSE));
            (NOT supp | copy(k))
          OD

        ELIF a = "a" THEN
          IF supp THEN j +:= rep
          ELSE TO rep DO copy(s[j +:= 1]) OD
          FI

        ELIF a = "+" OR a = "-" THEN k:= s[j +:= 1];
          (zs | (k = "0" | k:= "."
                |: k ≠ "+" AND k ≠ "-" AND k ≠ "." | zs:= FALSE));
          copy(k)
```

```
            ELIF a = "." THEN (NOT supp | copy("."); j +:= 1

            ELIF a = "e" OR a = "i" THEN
              (NOT supp | copy(a)); zs:= TRUE

            ELIF a = "b" THEN copy(s[j +:= 1])

            ELIF a = "r" THEN SKIP

            ELSE alignment(f, rep, a)
            FI;

            supp:= FALSE; rep:= 1
          FI
      ESAC;
      sfp +:= 1
    OD
  END;
```

```
n) PROC putf = (REF FILE f, [] UNION(OUTTYPE, FORMAT) x) VOID:
   BEGIN
    IF NOT (status OF cover OF f SAYS put char status)
    THEN ensure state(f, put char status)
    FI;

    FOR k TO UPB x
    DO

     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);    # mood ok #

         INT n =
           CASE picture IN
             (DPICT dp):
               (picture:= staticize picture(dp);
                  # now the mode of 'picture' is 'SPICT' #
                ensure state(f, put char status); 0),
             (CPICT cp): UPB i2 OF cp,
             (GPICT gp): UPB i2 OF gp,
             (VOIDPICT vp): UPB vp
           OUT 0
           ESAC;

         REF [] SFRAME sinsert:= LOC [1 : n] SFRAME;
           # space for the final insertion of a 'choice-', 'general-' or
             'void-pattern' #
```

```
CASE picture IN
 (SPICT sp):
  BEGIN INT sfp:= 1, REF [] SFRAME sf = sframes OF sp;

   PROC convert 1 real =
       (REALPATTERN rp, REF [] CHAR s, REF INT first, last,
        UNION(£L REAL£) x) BOOL:
      # 'x' is converted into 's' from position 'first' up to
        'last', using the information from 'rp' #
     IF INT sign1 = ABS s1 OF rp;
       INT before = b OF rp - SIGN sign1;
       e OF rp > 0
     THEN INT exp:= before, rplace;
       BOOL neg1 = subfixed(x, a OF rp, exp, s, TRUE);
       last:= a OF rp + before + point OF rp;

       first:=
         IF power10(s, rplace, last)
         THEN exp +:= 1; s[before]:= ".";
           s[before+1]:= (before = 0 | rplace:= 1; "0" | "9");
           last -:= 1; 0
             # xxx9.xxx => xxx.9xxx, .9xxx => 0.xxx
               {the last one will be turned into 1.xxx by
               the subsequent call of 'round'} #
         ELSE 1
         FI;

       round(s, rplace, last);

       IF sign1 ≠ 0
       THEN s[first -:= 1]:=
              (neg1 | "-" |: s1 OF rp > 0 | "+" | ".")
       FI;
         # now s[first:last] contains the stagnant part of x #

       exp -:= before;
         # note that "e" is not explicitly stored in 's', so the
           exponent starts at position 'last+1' in 's' #
```

```
    INT f:= last + 1;
    BOOL neg2 = subwhole(exp, f, last + e OF rp, s);
    INT sign2 = ABS s2 OF rp;

    IF last + SIGN sign2 ≥ f OR
       (sign2 = 0 AND neg2) OR (sign1 = 0 AND neg1)
    THEN FALSE
    ELSE
      (sign2 ≠ 0
      | s[(sign2 + last) MIN (f - 1)]:=
          (neg2 | "-" |: s2 OF rp > 0 | "+" | ".."));
      last +:= e OF rp; TRUE
    FI

  ELSE # e OF rp = 0 #

    INT bb, rplace;
    BOOL neg1 = subfixed(x, a OF rp, bb, s, FALSE);
    last:= bb + a OF rp + point OF rp - 1;
    first:= (power10(s, rplace, last) | bb +:= 1; 0 | 1);
    round(s, rplace, last);

    IF INT p = bb - 1 - b OF rp;
      # '-p' is equal to the space left unfilled #
      p + SIGN sign1 > 0 OR (sign1 = 0 AND neg1)
    THEN FALSE
    ELSE
      (sign1 ≠ 0
      | s[(sign1 + p) MIN (first - 1)]:=
          (neg1 | "-" |: s1 OF rp > 0 | "+" | ".."));
      first:= p + 1; TRUE
    FI
  FI;
```

