IA

stichting
mathematisch
centrum

MC

AFDELING INFORMATICA IW 10/73 OCTOBER

IA

H.W. ROOS LINDGREEN

A FILE SYSTEM FOR MULTI-SEQUENTIAL FILES

2e boerhaavestraat 49 amsterdam

BIBLIOTHEEK MATHEMATISCH CENTRUM
AMSTERDAM

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

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

ACM - Computing Reviews - category: 4.41

ABSTRACT

A user oriented description of a file system is given, and also an implementation of that system in the form of an ALGOL 60 program. It is a dynamic system in the sense that there are no fixed upper bounds for the number of files or the length of a file. A new type of file access, multi-sequential access, is introduced, allowing the user to have sequential access to a file at various entry points of that file simultaneously. All files within the system presented are of this access type and therefore termed multi-sequential files.
It is assumed that the system is embedded in an operating system having dynamic storage allocation features. The specific requirements of the file system with respect to that operating system are listed. Underlying concepts of both implementation and design of the file system are given.

TABLE OF CONTENTS

0. PREFACE 1

1. INTRODUCTION 3

1.1. Some general remarks 3

1.2. History 4

1.3. Results 4

1.4. Report get-up 6

2. GENERAL 7

2.1. Technical information 9

2.2. General concepts of the implementation 10

2.3. General design considerations 24

3. POINTER ROUTINES 26

3.1. The routines 27

3.2. Implementation 28

3.3. Pointer resetting 31

4. ACCESS ROUTINES 32

4.1. The routines 32

4.2. Implementation 34

4.3. Designing the access actions 35

5. FILE OPENING 40

5.1. The routines 40

5.2. Implementation 43

6. FILE CLOSING 49

6.1. The routines 50

6.2. Implementation 51

6.3. Implicit closing 54

7. FILE NAMING 55

7.1. The routine 55

7.2. Implementation 56

8. INQUIRY ROUTINES 58

8.1. The routines 58

8.2. Implementation 59

8.3. Design considerations 60

TABLE OF CONTENTS (continued)

9. DYNAMIC STORAGE ALLOCATION MODULE 61
9.1. External description 61
10. INTERFACE FILE SYSTEM / OPERATING SYSTEM 64
10.1. Routines, constants and variables 64
10.2. Actions 68
10.3. Hidden interface 69
11. TESTING THE SYSTEM 71
12. THE PROGRAM 73
12.1. The program text 74
REFERENCES 103

0. PREFACE

The present report contains a detailed description of a file system. This detailed description consists of an integral program text and also of an ample explanation of that text and of the underlying concepts.
In programming literature the term "file system" rather often occurs, mostly in one of the three following significations [5,8,9,11]:

- an operating system heavily leaning upon the use of files. Files then serve data transmission between users, between user and system, and between various parts of the operating system itself. Such a system preferably should be called a "file oriented system".
- one of the modules of an operating system. This module, the file module, deals with all file actions. In this report the terms "file module" and "file system" will be equivalent.
- an information retrieval system in which in general a lot of attention is paid to the data structures within the files. Such a system preferably should be called a "filing system".

The file system presented here can be considered to be a system for dynamic files. This term needs some clarification. A file often is considered as being either static or dynamic depending upon the frequency of file alterations. In this report, however, files are termed dynamic, because they can grow and shrink in number and length.

The files are named multi-sequential after their method of access. Some general types of access such as direct, random, word addressable, sequential, indexed sequential, etc. do not aptly apply to the method of access practiced here. The term multi-sequential is introduced because the way of accessing is of sequential nature and has some analogy with (the accepted term) multi-access [9]: the user has a variable number of reference points on his file and he accesses the file sequentially via those references.

Files may stay in the operating system permanently. Attention has been paid to such matters as unique naming, scratch files, multi-read files, private and public files, but not to matters such as creation date, retention period, extra cycles, passwords or priorities.

Special attention is paid to the use of the system by ordinary programs (opposed to system routines), which resulted, among other things, in extensive error checking and the distinction between fatal and venial errors.

No attention has been paid to special data structures within a file. A file is a linear memory that is not structured. Each file is of some species, the species dictating the type of the memory cells, that may be different for different files. To give an impression of what is meant: we can speak about "real files","boolean files", etc. This allows efficient storage in case all information to be stored in a file is of the same type.

1. INTRODUCTION

1.1. Some general remarks

A file is a macroscopic information unit. In general it is of no basic importance in what way a file is represented, nor which media are used. Therefore, a file can be considered a logical entity. When a file is observed more in detail, a microscopic structure might be noticed. In general a file consists of a number of blocks, each block consisting of a number of records and each record consisting of a number of elements. The files presented here consist of a number of elements and no other hierarchy is of importance of the user.

Files supply the following wants:

- Large operating systems need some means for internal data transmission. Using files an operating system is able to standardize communication between its modules.
- Communication between an interactive user and the system is done easily and smoothly via files.
- Files form an excellent medium for permanent storage of data, programs, etc.
- When handling large amounts of information, files can be of prime assistance.

A *file system* is a collection of routines that

- enables the user to create, read, alter, store permanent and/or delete files.
- forms a module of the operating system it is embedded in; a module that is pretty well autonomous.

In view of the important role files (are able to) play in nearly all system actions, a file system belongs to the basis, the kernel of any advanced operating system.

1.2. History

The development of a time sharing system for the computer tandem consisting of the Electrologica X8 and DEC PDP8-I led to the design of a file system. This design contained the concepts of both sequential access files and random access files. A proposal for the sequential access files was given in [2]. This proposal was a starting point for the investigation reported here. Although an accurate description of the file routines was given in [2], it did not answer some questions of vital importance to some actual implementation of the files, such as what to do in case of space exhaustion, what is the maximal file length, etc. (quoted from [2]: "This proposal does not go into the matter of the principally finite length of a file".). To these and other questions this report gives a full answer. While considering some of the aspects of the proposal just mentioned, it was decided to reconsider all of it, which resulted in a completely new proposal, done by L.J.M. Geurts, L.G.L.Th. Meertens and the author [1]. One of the main improvements is the concept of the multi-sequential file. All files in this proposal dispose of a method of access that facilitates the user to manipulate his files in a way that gives him pseudo random access.

The next question to be answered was if the proposal was implementable. It was answered in the affirmative with the proviso that some sort of dynamic storage allocation scheme should be available. It was decided to realize the file system within the existing operating system for the X8, Milli [3]. The first step in this process was made by L.G.L.Th. Meertens and the author. They developed and coded a dynamic storage allocation module for the X8 [6]. The next step was to implement the file system itself, which activity is given account of in this report.

1.3. Results

As mentioned before, an interesting by-product of the construction of the file system was a dynamic storage allocation module. A very brief description of its external behaviour will be given in chapter 9.

The file system as it is realized consists of a number of ALGOL 60 procedures that have a well determined interaction with routines that belong to the operating system. Those routines have to do with memory reservation, job scheduling, etc. and therefore lay beyond the scope of the file module itself. The dynamic storage features for a good deal dictated the data structures the files are administrated within. To test the file module without having it actually embedded in the operating system, vital system routines and most of the dynamic storage allocation routines were simulated in ALGOL 60. Since all tests were performed with the ALGOL 60 version, no measured figures can be given about rather important system characteristics, such as the access time for one file element. In order to give a fairly good estimate for the characteristic just mentioned, the routine *trans el* was coded in assembler language. It was concluded that on the X8, with disk drive(s) serving as mass storage and with the file system installation parameters, such as blocklength, set to some likely values, the access time for file elements read sequentially would be about 1 millisecond per element. This seems rather expensive for the X8 with a mean instruction time of 5 microseconds. It is accounted for by the great number of checks to be made at run time, due to the flexibility of the system.

A next step in the process of system development should be making the file system operative within the Milli system. For that purpose the following actions must be taken:

- incorporation of the dynamic storage module in Milli.
- extending Milli with routines for communication with disk drives.
- adapting the measures Milli takes upon program termination.
- coding the ALGOL 60 procedures of the file system into assembler code. This is not really needed since Milli disposes of an ALGOL 60 library, but it surely would make the system more efficient. Furthermore, in coded form the operating system itself can use the file routines in a way far more elegant and surveyable than in case they are in ALGOL 60.

The actions spoken of above were not taken after reaching the here presented state of affairs as one might have expected. They were not taken, because a new computer was about to arrive, meant to replace the X8. Since this new machine was to be equipped with a file system and since the days

of the old machine were numbered, it seemed a considerable waste of time and effort to make the system operative.

1.4. Report get-up

In this report different type fonts are used. Chapter 12 merely consists of the integral program text, which is given in lineprinter symbols. Parts of the program are reproduced elsewhere in this report and in some places programming language was used instead of English language if such seemed to increase clearness. Wherever program text is used, it is represented in italics, e.g. *begin*. At the introduction of a technical term, the new notion is given in *slope-writing*. File variables that have an equivalent program variable with a different identifier are apostrophed, e.g. 'spec' is a file variable that has as equivalent program variable (the location addressed as) *des - SPEC IN DES.*

Because the implementation was done in ALGOL 60, file actions are taken by calling the file system procedures. In the sequel it is preferred however to talk about "routines" that perform those actions, rather than "procedures", to emphasize the generality of the system presented.

The chapters 2 up to and including 8 all are structured in the same way, as is shown by the table of contents. It may be of service to the reader to know that this structuring is done according to the following schema (which is inspired by that of [4])

	2 3 4 5 6 7 8
	general information
.1	technical information
.2	implementation
.3	design considerations

2. GENERAL

A *file* is a virtual memory, consisting of consecutive *positions*, numbered in a natural way and defined from the *file begin*, the first selectable file position, up to and including the *file end*, the last selectable position. The natural numbers attached to the file positions can be considered to be the addresses of the locations of the virtual memory. It is emphasized that the user of a file in general will be completely ignorant about the physical resources called upon by the system to realize the file. Despite this allowed, and in a sense encouraged, ignorance about the exact where-abouts of his files, it might help the user in manipulating his files to have a vague notion about the implementation. Also it could have a favorable effect on the efficiency of file handling. In the present implementation the major part of a (large) file is on back store; a rather small part will be in main store. Nearly all of the inevitable file administration is kept in main store. In this report both the terms *back store* and disk are to be understood as a notion for some, so-called backing storage or mass storage device, preferably easy to access. Where the term *main store* or core is mentioned, some storage device easier to access than back store is meant, preferably central memory.

The contents of a file position are called a *file element*. All elements of a file are of the same species, the *file species*, which may be different for different files. It is the file species that dictates the size of a file position, i.e. the number of bits it takes.
The *file length*, i.e. the number of elements a file contains, is not a fixed value. The file is allowed to grow or shrink at user's command, though the existence of an upper bound is of perforce.
The *file claim*, i.e. the maximal number of elements a file can contain is not fixed either. If possible it will be enlarged automatically as soon as the user needs more elements than available.
Addressing a file element can be done only in an indirect way, via a pointer on that file. Such a pointer, a *file pointer*, thus plays a role of vital importance. The number of pointers on a file is not fixed, but left to the user of the file. Because more than one pointer on the file is al-

lowed, the files are called *multi-sequential*. The *value of a pointer* is the position the pointer points at.

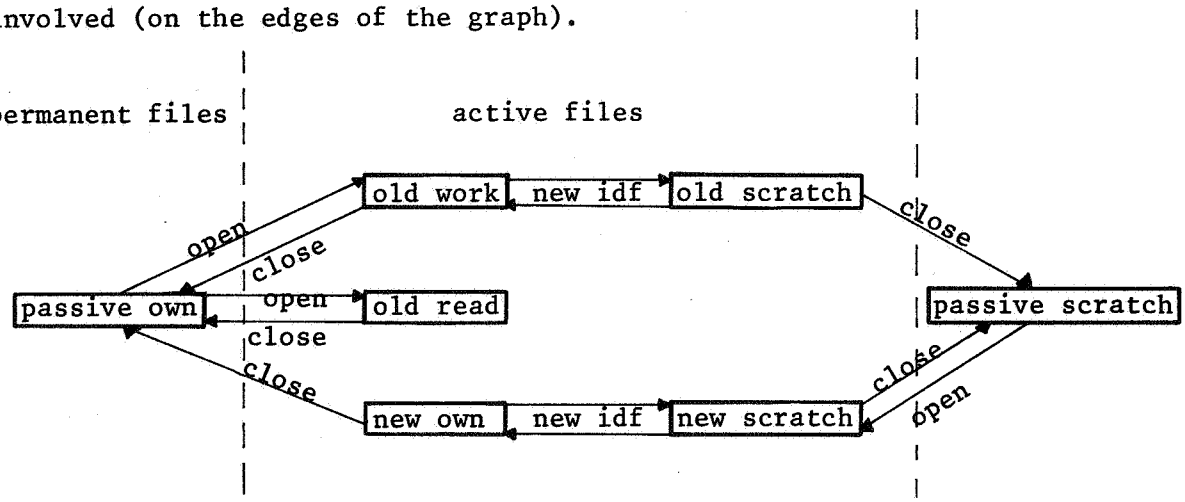
Once established a file can be preserved for some space of time, may be for ever. A preserved file is called *permanent*.
Any file carries a name, the *file name*, that serves two purposes:
. it allows identification of permanent files.
. it distinguishes between *scratch files*, which carry the scratch name, and *own files* which carry a name different from the scratch name.
In the present implementation the *scratch name* is identical to the empty name. The name of a permanent file cannot be the scratch name.

A file is called *active* if the user has access to it. If a file is not active, *inactive*, it is either an empty scratch file, only virtually existing, or a permanent file.
Active files can be classified according to the following criteria:
. a file is either an own file, i.e. a permanent file, or a scratch file.
. a file is either a work file or a read file. A file is called a *work file* if the user is allowed to change the file in whatever respect (the contents, its length, its name, etc.). File and user are said to have the *work permit*. If a file cannot be changed at all by its user, the file is called a *read file*.
. a file is either an old file or a new file. An *old file* is an activated permanent file. A *new file* is a file freshly created, possibly carrying a name different from the scratch name.
Activating a file is called *file opening*. Making a file inactive is called *file closing*.
Considering the three classification criteria: new/old, work/read, own/scratch, we are able to observe eight possible combinations of them. Only five actually exist. They are listed below, accompanied by the file class attached to them.

combination			file class		
new	scratch	work	*new*	*scratch*	*file*
new	own	work	*new*	*own*	*file*
old	scratch	work	*old*	*scratch*	*file*
old	own	work	*old*	*work*	*file*
old	own	read	*old*	*read*	*file*

new	scratch	read	"impossible"
new	own	read	
old	scratch	read	

The following digraph shows the connections and possible transitions between active and passive files (the nodes of the graph) and the routines involved (on the edges of the graph).



The user who created a permanent file is called the *owner* of the file. Permanent files can be changed by no one but their owner. The moment a permanent file is created, the file owner marks it as either public or private. *Private files* can be accessed only by their owners, whereas *public files* can be read by anyone, i.e. public files are multi-read files.

2.1. Technical information

Active files are identified not by their names, but by a *file number*, a positive integer distributed to the user by the file system when the file

is opened. File numbers are always denoted here as f in plain text and as *f* in program text. In the latter case "file *f*" means "file with file number f", where f equals the contents of the variable *f*.
As is pointed out in chapter 3, much the same holds for the integers that identify pointers, the *pointer names*. They are denoted here by p in plain text and by *p* in program text. The beginpointer, endpointer and workpointer are denoted by bp, ep and wp respectively. The value of pointer p is denoted by $\bar{p}$ and $\bar{p}$ denotes the value of the pointer the name of which is contained in the program variable *p*; $\overline{bp}$ stands for the first position of the file, etc.

Two types of errors can occur when using files:
- *fatal errors*. They produce some error message and cause job termination.
- *venial errors*. They do not produce an error message and the program is continued after being supplied with some information about the error detected. This passing of information always is done via the routine-identifier. If this routine is of type boolean, true means no error occurred and false means the only venial error possible was detected. If the routine is of type integer, a positive result means all was correct and a negative result means some venial error occurred. More information about the error will be contained in the value of the negative result.

Related with this approach is the fact that f, p and $\bar{p}$ always are positive integers.

2.2. General concepts of the implementation

Any file, passive or active, is structured as a logically consecutive row of blocks. Each *block* consists of a number of logically consecutive cells and each *cell* consists of one or more elements, depending upon the file species. In the present implementation a cell is realized in two consecutive machine words. A file block resides in back store or in main store. In both cases the cells contained in it are physically consecutive. All blocks of a permanent file are in back store. Those of an active file are either in main store or in back store, with an exception for read files: the blocks of a read file all are in back store and incarnations of them

may be existent in main store.

2.2.1. Main store management

Besides the info-space needed by an active file to accommodate file blocks in main store, administration space in main store is needed to keep all vital information about the active file. Reservation of info- and administration space, i.e. reservation of all main store required by a file, is not done statically, in a fixed region, nor in a run-time stack, but in another region, termed the *heap*, in which garbage collection techniques may be used for storage retrieval. (The term was borrowed from [10]). Claiming from the heap rather than from some other source is more or less demanded by the various dynamic file parameters such as length, number of pointers and even number of files.
The heap is controlled by the *dynamic storage allocation system*. This authority allows structured values on the heap, but they cannot be of great complexity. Therefore the file administration had to be "layered" in order to achieve the complexity needed.
The skeleton of the file administration is given below, in two versions:
. a description in terms of ALGOL 68 [10].
. a pictural approach in which the skeleton is linked to the program.

