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

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

MATHEMATICAL CENTRE TRACTS 110

ALGOL 68 TRANSPUT

PART I
HISTORICAL REVIEW AND DISCUSSION OF THE IMPLEMENTATION MODEL

J.C. VAN VLIET

MATHEMATISCH CENTRUM AMSTERDAM 1979

1980 Mathematics subject classification: 68B20, 68-03, 01A99

ACM-Computing Reviews-category: 4.41, 1.2

ISBN 90 6196 178 5

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. I am grateful to H. de J. LAIA LOPES for his assistance in testing the implementation model.

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

The monograph 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.

SUMMARY

This monograph 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 monograph 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.

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.

$$\begin{tikzcd}
\text{program} \arrow[rr, shift left] & & \text{logical interface} \arrow[ll, shift left] \arrow[rr, shift left] & & \text{physical device} \arrow[ll, shift left] \\
& \text{conversion} \arrow[u] & & \text{transmission} \arrow[u] &
\end{tikzcd}$$

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), $\quad$ (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 <u>end</u> 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. Both in [4] and in the ultimate ALGOL 68 report [8], the transput section comprises about 20 pages. 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 <u>generated</u>, 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 <u>intention</u>, 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 <u>nothing whatsoever</u> 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 <u>intention</u> 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.

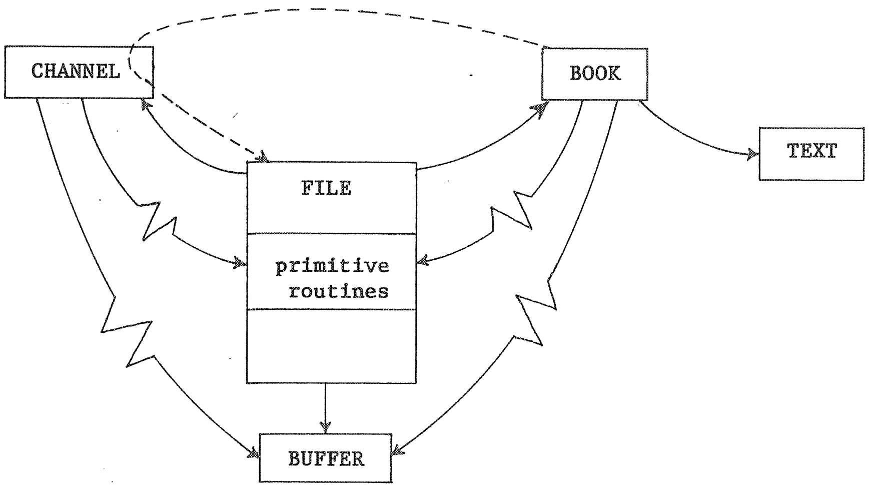
Various other miscellaneous comments (like the definitions of the modes INTYPE and OUTTYPE and some special generators in the formatted-transput sections) will not be further discussed here. These will sometimes necessitate special provisions in the compiler.

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 $A \longrightarrow B$ indicates that A contains a reference to B; $A \rightsquigarrow B$ 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.

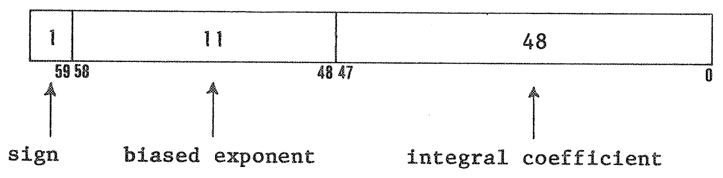
4.2. IMPLEMENTING REAL ARITHMETIC

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, $A = \text{entier}(\log_{10}V) + 1$. (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 ($E = \text{ENTIER}(\log_2 V)$). Note that A' may be one larger than the proper value of A.

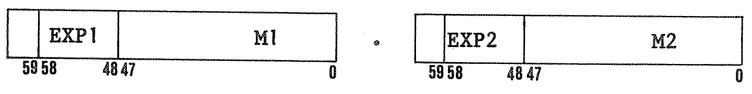
Converting a floating-point number X to an integer can be done very easily on the Cyber if X is such that

$$0.1 \leq X < 1.0.$$

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:

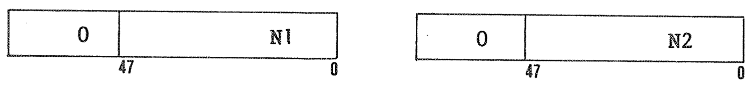


(The decimal point is in between the two parts of the number.) The value of X, being the scaled value of V, now is:

$2^{EXP1} * M1 + 2^{EXP2} * M2$, 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.

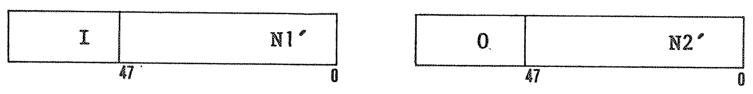
Extracting digits is very easy if the integer is stored as follows:



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 <u>14</u> (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, ALGOL Bulletin 44, pp 6-7, May 1979.

[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.