```
PROC edit l real =
    (UNION(⦃L REAL⦄) x, REALPATTERN rp) VOID:
  IF INT u, v;
    IF e OF rp > 0
    THEN u:= -1;
      v:= b OF rp + a OF rp + 2 + EXPWIDTH x MAX e OF rp
    ELSE INT b = log10(x);
      u:= 0 MIN (b - b OF rp) - 1;
      v:= b + a OF rp + 2
    FI;
      # guess lower and upper bound for 's' #
    [u : v] CHAR s, INT first, last;
    convert l real(rp, s, first, last, x)
  THEN edit string(f, sf, sfp, s[first : last], TRUE)
  ELSE incomp:= TRUE
  FI;
```

```
PROC edit l compl =
    (UNION(⦃L COMPL⦄) z, COMPLPATTERN cp) VOID:
  IF UNION(⦃L REAL⦄) re z =
       CASE z IN ⦃(L COMPL zz): re OF zz⦄ ESAC,
                 im z =
       CASE z IN ⦃(L COMPL zz): im OF zz⦄ ESAC,
    INT u1, v1, u2, v2;

    IF e OF re OF cp > 0
    THEN u1:= -1;
      v1:= b OF re OF cp + a OF re OF cp + 2 +
             EXPWIDTH re z MAX e OF re OF cp
    ELSE INT b = log10(re z);
      u1:= 0 MIN (b - b OF re OF cp) - 1;
      v1:= b + a OF re OF cp + 2
    FI;

    IF e OF im OF cp > 0
    THEN u2:= -1;
      v2:= b OF im OF cp + a OF im OF cp + 2 +
             EXPWIDTH im z MAX e OF im OF cp
    ELSE INT b = log10(im z);
      u2:= 0 MIN (b - b OF im OF cp) - 1;
      v2:= b + a OF im OF cp + 2
    FI;
      # guess lower and upper bound for 's re' and 's im' #

    [u1 : v1] CHAR s re, [u2 : v2] CHAR s im,
    INT f re, l re, f im, l im;

    convert l real(re OF cp, s re, f re, l re, re z) AND
    convert l real(im OF cp, s im, f im, l im, im z)
  THEN edit string(f, sf, sfp, s re[f re : l re], FALSE);
    edit string(f, sf, sfp, s im[f im : l im], TRUE)
  ELSE incomp:= TRUE
  FI;
```

```
CASE p OF sp IN

  (INTPATTERN ip):
    (y[j]
    | (UNION(⦃L INT⦄) i):
        IF INT upbs = INTWIDTH i MAX width OF ip;
          [0 : upbs] CHAR s, INT first:= 0;
          BOOL neg = subwhole(i, first, upbs, s);
            # 'first' points to the first digit of 'i' in 's' #
          INT p = upbs - width OF ip,
                # 'p+1' is the first usable position in 's' #
              abssign = ABS sign OF ip;
          p + SIGN abssign ≥ first OR (abssign = 0 AND neg)
        THEN incomp:= TRUE
        ELSE
          (abssign ≠ 0
          | s[(abssign + p) MIN (first - 1)]:=
              (neg | "-" |: sign OF ip > 0 | "+" | "."));
            # place sign #
          edit string(f, sf, sfp, s[p + 1 : ], TRUE)
        FI
    | incomp:= TRUE),

  (REALPATTERN rp):
    (y[j]
    | ⦃(L REAL r): edit 1 real(r, rp)⦄,
      ⦃(L INT i): edit 1 real(L REAL(i), rp)⦄
      # knowing the internal representation of numbers, this
        case clause can presumably be optimized #
    | incomp:= TRUE),

  (BOOLPATTERN bp):
    (y[j]
    | (BOOL b):
        edit string(f, sf, sfp,
                    LOC [1:1] CHAR:= (b | flip | flop),
                    TRUE)
    | incomp:= TRUE),
```

```
(COMPLPATTERN cp):
  (y[j]
  | ¢(L COMPL z): edit 1 compl(z, cp)¢,
    ¢(L REAL r): edit 1 compl(L COMPL(r), cp)¢,
    ¢(L INT i): edit 1 compl(L COMPL(i), cp)¢
    # knowing the internal representation of numbers, this
      case clause can presumably be optimized #
  | incomp:= TRUE),

(STRINGPATTERN stp):
  (y[j]
  | (CHAR c):
      IF stp = 1
      THEN edit string(f, sf, sfp,
                       LOC [1:1] CHAR:= c, TRUE)
      ELSE incomp:= TRUE
      FI,
    ([] CHAR t):
      IF stp = (UPB t - LWB t + 1) MAX 0
                  # mind superflat rows #
      THEN edit string(f, sf, sfp,
                       LOC [1:stp] CHAR:= t, TRUE)
      ELSE incomp:= TRUE
      FI
  | incomp:= TRUE),
```