It was decided to give this skeleton with the emphasis shown, because this information seems of crucial importance to a proper understanding of large parts of the program.

```
mode  fad    = [1:0]flex ref cad;
struct cad   = (ref des des,ref sad sad,ref pad pad,ref bad bad,
                ref bic trans);
struct des   = (int spec,bp,ep,nsegm,offset,nbics,nptrs,nelpw,bpel,
                nblocks,nelpb,nfree,last block,catpos,transcor,nfreebics,
                bool new,scratch,work,segmad backad,string idf);
mode  sad    = [1:0 flex] sadcell;
struct sadcell = (segmad segmad, [1:nbps] bool bitwrd);
mode  pad    = [1:0 flex] padcell;
```

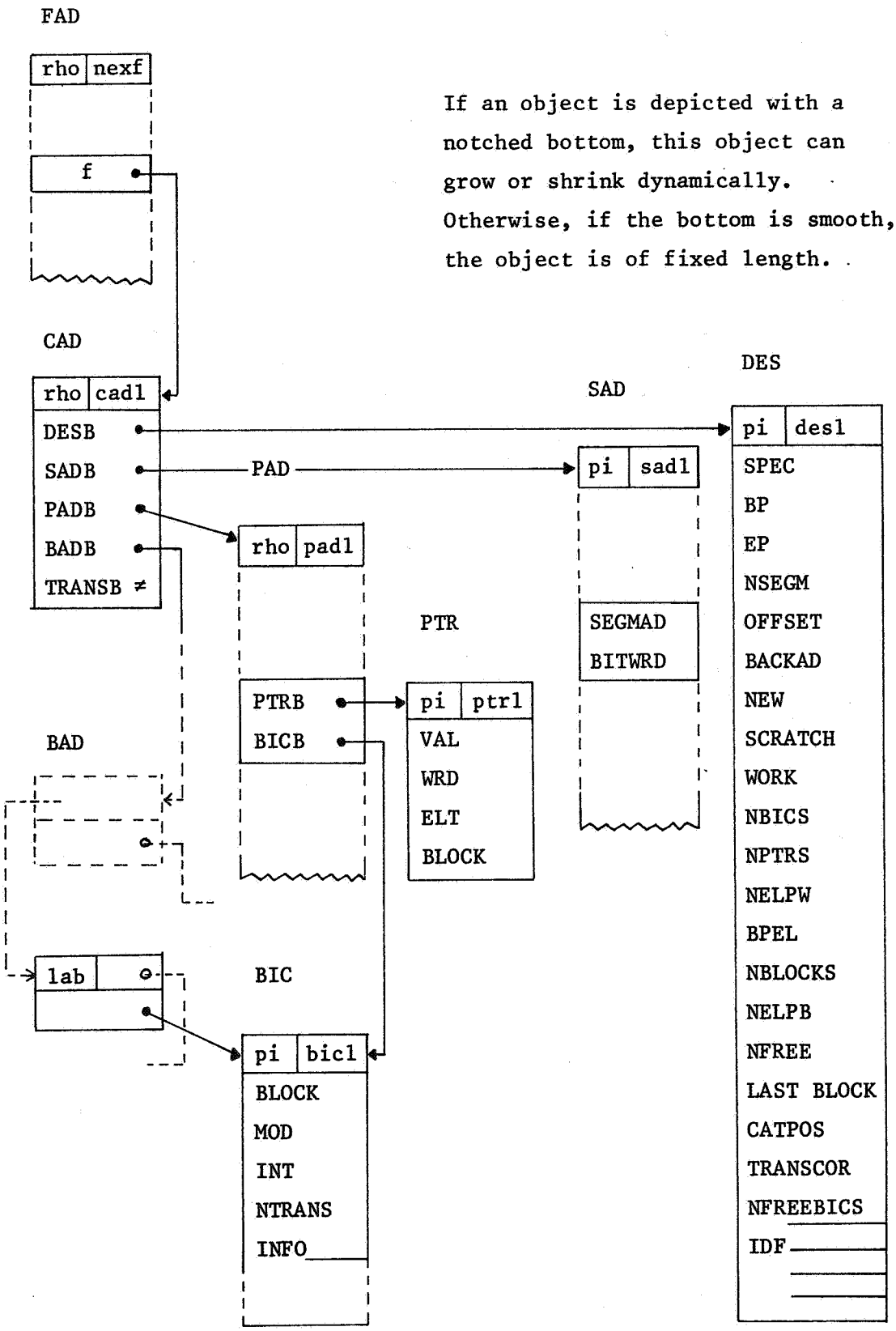
```
struct padcell = (ref ptr ptr,ref bic bic);
struct ptr     = (int val,wrd,elt,block);
struct bic     = (int block,int,ntrans,bool mod,info info);
mode   info    = [1:infol * nelpw] elt;
struct bad     = (ref bad next,ref bic bic);
mode   segmad  = [1:nbps] blockad;

mode   blockad = c an actual-declarer specifying the mode of a file block
                   on back store c;
mode   elt     = c an actual-declarer specifying the mode of a file ele-
                   ment. This mode depends on the file species c;
```

Access to a file is obtained through an index for a multiple value of mode *fad*. This index is called the file number. So:

```
mode   file    = int;
```

As said before, the actual form of the file administration depends on the information structures allowed by the dynamic storage allocation system. This system allows:

```
mode   pi      = [1:0 flex] int;
mode   rho     = [1:0 flex] ref str;
struct lab     = (ref lab lab,ref str str);
union  str     = (pi,lab,rho);
```



If an object is depicted with a notched bottom, this object can grow or shrink dynamically. Otherwise, if the bottom is smooth, the object is of fixed length.

The administration can be split up into the following parts:
. a general file administration (per job), *fad*.
. a segment administration (per file), *sad*.
. a pointer administration (per file), *pad*.
. a block-in-core administration (per file), *bad*.
. a core descriptor (per file), *des*.

These parts will be discussed in some detail in the next sections.

2.2.1.1. Fad

If a program uses the file system it will have a fad, otherwise it will not. Each active file of the program has exactly 1 entry in fad, fad being a table of references. This entry yields the reference to the rest, i.e. nearly all, of the administration of the file. Entering fad is done via the file number. As soon as the program tries to activate its first file, fad is claimed, from the heap, and initialized. Activating a file means, among other things, selecting a free entry in fad, i.e. picking a file number; if no such entry is available fad is extended.

2.2.1.2. Sad

Before explaining the segment administration, the notion segment is introduced. It is evident that the choice of how to perform the bookkeeping of the back store is strongly related to the choice of how to do back store distribution. Both choices are made, simultaneously, in the next section. And it is there that the notion segment pops up.

2.2.1.2.1. Segments and blocks

File blocks remain either in main store or in back store. In the latter case their addresses, *back addresses*, in general are not consecutive. So we need some administration to be able to get hold of the back address of any particular block, any particular time. Four ways of performing this administration are discussed below and one is selected.

A. A very simple way to do the bookkeeping is to link all blocks in a linear (singly- or multi-) linked list, e.g. each block indicates its successor (or/and predecessor). Such a method will be inefficient if some form of random access to the file is allowed. Since this is the case in the file system discussed, method A is rejected.

B. The administration has the form of a table, having an entry for each block and containing its physical address. For a large file with relatively small blocks such a table will be of formidable proportion. It will be so big, it has to be accommodated in a file - a file of block addresses. This file needs block administration, and so on. Indeed it is possible to realize this kind of administration, but in the present implementation it is rejected because a simpler form could be used.
N.B. The solution just ruled out, suits fine the implementation of so-called "sparse files" - files containing large gaps, maybe 0 by default. For such files very little space, say 1 word, is needed to describe a whole block as empty (non-existent, zero). At the cost of multi-stage-addressing of the blocks sparse files thus can be stored in a very compact way.

C. A combination of the methods A and B: a number of logically consecutive blocks form a *segment*. The segment address is to be found in a table and within the segment the blocks are linked in some way. This seems o.k. but we can still do better by taking advantage of the specific qualities of the application in the file system.

D. A practical variant of C: the segment address resides in a table. The table contains the address of the first block of the segment. The locations of the other blocks within the segment are found easily, because we choose all blocks in a segment to be physically consecutive. That is, on back store. This way of administration is selected for the file system. It can be described in short as "a form of indexed sequential addressing".

If a block resides in main store its exact location is provided by a linked list in core. This list cannot possibly grow beyond reasonable pro-

portions since it deals with core blocks, and only relatively few blocks are allowed in core simultaneously. Locations of core blocks are found through a linear search in the list mentioned.

Whether a block is in core or on back store is told by one bit especially reserved for each block. These bits form a bit table, that in the implementation is split up and distributed over the segment-address-table in a convenient way.

2.2.1.3. Pad

The pointer administration consists of a table of padcells. Each pointer is connected with a padcell, which can be selected from pad by the pointer name. When an own pointer is created a free cell is selected from pad, i.e. a pointer name is picked. If such a free cell is not available pad is extended. Standard pointers have fixed entries in pad.
A *padcell* consists of two items: a reference to a bic and a reference to a ptr. More information about the bic will be given in the next section.
The *ptr* consists of four plain variables:

'val' - the value of the pointer.

'block' -)
'wrd' -) The 'elt'-th element in the 'wrd'-th word of the
'elt' -) 'block'-th block of the file is the element in the 'val'-th position of that file.

2.2.1.4. Bad

Bic stands for "space-for-a-block-in-core". How a bic is structured and made use of is explained later on. First the administration of bics is dealt with.
Bad is a "normal", one-way-linked list, containing as information no more than references to bics. Each time a cell has to be added to the list, a next cell is claimed from - and each time a cell becomes obsolete, it is returned to - the heap. Bad is a list of all bics, a list that is consulted whenever a bic has to be located that is not necessarily connected with some pointer. The routines *free semifree*, *try bic release* and *ask for block*

all essentially are based upon the existence of bad.

2.2.1.4.1. On bics

A *bic* consists of space to contain a file block and of space to keep the bic administration. This administration space takes four variables:

'block' - the block number of the file block contained in the bic. If the bic does not contain any block (yet) the bic is said to be a *free bic* and its 'block' will be a negative value.

'mod' - a boolean value, giving information about the block being modified or not. If 'mod' > 0 then the block is modified by the user: it is not, or no longer, a copy of a block on back store, so, sooner or later the block must be transported to the back store.

'int' - a counter, indicating the number of pointers that are interested in the block contained in the bic. The values of those pointers all select an element of this block. If 'int' = 0 no pointer is interested in the block and if at the same time 'mod' < 0 we call the block *passive*.

'ntrans'- a counter, indicating the number of transports required for the block and not yet completed. Usually 'ntrans' will be 0 or 1 but under circumstances it can become more than 1. If 'ntrans' = 0 then the bic is said to be *at rest*.

Overall administration about the bics is partly conducted in the list bad, and partly in some variables of the core descriptor des. These variables are 'nbics' and 'nfreebics'. Further descriptor variables involved are 'nptrs' and 'last block'.
An active file always has at least as many bics as it has pointers, with a lower bound of 1. This guarantees:

. once a pointer is established, moving this pointer along the file never will cause fatal errors due to space exhaustion.

. in all situations at least one active pointer can be attached to the file without causing such errors.

To give an impression of bic-handling, a bird's-eye view of the most important system routines dealing with bics is given below.

integer procedure claim bic (f,al); value f; integer f; label al;

function: Heap space is claimed for a bic of file *f*. If this space is not available, control is transferred to *al*, otherwise the bic address is delivered. The bic is initialized as free, unmodified, passive and at rest. Its address is added to the list bad; 'nbics' and 'nfreebics' both are increased by 1.

integer procedure claim free bic (f); value f; integer f;

function: This routine is called only if 'nfreebics' is at least 1. The list bad is scanned until a free bic is found, the address of this bic is delivered and 'nfreebics' is decreased by 1.

procedure try bic release (f); value f; integer f;

function: All bics of file *f* are inspected by scanning the list bad. A bic that is not, and never more can become, of any importance to the file is returned to the heap, unless that bic is modified or not at rest. If it is modified a transport is asked for, so the status of the bic becomes not-at-rest. Bics not at rest are ignored and taken care of the next time the routine is called. Returning bics to the heap is stopped as soon as all bics are inspected, or in an earlier stage: it is guaranteed that at least "the maximum of 'nptrs' and 1" bics will stay connected with the file.

procedure decr int in bic (f,p); value f,p; integer f,p;

function: Pointer *p* of file *f* looses its interest in the bic it is connected with. The bic address is taken from the table pad and replaced by the reference nil. The 'int' of the bic is decreased by 1 and if it becomes 0 the next steps are carried out:

if 'mod' > 0 then a transport to back store is asked for, otherwise if *p* is the beginpointer or the endpointer the bic is discarded through a call of *try bic release*.

procedure ask for block (f,p); value f,p; integer f,p;

function: The block connected with pointer *p* of file *f* is wanted in core. The bic containing that block is to be linked to that pointer via pad. If the block already is in core, the relevant bic is selected from bad, otherwise a free bic is picked from it and the transport from back store to bic is started.

2.2.1.5. Des

Des is a descriptor, the core descriptor of a file. It contains an overall administration of it, consisting of some variables, global to the file. The descriptor is of fixed length. Its variables, together with a short description of each one of them are listed below.

'spec' - the file species. In the present implementation this is an integer value meaning the number of elements per word, with the exception that 0 means 1 element per 2 words. So, relating the value of the species to a type could be done as follows (supposing the word to be an X8 word, i.e. a word of 27 bits):

0 - real	2 - card column
1 - integer	6 - plotsym
27 - boolean	
3 - char	

'bp' - the value of the beginpointer.

'ep' - the value of the endpointer.

'nsegm' - the number of segments assigned to the file.

'offset' - an auxiliary variable of block administration.

'backad' - the back address of the long descriptor of the file if the file is not a scratch file. Otherwise the value is undefined.

'new' - a boolean variable telling whether the file is new or old.

'scratch' - a boolean variable telling whether the file is scratch or own.

'work' - a boolean variable telling whether the file is work or read.

'nbics' - the number of bics attached to the file. All these bics are kept trail of in the list bad.

'nptrs' - the number of active pointers of the file.

'nelpw' - the number of elements per word. In this implementation this variable is a copy of 'spec'.

'bpel' - the number of bits per element. This is calculated directly from the number of bits per word and 'nelpw'. This calculation is done only once, in order to increase the speed of the access routines.

'nblocks' - the number of blocks that contain information of this file.

'nelpb' - the number of elements per block, easily derived from the number of words per block and 'nelpw'.

'nfree' - the number of free positions of the file, i.e. the maximal number of elements that can be written into the file without intermediate destructive read, unstack or file space extension.

'last block' - the number of the block pointed at by the endpointer.

'catpos' - the position of the short descriptor of the file in the catfile, if the file is an old read file, otherwise the value is undefined.

'transcor' - an auxiliary variable of file space extension.

'nfreebics' - the number of free bics of the file.

'idf' - a consecutive number of integer variables, each of them containing one or more characters. Together they form a representation of the file name. In the present implementation each word contains three characters.

2.2.2. Back store management

The back store is divided in a scratch pool and in an own pool. This division is of an administrative kind and need not be a physical one. The *scratch pool* consists of scratch segments only; the *own pool* consists of own segments and of own blocks as well. These blocks are claimed one at a time and do not serve to keep file elements, but they keep information about the file, as will be pointed out in the next section.

Permanent files are stored in back store. The file catalogue is to comprise all information about those files, needed to enable the system activating a specific own file. This information is not static of nature. It will vary in length and in meaning as well. Because of this dynamic character the

file catalogue is accommodated in a file, the *catalogue file*.

2.2.2.1. The file catalogue

The *catfile*, short for catalogue file is the only file that can be called a system's file. It diverges from a normal file in no more than a tiny detail: some of the information about the catfile is kept in main store, instead of in the catfile.
Each own file is represented in the catfile by a *short descriptor*, giving details about the file name, owner, use and where to find additional information (the back address of the long descriptor):

short descriptor:

file name	use	owner	backad of long descriptor block

The variable 'use' tells something about the momentary use of the file:
'use' < 0 - the file is used as work file.
'use' = 0 - the file is inactive.
'use' > 0 - the file is used as read file by 'use' users.

The *long descriptor* of a file is a block in back store, giving details about some file parameters, namely 'spec', 'bp', 'ep', 'nsegm' and 'offset'. Furthermore it contains the segment administration, in other words, tells where file segments are to be found in back store. The difference between this administration and the one in main core (sad) is that sad contains an additional bit table for the blocks.
The tiny difference between the catfile and an own file is that the catfile is not represented in the catfile by a short descriptor. Its long descriptor resides in a back block with a location well known to the system.

The catfile capacity will always be sufficient; it is guaranteed that never an error occurs due to catfile exhaustion. This can be guaranteed because of the very small size of the short descriptor.

Since the long descriptor contains all segment addresses and is of the same

size as file blocks, a file cannot consist of more than (block length - 5) segments. This upper bound can very easy be raised if a long descriptor is allowed to be continued in one or more extra blocks. However, this raising seemed unnecessary in the present implementation because of the following calculation. For reasons of X8-disk limitations, the block length is decided to be 260 words. Therefore the long descriptor is able to contain the administration of 255 segments. A segment is to contain 10400 words in this case, so the maximal file length that can be realized in the present implementation is 2652K words which seems a very reasonable upper bound.

2.2.2.2. Catfile core space

The minimal amount of core space needed by an active catfile equals, of course, the minimal amount of core space needed by any active file. Straightforward implementation of opening an own file therefore would require twice as much space as would be required once the opening were done. Furthermore, straightforward implementation of closing an own file would require extra space at the moment of closing. In the worst case this space might be unavailable, so preserving the particular file would be out of the question; a most inconvenient situation. The obvious solution, to reserve space for the catfile permanently is rejected, because it takes up more space than really necessary. The answer to the space problem lies in the fact that the file being opened or closed is in fact inactive, part of the time. The programming "trick" performed in both opening and closing allows the catfile administration to be accommodated in the file space of the file to be opened or closed (the file then being inactive).