```
    (BITSPATTERN bp):
      (y[j]
      | ¢(L BITS lb):
          IF INT upbs = L bits width MAX width OF bp;
            [1 : upbs] CHAR s, L INT n:= ABS lb, INT first:= upbs;

            WHILE s[first]:= dig char(S(n MOD K radix OF bp));
              n OVERAB K radix OF bp; n ≠ L 0
            DO first -:= 1 OD;

            INT p = upbs - width OF bp + 1;
              # 'p' is the first usable position in 's' #
            p > first
          THEN incomp:= TRUE
          ELSE
            WHILE p < first DO s[first -:= 1]:= "." OD;
            edit string(f, sf, sfp, s[p : ], TRUE)
          FI¢
      | incomp:= TRUE)
    ESAC;

    IF incomp
    THEN sfp:= UPB sf;
      WHILE
        CASE sf[sfp] IN
          (CHAR a): char in bag(a, xylpkq)
        OUT TRUE
        ESAC
      DO sfp -:= 1 OD;
      sinsert:= sf[sfp + 1 : ]
        # the last insertion of 'sf' is searched for; this
          insertion will be performed after the 'value error
          mended' routine has been called and (possibly) some
          default action has been taken #
    FI
  END,
```

```
(CPICT choice):
 BEGIN [1 : UPB i1 OF choice] SFRAME si;
   staticize frames(i1 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 INSERTION c1 = (c OF choice)[l];
     [1 : UPB c1] SFRAME ci;
     staticize frames(c1, ci);
     put insertion(f, ci)
   FI;
   staticize frames(i2 OF choice, sinsert)
 END,

(FPICT fpict):
 BEGIN j -:= 1; # since 'y[j]' has not been output yet #
   do fpict(f, fpict)
 END,
```

```
(GPICT gpict):
 BEGIN [1 : UPB i1 OF gpict] SFRAME si;
   staticize frames(i1 OF gpict, si);
   staticize frames(i2 OF gpict, sinsert);
   INT n = UPB spec OF gpict; [1 : n] INT s;
   FOR i TO n DO s[i]:= (spec OF gpict)[i] OD;
   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]))
             # For optimization purposes, one might want to
               generate different code here. #
       ESAC
     FI
   FI
  END,

 (VOIDPICT v):
   BEGIN j -:= 1;  # since 'y[j]' has not been output yet #
     staticize frames(v, sinsert)
   END

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;

        IF UPB sinsert > LWB sinsert
        THEN put insertion(f, sinsert)
        FI;
          # put the final insertion of the picture at hand #

        IF cp OF piece OF f > UPB p OF piece OF f  # piece ended #
        THEN update cp(f, TRUE, put char status)
        FI
          # search for the next complete picture or collection list, in
            the meantime outputting final insertions #

      OD
    END
  ESAC
 OD
END;
```

```
o) PROC indit string =
      (REF FILE f, REF [] SFRAME sf, REF INT sfp, REF [] CHAR s,
       INT sign, radix, BOOL end) VOID:
          # The frames in 'sf' from the position indicated by 'sfp' onwards
            are used to indit 'index' characters, which are placed in 's'
            from position 1 onwards. At the end, 'sfp' will point just
            beyond the last frame in 'sf' that is used. If 'end' is true,
            the possible trailing frames in 'sf' are handled also. {Note
            that the ".", "e" and "i" are not stored in 's'; their positions
            are known from the simple picture directly.} #
    BEGIN
      CHARBAG digits = (  radix = 10 | radix10digit
                       |: radix =  2 | radix 2digit
                       |: radix =  4 | radix 4digit
                       |: radix =  8 | radix 8digit
                                     | radix16digit);
      CHARBAG digits and space     = digits + STRINGTOBAG STRING(".");
              digits and signspace = digits + signspace;

      PROC expect char = (CHARBAG s, CHAR c) CHAR:
        # expects a character contained in 's'; if the character read is
          not in 's', then the event routine corresponding to the 'on char
          error' event is called, with the suggestion 'c' #
        IF CHAR k;
          IF
            IF status OF cover OF f SAYS line ok
            THEN TRUE
            ELSE check pos(f)
            FI
          THEN get char(f, k)
          ELSE error(nocharpos); abort
          FI;
          char in bag(k, s)
        THEN k
```

```
  ELSE k:= c; BOOL mended = (char error mended OF f)(f, k);
    ensure state(f, get char status);
    IF (mended | char in bag(k, s) | FALSE)
    THEN k
    ELSE error(wrongchar); c
    FI
  FI;

INT index:= ABS sign, rep:= 1,
BOOL sign found:= FALSE, first space:= FALSE, supp:= FALSE;
CHARBAG allowed:= CASE sign + 2 IN
  # "-"-frame #  (first space:= TRUE; signspace),
  #  no frame #  (sign found:= TRUE; digits and space),
  # "+"-frame #  signspace
                 ESAC;

WHILE index < UPB s OR (end AND sfp ≤ UPB sf)
DO

  CASE sf[sfp] IN

   (INT count): rep:= count,

   (REF [] CHAR s): (get insert string(f, rep, s); rep:= 1),

   (CHAR a):

    IF a = "s" THEN supp:= TRUE
    ELSE
```

```
IF a = "d" THEN
     TO rep
     DO s[index +:= 1]:=
          (supp | "0" | expect char(digits, "0"))
     OD;
     allowed:= digits and space

ELIF a = "z" OR a = "+" OR a = "-" THEN
     TO rep
     DO
       IF supp
       THEN s[index +:= 1]:= "0"
       ELIF sign found
       THEN s[index +:= 1]:=
               (CHAR c = expect char(allowed, "0");
                (c ≠ "." | allowed:= digits; c | "0"))
       ELSE CHAR c:=
              expect char((a = "+" | plusminus
                                   | allowed),
                          "+");
         IF c = "." AND a = "z"
         THEN (first space | allowed:= digits and signspace;
                             first space:= FALSE);
           c:= "0"
         ELSE sign found:= TRUE; allowed:= digits;
           (c = "." | c:= "+")
         FI;
         (c = "+" OR c = "-" | s[1]:= c | s[index +:= 1]:= c)
       FI
     OD

ELIF a = "." THEN
     (NOT supp | expect char(point, "."))

ELIF a = "e" THEN
     (NOT supp | expect char(times ten to the power, "e"))
```

```
        ELIF a = "i" THEN
             (NOT supp | expect char(plus i times, "i"))

        ELIF a = "r" THEN SKIP

        ELIF a = "b" THEN
             s[index +:= 1]:= expect char(flipflop, flop)

        ELIF a = "a" THEN
             TO rep
             DO s[index +:= 1]:=
                  IF supp
                  THEN "."
                  ELSE CHAR c;
                    IF
                      IF status OF cover OF f SAYS line ok
                      THEN TRUE
                      ELSE check pos(f)
                      FI
                    THEN get char(f, c)
                    ELSE error(nocharpos); abort
                    FI;
                    c
                  FI
             OD
        ELSE alignment(f, rep, a)
        FI;
        rep:= 1; supp:= FALSE
      FI
    ESAC;
    sfp +:= 1
  OD
END;
```

```
p) PROC getf = (REF FILE f, [] UNION(INTYPE, FORMAT) x) VOID:
   BEGIN
    IF NOT (status OF cover OF f SAYS get char status)
    THEN ensure state(f, get char status)
    FI;

    FOR k TO UPB x
    DO

     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);

         INT n =
           CASE picture IN
             (DPICT dp): (picture:= staticize picture(dp);
                          # now the mode of 'picture' is 'SPICT' #
                         ensure state(f, get char status); 0),
             (CPICT cp): UPB i2 OF cp,
             (GPICT gp): UPB i2 OF gp,
             (VOIDPICT vp): UPB vp
           OUT 0
           ESAC;

         [1 : n] SFRAME sinsert;
           # since the last insertion of a simple picture is always handled
             by 'indit string', no special arrangements to recover this
             final insertion need be made here (as had to be done in 'putf')
             #
```

```
CASE picture IN

 (SPICT sp):
  BEGIN INT sfp:= 1, REF [] SFRAME sf = sframes OF sp;

   PROC convert 1 real =
        (REALPATTERN rp, UNION(⦃REF L REAL⦄) rr, BOOL end) BOOL:
     BEGIN INT upbs = a OF rp + b OF rp, exp width = e OF rp;
       [1 : upbs] CHAR s;
       indit string(f, sf, sfp, s, SIGN s1 OF rp, 10,
                       exp width = 0 AND end);
       BOOL ok:= TRUE, INT first:= (s1 OF rp = 0 | 1 | 2);
         # 's' contains the digits from the stagnant part;
           'first' points to the first digit #

       WHILE first < upbs AND s[first] = "0"
       DO first +:= 1 OD;
         # skip leading zeroes; 'first' now points to the first
           significant digit #

       INT last = (first + RREALWIDTH rr) MIN upbs;
       INT exp:= b OF rp - last;
         # distance from the (implicit) decimal point to the
           last significant digit #

       IF exp width > 0
       THEN [1 : exp width] CHAR s;
         indit string(f, sf, sfp, s, SIGN s2 OF rp, 10, end);
           # 's' contains the exponent, including the sign,
             if any #
         INT e:= 0;

         FOR i FROM (s2 OF rp = 0 | 1 | 2) TO exp width
         WHILE ok
         DO ok:= e ADD char dig(s[i]) OD;
```