2.2.2.3. Catfile routines

Some of the routines concerned with the catfile are mentioned and discussed below.

integer procedure look up (f,idf); value f; integer f; array idf;

f is the file number of the file that accommodates the catfile, so it can be regarded as the file number of the catfile itself. The work-

pointer of the file is assumed to be active.
The catfile is scanned from the position indicated by the workpointer till the end of the file. As soon as a short descriptor containing the file name *idf* is detected the scanning is stopped and a catfile position is delivered. This position indicates the rest of the information (beyond the file name) comprised in the selected short descriptor. If, on the other hand, the file name *idf* does not occur in the catfile, then a negative value is delivered.

integer procedure pos in catfile (f,idf,work,user); value f;work,user; integer f;user; boolean work; array idf;

Again *f* is the file number of the catfile. The workpointer on *f* is assumed to be active, its value 1. By means of a call of *look up* some file of name *idf* is traced (the first one to appear in the catfile). If the user *user* appears to be the owner of this file then the file wanted is found, otherwise the selected file is kept in mind and search is continued by another call of *look up*, and so on. In the end one of the following situations is achieved:

. no file of name *idf* is present in the catfile and the negative value ER UK is delivered.

. no file of name *idf*, owned by *user*, is present in the catfile, but one or more files of name *idf* belonging to other owners are there. These files were kept in mind during the search. If none of these files is a public file then the negative value ER NY is delivered. Otherwise the public file of name *idf* is looked at more closely: if *work*, i.e. the file is wanted as work file, then the negative value ER NP is delivered, otherwise if the file is wanted as read file it is possible that the public file is involved in some updating process performed by its owner. In that case the negative value ER NN is delivered. The one possibility left - the file is a public file, not used by someone else as work file, and is wanted as read file - yields a positive value: the position in the catfile of the rest of the selected short descriptor.

. a file of name *idf*, owned by *user*, is present in the catfile. If

work and the file is a public one being read by someone (or being updated by the user himself) then the negative value ER NN is delivered. Otherwise this routine gives as result a positive value: the position in the catfile of the rest of the information contained in the selected short descriptor.

boolean procedure update catfile (f,idf,des,private,work,new,scratch);
value f,des,private,work,new,scratch; integer f,des;
boolean private,work,new,scratch; array idf;

Again *f* is the file number of the catfile. The parameter *des* gives the address of the core descriptor of *f*. The file of the present user with file name *idf* is about to be closed as (if *private* then private- else public-) file, so the catfile has to be updated. Additional information about the file is given in the self evident boolean parameters *work*, *new* and *scratch*.

In case the file is a scratch file or a read file the updating is trivial and not discussed further at this place. If the file is an old file, its position (of the short descriptor) in the catfile is taken from the core descriptor and the short descriptor in the catfile is erased, i.e. replaced by a scratch descriptor. Both old and new own files to be recorded in the catfile now find themselves in exactly the same situation. First the catfile is scanned for other files of the same name, to check upon possible ambiguities that would arise from adding the new name to the catalogue. This is done through a call of *pos in catfile*. If trouble occurs, appropriate measures are taken, such as giving the file some other name. The ambiguity matter settled, the new short descriptor is added to the catfile, replacing the first scratch short descriptor to be found. If no such "gap" exists the catfile is enlarged through writing elements via the end-pointer instead of via the workpointer. The value true is delivered if no new name had to be given to the file, the value false otherwise.

2.3. General design considerations

2.3.1. File length and file extension

It was considered desirable that, as soon as a user should reach his maximal file length, i.e. exhaust his file claim, the file be extended automatically. This extension is easily described from the user's point of view; it means the file claim is enlarged. Its implementation is not that simple, because of the cyclic block administration that is chosen to comfort the sort of file usage that stacks and reads destructive alternately. Apart from this complication, dealt with by the routine *extend file*, another matter had to be settled. Namely, how to inform the user in case file extension cannot be managed because of file space exhaustion (on the physical devices). To obtain the possibility of a venial error - the fatal one of course means no trouble at all - file extension is done in the following manner. In case *write el* meets the situation that after writing the element the file is filled completely it tries to extend the file. If this is impossible at the moment, no further actions are taken. The user can observe that the file length equals the file claim and conclude that file extension is not possible, so, without special measures (such as the deletion of another file), the next call of *write el* for that file will cause a fatal error.

In the present implementation the initial claim of a file is one segment. File extension adds one segment to the claim each time it is successfully tried.

3. POINTER ROUTINES

Two kinds of file pointers exist: standard pointers and own pointers. A pointer is identified by its name, the pointer name, a (small) positive integer value. *Standard pointers* have standard names - they are identified by integers chosen once and for all, known to the system and all users. *Own pointers* are created as a private enterprise of the user. Their names are invented by the system at the moment of pointer creation and passed on to the user.
A pointer is said to be *active* if a file access can be achieved via that pointer. All own pointers are active during their entire existence, but standard pointers may be inactive and still in existence.

Pointer activation, i.e. activating a standard pointer or creating an own one, will seize a part of main storage for reasons of administration. If this amount of space is not available the pointer activation will not get any further than the attempt. Whether a pointer can be activated or not depends on the actual core distribution. However, it is guaranteed that any active file can have at least one active pointer.
If a pointer is *deleted* the occupied space in main storage will be released. If the pointer deleted is an own pointer then the pointer ceases to exist. If it is a standard pointer it survives as an inactive pointer that does not take up considerable amounts of core space.

Three standard pointers exist:

- the *beginpointer* bp, always pointing at the first file position, the filebegin. Its name, 1, is fixed in and obtainable from the systems constant BP. The pointer will never be active in a spontaneous way: if needed it must be activated explicitly by the user.
- the *endpointer* ep, always pointing at the post-last file position, the (in fact non-existing) position "file end + 1". Its name, 2, is fixed in and obtainable from the systems constant EP. The pointer will be active directly after file creation or, of course, an explicit activation.
- the *workpointer* wp. Its name, 3, is fixed in and obtainable from the systems constant WP. The pointer will be active and pointing at the

first file position, directly after opening an old file or, of course, after explicit activation.

3.1. The routines

3.1.1. *procedure standard ptr (f,p); value f,p; integer f,p;*

fatal errors: ER WF - *f* is not a file.
ER ST - *p* is not a standard pointer.
ER RE - *p* is a standard pointer already active.
ER CE - core space exhausted.

function: The standard pointer *p* is activated. If *p* is the work-pointer its value is set to that of the beginpointer. Otherwise *p* has to be the beginpointer or the endpointer and already possesses a value - which it keeps.

3.1.2. *boolean procedure standard ptr 1 (f,p); value f,p; integer f,p;*

fatal errors: ER WF - *f* is not a file.
ER ST - *p* is not a standard pointer.
ER RE - *p* is a standard pointer already active.

venial error: (ER CE) - core space exhausted.

function: As standard ptr.
If a venial error occurs then the value false is delivered, true otherwise.

3.1.3. *integer procedure new ptr (f,pos); value f,pos; integer f,pos;*

fatal errors: ER WF - *f* is not a file.
ER PO - *pos* is not a position within file *f*:
$pos < \overline{bp}$ or $pos \geq \overline{ep}$.
ER CE - core space exhausted.

function: a new, own pointer is created and activated. Its initial value is set to *pos*. The pointer name, a positive integer value,

is delivered.

remark: As a consequence of the test on the fatal error ER PO it is not possible to create an own pointer on an empty file.

3.1.4. *integer procedure new ptr 1 (f,pos); value f,pos; integer f,pos;*

fatal errors: ER WF - *f* is not a file.
ER PO - *pos* is not a position within file :
pos < $\overline{\text{bp}}$ or *pos* ≥ $\overline{\text{ep}}$.

venial error: (ER CE) - core space exhausted.

function: As new ptr.
If a venial error occurs a negative value is delivered.

remark: As new ptr.

3.1.5. *procedure delete ptr (f,p); value f,p; integer f,p;*

fatal errors: ER WF - *f* is not a file.
ER WP - *p* is not a pointer of file .

function: If *p* is an active standard pointer on *f* then it is made inactive and if *p* is an own pointer it is deleted. In both cases the pointer no longer can serve as a means to access the file. The core space seized by the pointer *p* is released.

3.1.6. *procedure reset wp (f); value f; integer f;*

fatal errors: ER WF - *f* is not a file.
ER WP - the workpointer of file *f* is not active.

function: The value of the active workpointer is set to that of the beginpointer.

3.2. Implementation

The pointer routines are implemented in a way that does not need very

much explanation. They merely consist of the claiming and returning of main store.

```
3.2.1.1. procedure standard ptr (f,p); value f,p; integer f,p;
                    if ¬ standard ptr 1 (f,p) then error (ER CE);

3.2.1.2. integer procedure new ptr (f,pos); value f,pos; integer f,pos;
         begin integer p; new ptr:= p:= new ptr 1 (f,pos);
               if p < 0 then error (ER CE)
         end;

3.2.2. boolean procedure standard ptr 1 (f,p); value f,p; integer f,p;

begin hard check on f (f);
      if p ≠ BP ∧ p ≠ WP ∧ p ≠ EP then error (ER ST); comment p must de-
                                                      note a standard pointer;
      if ptrb (f,p) ≠ 0 then error (ER RE); comment standard pointer p has
                                                      to be inactive;
      SYS not;
      create ptr space (f,p,no); comment administration space for the
                                 standard pointer p is claimed, such as ptr and bic.
                                 If the space required is not available, control is
                                 transferred to no;
      initialize ptr (f,p,val of ptr(f,if p = WP then BP else p));
                 comment the ptr of the pointer p is initialized. If p de-
                 notes the endpointer, the pointer value will be set to the
                 post-last position of the file, otherwise it will be set to
                 the first position. Furthermore the file block connected
                 with p is assured to be in core;
      if true then standard ptr 1:= true else
no:   standard ptr 1:= false;
      SYS ton
end standard ptr 1;
```

3.2.3. *integer procedure new ptr 1 (f,pos); value f,pos; integer f,pos;*

```
begin integer p; hard check on f (f); hard check on pos (f,pos);
        comment if pos < bp or pos ≥ ep then a fatal error is re-
                                                          ported;
    SYS not;
    p:= new ptr number (f,no); comment a padcell is claimed from pad.
        If not available, pad is extended. If this extension is not
        possible due to core space exhaustion, a transfer of control
        to no is performed;
    create ptr space (f,p,no); comment see explanation in 3.2.2.;
    initialize ptr (f,p,pos); comment the ptr of pointer p is initia-
        lized. The value of the pointer is set to pos. The file block
        that is connected with the pointer p is assured to be in core;
    if true then new ptr 1:= p else
no: new ptr 1:= ER CE;
    SYS ton
end new ptr 1;
```

3.2.4. *procedure delete ptr (f,p); value f,p; integer f,p;*

```
begin integer adp,adn; hard check on f and p (f,p);
    adn:= desb (f) - NPTRS IN DES;
    adp:= padb (f) - (p-1) * PADCELL - PTRB IN PAD;
    SYS not;
    incr (adn,-1); comment decrease 'nptrs' by 1;
    decr int in bic (f,p); comment see 2.2.1.4.1.;
    if fetch (adn) > 0 then delete core (adp) else
    begin comment core space required by one pointer is preserved, even
            if all pointers of the file are deleted;
        store ref (cadb(f) - TRANSB IN CAD, fetch ref(adp));
        comment 'transb' serves as temporarily depository for the
            reference to the preserved pointer space;
        store ref (adp,0)
    end;
```

```
        SYS ton
end delete ptr;
```

3.2.5. *procedure reset wp (f); value f; integer f;*

```
begin hard check on f and p (f,WP);
        SYS not;
        decr int in bic (f,WP); comment see 2.2.1.4.1.;
        initialize ptr (f,WP,val of ptr(f,BP)); comment the ptr of the work-
                pointer is initialized. The value of the workpointer is
                reset to the first position of the file. Furthermore the
                block connected with that position is assured to be in core;
        SYS ton
end reset wp;
```

3.3. Pointer resetting

A natural routine not present in the file system would have been:

procedure set ptr (f,p) to position: (k);

function: set the value of pointer p of file f to position k.

This routine is omitted because it would encourage the user to have random access to a file quite easy and very inefficient. Discouraged users who still need such an operation can always help themselves via a detour over *delete ptr* and *new ptr*.
The routine *reset wp* is of the type *set ptr*. It is selected because it performs the operation "rewind", undoubtly indispensable.

4. ACCESS ROUTINES

Accessing a file, i.e. reading an element from it or writing an element into it, is done via a pointer on that file. All access actions have a side-effect on the value of the pointer that is involved. This value will always be increased or decreased by 1 when reading or writing an element. The three actions that will be discussed are named after their pointer effect: *write forward*, *read forward* and *read backward*.

. Writing forward an element via pointer p can be done by calling the routine *write el*. The element is written into position $\bar{p}$ of the file. Writing forward via the endpointer is called *stacking* and causes file growth.

. Reading forward via pointer p can be done by calling the routine *next el*. The element in position $\bar{p}$ of the file is delivered. Reading forward via the beginpointer is called *reading destructively* and causes file diminution.

. Reading backward via pointer p can be done by calling the routine *prev el*. The element in position $\bar{p}-1$ of the file is delivered. Reading backward via the endpointer is called *unstacking* and causes file diminution.

Repeatedly stacking without a countering unstacking or reading destructively causes continued file growth, so, one moment the file claim will become insufficient. The system will take steps to prevent fatal errors resulting from file overflow; the system tries to enlarge the file claim in time.

4.1. The routines

4.1.1. *procedure write el (f,p,el); value f,p,el; integer f,p; real el;*

fatal errors: ER WF - *f* is not a file.
ER WP - *p* is not a pointer of file *f*.
ER NW - the file is not a workfile.

ER PL - p points below the beginpointer ($\bar{p} < \overline{bp}$)
ER PH - p is not the endpointer and p points above the file end ($\bar{p} \geq \overline{ep}$ & $p \neq$ ep).
ER FE - the file is filled to capacity and can not be enlarged, due to filespace-exhaustion.

function: 1. If the file is filled to capacity an attempt is made to enlarge the file claim. If this attempt is not succesful a fatal errormessage is given.
2. If the size of *el* does not match the filespecies of f, *el* is made to measure by cutting the most significant bits of *el*.
3. the (possibly truncated) element *el* is written into the file on position $\bar{p}$.
4. The value of p is increased by 1.
5. If the file is filled completely, i.e. file claim = $\overline{ep} - \overline{bp}$, then an attempt is made to enlarge the file claim. The result of this attempt is not of immediate importance to the system. (However, it can be to the user, see 2.3.2.)

remark: in case p is the beginpointer of the file, the file shrinks at the front and the element just written is inaccessible ever after.

4.1.2. *real procedure next el (f,p); value f,p; integer f,p;*

fatal errors: ER WF - f is not a file.
ER WP - p is not a pointer of file f.
ER PL - p points below the beginpointer ($\bar{p} < \overline{bp}$).
ER PH - p points above the file end ($\bar{p} \geq \overline{ep}$).

function: 1. The element on position $\bar{p}$ of file f is delivered (through the procedure identifier).
2. The value of pointer p is increased by 1.

4.1.3. *real procedure prev el (f,p); value f,p; integer f,p;*

fatal errors: ER WF - f is not a file.
ER WP - p is not a pointer of file f.
ER PL - p points below the second element ($\bar{p} \leq \overline{bp}$).
ER PH - p points above the endpointer ($\bar{p} > \overline{ep}$).

function: 1. The value of pointer p is decreased by 1.
2. The element on position $\bar{p}$ of the file is delivered (through the procedure identifier).

remark: pointer p always points to the element next to the element to be delivered.

4.2. Implementation

The three routines run very much along the same lines so their bodies are swept together and form the routine *trans el*, the kernel of the file system.

4.2.1.1. *procedure write el (f,p,el); trans el (f,p,1,el);*

4.2.1.2. *real procedure next el (f,p); next el:= trans el (f,p,2,0);*

4.2.1.3. *real procedure prev el (f,p); prev el:= trans el (f,p,3,0);*

4.2.2. *Trans el.*

This routine performs all kinds of accesses to a file and in order to get an impression of its degree of complexity it is programmed in a form of about the lowest degree of complexity possible. Dummy labels serve to regain readability. It was felt useful also with respect to clearness to use bit manipulation procedures [7] for fumbling elements in and out of the file cells. The routine does not need much functional elucidation. Each time a file element is accessed, the pointer value involved is stepped up or down, depending on the direction of the operation. This stepping the pointer value influences element count, possibly word count or even block number count. In the latter case special care is taken to evoke the "next block". This is done through a call of *ask for block*. As soon as the new

block is referenced, its presence is assured through a call of *assure presence block*. In case of a forward operation this is done at the moment the next element is wanted (for read or write); in case of a backward operation it is done immediately. Moving the beginpointer (in forward direction) causes special actions such as a possible change of 'offset' through a call of *consider offset*. This is done because the file shrinks at the front. If the endpointer is moved in backward direction the file shrinking is administrated via a call of *one block down*.