```
        IF ok:= ok AND ABS exp ≤ max int - e
        THEN exp +:= (s[1] = "-" | -e | e)
        FI
          # test for integer overflow, and determine exponent #
      FI;

      IF NOT ok
      THEN FALSE
      ELSE string to l real(s[first:last], exp, s[1] = "-", rr)
      FI
    END;

  CASE p OF sp IN

    (INTPATTERN ip):
      (y[j]
      | ↰(REF L INT ii):
          BEGIN [1 : width OF ip] CHAR s;
            indit string(f, sf, sfp, s, sign OF ip, 10, TRUE);
              # 's' contains all digits, and the sign, if any #
            BOOL ok:= TRUE, L INT j:= L 0;

            FOR i FROM (sign OF ip = 0 | 1 | 2)
            TO width OF ip WHILE ok
            DO ok:= j ADD char dig(s[i]) OD;

            (ok | ii:= (s[1] = "-" | -j | j));
            incomp:= NOT ok
          END↱
      | incomp:= TRUE),

    (REALPATTERN rp):
      (y[j]
      | (UNION(↰REF L REAL↱) rr):
          incomp:= NOT convert l real(rp, rr, TRUE)
      | incomp:= TRUE),
```

```
(BOOLPATTERN bp):
  (y[j]
  | (REF BOOL bb):
      BEGIN [1 : 1] CHAR s;
        indit string(f, sf, sfp, s, 0, 0, TRUE);
        bb:= s[1] = flip
      END
  | incomp:= TRUE),

(COMPLPATTERN cp):
  (y[j]
  | ¢(REF L COMPL zz):
      incomp:= NOT
        IF convert 1 real(re OF cp, re OF zz, FALSE)
        THEN convert 1 real(im OF cp, im OF zz, TRUE)
        ELSE FALSE
        FI¢
  | incomp:= TRUE),

(STRINGPATTERN stp):
  (y[j]
  | (REF CHAR cc):
       IF stp = 1
       THEN indit string(f, sf, sfp, REF [] CHAR(cc),
                         0, 0, TRUE)
       ELSE incomp:= TRUE
       FI,
    (REF [] CHAR ss):
       IF stp = (UPB ss - LWB ss + 1) MAX 0
       THEN indit string(f, sf, sfp, ss[@1], 0, 0, TRUE)
       ELSE incomp:= TRUE
       FI,
    (REF STRING ss):
       BEGIN [1 : stp] CHAR s;
         indit string(f, sf, sfp, s, 0, 0, TRUE);
         ss:= s
       END
  | incomp:= TRUE),
```

```
(BITSPATTERN bp):
  (y[j]
  | ¢(REF L BITS lb):
       BEGIN [1 : width OF bp] CHAR s,
         INT radix = radix OF bp;
         indit string(f, sf, sfp, s, 0, radix, TRUE);
         INT r = (radix = 2 | 1 |: radix = 4 | 2
               |: radix = 8 | 3 | 4),
         INT w:= width OF bp, n:= 0, d,
         [1 : L bits width] BOOL b;

         FOR i FROM L bits width BY -1 TO 1
         DO
           IF n = 0
           THEN d:= (w > 0 | char dig(s[w]) | 0);
             w -:= 1; n:= r
           FI;
           b[i]:= ODD d; d:= d OVER 2; n -:= 1
         OD;
           # convert radix integer to [] BOOL #

         lb:= L bits pack(b)
       END¢
    | incomp:= TRUE)
 ESAC
END,
```

```
(CPICT choice):
  BEGIN [1 : UPB i1 OF choice] SFRAME si;
    staticize frames(i1 OF choice, si);
    get insertion(f, si);
    INT c = c OF cpos OF cover OF f, CHAR kk,
    INT k:= 0, BOOL found:= FALSE;

    IF NOT (possibles OF cover OF f SAYS backspace poss)
    THEN error(nobackspace); abort
    FI;

    WHILE k < UPB c OF choice AND NOT found
    DO INSERTION ck = (c OF choice)[k +:= 1];
      [1 : UPB ck] SFRAME si;
      staticize frames(ck, si);
      ensure state(f, get char status);
      BOOL bool:= TRUE, INT rep:= 1;

      FOR i TO UPB si WHILE bool
      DO
        CASE si[i] IN
         (INT count): rep:= count,
         (REF [] CHAR ss):
           (FOR j TO rep WHILE bool
            DO
              FOR l TO UPB ss
              WHILE bool:= bool AND
                       status OF cover OF f SAYS line ok
              DO get char(f, kk); bool:= kk = ss[1] OD
            OD;
            rep:= 1)
        ESAC
      OD;
        # try to read 'si' #

      (NOT (found:= bool) | set char number(f, c))
    OD;
      # try successive insertions until 'si' is found #
```

```
      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 frames(i2 OF choice, sinsert)
    END,

  (FPICT fpict): (j -:= 1; do fpict(f, fpict)),

  (GPICT gpict):
    BEGIN [1 : UPB i1 OF gpict] SFRAME si;
      staticize frames(i1 OF gpict, si);
      staticize frames(i2 OF gpict, sinsert);
      get insertion(f, si); get(f, y[j]);
      incomp:= UPB spec OF gpict > 0
    END,

  (VOIDPICT v):
    (j -:= 1; staticize frames(v, 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;
```

```
        IF UPB sinsert > LWB sinsert
        THEN get insertion(f, sinsert)
        FI;
          # get the final insertion of a 'choice-', 'general-' or 'void-
            pattern' #

        IF cp OF piece OF f > UPB p OF piece OF f
        THEN update cp(f, TRUE, get char status)
        FI
          # search for the next complete picture or collection list, in the
            meantime inputting final insertions #

      OD
    END
  ESAC
 OD
END;
```

11.3. EFFICIENCY

Considerable attention has been paid to the efficiency of formatted transput. The data structures have been chosen with great care, so as to minimize both space and time. Firstly, the gain in speed obtained is considerable if a picture containing only integer replicators leads to a so-called simple picture. Typically, the gain obtained hereby is linear in the number of constituent replicators of the picture. {To give a rough idea, for the picture '12d', a 20 percent increase in speed may well be expected, while for the picture 'ddddddddddd' the gain may well be a 100 percent increase in speed.}

By using the internal structure which does not explicitly store the default replicators and insertions, a considerable gain in both space and time is obtained.

Measurements have shown that, using this optimized version of formatted transput, more than 50 percent of the time needed for a typical formatted transput operation is spent inside the routines 'edit string' and 'indit string'. The efficiency may therefore be further increased by carefully

rewriting these routines in machine code.

A very reasonable intermediate form is obtained when only the routines 'copy' in 'edit string' and 'expect char' in 'indit string' are optimized. In some ALGOL 68 implementations, procedure calls will be relatively expensive. It may therefore be worthwhile to minimize the number of procedure calls to be made for each character that is to be transput. If implemented straightforwardly, each call of the routine 'copy' in 'edit string' leads to at least 2 other procedure calls, one of which can be easily circumvented by just copying the code of 'put char' instead of calling the routine. Similarly for the routine 'get char' that is called from inside the body of 'expect char'.

Another worthwhile optimization is not to incorporate information on the environs necessary for the procedures of a format inside objects of the mode FORMAT (see also the last part of section 10.2.2). Those objects then become completely static, and may therefore be constructed at compile time.

In both the Revised Report and the present model, frames are coded as characters. Since these frames are heavily used inside conditional-clauses (for example in 'edit string' and 'indit string'), recoding them as integers may lead to a significant increase in speed. As a special case, the 'q'-alignment may be represented internally as the string "."; this obviates its special treatment in the routine 'alignment' (11.2.3.j).

12. BINARY TRANSPUT

12.1. DIFFERENCES

- {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.

- {C30} An implementer of the binary transput as defined in the Revised Report is not, in general, able to output stowed values in the form in which they are stored internally. This is because of the requirement to be able subsequently to re-input their component fields or elements either singly, or stowed in other ways. Clearly, it is much easier (and more efficient) to implement a sublanguage feature in which output values of some mode may only be re-input into variables of that same mode. This will detract little from the power of the language, and even increase its safety. This scheme is followed in the present model. Therefore, the elements of the data list are not straightened; rather, the routines 'to bin' and 'bin length' are given values of the mode OUTTYPE and INTYPE, respectively.

- {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 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 OUTTYPE 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 a 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 is transput via some 'row of BINCHAR', the length of the row being determined by the book to which the transput takes place, the mode of the value to be transput (and its length in the case of a multiple value). C;

b) PROC ? to bin = (REF FILE f, OUTTYPE 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, OUTTYPE y, [] BINCHAR c) OUTTYPE:
    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, INTYPE y) INT:
     C The upper bound of the multiple value which is needed to input a
       value into 'y'. C;

e) PROC put bin = (REF FILE f, [] OUTTYPE ot) VOID:
     BEGIN
       IF NOT (status OF cover OF f SAYS put bin status)
       THEN ensure state(f, put bin status)
       FI;
       FOR k TO UPB ot
       DO [] BINCHAR bin = to bin(f, ot[k]);
         FOR i TO UPB bin
         DO
           IF NOT (status OF cover OF f SAYS line ok)
           THEN next pos(f)
           FI;
           (write bin char OF f)(f, bin[i]);
           REF COVER cover = cover OF f;
           IF c OF cpos OF cover > char bound OF cover
           THEN status OF cover ANDAB line end
           FI
         OD
       OD
     END;
```

```
f) PROC get bin = (REF FILE f, [] INTYPE it) VOID:
     BEGIN
       IF NOT (status OF cover OF f SAYS get bin status)
       THEN ensure state(f, get bin status)
       FI;
       FOR k TO UPB it
       DO [1 : bin length(f, it[k])] BINCHAR bin;
         FOR i TO UPB bin
         DO
           IF NOT (status OF cover OF f SAYS line ok)
           THEN next pos(f)
           FI;
           (read bin char OF f)(f, bin[i]);
           REF COVER cover = cover OF f;
           REF STATUS status = status OF cover;
           IF c OF cpos OF cover = c of lpos OF cover
           THEN
             IF status SUGGESTS lfe in current line
             THEN status ANDAB logical file ended
             ELSE status ANDAB line end
             FI
           FI
         OD;
         C the name yielded by 'it[k]' C:= from bin(f, it[k], bin)
       OD
     END;
```