4.3. Designing the access actions

Sequential use of a file is extorted more or less from the user. He can access a file in just one way, via a pointer, and the number of pointers at his disposal will be relatively small (as opposed to the number of file elements). Moreover it is not easy-and-free to change the value of a pointer to an arbitrary position of the file. Now that haphazard accesses to a file are ruled out, to emphasize the sequential nature of the files, we focus on typical sequential operations to be offered to the user. They must be intrinsically sequential, that is, couple an access with stepping up or down the pointer involved. Furthermore efficiency should be guaranteed. Having in mind a file that can grow and shrink at both head and tail, four actions at least must be available, all having a side-effect on the value of the pointer: it is increased or decreased by 1. The routines are named after their effect on the pointer:

WF = write element in forward direction.
WB = write element in backward direction.
RF = read element in forward direction.
RB = read element in backward direction.

Implementation can be done in a number of ways, so, before proceeding three desiderata are stated:

i. The actions must be clear to the user (almost at once, at first sight) All of them should be easy to remember without the possibility of serious confusion between some of them.

ii. The whole pack of operations should be consistent and work together,

gear into each other, in a direct and obvious way.

iii. The actions should be easy and efficient to realize, to implement within the system, because here we have to do with the most elementary file actions.

If each action consists of: write (or read) an element and increase (or decrease) the pointer, or the other way round, then surely i. and iii. will be satisfied. So we concentrate upon ii.

In the sequel the following short-hand-notation will be used:

x : write an element , o : read an element ,

+ : increase pointer , - : decrease pointer .

An example: x+ stands for "write an element and afterwards increase the value of the pointer involved by 1".

The notation serves both as external description and as way of implementation of an operation. So, in the example x+ is to be conceived as both the description and implementation of the operation WF.

The four operations mentioned above now can be implemented in 16 different ways by using all combinations of the given primitives. We are looking for one that is in accordance with ii.

Let us choose the implementation x+ for WF and proceed from this choice.

A reasonable consequence of ii. can be described as follows:

"WF(a); WF(b); x:= RB; y:= RB" should yield the result "x=a, y=b". In other words, WF and RB should act as stacking and unstacking actions. This leads to the choice of -o for RB. Now by ii.:

"x:= RB; y:= RB; yy:= RF; xx:= RF" should yield "xx=x, yy=y". This inevitably suggests the choice of o+ for RF. By ii.:

"WB(a); WB(b); x:= RF; y:= RF" should yield "x=b, y=a" (reverse stack mechanism). So we choose -x for WB. Summarizing the results we see that the one acceptable set of operations starting with the choice x+ is:

WF = x+ , WB = -x , RF = o+ , RB = -o .

In case the initial choice were +x, the result would be, of course:

WF = +x , WB = x- , RF = +o , RB = o- .

We have now dealt with all 16 possibilities, leaving two acceptable ones. These two implementations are equivalent (the mirror image of each other) and the first one mentioned is adopted.

We have defined four elementary actions, all having side effects on the pointer involved and we wish to add some operations that do not influence the pointer, like:
W = write an element without changing the pointer value.
R = read an element without changing the pointer value.

Again we want the set of operations to be in agreement with i., ii. and iii. In the end it appears that this is not possible, so, in the end the two operations just mentioned are rejected.
We consider two evident ways to satisfy i.

A. The operation (e.g. R) has to do with the element most recently involved with the pointer via which the operation is done. We do not worry about difficulties of "initial kind". Assume the sequence "WF(a); WF(b)" to have taken place via the pointer observed immediately before we start our investigation:

actions	results wanted (demanded by ii.)	
RB RB	b	a
R R	b	b
RB R	b	b
R RB	b	?

If we choose R not to change the value of the pointer at all, then ? has to be b and so "R; RB; R; RB" should yield b, b, b, a. Clearly ii. is violated this way.
If we choose R to change the pointer value in convenient situations, the implementation will not be simple at all, thus violating iii.
Taking into account the operation W and making combinations of all operations allowed, thus increasing complexity, it soon becomes clear that it is not possible to serve both ii. and iii.

B. The operation (e.g. R) has to do with the element pointed at by the pointer involved. The thus created set of actions gives the illusion of being very symmetrical, but they act, on the contrary, highly asymmetrical, in a way that violates ii. This is shown in the next two examples, where we assume the sequences "WF(a); WF(b)" (in example 1) and "WB(a); WB(b)" (in example 2) to have taken place immediately before the actions considered.

actions (after WF)		results	
RB	RB	b	a
R	R	-	-
RB	R	b	b
R	RB	-	b

(example 1)

actions (after WB)		results	
RF	RF	b	a
R	R	b	b
RF	R	b	a
R	RF	b	b

(example 2)

In the examples - stands for an element different from both a and b.

Conclusion: the elementary actions on a sequential file acceptable in the sense of i., ii. and iii. are:

WF : x+ RF : o+
WB : -x RB : -o

A disadvantage of the pack of operations thus left is that they are still somewhat asymmetrical in behaviour. This should be considered the price to be paid for the two-sidedness of the operations.

In determining upon the set of operations to be submitted to the user, it was decided to drop the operation WB. Reasons:

- implementation as a whole becomes a lot easier, mainly because a file now cannot grow at its begin.
- this way the set of operations suggests an asymmetry that exists in fact, instead of suggesting a symmetry that does not.
- the drawback that writing in backward direction is not possible should

not be considered a very serious one, since some thinking nor some investigation among programming colleagues did actually give a particular good application of this operation. (It is admitted that taking decisions this way is taking the risk of throwing the baby out with the bath).

Thus we arrived at a final choice that links up with the proposals made in [2]. Those proposals now can be seen as special cases of the operations mentioned above via some implicit standard pointers.

5. FILE OPENING

File opening is the activation of a passive file. If this passive file is a scratch file the opening is termed *file creation*. If the passive file is a permanent file the action is called opening an old file. In both cases the file is made accessible to the user who, for a start, gets one pointer at his disposal.

File creation always produces an empty scratch file starting at position 1. The species of this file has to be specified by the user at the moment of creation and can not be altered afterwards. The user is (of course) allowed to write into the new file.

Opening an old file activates a permanent file that carries the name specified by the user. Not only the file name of the file wanted has to be supplied, but also whether the file is wanted as a read file or as a work file. Searching through the file catalogue for the desired file, the files of the user are examined first. If one of his own files carries the specified file name it is this file that is activated. In case none of his own files has the right name, search is continued in the area of all public files. This last part of the search can only be successful if the user asked for a read file.

The multi-read aspect of the files gives rise to some statically unavoidable errors, such as "sorry, the file you want to write into is yours alright, but you saved it as a public file and now someone else is reading it - you'll have to wait till he is ready". All errors of this kind are handled in such a way that the user can get aware of them without being kicked off.

5.1. The routines

5.1.1. *integer procedure new file (spec); value spec; integer spec;*

fatal errors: ER WS - *spec* is not a file species.
ER CE - core space exhausted.
ER BE - file space exhausted.

function: 1. A brand new file is created carrying the scratch name and with species *spec*. The file is empty, both beginpointer and endpointer initialized as pointing to position 1. The file claim gets its initial value depending upon the file species; it will be the equivalent of an initial segment of back store. One pointer, the endpointer, is activated, thus enabling the user to write into the new file straight away.
2. A positive integer value is delivered, that serves as identification of this file while it is active, the file number.

5.1.2. *integer procedure new file 1 (spec); value spec; integer spec;*

fatal error: ER WS - *spec* is not a file species.

venial errors: (ER CE) - core space exhausted.
(ER BE) - file space exhausted.

function: as new file.

remark: if a venial error occurs a significant negative value is delivered.

5.1.3. *integer procedure old file (idf);*

fatal errors: ER CE - core space exhausted.
ER UK - no file of file name *idf* exists.
ER NY - the file is a private file of another user.
ER NN - the file is temporarily unavailable because it is a public file being updated by its owner.

function: 1. The parameter *idf* must be either a string or an integer-procedure-without-parameters. In the latter case the procedure is called repeatedly by the file system, which expects internal representations of characters and assembles them to a string; after thus producing a file name the procedure *idf* should deliver a string endmarker, a deletion symbol. So the parameter

idf is or stands for a string, the file name N of the desired file.

2. If the user owns a file called N, that file is activated, otherwise it is a public file of name N. The file is activated as old read file.

3. Appropriate values are assigned to beginpointer and endpointer of the file (values as they were at the moment of closing the file). The workpointer is initialized as pointing to position $\overline{bp}$ of the file, and is activated so the user can start scanning the file without further preparation.

4. A positive integer value, the file number, is delivered. It serves as identification for this file while it is active.

5.1.4. *integer procedure old file 1 (idf);*

venial errors: (ER CE) - core space exhausted.
(ER UK) - no file of filename *idf* exists.
(ER NY) - the file is a private file of another user.
(ER NN) - the file is temporarily unavailable because it is a public file being updated by its owner.

function: as old file.

remark: in case a venial error occurs a significant negative value is delivered.

5.1.5. *integer procedure old work file (idf);*

fatal errors: ER CE - core space exhausted.
ER UK - no file of file name *idf* exists.
ER NY - the file is a private file of someone else.
ER NN - the file is a public file of the user but it is temporarily unavailable as work file because it is being read by another user.
ER NP - the file is a public file of someone else.

function: 1. As old file.

2. If the user owns a file of name N, that file will be activated as old work file, with a file claim as at the moment of closing the file.
3. As old file.
4. As old file.

remark: If a public file is opened as workfile it automatically looses its public state, which state can be regained at the moment of closing the file (as public file).

5.1.6. *integer procedure old work file 1 (idf);*

venial errors: (ER CE) - core space exhausted.
(ER UK) - no file of file name *idf* exists.
(ER NY) - the file is a private file of someone else.
(ER NN) - the file is a public file of the user, but it is temporarily unavailable as work file because it is being read by another user.
(ER NP) - the file is a public file of someone else.

function: As old work file.

remarks: As old work file.
In case a venial error occurs a significant negative value is delivered.

5.2. Implementation

The routines concerned with file creation have *new file 1* as common part. The routines concerned with the activation of a permanent file all lean most heavily upon the routine *open old file.*

5.2.1.1. *integer procedure new file (species);*
begin integer f; new file:= f:= new file 1 (species);
if f < 0 then error (f)
end;

5.2.1.2. *integer procedure old file (idf);*
begin integer f; old file:= f:= open old file (idf,false);
if f < 0 then error (f)
end;

5.2.1.3. *integer procedure old file 1 (idf);*
old file 1:= open old file (idf,false);

5.2.1.4. *integer procedure old work file (idf);*
begin integer f; old work file:= f:= open old file (idf,true);
if f < 0 then error (f)
end;

5.2.1.5. *integer procedure old work file 1 (idf);*
old work file 1:= open old file (idf,true);

5.2.2. Opening a new file

The opening of a new file is done by the routine *new file 1* and consists of the claiming of file space and the initialization of it. The claiming is done in portions, because the data structure of the core space needed does not allow to claim all space at once. Moreover this way the opening of a new file and the opening of an old file can share claiming routines. Claiming core space is done carefully, that is, as soon it is detected that some space required is not available, the space already claimed for the file-in-the-making is returned to the heap.

5.2.2.1. *integer procedure new file 1 (species); value species;*
integer species;
comment species: the species of the file to be created;
begin integer k, f, segmad;
hard check on species (species); SYS not;
f:= set up first part skeleton (true,no); comment if during this run the user has not yet opened a file, the general file administration space fad is claimed from the heap and initialized. From the file administration fad a file number is obtained.

```
                If no file number is available fad is extended, an action
                that includes claiming space from the heap. Parts of the
                administration needed for one file are claimed from the heap
                (cad,des and sad) and initialized ('nptrs' := 'nbics' :=
                'nfreebics' := 0). If any of the claims is rejected, all
                space that is claimed for this file is returned to the heap
                and a jump to no is executed;
initialize descriptor new file (f,species); comment 'bp':= 'ep':= 'nsegm':=
                'offset':= 1 / 'spec':= species / 'new':= 'work':=
                'scratch':= true / 'idf':= "" / 'last block':= 0 / 'nelpw',
                'nelpb', 'bpel', 'nblocks' and 'nfree' are set to their
                initial obvious values;
set up second part skeleton (f,EP,no); comment pointer administration is
                claimed from the heap, as well as core space for one data
                block. Pointer ep is activated. In case a claim from the
                heap is rejected all space that is claimed for this file
                is returned to the heap and a jump to no is executed;
segmad:= SYS claim segment (true); comment one file segment is claimed
                from the scratch pool;
if segmad < 0 then
begin comment no segment available from scratch pool;
      delete core (fadb - f); comment return to the heap all space claimed
                                    for this file;
      goto nob
end;
store (sadb(f) - SEGMAD IN SAD,segmad);
for k:= 1 step 1 until NBPS do
mark block in core (f,k,false); comment the bit table over all file blocks
                is initialized: no file block is in core;
initialize ptr (f,EP,1); comment the already active endpointer is initi-
                alized: its value is set to 1 and the core block reserved
                previously is now attached to the endpointer (so the inter-
                est count 'int' of the bic is set to 1 and file block 1 is
                marked "in core");
if true then new file 1:= f else
```

```
no: if true then new file 1:= ER CE else
nob:              new file 1:= ER BE;
  SYS ton
end new file 1;
```

5.2.3. Opening an old file

Opening an old file is done by the routine *open old file*. The remarks made in 5.2.2. about opening a new file hold for opening an old file also. The initialization however is a lot more complicated since the file catalogue has to be consulted. Opening and closing the catfile is done in the space just claimed for the file to be created as described in 2.2.2.2.

```
5.2.3.1. integer procedure open old file (ident,work); value work;
                                                    boolean work;
        comment ident = string or integer procedure yielding the file
                        name of the file wanted to be opened.
                work = a boolean value deciding whether the file is to be
                       opened as work file (true) or as read file (false);
begin integer array idf [1:IDFL]; integer f,pos,backad,k;
  make idf (idf,ident); comment the characters of the file name are
                obtained from ident and reassembled, 3 characters per
                word, in array idf;
  SYS not;
  f:= set up first part skeleton (false,no); comment see description at
                new file 1. The first actual parameter indicates an old
                file is involved, so an ample amount of space is initially
                claimed for the segment administration sad, in order that
                the largest file possible can be opened;
set up second part skeleton (f,WP,no); comment see description at new
                file 1. The workpointer is made active (instead of the
                endpointer as in new file 1);
for k:= 1 step 1 until IDFL do
store (desb(f) - IDF IN DES - k + 1,idf[k]); comment the file name is
                stored into the core descriptor des of file f;
get access to catfile; comment a P-operation [0] with respect to the
```

catfile facility;

simple open file (f,fetch(SYSVAR CATFIBACKAD),true); *comment* the catfile is opened as own work file in such a way that no extra main store is needed to be claimed from the heap;

pos:= pos in catfile (f,idf,work,fetch(SYSVAR USER)); *comment* the catalogue is searched for the desired file. The position of its short descriptor in the catfile is assigned to *pos*. If the request for the file is not to be honoured for one reason or another, *pos* will be assigned a negative value indicating that reason;

if pos < 0 then

begin *comment* the required opening failed;

delete core (fadb - f); *comment* all space claimed for this file is returned to the heap;

f:= pos

end else

begin *comment* the required opening is successful;

mark interest in catfile (f,pos,work); *comment* information about the kind of use is made of the file is entered into the catalogue;

backad:= backad in catfile (f,pos);

simple close file (f); *comment* the catfile is closed now;

reopen file (f,backad,work); *comment* the file wanted now is opened as own file. If *work* then it is opened as work file, otherwise it is opened as read file. In both cases the work-pointer is active, with value $\overline{bp}$. Initialization of the file has taken place;

comment the space claimed for the segment administration was sufficient to contain the administration of even the largest file possible. This space now is reduced to the proportions actually needed by the file *f* : ;

shrink sad (f)

end;

return access to catfile; *comment* the V-operation [0] for the catfile

```
                                        facility;
   if true then open old file:= f else
no: open old file:= ER CE;
  SYS ton
end open old file;
```

6. FILE CLOSING

If a file is closed it will no longer occupy any core space. If the file closed is a scratch file, all information contained in it will get out of reach (of all users) permanently. In other words, the file is deleted. If the file closed is an own file, the information gets out of reach of the user temporarily; the file will be saved in back store; the file state changes from active to permanent. A permanent file can be (re)activated by opening it as an old file.
Thus, closing a file means either destroying the file info - in case the file name is the scratch name - or saving the file info otherwise.
If a file is closed as permanent file and this is done by the creator of the file having a work permit, then the file can be destined public or private, optional with the user.

When a program is terminated by the operating system some of its files may still be active. The system shall take care of those files and close them. In case a choice between private and public has to be made the system closes a file as a private one.
Although explicit file closing is not demanded from the user it is advised to do so (as soon as possible) since closing
. frees core storage
. frees back store space (if the file is a scratch file)
and thus possibly prevents the occurring of errors due to storage exhaustion. Furthermore it will in general make the user's program more comprehensible.

If the file closed is an old read file, no problems arise in updating the file catalogue; in the catalogue it is entered that for this particular file a reader cancelled his subscription.
If the file closed is an own work file, i.e. an old work file or a new own file, that file must be saved under file name and creator. The catalogue contains this kind of information about all permanent files. A difficulty crops up if the file name already occurs in the catalogue in a situation that will cause ambiguities:

. some other permanent file of this user has the same name as the file that is to be closed.
. the file is closed as public file and some other public file has the same name as the file to be closed.

In both cases the file system changes the file name into one that is unique and the file is closed as private file under the new name. These actions are reported by the system:
. via normal system's report, e.g. a monitor-report-sheet; in other words, the actions are protocolized.
. directly to the user who asked for file closing, via an output parameter of the routine called.

The file is reopened as old work file under the new name in order to enable the user:
. to get hold of the name the system invented so he will not loose his grip on this file during this run of the program.
. to suggest another name for the file to the file system.

6.1. The routines

6.1.1. *boolean procedure close file (f); value f; integer f;*

fatal error: ER WF - *f* is not a file.

function: 1. If *f* is a scratch file the file is deleted.
2. If *f* is an old read file the file is returned to the system, that is, subscription is cancelled.
3. If *f* is an old work file the file is closed as private file, possibly after a renaming of the file by the system.
4. In case a renaming took place the value false, otherwise the value true is delivered.

6.1.2. *boolean procedure close file public (f); value f; integer f;*

fatal errors: ER WF - *f* is not a file.
ER PC - *f* is not an own work file.

function: 1. The file *f* is closed as public file, possibly after a renaming of the file by the system.
2. If a renaming took place the value false is delivered, true otherwise.

6.2. Implementation

Both routines available to the user consist of a call of the routine *close*, which will be treated extensively in the sequel.

6.2.1.1. *boolean procedure close file (f); close file:= close (f,true);*

6.2.1.2. *boolean procedure close file public (f);*
close file public:= close (f,false);

6.2.2. The routine *close*

```
boolean procedure close (f,priv); value f,priv; integer f; boolean priv;
comment    f - the file number of the file to be closed.
       priv  - true: if the file is an own work file it is closed as a
                     private file, otherwise it is obliterated or returned.
               false: if the file is an own work file it is closed as
                     a public file, otherwise a fatal error message is
                     given;
begin integer des,catpos,k;
      boolean work,new,scratch,ok;
      integer array idf,old idf [1:IDFL];

hard check on f(f); comment if f is not a file a fatal error message
                           follows;
des:= desb(f); work:= fetch (des - WORK IN DES) > 0;
scratch:= fetch (des - SCRATCH IN DES) > 0;
if if ¬ priv then ¬ work ∨ scratch else false
then error (ER PC); comment if the file has to be closed as a public file,
                           then it has to be an own work file;
new:= fetch (des - NEW IN DES) > 0;
```

```
simple close file (f); comment now all file blocks are safely stored on
                                    back store;
SYS not; comment program termination is not allowed from now on since
                    vital administration will be inconsistent for some time;
if scratch then
begin delete all segments (f); comment all segments of the scratch file
                    are returned to the scratch pool. The file becomes pointer-
                    less, i.e. all pointer administration is deleted;
                    comment f is a scratch file but in case it was an old file
                    when opened the file catalogue must be updated: ;
      if new then goto true
end else
begin if work then make adm block (f,false); comment for a new own work
                    file a long descriptor must be created and for an old work
                    file it must be updated;
                    comment the file becomes pointerless: ;
      release second part skeleton (f)
end;
for k:= 1 step 1 until IDFL do
idf[k]:= fetch (des - IDF IN DES - k + 1); comment the file name is copied;
ecch: set up second part skeleton (f,WP,ecch); comment the pointerless file
                    gets equipped with one pointer: the workpointer wp, such
                    that wp = beginpointer. The dummy label ecch occurs because
                    in general an operation as sketched above might require
                    main store not available. In that case control would be
                    transferred to the process identified by the label. Not so,
                    however, in this case where the storage surely will be
                    available;
get access to catfile; comment this can be considered a P-operation with
                                    respect to the catfile;
simple open file (f,fetch(SYSVAR CATFIBACKAD),true); comment the catfile is
                    opened as own work file in such a way that no main store
                    is needed but the administration space of f;
ok:= update catfile (f,idf,des,priv,work,new,scratch); comment the routine
```

```
                update catfile really does all the work;
simple close file (f); comment the catfile is closed now;
return access to catfile; comment the V-operation for the catfile facility;
if ok then true:
begin delete core (fadb - f); comment delete all core space occupied by f;
      close:= true
end else
begin for k:= 1 step 1 until IDFL do
      old idf[k]:= fetch (des - IDF IN DES - k + 1);
      SYS idf fancy (f,old idf,idf); comment the operating system is
                informed about the renaming of file f. It can take actions
                for an accurate report of it. The old name is contained in
                old idf, the new one in idf;
      for k:= 1 step 1 until IDFL do
      store (des - IDF IN DES - k + 1,idf[k]); comment the new name is
                stored in the core descriptor des;
      reopen file (f,fetch(des - BACKAD IN DES),true); comment the file is
                opened as own work file;
      close:= false
end;            comment program termination is allowed again: ;
SYS ton
end close;
```

6.2.3. Other aspects

When terminating a program, the operating system shall take the following actions:

```
comment if the program used any files: ; if fadb ≠ 0 then
begin integer f; for f:= 1 step 1 until fmax do
      if cadb (f) ≠ 0 then
      begin if ¬ close file (f) then close file (f) end
end;
comment cadb (f) ≠ 0 iff f is an active file;
```

6.3. Implicit closing

It is not demanded from the user to close explicitly all the files he has opened. Extorting explicit closing from the user cannot be done, so demanding it, i.e. not doing it implicitly at program termination, would cause the following undesirable situations:

- the user opens a public file as read file. If this file is not closed before program termination, then the file will be listed as "subscripted" for ever. So the file owner will no longer be able to alter this file.
- the user creates a scratch file, renames it as an own file and stuffs it with valuable information. Due to some innocent programming error the program is terminated and all information is lost.

7. FILE NAMING

File creation produces a scratch file, i.e. a file with the scratch name. Closing a scratch file means exterminating it. The only way to preserve a file for a longer period than the run it is created in, is to give that file a name different from the scratch name. Thus the file becomes an old file and closing the file will make it permanent. Since file closing preserves all own files, the only way to delete an own file is to give it the scratch name before closing it.

Changing the name of a file is possible only if the file is a work file. Changing the name from non-scratch to scratch causes a transfer of the file space occupied by the file from the own pool to the scratch pool. This transfer can always be done - it cannot prohibit the name alternation. On the other hand, if the name is changed from scratch to non-scratch, the occupied file space has to be transferred from the scratch pool to the own pool, which might not always be possible, due to the file space distribution key. (The transfers, mentioned above, should not bother the user for their efficiency aspect: they are made not in physical but in administrative sense.)

7.1. The routine

boolean procedure new idf (f,idf); value f; integer f;

fatal errors: ER WF - *f* is not a file.
ER NW - *f* is not a work file.

function: 1. As *old file* (*idf* yields a file name N).
2. The file name of *f* is changed into N, except when *f* is a scratch file and N is not the scratch name and the transfer from the scratch pool to the own pool is considered impossible by the system.
3. If the file afterwards carries its new name, the value true is delivered, false otherwise.

7.2. Implementation

```
boolean procedure new idf (f,ident); value f; integer f;
begin integer array idf [1:IDFL]; boolean scratch; integer k,des,b;
make idf (idf,ident); comment the characters of the new file name are
                      obtained from ident and reassembled, 3 characters/word
                      in array idf;
hard check on f (f);
hard check on work permit (f);
SYS not; scratch:= true; des:= desb (f);
for k:= 1 step 1 until IDFL do
scratch:= scratch ∧ idf[k]=SCRATCHIDF; comment scratch≡the new name is the
                                                    scratch name;
if fetch (des - SCRATCH IN DES) > 0 then
begin comment f is a scratch file;
  if ¬scratch then
  begin comment the user asks to change the scratch file into an own file;
    if  SYS no longer scratch (fetch(des - NSEGM IN DES))
    then goto false; comment the transfer of 'nsegm' file segments from
                      the scratch pool to the own pool is not allowed. The file
                      name is not changed and the value false is delivered;
    b:= SYS claim block; comment a block is claimed from the own pool.
                      It shall contain the long descriptor of f;
    if b < 0 then
    begin comment how sad, the alteration seemed to be in the bag, but
                      no block for the long descriptor is available;
      SYS scratch now (fetch(des - NSEGM IN DES)); comment undo the
                      transfer from scratch pool to own pool;
      goto false
    end;
    store (des - SCRATCH IN DES,-777);
    store (des - BACKAD IN DES,b)
  end scratch to non-scratch
end f is scratch file else
begin if scratch then
```

```
  begin comment the file must be changed from own file to scratch file;
    SYS scratch now (fetch(des - NSEGM IN DES)); comment transfer the
                segments of f from the own pool to the scratch pool;
    store (des - SCRATCH IN DES,777); comment the block for the long
                descriptor of f is no longer needed : ;
    SYS delete block (fetch(des - BACKAD IN DES))
  end non-scratch to scratch
end;
for k:= 1 step 1 until IDFL do
store (des - IDF IN DES - k + 1,idf[k]); comment the file now carries the
                new name as specified by the user;
if true then new idf:= true else
false:       new idf:= false;
SYS ton
end new idf;
```

8. INQUIRY ROUTINES

When manipulating files certain questions may arise to the user. Questions concerning the size of the file claim, the value of active pointers, the file species, the file name. These, and other questions are answered by the seven inquiry routines given below.

8.1. The routines

8.1.1. *integer procedure file species (f); value f; integer f;*

venial error: (ER WF) - *f* is not a file.

function: the species of *f* is delivered.

remark: the species of a file is coded in a non-negative integer. If *f* is not a file a negative value is delivered.

8.1.2. *integer procedure file claim (f); value f; integer f;*

fatal error: ER WF - *f* is not a file.

function: the maximal number of elements to be contained in *f* is delivered.

8.1.3. *boolean procedure work permit (f); value f; integer f;*

fatal error: ER WF - *f* is not a file.

function: if the file *f* is a work file the value true is delivered, the value false otherwise.

8.1.4. *integer procedure idf sym (k,f); value k,f; integer k,f;*

fatal error: ER WF - *f* is not a file.

function: suppose the file name is represented by a suitable string S. The value *idf sym* delivers will be that of stringsymbol (k,S): the internal representation of

. the deletion-symbol if $k < 0$ or $k \geq$ the length of S,
. the k-th symbol of S otherwise.

remark: in case a routine has a file number as one of its parameters it is the first parameter; this holds for all routines except for *idf sym*. The order of the parameters is identical to that of *stringsymbol*, a routine with a function analogous to *idf sym*.

8.1.5. *integer procedure value of bp (f); value f; integer f;*

fatal error: ER WF - f is not a file.

function: the value of the beginpointer of f is delivered, whether the beginpointer is an active pointer or not.

8.1.6. *integer procedure value of ep (f); value f; integer f;*

fatal error: ER WF - f is not a file.

function: the value of the endpointer of f is delivered, whether the endpointer is an active pointer or not.

8.1.7. *integer procedure value of ptr (f,p); value f,p; integer f,p;*

fatal error: ER WF - f is not a file.

venial error: (ER WP) - p is not a pointer of f.

function: the value of pointer p is delivered.

remark: in case p is not a (n active) pointer of f a negative value is delivered.

8.2. Implementation

In all cases the implementation of these routines is as straightforward and obvious as possible - mostly the main part of the routine consists of the examination of one system variable -, so no more words will be

wasted upon it here.

8.3. Design considerations

The user should be able to avoid the occurrence of fatal errors in all or nearly all possible situations. Therefore no vital information may be hidden; all useful information must be attainable in some way. The inquiry routines enable the user to obtain that information, even if he is "blindfolded", such as a general purpose routine, fully dependent on its parameters. It was decided to combine several inquiry functions into one routine, if such an approach seemed quite natural, rather than introducing the umpty-umpth routine. This combining is done by means of the venial errors in the routines *file species* and *value of ptr*.
The routine *file claim* has an additional function, maybe not obvious at first sight, which has to do with file extension. It enables the user to avoid a fatal error resulting from file space exhaustion.

9. DYNAMIC STORAGE ALLOCATION MODULE

The here presented file system requires storage handling that allows garbage collecting techniques. The run-time stack is not an adequate part of the storage for this kind of use, so another part of the memory, the *counter stack*, has to be called into existence. If we locate the bottom of the stack in the low addresses and let it grow upwards, the bottom of the counter stack is located in the high addresses and grows downwards. The counter stack can be considered as hanging from the ceiling. Once stack and counter stack are established there will be no third part of storage that can easily be placed at the disposal of an authority that needs some other kind of dynamic storage. Therefore the counter stack in the present system, the heap, is modelled in such a way that it allows far more than needed by the file system; e.g. string operations in ALGOL 60 would be quite easy to implement once the dynamic storage allocation (*DSA*) module is present in the operating system.

The way the DSA module functions, internally, is hardly of any interest to the file system. It is not important if and how the module does things like garbage collection and compaction. The things the (programmer programming the) file system should know could be termed the face, the outside of the module. This face, a description of the DSA module in user's terms, is given below.

9.1. External description

A description is given of the heap, as implemented for the X8, so, for one thing, word length from now on will be 27 bits.
The heap consists of objects, each *object* being a consecutive row of words. An object can be considered an information unit; in general the interpretation of the information conceiled in an object is entirely at the responsibility of the user, e.g. the file system, and not of the DSA module. To help the user in recognizing the object's meaning, he is allowed to attach a *type* to each object. How this is done follows from the description of the representation of an object in the heap as one word. Such a

representation is called a *struct*.

a struct:

d26	d25...d19	d18	d17...d0
0	"type"	0	"ref"

The choice of "type" is left to the user, the file system uses "type" = 0 for file structs.
If "ref" ≠ 0 it points to the first word of the object in the heap. This word is called the *genus word*. It gives information about the object as can be seen below. If "ref" = 0 it is a nil-reference; it does not refer to any object.
Three genera of objects exist:

. *rho* - an array of structs, i.e. all elements of the object are structs.
. *pi* - an array of plain values, i.e. none of the elements of the object is a struct.
. *lambda* - two words, one being the genus word that contains a reference to an object, the other being a struct. So an object of genus lambda contains two references; it is called a list cell.

The layout of the genus words is:

	d26	d25...d19	d18	d17...d0
rho	1	refcnt	1	length
pi	1	refcnt	0	length
lambda	0	refcnt	1	ref

The *reference count* "refcnt" gives the number of references made to this object. If the number of references exceeds 125 it is fixed at 126, meaning 126 or more. The number is coded as the inverse of the binary representation of the reference count.
The length of the object is coded in "length" as the inverse of the binary representation of the object length minus 1 (= length of data field).
All objects in the heap are positioned upside down, that is, if m is the

address of the genus word of object M, the second word of M is to be found at address m-1, and so on.

The routines that have to do with the DSA module are presented in section 10.

10. INTERFACE FILE SYSTEM / OPERATING SYSTEM

The *interface* between the file system and the (rest of the) operating system can be defined as follows: suppose the operating system is a fully self-supporting piece of software; suppose furthermore the file system consists of a bundle of routines, forming a module to be added to the operating system. The interface then consists of:

- all routines, variables and constants needed by the file system and not contained in it. The file system assumes these tools to be present in the operating system; if they are not, the operating system has to be extended.
- actions the file system (cannot possibly take by itself and) expects to be taken by the operating system under certain circumstances. This easily leads to modifications of the operating system. Since these actions may involve the call of one or more routines of the file system these routines in fact should be considered as part of the interface.

Besides the interface thus defined, another, weaker form of connection exists which might be called the *hidden interface*. This hidden interface has to do with actions taken by the file system which might have been taken by the operating system in a plain and more efficient way. Such actions, in fact, have been implemented in the present file system. It seems a matter of taste at what side of the dividing-line they should be situated. If the implementation is seen as a general approach, one can argue, they belong to the file system; otherwise, if the implementation is done for a particular operating system, they belong to that operating system. Other grounds for the decisions made will be put forward at the detailed discussion of the hidden interface.

10.1. Routines, constants and variables

10.1.1. Routines concerning main store

10.1.1.1. The simple main store routines

procedure store (ad,w); stores *w* at address *ad*.

procedure sstore(ad,real); double-length store: stores *real* at addresses *ad* and *ad*+1.

integer procedure fetch (ad); delivers contents of address *ad*.

real procedure ffetch (ad); double-length fetch: delivers the contents of addresses *ad* and *ad*+1.

procedure incr (ad,i); adds *i* to the contents of address *ad*.

10.1.1.2. Dynamic storage allocation routines

integer procedure SYS claim (lh); claims heap space of length *lh* and delivers the first address of the space claimed. If space not available a negative value is delivered.

procedure SYS shrink (ad,lh); the length of the object starting in location *ad* is reduced to the new length *lh*.

integer procedure SYS extend (ad,extra); the length of the object starting in location *ad* is increased by *extra*. Possibly the increase changed the position of the object in the heap, so the (new) starting address of the object is delivered. If the space extension is a failure, due to core space exhaustion, a negative value is delivered.

procedure SYS delete (ad); the space occupied by the object starting in location *ad* is returned to the heap. Appropriate measures are taken in case the object deleted referenced other objects.

procedure store ref (ad,ref); stores *ref* in the reference part of the contents of address *ad*.

integer procedure fetch ref (ad); delivers the reference part of the contents of address *ad*.

procedure SYS decr refcnt (ad); decreases the reference count of the object starting at *ad* by 1.

procedure SYS incr refcnt (ad); increases the reference count of the object starting at *ad* by 1.

integer procedure SYS length (ad); delivers the length of the object starting at *ad*.

integer procedure SYS genwrd (gen,lh,rc); delivers the genus word of an object of genus *gen*, length *lh* and reference count *rc*.