13. OPENING OF STANDARD FILES

13.1. DIFFERENCES

- {S} The Revised Report defines the identification string of the files declared and opened in the standard prelude (section 10.5.1 of the Revised Report) to be the empty string. In the present model, these identification strings are considered to be implementation-dependent. It is expected that on many systems empty strings cannot be used as the identification string of a book. Moreover, systems may well not allow two books to have the same identification string.

  If a book is linked to a file via the routine 'open' (5.2.c), it must in some sense be available already. If the user does not employ one or more of the standard files, this may not be true, and 'open' may therefore fail. In the present model, empty books are created in those cases.

13.2. NEW DEFINITION

Below, only the revised version of 10.5.1.c is given.

a) FILE standin, standout, standback;

b) IF STRING s = C the identification string of the standard input file C;
   open(standin, s, stand in channel) = 0
THEN SKIP
ELIF C construct an empty book with identification string 's' C;
  INT er = open(standin, s, stand in channel);
  er ≠ 0
THEN error(er); abort
FI;

```
c) IF STRING s = C the identification string of the standard output file C;
     open(standout, s, stand out channel) = 0
   THEN SKIP
   ELIF POS default = default size(stand out channel);
     INT er = establish(standout, s, stand out channel, p OF default,
                        1 OF default, c OF default);
     er ≠ 0
   THEN error(er); abort
   FI;

d) IF STRING s = C the identification string of the standard standback file
                  C;
     open(standback, s, stand back channel) = 0
   THEN SKIP
   ELIF POS default = default size(stand back channel);
     INT er = establish(standback, s, stand back channel, p OF default,
                        1 OF default, c OF default);
     er ≠ 0
   THEN error(er); abort
   FI;
```

REFERENCES

[1] WIJNGAARDEN, A. VAN, et al (eds.), Revised Report on the Algorithmic Language ALGOL 68, Acta Informatica 5 (1975), pp 1-236, SIGPLAN Notices 12, no 5, pp 1-70, May 1977.

[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, Revision B, 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] FISKER, R.G., Manchester University portable ALGOL 68 transput implementation, July 1977.

[12] VLIET, J.C. VAN, Towards an Implementation-Oriented Definition of the ALGOL 68 Transput, Report IW90, Mathematical Centre, October 1977.

[13] VLIET, J.C. VAN, An Implementation Model of the ALGOL 68 Transput, Draft version, Report IN15, Mathematical Centre, July 1978.

[14] VLIET, J.C. VAN, An Implementation Model of the ALGOL 68 Transput, Part I and II, Working Document Transput Task Force Meeting, Mathematical Centre, December 1978.

[15] VLIET, J.C. VAN, An Implementation Model of the ALGOL 68 Transput, Working Document Transput Task Force Meeting, Mathematical Centre, April 1979.

[16] EHLICH, H. & H. WUPPER, Experiences with ALGOL 68 Transput and its Implementation, Arbeitsberichte des Rechenzentrums der Ruhr-Universitaet Bochum, ISSN 0341-0358, Nr. 7901.

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

[18] IFIP WG 2.1 - Subcommittee on ALGOL 68 Support, Commentaries on the Revised Report, ALGOL Bulletin 43, AB43.3.1, December 1978, ALGOL Bulletin 44, AB44.3.1, May 1979.

[19] LINDSEY, C.H., ALGOL 68 and your friendly neighbourhood operating system, ALGOL Bulletin 42, AB42.4.4, May 1978.

[20] FISKER, R.G., The transput section for the Revised ALGOL 68 Report, M. Sc. thesis, Dept. of Computer Science, University of Manchester, August 1974 (revised version).

[21] CMU - ALGOL 68 User's Manual, Chapter 5: Transput User's Guide, Technical Report, Carnegie Mellon University, to be published.

[22] CMU - ALGOL 68 Operating System Interface Guide, Technical Report, Carnegie Mellon University, to be published.