10.1.2. Routines concerning back store

integer procedure SYS claim block; delivers the back address of a block from the own pool. If not available a negative value is delivered.

procedure SYS delete block (bad); returns the block at the back address *bad* to the own pool.

integer procedure SYS claim segment (scratch); delivers the back address of a segment from the (if *scratch* then scratch else own) pool, if available, a negative value otherwise.

procedure SYS delete segment (sad,scratch); returns the segment at the back address *sad* to the (if *scratch* then scratch else own) pool.

procedure SYS scratch now (n); *n* own segments are scratch from now.

boolean procedure SYS no longer scratch (n); *n* scratch segments are own from now on, if allowed, and the value true is delivered. If not allowed (because there are too many own segments already) the value false is delivered.

integer procedure SYS compute backad (sad,b); delivers the back address of the *b*-th block of the segment at the back address *sad*.

procedure SYS to disk (m,l,b,c); transports *l* consecutive locations of core storage, starting at *m*, to the disk sector with back address *b*. If and when transport is done the contents of core location *c* are increased by 1.

procedure SYS from disk (m,l,b,c); transports the disk sector with back address *b* to *l* consecutive core locations starting at *m*. If and when the transport is completed the contents of the core location *c* are increased by 1.

10.1.3. Routines concerning the supervisor

The routines in this section have to do with a very special part of the

operating system dealing with interrupt handling, swapping and typical monitoring functions.

procedure error (e); the fatal error *e* occurred and control is transferred to the monitor.

procedure SYS not; a critical section is entered, no program termination is allowed. It is supposed that this wish is honoured by a mechanism using a counting device, rather than a boolean flag.

procedure SYS ton; a critical section is left. This is the reverse operation of *SYS not*.

procedure SYS el; an elementary action has to be taken, no interrupt is allowed. A mechanism like that of *SYS not* should handle it.

procedure SYS le; an elementary action is completed. This is the reverse operation of *SYS el*.

procedure SYS swap; the program can be swapped out immediately; (and swapped in any time, though it is understood that) the swap reason is: this program needs the catfile facility, which facility is occupied by someone else.

10.1.4. Miscellaneous routines

boolean procedure SYS is string (p); if *p* is a string the value true is delivered, false otherwise.

boolean procedure SYS is int proc (p); if *p* is an integer procedure without parameters the value true is delivered, false otherwise.

procedure SYS fancy idf (idf); changes the contents of the integer array *idf* [*1:IDFL*]. This array contains the representation of a file name that is changed into another one in some neat way (for instance: repetitive calls of this routine should yield as many different file names as possible, within reasonable bounds).

10.1.5. Constants

The operating system has knowledge of the constants BP, EP and WP so it

can inform the user of the system if needed.
All fatal errors have a unique integer identifying them and it may be of interest to the operating system to know these error numbers, in order to produce some intelligible message instead of a cryptic error code in case a fatal error occurs.

10.1.6. Variables

Four variables with fixed locations within the operating system must be mentioned:

SYSVAR catfibackad - the back address of the long descriptor of the catalogue file.

SYSVAR catfinacc - a boolean variable telling whether the catalogue file is temporarily inaccessable or not.

SYSVAR user - the code(name) of the user. This is a *swap variable*, i.e. a variable that has to be swapped in and out with the program. For all programs a certain swap variable is in the same location.

SYSVAR filehandle - the reference to all file space (in the heap) of the active program. This variable is a swap variable.

10.2. Actions

The file system can ask for transport of information from main store to back store or vice versa. Any request for such a transport has, besides the parameters describing the transport wanted, an additional parameter: the *report address*. The operating system is expected to carry out transport and, if it is finished and done, to report the completion by adding 1 to the contents of the report address.

Whenever a file routine needs to consult or update the catalogue file, the *catfile facility* is wanted by that routine. If some other authority occupies the facility at that moment, the routine asking for it explicitly allows to be swapped out by the operating system. If the operating system decides to swap out on this ground (catfile facility occupied) it is due

to swap in at some time or another, preferably as soon as, but not before, the facility is free for this user.
Some of the heapspace occupied by the file system possibly is not directly needed by it. That is, it can function properly, though maybe less efficient, without that space. In case some authority badly needs space, none available, the garbage collection routine of the dynamic storage allocation system can demand the release of the space mentioned. This can be done by a call of the routine *free semifree*.

If for some reason a program is terminated, the operating system is obliged to close all files still active for the program. This closing can be done by an appropriate number of calls of the routine *close*.

10.3. Hidden interface

The catfile facility is claimed by the routine *get access to catfile* and the facility is released by the routine *return access to catfile*. The latter routine, in fact to be considered as a V-operation, is implemented in a very simple way; the former routine, actually a P-operation, is realized through calls of *SYS el*, *SYS le* and *SYS swap*. The implementation chosen could easily be replaced by another one, e.g. real P- and V-operations. It is done the way it is because of simplicity, test runs in a simulated environment in mind.

Since the file system is presented as a lot of ALGOL 60 procedures and not, say, code routines, no use is made of routines that are supposed to be driven by interrupts. So, parts of the system that fundamentally rely upon interrupt sensitive actions, such as the transport routines, have been implemented in a rather clumsy way. In a machine code version of the system these parts surely should be modified.

A role of very special importance is played by the routine *initialize file system*. It is called upon the moment a user for the first time during the program activates a file. This role is sketched in the next lines, though the routine as presented in the sequel does not act that way at all, since such action is very much operating system dependent. Adding the file system

in ALGOL 60 form to an operating system that has an ALGOL 60 library at its disposal, can be done by modifying the operating system a bit, and extending the library with the file system procedures.
The operating system is supposed not to rely on the file system in this approach, so it has no direct ways to access file routines. Under certain circumstances, however, the operating system is assumed, not to say obliged, to call file routines: *close* at program termination, *free semifree* if heap space trouble occurs. Luckily these routines are in core in case the operating system needs them, but their explicit locations in main store may be unknown to the system. The system possibly can get hold of them by examining some library tables at a convenient moment. If not, the core positions that are of importance can be handed over to the system by the routine *initialize file system*, in passing also informing the system about the fact that the program considered uses files. The routine *initialize file system* itself can get hold of the relevant core addresses either directly - in a code version - or by some trick allowed by the system.

11. TESTING THE SYSTEM

The file system was tested without having it added to some operating system. Therefore nearly all parts of the interface belonging to the operating system are simulated in an ALGOL 60 environment:

. The heap is situated in an integer array, *mem*, big enough to do some testing. This way a crash between a growing stack and a sagging heap is easily averted. The simple main store routines are done with by simple accesses of *mem*, all of them provided with software checks on the bounds of *mem*.
The routines of the dynamic storage allocation system asked for a more sophisticated approach. They are simulated fairly good, but for an inefficient (and somewhat incorrect) behaviour of *SYS extend* and an implementation of *SYS claim* that does no garbage collection nor compaction. Since nearly all data structures involved in the heap are of the same sort in cases that matter (of sort bic), the latter defect of *SYS claim* will not bother too much - the ALGOL 60 version is perfectly well capable of re-using returned heapspace. Besides their functional meaning in testing the system, the simulation of the heap routines may serve to brighten the insight into some parts of the dynamic storage allocation system.

. The info transports from and to disk are replaced by drum accesses. All transports taken up are waited for until they are completed (no interrupt business).
The claiming of segments and blocks is done from different parts of drum storage.

. Fatal errors are reported by printing their number, accompanied with a "coredump" (the contents of the array *mem* are printed). All other supervisor routines are supplied in the most simple way - as empty routines.

. In case an actual parameter is allowed to be either a string or a parameterless procedure delivering an integer value, the latter possibility is prohibited in the testing phase. The routine *SYS fancy idf* is supplied in

a rather silly form, though it is good enough to perform test runs.

Testing the system gives rise to many situations in which it is convenient to the performer to know exactly the contents of the heap. These contents can be made visible by a call of *dump*. The procedure *dump* is added for reasons of testing only. This auxiliary procedure dumps the heap contents over the lineprinter in a structured lay-out, closely resembling the actual structures on the heap. So the performer can easily read the heap at any particular moment he wants to.

Whenever an old file is opened or an own file is closed, it is assumed by the system that a file catalogue exists. Therefore, initializing the system (for test reasons) means, among other things, the founding of a library and a matching catalogue. This is done by the procedure *found catalogue*. It creates an empty file that serves as catalogue file - the library all tests start with is empty.
Some system parameters have to be chosen, such as word length, block length, segment length, et cetera. These parameters are assigned a value only once and can be characterized as assembly parameters. Their values are aptly chosen in the test phase, so, that they

- allow nearly all realistic situations that are interesting to occur in relatively small test samples.
- do not demand the use of enormous amounts of heap space, so for one thing the dumps are kept surveyable.

12. THE PROGRAM

The ALGOL 60 program presented in the next section consists of all file routines, embedded in a rather small test envelope. The main program shows some aspects of the use of the file procedures discussed in the previous chapters. It mainly serves to give some illustration to the reader, rather than to test the system, which was done to some extent with a lot of small testing samples not presented here.

In order to facilitate a transscription, if any, of the system from ALGOL 60 to some convenient assembler code, the program is of a very simple structure. For instance, complicated statements are avoided and procedures are not nested.

The source code of the program was on cards, so the actual text differs from that on the previous pages with respect to the representation language. Instead of underlining word delimiters they are apostrophed and only capital letters occur. Furthermore certain abbreviations are used, such as '*INT*' for '*INTEGER*', etc. (see [3]).

Values of boolean nature are represented in the heap by integers. A positive value always means true and a negative one always means false. In fact, all values that serve this purpose are chosen to be 777 and -777 respectively.

A lot of constants appear in the system; constants of type "assembly literal". ALGOL 60 does not provide the possibility of using these literals, so the following peculiar solution is chosen: all such constants are delivered by integer procedures (of course this is very "expensive", but surely admissible in an experimentary model of this type). Why procedures and not variables? Variables must be declared and initialized separately, which seemed inconvenient to the programmer of the system. Procedures do not have this inconveniency.

A procedure the name of which starts with *hard check* causes a fatal error if the checking is unsatisfactory.

In some parts of the program it was needed to do some bit manipulation. Of course this could have been done in ALGOL 60 by combinations of integer division, multiplication, etc. but it was decided to use the bit manipulation procedures offered by the system's library [7].