INDEX

Alphabetic listing of all defining occurrences of mode-indications and identifiers. The number in front of each entry indicates the page where the defining occurrence can be found. The applied occurrences are listed between { and }; if some entry is used more than once in a given routine-text, the number of applied occurences is placed between (and). In case an indicant is applied in almost all of the routines, this is denoted by {MANY}. References to Chapter 13 (Opening of standard files) are denoted by 'Ch13'. If a mode-indication or identifier is prefixed with an *, this means that it is not (completely) defined in ALGOL 68.

```
140  get char {get(9), get insert string, indit string(2), getf}
 81  get char status {get(6), get char, get insertion(2), get insert
         string, indit string, getf(7)}
171  get insert string {get insertion, alignment, indit string}
171  get insertion {get next picture, update cp, do fpict, getf(3)}
163  get next picture {putf, getf}
 25  get poss {get possible, open, set read mood, reset}
 36  get possible {}
192  getf {}
151  GPICT {PICTURE, putf(2), getf(2)}
188  indit string {getf(8)}
 43 *init buffer {do newline, do newpage, do set, associate(2), set write
         mood, ensure logical file, ensure physical file, set char number,
         get(2)}
151  INSERTION {COLLITEM(2), CPICT(3), FPICT(2), GPICT(2), VOIDPICT, update
         cp, putf, getf}
151  INTPATTERN {SPICT, putf, getf}
117  INTWIDTH {whole, putf}
119 *INTYPE {STRAIGHTIN, get(2), getf(2), bin length, get bin}
117 *L exp width {EXPWIDTH, put(2)}
116  L int width {INTWIDTH, put}
117 *L real width {REALWIDTH, RREALWIDTH, put(2)}
 81  lfe in current line {set logical pos, init buffer(2), write buffer(2),
         do set, set read mood, newline, get char, get bin}
 81  line end {init buffer, do set(2), associate(2), space(3), put(2), put
         char, get char, put bin, get bin}
 79  line number {}
 81  line ok {set write mood, space, backspace, ensure line, check pos, set
         char number, put(5), get(8), put insert string, get insert string,
         edit string, indit string(2), getf, put bin, get bin}
 76 *lock {}
114 *log10 {fixed, putf(3)}
 81  logical file ended {init buffer, do newpage, do set, associate(2),
         mind logical pos, space, newline, get char, get bin}
 81  logical pos ok {set write mood, ensure logical file, ensure physical
         file}
 38 *make conv {}
 38  make term {}
```

```
141  radix 2digit {indit string}
141  radix 4digit {indit string}
141  radix 8digit {indit string}
141  radix10digit {get(3), indit string}
141  radix16digit {indit string}
 54 *read bin char {establish, open, get bin}
 53 *read char {establish, open, space, get char}
 82  read mood {init buffer, do set, open, associate(2), set write mood,
         set read mood, mind logical pos, space, newline, ensure state,
         reset, get next picture, update cp, alignment, do fpict}
 82  read or write mood {associate}
 82  read to write not possible {set write mood}
152  REALPATTERN {SPICT, COMPLPATTERN, staticize picture, putf(3), getf(2)}
117  REALWIDTH {put}
117  RREALWIDTH {get, getf}
 39 *reidf {}
 25  reidf poss {reidf possible}
 37  reidf possible {}
 98  reset {}
 25  reset poss {reset possible}
 36  reset possible {reset}
114  round {fixed, float, put, putf(2)}
 24,80  SAYS {MANY}
 76 *scratch {}
 97  set {}
 75  set bin mood {ensure state}
 75  set char mood {ensure state}
 99  set char number {alignment, getf}
 18 *set logical pos {write buffer, set write mood}
 25  set poss {set possible, establish, open, reset}
 37  set possible {set}
 75  set read mood {ensure state}
 74 *set write mood {open, ensure state, reset}
151  SFRAME {SPICT, get next picture, update cp, staticize frames,
         staticize picture, put insertion, get insertion, do fpict, edit
         string, putf(5), indit string, getf(5)}
141  signspace {indit string(3)}
119  SIMPLIN {STRAIGHTIN, get, getf}
```

```
 46 *write buffer {do newline, do newpage, do reset, do set, close, lock,
        scratch}
 53 *write char {establish, open, space, put(3), put char}
 82  write mood {do newline, do newpage, set write mood, set read mood}
 81  write sequential {init buffer, set write mood}
 82  write to read not possible {set read mood}
 64  wrongbacksp {backspace}
 64  wrongbin {}
 64  wrongchar {get, get char, indit string}
 64  wrongformat {get next picture(3), do fpict(3)}
 63  wrongmult {associate}
 64  wrongpos {set char number, get insert string}
 64  wrongset {do set, associate}
 64  wrongval {get, putf, getf}
141  xylpkq {putf}
141 *+ {indit string(2)}
```