12.1. The program text

The integral ALGOL 60 text of the system is reproduced on the next pages.

```
1
2   'BEGIN' 'COMMENT' A FILE SYSTEM FOR MULTI-SEQUENTIAL FILES ,
3                                                   H.W.ROOS LINDGREEN;
4
5
6   'COMMENT'  THE TEST ENVELOPE  ****************************************;
7
8   'INT' T18,T19,T26,MEM END,FREEPTR,NBBFREE,NBB,NBSFREE,NBS,
9         BBOFFSET,BSOFFSET;
10  T18:= 2**18; T19:= 2*T18; T26:= 1=2**26;
11  MEM END:= 3000; NBB:= 10; NBS:= 100;
12
13  'BEGIN' 'BOOL' 'ARRAY' BBFREE[1:NBB],BSFREE[1:NBS];
14          'INT' 'ARRAY' MEM[0:MEM END];
15
16
17  'COMMENT' USER ROUTINES ***********************************************;
18
19
20  'INT' 'PROC' NEW FILE(SPECIES); 'VAL' SPECIES; 'INT' SPECIES;
21  'BEGIN' 'INT' F; NEW FILE:= F:= NEW FILE 1(SPECIES);
22    'IF' F<0 'THEN' ERROR(F)
23  'END';
24
25  'INT' 'PROC' NEW FILE 1(SPECIES); 'VAL' SPECIES; 'INT' SPECIES;
26  'BEGIN' 'INT' F,K,SEGMAD;  HARD CHECK ON SPECIES(SPECIES);
27    SYS NOT;
28    F:= SET UP FIRST PART SKELETON('TRUE',NO);
29    INITIALIZE DESCRIPTOR NEW FILE(F,SPECIES);
30    SET UP SECOND PART SKELETON(F,EP,NO);
31    SEGMAD:= SYS CLAIM SEGMENT('TRUE');
32    'IF' SEGMAD < 0 'THEN'
33    'BEGIN' DELETE CORE(FADB = F); 'GOTO' NOB 'END';
34    STORE(SADB(F) = SEGMAD IN SAD,SEGMAD);
35    'FOR' K:= 1 'STEP' 1 'UNTIL' NBPS 'DO'
36    MARK BLOCK IN CORE(F,K,'FALSE');
37    INITIALIZE PTR(F,EP,1);
38    'IF' 'TRUE' 'THEN' NEW FILE 1:= F 'ELSE'
39  NO:  'IF' 'TRUE' 'THEN' NEW FILE 1:= ER CE 'ELSE'
40  NOB:  NEW FILE 1:= ER BE;
41    SYS TON
42  'END';
43
44  'INT' 'PROC' OLD FILE(IDF);
45  'BEGIN' 'INT' F; OLD FILE:= F:= OPEN OLD FILE(IDF,'FALSE');
46    'IF' F<0 'THEN' ERROR(F)
47  'END';
48
49  'INT' 'PROC' OLD FILE 1(IDF);
50    OLD FILE 1:= OPEN OLD FILE(IDF,'FALSE');
51
52  'INT' 'PROC' OLD WORK FILE(IDF);
53  'BEGIN' 'INT' F; OLD WORK FILE:= F:= OPEN OLD FILE(IDF,'TRUE');
54    'IF' F<0 'THEN' ERROR(F)
55  'END';
56
```

```
'INT' 'PROC' OLD WORK FILE 1(IDF);
  OLD WORK FILE 1:= OPEN OLD FILE(IDF,'TRUE');

'REAL' 'PROC' NEXT EL(F,P); 'VAL' F,P; 'INT' F,P;
  NEXT EL:= TRANS EL(F,P,2,0);

'REAL' 'PROC' PREV EL(F,P); 'VAL' F,P; 'INT' F,P;
  PREV EL:= TRANS EL(F,P,3,0);

'PROC' WRITE EL(F,P,EL); 'VAL' F,P,EL; 'INT' F,P; 'REAL' EL;
  TRANS EL(F,P,1,EL);

'PROC' STANDARD PTR(F,P); 'VAL' F,P; 'INT' F,P;
  'IF' ¬STANDARD PTR 1(F,P) 'THEN' ERROR(ER CE);

'BOOL' 'PROC' STANDARD PTR 1(F,P); 'VAL' F,P; 'INT' F,P;
'BEGIN'  HARD CHECK ON F(F);
  'IF' P 'NE' BP ∧ P 'NE' WP ∧ P 'NE' EP 'THEN' ERROR(ER ST);
  'IF' PTRB(F,P) 'NE' 0 'THEN' ERROR(ER RE);
  SYS NOT;
  CREATE PTR SPACE(F,P,NO);
  INITIALIZE PTR(F,P,VAL OF PTR(F,'IF' P = WP 'THEN' BP 'ELSE' P));
  'IF' 'TRUE' 'THEN' STANDARD PTR 1:= 'TRUE' 'ELSE'
NO:  STANDARD PTR 1:= 'FALSE';
  SYS TON
'END';

'INT' 'PROC' NEW PTR(F,POS); 'VAL' F,POS; 'INT' F,POS;
'BEGIN' 'INT' P; NEW PTR:= P:= NEW PTR 1(F,POS);
  'IF' P<0 'THEN' ERROR(ER CE)
'END';

'INT' 'PROC' NEW PTR 1(F,POS); 'VAL' F,POS; 'INT' F,POS;
'BEGIN'  'INT' P;
  HARD CHECK ON F(F); HARD CHECK ON POS(F,POS);
  SYS NOT;
  P:= NEW PTR NUMBER(F,NO); CREATE PTR SPACE(F,P,NO);
  INITIALIZE PTR(F,P,POS); 'IF' 'TRUE' 'THEN'
  NEW PTR 1:= P 'ELSE'
NO:  NEW PTR 1:= ER CE;
  SYS TON
'END';

'PROC' DELETE PTR(F,P); 'VAL' F,P; 'INT' F,P;
'BEGIN' 'INT' ADP,ADN;
  HARD CHECK ON F AND P(F,P);
  ADN:= DESB(F) + NPTRS IN DES;
  ADP:= PADB(F) + (P-1)*PADCELL + PTRB IN PAD;
  SYS NOT;
  INCR(ADN,-1); DECR INT IN BIC(F,P);
  'IF' FETCH(ADN)>0 'THEN' DELETE CORE(ADP) 'ELSE'
  'BEGIN' STORE REF(CADB(F) + TRANSB IN CAD,FETCH REF(ADP));
    STORE REF(ADP,0)
  'END';
  SYS TON
'END';

'INT' 'PROC' VALUE OF PTR(F,P); 'VAL' F,P; 'INT' F,P;
'BEGIN'  HARD CHECK ON F(F);
  VALUE OF PTR:= 'IF' P OK(F,P) 'THEN'
```

```
117     VAL OF PTR(F,P) 'ELSE' -777
118   'END';
119
120   'INT' 'PROC' VALUE OF BP(F); 'VAL' F; 'INT' F;
121   'BEGIN'  HARD CHECK ON F(F);
122     VALUE OF BP:= VAL OF PTR(F,BP)
123   'END';
124
125   'INT' 'PROC' VALUE OF EP(F); 'VAL' F; 'INT' F;
126   'BEGIN'  HARD CHECK ON F(F);
127     VALUE OF EP:= VAL OF PTR(F,EP)
128   'END';
129
130   'PROC' RESET WP(F); 'VAL' F; 'INT' F;
131   'BEGIN'  HARD CHECK ON F AND P(F,WP);
132     SYS NOT;
133     DECR INT IN BIC(F,WP);
134     INITIALIZE PTR(F,WP,VAL OF PTR(F,BP));
135     SYS TON
136   'END';
137
138   'INT' 'PROC' FILE CLAIM(F); 'VAL' F; 'INT' F;
139   'BEGIN' 'INT' DES;
140     HARD CHECK ON F(F); DES:= DESB(F);
141     FILE CLAIM:= (FETCH(DES + NBLOCKS IN DES) - 1)*
142                  FETCH(DES + NELPB IN DES) - 1
143   'END';
144
145   'INT' 'PROC' FILE SPECIES(F); 'VAL' F; 'INT' F;
146     FILE SPECIES:= 'IF' F OK(F) 'THEN' FETCH(DESB(F) + SPEC IN DES)
147                    'ELSE' -777;
148
149   'BOOL' 'PROC' WORK PERMIT(F); 'VAL' F; 'INT' F;
150   'BEGIN'  HARD CHECK ON F(F);
151     WORK PERMIT:= FETCH(DESB(F) + WORK IN DES) > 0
152   'END';
153
154   'INT' 'PROC' IDF SYM(K,F); 'VAL' K,F; 'INT' K,F;
155   'BEGIN'  HARD CHECK ON F(F);
156     'IF' K < 0 ∨ K 'GE' IDFL * 3 'THEN' IDFSYM:= DEL SBL 'ELSE'
157     'BEGIN'  'INT' N,S;
158       N:= K '/' 3; S:= FETCH(DESB(F) + IDF IN DES + N);
159       N:= K - N * 3;
160       IDF SYM:= 'IF' N = 0 'THEN' S '/' 262144 'ELSE'
161         'IF' N=1 'THEN' BITSTRING(17,9,S) 'ELSE' BITSTRING(8,0,8)
162     'END'
163   'END';
164
165   'BOOL' 'PROC' NEW IDF(F,IDENT); 'VAL' F; 'INT' F;
166   'BEGIN'  'INT' 'ARRAY' IDF[1: IDFL]; 'BOOL'  SCRATCH; 'INT' K,DES,B;
167     MAKE IDF(IDF,IDENT); HARD CHECK ON F(F);
168     HARD CHECK ON WORK PERMIT(F);
169     SYS NOT; SCRATCH:= 'TRUE'; DES:= DESB(F);
170     'FOR' K:= 1 'STEP' 1 'UNTIL' IDFL 'DO'
171     SCRATCH:= SCRATCH ∧ IDF[K] = SCRATCH!IDF;
172     'IF' FETCH(DES + SCRATCH IN DES) > 0 'THEN'
173     'BEGIN'  'IF' ¬ SCRATCH 'THEN'
174       'BEGIN' 'IF' ¬SYS NO LONGER SCRATCH(FETCH(DES + NSEGM IN DES))
175         'THEN' 'GOTO' FALSE; B:= SYS CLAIM BLOCK; 'IF' B<0 'THEN'
176         'BEGIN'  SYS SCRATCH NOW(FETCH(DES + NSEGM IN DES));
```

```
177           'GOTO' FALSE
178         'END';
179         STORE(DES = SCRATCH IN DES, =777);
180         STORE(DES = BACKAD IN DES,B)
181       'END'
182     'END' 'ELSE'
183     'BEGIN'  'IF' SCRATCH 'THEN'
184       'BEGIN'  SYS SCRATCH NOW(FETCH(DES = NSEGM IN DES));
185         STORE(DES = SCRATCH IN DES, =777);
186         SYS DELETE BLOCK(FETCH(DES = BACKAD IN DES))
187       'END'
188     'END';
189     'FOR' K:= 1 'STEP' 1 'UNTIL' IDFL 'DO'
190     STORE(DES = IDF IN DES = K + 1,IDF[K]);
191     'IF' 'TRUE' 'THEN' NEW IDF:= 'TRUE' 'ELSE'
192   FALSE;  NEW IDF:= 'FALSE';
193     SYS TON
194   'END';
195
196   'BOOL' 'PROC' CLOSE FILE(F); 'VAL' F; 'INT' F;
197     CLOSE FILE:= CLOSE(F,'TRUE');
198
199   'BOOL' 'PROC' CLOSE FILE PUBLIC(F); 'VAL' F; 'INT' F;
200     CLOSE FILE PUBLIC:= CLOSE(F,'FALSE');
201
202
203   'COMMENT' OPEN/CLOSE ROUTINES ***************************************;
204
205
206   'INT' 'PROC' OPEN OLD FILE(IDENT,WORK); 'VAL' WORK; 'BOOL' WORK;
207   'BEGIN'  'INT' 'ARRAY' IDF[1:IDFL];
208     'INT' F,POS,BACKAD,K;
209     MAKE IDF(IDF,IDENT);
210     SYS NOT;
211     F:= SET UP FIRST PART SKELETON('FALSE',NO);
212     SET UP SECOND PART SKELETON(F,WP,NO);
213     'FOR' K:= 1 'STEP' 1 'UNTIL' IDFL 'DO'
214     STORE(DESB(F) = IDF IN DES = K + 1,IDF[K]);
215     GET ACCESS TO CATFILE;
216     SIMPLE OPEN FILE(F,FETCH(SYSVAR CATFILBACKAD),'TRUE');
217     POS:= POS IN CATFILE(F,IDF,WORK,FETCH(SYSVAR USER));
218     'IF' POS < 0 'THEN'
219     'BEGIN' DELETE CORE( FADB = F); F:= POS 'END' 'ELSE'
220     'BEGIN'  MARK INTEREST IN CATFILE(F,POS,WORK);
221       BACKAD:= BACKAD IN CATFILE(F,POS);
222       SIMPLE CLOSE FILE(F);
223       REOPEN FILE(F,BACKAD,WORK);
224       SHRINK SAD(F)
225     'END';
226     RETURN ACCESS TO CATFILE;
227     'IF' 'TRUE' 'THEN' OPEN OLD FILE:= F 'ELSE'
228   NO:  OPEN OLD FILE:= ER CE;
229     SYS TON
230   'END';
231
232   'PROC' SIMPLE OPEN FILE(F,BACKAD,WORK);
233         'VAL' F,BACKAD,WORK; 'INT' F,BACKAD; 'BOOL' WORK;
234   'BEGIN'  'INT' BIC,DES,CAT,K,AD,T;
235     BIC:= FETCH REF(BADB(F) = 1); DES:= DESB(F);
236     BIC FROM BACK(F,BIC,BACKAD);
```

```
237     DEMAND REST(BIC);
238     'IF' BACKAD 'NE' FETCH(SYSVAR CATF!BACKAD) 'THEN'
239     STORE(DESB(F) := BACKAD IN DES,BACKAD);
240     CAT:= BIC := INFO IN BIC + 1;
241     INITIALIZE DESCRIPTOR OLD FILE(F,CAT,WORK);
242     'FOR' K:= FETCH(DES := NSEGM IN DES) -1 'STEP' -1 'UNTIL' 0 'DO'
243     'BEGIN'  AD:= SADB(F) := SADCELL + K := SEGMAD IN SAD;
244       STORE(AD,FETCH(CAT := SEGMAD IN CAT := K));
245       'FOR' T:= NBITWRDS -1 'STEP' -1 'UNTIL' 0 'DO'
246       STORE(AD+SEGMAD IN SAD := BITWRD IN SAD := T,ALLNINC)
247     'END';
248     INITIALIZE PTR(F,WP,1)
249   'END';
250
251   'INT' 'PROC' SET UP FIRST PART SKELETON(NEW,ALARM);
252                'VAL' NEW; 'BOOL' NEW; 'LABEL' ALARM;
253   'BEGIN'  'INT' F;
254     F:= NEW FILENUMBER(ALARM);
255     CLAIM CORE(RHO,CADL,FADB := F,REL);
256     CLAIM CORE(PI,DESL,CADB(F) := DESB IN CAD,REL);
257     CLAIM CORE(PI,'IF' NEW 'THEN' NEXS*SADCELL 'ELSE' SADL,
258                CADB(F) := SADB IN CAD,REL);
259     STORE(DESB(F) := NPTRS IN DES,0);
260     STORE(DESB(F) := NBICS IN DES,0);
261     STORE(DESB(F) := NFREEBICS IN DES,0);
262     'IF' 'TRUE' 'THEN' SET UP FIRST PART SKELETON:= F 'ELSE'
263   REL:  'BEGIN'  DELETE CORE(FADB := F); 'GOTO' ALARM 'END'
264   'END';
265
266   'PROC' SET UP SECOND PART SKELETON(F,P,ALARM);
267          'VAL' F,P; 'INT' F,P; 'LABEL' ALARM;
268   'BEGIN'  CLAIM CORE(RHO,PADL,CADB(F) := PADB IN CAD,REL);
269     CLAIM BIC(F,REL); CLAIM CORE(PI,PTRL,CADB(F) := TRANSB IN CAD,REL);
270     CREATE PTR SPACE(F,P,REL);
271     'IF' 'FALSE' 'THEN'
272   REL:  'BEGIN'  DELETE CORE(FADB := F); 'GOTO' ALARM 'END'
273   'END';
274
275   'PROC' INITIALIZE DESCRIPTOR NEW FILE(F,SPECIES); 'VAL' F, SPECIES;
276          'INT' F,SPECIES;
277   'BEGIN' 'INT' DES,K; DES:= DESB(F);
278     STORE(DES := SPEC IN DES,SPECIES);
279     STORE(DES := BP IN DES,1);
280     STORE(DES := EP IN DES,1);
281     STORE(DES := NSEGM IN DES,1);
282     STORE(DES := OFFSET IN DES,1);
283     STORE(DES := NEW IN DES,+777);
284     STORE(DES := SCRATCH IN DES,+777);
285     STORE(DES := WORK IN DES,+777);
286     'FOR' K:= IDFL := 1 'STEP' -1 'UNTIL' 0 'DO'
287     STORE(DES := IDF IN DES := K,SCRATCH!DF);
288     INITIALIZE REST OF DESCRIPTOR(F)
289   'END';
290
291   'PROC' INITIALIZE DESCRIPTOR OLD FILE(F,CAT,WORK); 'VAL' F,CAT,WORK;
292          'INT' F,CAT; 'BOOL' WORK;
293   'BEGIN' 'INT' DES; DES:= DESB(F);
294     STORE(DES := SPEC IN DES,FETCH(CAT := SPEC IN CAT));
295     STORE(DES := BP IN DES,FETCH(CAT := BP IN CAT));
296     STORE(DES := EP IN DES,FETCH(CAT := EP IN CAT));
```

```
297     STORE(DES = NSEGM IN DES,FETCH(CAT = NSEGM IN CAT));
298     STORE(DES = OFFSET IN DES,FETCH(CAT = OFFSET IN CAT));
299     STORE(DES = NEW IN DES,=777);
300     STORE(DES = SCRATCH IN DES,=777);
301     STORE(DES = WORK IN DES,'IF' WORK 'THEN' *777 'ELSE' =777);
302     INITIALIZE REST OF DESCRIPTOR(F)
303   'END';
304
305   'PROC' INITIALIZE REST OF DESCRIPTOR(F); 'VAL' F; 'INT' F;
306   'BEGIN' 'INT' DES; DES:= DESB(F);
307     STORE(DES = NELPW IN DES, NELPW TO SPEC(FETCH(DES = SPEC IN DES)));
308     STORE(DES = BPEL IN DES, BPEL TO SPEC(FETCH(DES = SPEC IN DES)));
309     STORE(DES = NBLOCKS IN DES,NBPS*FETCH(DES = NSEGM IN DES));
310     STORE(DES= NELPB IN DES,NWPB*FETCH(DES = NELPW IN DES));
311     STORE(DES = LAST BLOCK IN DES,BLOCK TO POS(F,FETCH(DES = EP IN DES)
312           =1));
313     STORE(DES = NFREE IN DES,(FETCH(DES = NBLOCKS IN DES) = 1) *
314         FETCH(DES = NELPB IN DES) = 1 = FETCH(DES = EP IN DES)
315         + FETCH(DES = BP IN DES))
316   'END';
317
318   'PROC' REOPEN FILE(F,BACKAD,WORK); 'VAL' F,BACKAD,WORK;
319       'INT' F,BACKAD; 'BOOL' WORK;
320   'BEGIN' DECR INT IN BIC(F,WP); SIMPLE OPEN FILE(F,BACKAD,WORK)
321   'END';
322
323   'BOOL' 'PROC' CLOSE(F,PRIV); 'VAL' F,PRIV; 'INT' F; 'BOOL' PRIV;
324   'BEGIN' 'INT' DES,CATPOS,K;
325           'BOOL' WORK,NEW,SCRATCH,OK;
326           'INT' 'ARRAY' IDF,OLD IDF[1:IDFL];
327     HARD CHECK ON F(F); DES:= DESB(F); WORK:= FETCH(DES= WORK IN DES)>0;
328     SCRATCH:= FETCH(DES = SCRATCH IN DES)>0;
329     'IF' 'IF' =PRIV 'THEN' =WORK ^ SCRATCH 'ELSE' 'FALSE'
330     'THEN' ERROR(ER PC);
331     NEW:= FETCH(DES = NEW IN DES)>0;
332     SIMPLE CLOSE FILE(F);
333     SYS NOT;
334     'IF' SCRATCH 'THEN'
335     'BEGIN' DELETE ALL SEGMENTS(F);'IF' NEW 'THEN' 'GOTO' TRUE
336     'END' 'ELSE'
337     'BEGIN' 'IF' WORK 'THEN' MAKE ADM BLOCK(F,'FALSE');
338       RELEASE SECOND PART SKELETON(F)
339     'END';
340     'FOR' K:= 1 'STEP' 1 'UNTIL' IDFL 'DO'
341     IDF[K]:= FETCH(DES = IDF IN DES = K + 1);
342   ECCH:  SET UP SECOND PART SKELETON(F,WP,ECCH);
343     GET ACCESS TO CATFILE;
344     SIMPLE OPEN FILE(F,FETCH(SYSVAR CATFIBACKAD),'TRUE');
345     OK:= UPDATE CATFILE(F,IDF,DES,PRIV,WORK,NEW,SCRATCH);
346     SIMPLE CLOSE FILE(F);
347     RETURN ACCESS TO CATFILE;
348     'IF' OK 'THEN'
349   TRUE:  'BEGIN' DELETE CORE(FADB = F); CLOSE:= 'TRUE' 'END' 'ELSE'
350     'BEGIN' 'FOR' K:= 1 'STEP' 1 'UNTIL' IDFL 'DO'
351       OLD IDF[K]:= FETCH(DES = IDF IN DES = K + 1);
352       SYS IDF FANCY(F,OLD IDF,IDF);
353       'FOR' K:= 1 'STEP' 1 'UNTIL' IDFL 'DO'
354       STORE(DES = IDF IN DES = K + 1,IDF[K]);
355       REOPEN FILE(F,FETCH(DES = BACKAD IN DES),'TRUE');
356       CLOSE:= 'FALSE'
```

```
357     'END';
358     SYS TON
359   'END';
360
361   'PROC' RELEASE SECOND PART SKELETON(F); 'VAL' F; 'INT' F;
362   'BEGIN'
363     DELETE CORE(CADB(F) - PADB IN CAD);
364     STORE(DESB(F) - NPTRS IN DES,0)
365   'END';
366
367   'PROC' DELETE ALL SEGMENTS(F); 'VAL' F; 'INT' F;
368   'BEGIN' 'INT' SAD,K,DES; 'BOOL' SCRATCH;
369     DES:= DESB(F); SCRATCH:= FETCH(DES - SCRATCH IN DES) > 0;
370     SAD:= SADB(F);
371     'FOR' K:= SADCELL * (FETCH(DES - NSEGM IN DES) - 1)
372         'STEP' -SADCELL 'UNTIL' 0 'DO'
373     SYS DELETE SEGMENT(FETCH(SAD - K - SEGMAD IN SAD),
374         SCRATCH);
375     RELEASE SECOND PART SKELETON(F)
376   'END';
377
378   'PROC' SIMPLE CLOSE FILE(F); 'VAL' F; 'INT' F;
379   'BEGIN' 'INT' AD,DES,BIC; 'BOOL' WRITE;
380     DES:= DESB(F); WRITE:= FETCH(DES - SCRATCH IN DES)<0;
381   REP:  AD:= BADB(F);
382   LOCP:  BIC:= FETCH REF(AD - 1);
383     'IF' WRITE 'THEN'
384     'BEGIN'  'IF' FETCH(BIC - MOD IN BIC) > 0 'THEN'
385       BIC TO BACK(F,BIC,BACKAD OF BLOCK(F,FETCH(BIC - BLOCK IN BIC)))
386     'END' 'ELSE' DEMAND REST(BIC);
387     AD:= FETCH REF(AD); 'IF' AD 'NE' 0 'THEN' 'GOTO' LOCP
388     'ELSE' 'IF' WRITE 'THEN' 'BEGIN' WRITE := 'FALSE'; 'GOTO' REP 'END';
389     STORE(DES - LAST BLOCK IN DES,-1); TRY BIC RELEASE(F)
390   'END';
391
392
393   'COMMENT' CATALOGUE ROUTINES ***************************************;
394
395
396   'PROC' GET ACCESS TO CATFILE;
397   'BEGIN'
398   WAIT:  SYS EL; 'IF' FETCH(SYSVAR CATFINACC) > 0 'THEN'
399     'BEGIN' SYS LE; SYS SWAP; 'GOTO' WAIT 'END';
400     STORE(SYSVAR CATFINACC,+777);
401     SYS LE
402   'END';
403
404   'PROC' RETURN ACCESS TO CATFILE; STORE(SYSVAR CATFINACC,-777);
405
406   'PROC' POSITION(F,POS); 'VAL' F,POS; 'INT' F,POS;
407   LOCP:  'IF' VAL OF PTR(F,WP) 'NE' POS 'THEN'
408   'BEGIN'  'IF' VAL OF PTR(F,WP) > POS 'THEN'
409     PREV EL(F,WP) 'ELSE' NEXT EL(F,WP);
410     'GOTO' LOCP
411   'END';
412
413   'INT' 'PROC' BACKAD IN CATFILE(F,POS); 'VAL' F,POS; 'INT' F,POS;
414   'BEGIN'  POSITION(F,POS + 2);
415     BACKAD IN CATFILE:= NEXT EL(F,WP)
416   'END';
```

```
417
418   'INT' 'PROC' POS IN CATFILE(F,IDF,WORK,USER);
419   'VAL' F,WORK,USER; 'INT' F,USER; 'BOOL' WORK; 'ARRAY' IDF;
420   'BEGIN'  'INT' POS,FPOS,RPOS,USE,RUSE,OWNER;
421     RPOS:= -777;
422     FPOS:= POS:= LOOK UP(F,IDF);
423   EXAM:  'IF' POS > 0 'THEN'
424     'BEGIN' USE:= NEXT EL(F,WP); OWNER:= NEXT EL(F,WP);
425       'IF' USER = ABS(OWNER) 'THEN'
426       'BEGIN' 'IF' ( 'IF' USER 'NE' OWNER 'THEN' ( 'IF' WORK 'THEN' USE
427         'NE' 0 'ELSE' USE<0 ) 'ELSE' 'FALSE' ) 'THEN' POS:= ER NN;
428         'GOTO' EXIT
429       'END' 'ELSE'
430       'BEGIN' RPOS:= 'IF' OWNER<0 'THEN' POS 'ELSE' -POS;
431         SKIP REST OF CATDESCR(F); POS:= LOOK UP(F,IDF); 'GOTO' EXAM
432       'END'
433     'END';
434     POS:= 'IF' FPOS < 0 'THEN' ER UK 'ELSE'
435           'IF' RPOS < 0 'THEN' ER NY 'ELSE'
436           'IF' WORK     'THEN' ER NP 'ELSE'
437           'IF' RUSE < 0 'THEN' ER NN 'ELSE' RPOS;
438   EXIT:  POS IN CATFILE:= POS
439   'END';
440
441   'BOOL' 'PROC' UPDATE CATFILE(F,IDF,DES,PRIV,WORK,NEW,SCRATCH);
442       'VAL' F,DES,PRIV,WORK,NEW,SCRATCH; 'ARRAY' IDF;
443       'INT' F,DES; 'BOOL' PRIV,WORK,NEW,SCRATCH;
444   'BEGIN'  'INT' POS,K,USE; 'BOOL' OK;
445     OK:= 'TRUE';
446     'IF' SCRATCH 'THEN' 'BEGIN' POS:= FETCH(DES - CATPOS IN DES);
447                                 'GOTO' UPD IDF 'END';
448     'IF' -WORK  'THEN' 'BEGIN' POS:= FETCH(DES - CATPOS IN DES);
449                               'GOTO' UPD USE 'END';
450     'IF' - NEW 'THEN'
451     'BEGIN' POSITION(F,FETCH(DES - CATPOS IN DES) - IDFL);
452       'FOR' K:= 1 'STEP' 1 'UNTIL' IDFL 'DO'
453       WRITE EL(F,WP,SCRATCH[IDF]); RESET WP(F)
454     'END';
455   TRY:  POS:= POS IN CATFILE(F,IDF,PRIV,FETCH(SYSVAR USER));
456     'IF' POS > 0 'THEN'
457     'BEGIN' OK:= 'FALSE'; SYS FANCY IDF(IDF); 'GOTO' TRY 'END';
458     'BEGIN' 'INT' 'ARRAY' SCRIDF[1:IDFL];
459       'FOR' K:= IDFL 'STEP' -1 'UNTIL' 1 'DO'
460       SCRIDF[K]:= SCRATCH IDF; RESET WP(F);
461       POS:= POS IN CATFILE(F,SCRIDF,'TRUE',0)
462     'END';
463     'IF' POS < 0 'THEN'
464     'BEGIN' POS:= VAL OF PTR(F,EP) + IDFL;
465       DELETE PTR(F,WP); STANDARD PTR 1(F,EP);
466     'FOR' K:= 1 'STEP' 1 'UNTIL' DESCRIPTORL 'DO' WRITE EL(F,EP,0);
467       DELETE PTR(F,EP); STANDARD PTR 1(F,WP);
468     MAKE ADM BLOCK(F,'TRUE')
469     'END';
470   UPD IDF:  POSITION(F,POS - IDFL);
471     'FOR' K:= 1 'STEP' 1 'UNTIL' IDFL 'DO'
472     WRITE EL(F,WP,IDF[K]); NEXT EL(F,WP);
473     WRITE EL(F,WP,('IF' - PRIV ^ WORK 'THEN' -1 'ELSE' 1)
474         * FETCH(SYSVAR USER));
475     WRITE EL(F,WP,FETCH(DES - BACKAD IN DES));
476   UPD USE: POSITION(F,POS + 1); USE:= PREV EL(F,WP);
```

```
477     WRITE EL(F,WP,'IF' ¬ WORK 'THEN' USE = 1 'ELSE'
478               'IF' ¬ OK 'THEN' -777 'ELSE' 0);
479     UPDATE CATFILE:= OK
480   'END';
481
482   'INT' 'PROC' LOOK UP(F,IDF); 'VAL' F; 'INT' F; 'ARRAY' IDF;
483   'BEGIN'  'INT' K; 'BOOL' OK;
484   SEARCH ON: 'IF' VAL OF PTR(F,WP) 'GE' VAL OF PTR(F,EP) 'THEN'
485     LOOK UP:= -777 'ELSE'
486     'BEGIN'  OK:= 'TRUE';
487       'FOR' K:= 1 'STEP' 1 'UNTIL' IDFL 'DO'
488       OK:= OK ∧ IDF[K] = NEXT EL(F,WP);
489       'IF' ¬ OK 'THEN'
490       'BEGIN'  NEXT EL(F,WP); NEXT EL(F,WP);
491         SKIP REST OF CATDESCR(F);
492         'GOTO' SEARCH ON
493       'END';
494       LOOK UP:= VAL OF PTR(F,WP)
495     'END'
496   'END';
497
498   'PROC' SKIP REST OF CATDESCR(F); 'VAL' F; 'INT' F;
499       NEXT EL(F,WP);
500
501   'PROC' MAKE IDF(IDF,IDENT); 'ARRAY' IDF;
502   'BEGIN'  'INT' SYM,K,P,C,INT; 'BOOL' STRING;
503     STRING:= SYS IS STRING(IDENT);
504     'IF' ¬STRING ∧ ¬SYS IS INT PROC(IDENT) 'THEN' ERROR(ER WT);
505     K:= P:= C:= INT:= 0;
506   RESBL:  SYM:= 'IF' STRING 'THEN' STRINGSBL(K,IDENT)
507         'ELSE' PROCSBL(IDENT);
508     'IF' SYM = SPACESBL ∨ SYM = TABSBL ∨
509        SYM = NLCRSBL 'THEN' 'GOTO' RESBL;
510     'IF' SYM < 0 ∨ SYM > 35 'THEN' SYM:= DELSBL;
511   TREAT:  INT:= INT * 512 + SYM; C:= C + 1;
512     'IF' C = 3 'THEN'
513     'BEGIN'  P:= P + 1; IDF[P]:= INT; INT:= C:= 0 'END';
514     'IF' P 'NE' IDFL 'THEN'
515     'GOTO' 'IF' SYM = DELSBL 'THEN' TREAT 'ELSE' RESBL
516   'END';
517
518   'PROC' MAKE ADM BLOCK(F,CATF); 'VAL' F,CATF; 'INT' F; 'BOOL' CATF;
519   'BEGIN'  'INT' DES,BIC,CAT,K;
520     DES:= DESB(F); BIC:= FETCH REF(BADB(F) - 1);
521     CAT:= BIC - INFO IN BIC + 1;
522     STORE(CAT - SPEC IN CAT,FETCH(DES - SPEC IN DES));
523     STORE(CAT - BP IN CAT,FETCH(DES - BP IN DES));
524     STORE(CAT - EP IN CAT,FETCH(DES - EP IN DES));
525     STORE(CAT - NSEGM IN CAT,FETCH(DES - NSEGM IN DES));
526     STORE(CAT - OFFSET IN CAT,FETCH(DES - OFFSET IN DES));
527     'FOR' K:= FETCH(DES - NSEGM IN DES) -1 'STEP' -1 'UNTIL' 0 'DO'
528     STORE(CAT - SEGMAD IN CAT - K,FETCH(SADB(F) -
529           SADCELL * K - SEGMAD IN SAD));
530     BIC TO BACK(F,BIC,FETCH('IF' CATF 'THEN' SYSVAR CATF!BACKAD
531       'ELSE' DES - BACKAD IN DES));  DEMAND REST(BIC);
532   'END';
533
534   'PROC' MARK INTEREST IN CATFILE(F,POS,WORK); 'VAL' F,POS,WORK;
535   'INT' F,POS; 'BOOL' WORK;
536   'BEGIN'  'INT' USE;
```

```
537     POSITION(F,POS + 1); USE:= PREV EL(F,WP);
538     WRITE EL(F,WP,'IF' WORK 'THEN' -777 'ELSE' USE + 1);
539     STORE(DESB(F) - CATPOS IN DES,POS)
540   'END';
541
542
543   'COMMENT' POINTER ROUTINES ********************************************;
544
545
546   'PROC' INITIALIZE PTR(F,P,POS); 'VAL' F,P,POS; 'INT' F,P,POS;
547   'BEGIN' 'INT' PTR; PTR:= PTRB(F,P);
548     STORE(PTR - VAL IN PTR,POS);
549     STORE(PTR - WRD IN PTR,WRD IN BLOCK(F,POS));
550     STORE(PTR - ELT IN PTR,ELT IN WRD(F,POS));
551     STORE(PTR - BLOCK IN PTR,BLOCK TO POS(F,POS));
552     ASK FOR BLOCK(F,P);
553     ASSURE PRESENCE BLOCK(F,P)
554   'END';
555
556   'INT' 'PROC' NEW PTR NUMBER(F,ALARM); 'VAL' F; 'INT' F; 'LABEL' ALARM;
557   'BEGIN'  'INT' MAX,P;
558     MAX:= PMAX(F);
559     'FOR' P:= FFP 'STEP' 1 'UNTIL' MAX 'DO'
560     'IF' PTRB(F,P) = 0 'THEN' 'GOTO' FOUND;
561     EXTEND PAD(F,ALARM); P:= MAX + 1;
562   FOUND;  NEW PTR NUMBER:= P
563   'END';
564
565   'PROC' CREATE PTR SPACE(F,P,ALARM); 'VAL' F,P; 'INT' F,P; 'LABEL' ALARM;
566   'BEGIN'
567     CLAIM CORE(PI,PTRL,PADB(F) - (P-1)*PADCELL - PTRB IN PAD,ALARM);
568     'IF' FETCH(DESB(F) - NPTRS IN DES) > 0 'THEN' CLAIM BIC(F,NO) 'ELSE'
569     'BEGIN' STORE REF(PADB(F) - (P-1)*PADCELL - PTRB IN PAD,
570               FETCH REF(CADB(F) - TRANSB IN CAD));
571       STORE REF(CADB(F) - TRANSB IN CAD,0)
572     'END';
573     INCR(DESB(F) - NPTRS IN DES,1); 'IF' 'FALSE' 'THEN'
574   NO:  'BEGIN' DELETE CORE(PADB(F) - (P - 1) * PADCELL - PTRB IN PAD);
575          'GOTO' ALARM
576     'END'
577     'END';
578
579   'INT' 'PROC' VAL OF BP(F); 'VAL' F; 'INT' F;
580      VAL OF BP:= FETCH(DESB(F) - BP IN DES);
581
582   'INT' 'PROC' VAL OF EP(F); 'VAL' F; 'INT' F;
583      VAL OF EP:= FETCH(DESB(F) - EP IN DES);
584
585   'INT' 'PROC' VAL OF PTR(F,P); 'VAL' F,P; 'INT' P,P;
586      VAL OF PTR:=
587      'IF' P = BP 'THEN' VAL OF BP(F) 'ELSE'
588      'IF' P = EP 'THEN' VAL OF EP(F) 'ELSE'
589      FETCH(PTRB(F,P) - VAL IN PTR);
590
591   'INT' 'PROC' BLOCK OF PTR(F,P); 'VAL' F,P; 'INT' F,P;
592     BLOCK OF PTR:= FETCH(PTRB(F,P) - BLOCK IN PTR);
593
594
595   'COMMENT' ROUTINES FOR BLOCKS IN CORE **********************************;
596
```

```
597
598   'PROC' ASK FOR BLOCK(F,P); 'VAL' F,P; 'INT' F,P;
599   'BEGIN' 'INT' BLOCK,AD,BIC;
600   ECCH;  BLOCK:= BLOCK OF PTR(F,P);
601     'IF' BLOCK IN CORE(F,BLOCK) 'THEN'
602     'BEGIN' AD:= BADB(F);
603     LOCP;  BIC:= FETCH REF(AD - 1);
604       'IF' FETCH(BIC - BLOCK IN BIC) 'NE' BLOCK 'THEN'
605       'BEGIN'  AD:= FETCH REF(AD); 'GOTO' LOCP 'END'
606     'END' 'ELSE'
607     'BEGIN' 'IF' FETCH(DESB(F) - NFREEBICS IN DES) = 0 'THEN'
608       CLAIM BIC(F,ECCH); BIC:= CLAIM FREE BIC(F);
609       MARK BLOCK IN CORE(F,BLOCK,'TRUE');
610       STORE(BIC - BLOCK IN BIC,BLOCK);
611       'IF' BLOCK 'LE' LAST BLOCK(F) 'THEN'
612       BIC FROM BACK(F,BIC,BACKAD OF BLOCK(F,BLOCK))
613     'END';
614     INCR INT IN BIC(BIC);
615     STORE REF(PADB(F) - (P - 1) * PADCELL - BICB IN PAD,BIC)
616   'END';
617
618   'PROC' ONE BLOCK DOWN(F,BIC); 'VAL' F,BIC; 'INT' F,BIC;
619   'BEGIN' INCR(DESB(F) - LAST BLOCK IN DES,-1);
620     STORE(BIC - MOD IN BIC, -777)
621   'END';
622
623   'PROC' BIC TO BACK(F,BIC,BACK);
624         BICTRANS(F,BIC,BACK,'TRUE');
625
626   'PROC' BIC FROM BACK(F,BIC,BACK);
627         BICTRANS(F,BIC,BACK,'FALSE');
628
629   'PROC' BICTRANS(F,BIC,BACK,WRITE); 'VAL' F,BIC,BACK,WRITE;
630   'BOOL' WRITE; 'INT' F,BIC,BACK;
631   'BEGIN'
632     STORE(BIC - MOD IN BIC,-777);
633     SYS EL; INCR(BIC - NTRANS IN BIC,1); SYS LE;
634     'IF' WRITE 'THEN'
635     SYS  TO  DISK(BIC - BICL,INFOL,BACK,BIC - NTRANS IN BIC) 'ELSE'
636     SYS FROM DISK(BIC - BICL,INFOL,BACK,BIC - NTRANS IN BIC)
637   'END';
638
639   'PROC' FREE SEMIFREE;
640   'BEGIN' 'INT' F,DES,N,LAD,AD,BIC,NAD; 'BOOL' TRANS;
641     TRANS:= 'TRUE';
642     'FOR' F:= F 'WHILE' TRANS 'DO'
643     'BEGIN' TRANS:= 'FALSE'; 'FOR' F:= 1 'STEP' 1 'UNTIL' FMAX 'DO'
644       'BEGIN' DES:= DESB(F); N:= FETCH(DES - NPTRS IN DES);
645         N:= FETCH(DES - NBICS IN DES) - ('IF' N=0 'THEN' 1 'ELSE' N);
646         'IF' N>0 'THEN'
647         'BEGIN' LAD:= CADB(F) - BADB IN CAD; AD:= FETCH REF(LAD);
648   NEXT;   BIC:= FETCH REF(AD-1); 'IF' FETCH(BIC - INT IN BIC)=0 'THEN'
649           'BEGIN' 'IF' FETCH(BIC - MOD IN BIC)>0 'THEN'
650             'BEGIN' TRANS:= 'TRUE'; BIC TO BACK(F,BIC,BACKAD OF BLOCK(
651               F,FETCH(BIC - BLOCK IN BIC)))
652             'END' 'ELSE' 'IF' FETCH(BIC - NTRANS IN BIC)>0 'THEN'
653             TRANS:= 'TRUE' 'ELSE'
654             'BEGIN' NAD:= FETCH REF(AD); STORE REF(AD,0);
655               SYS DELETE(AD); STORE REF(LAD,NAD);
656               INCR(DES - NBICS IN DES,-1); N:= N-1;
```

```
657                 'IF' N>0 'THEN' AD:= LAD
658               'END'
659             'END';
660             LAD:= AD; AD:= FETCH REF(AD); 'IF' AD 'NE' 0 'THEN' 'GOTO' NEXT
661           'END'
662         'END'
663       'END'
664     'END';
665
666     'PROC' TRY BIC RELEASE(F); 'VAL' F; 'INT' F;
667     'BEGIN' 'INT' DES,N,FB,LB,BIC,B,LAD,AD,NAD;
668       DES:= DESB(F);
669       N:= FETCH(DES - NBICS IN DES) - FETCH(DES - NPTRS IN DES);
670       'IF' N>0 'THEN'
671       'BEGIN' FB:= BLOCK TO POS(F,FETCH(DES - BP IN DES));
672         LB:= FETCH(DES - LAST BLOCK IN DES);
673         LAD:= CADB(F) - BADB IN CAD; AD:= FETCH REF(LAD);
674     NEXT: BIC:= FETCH REF(AD - 1); 'IF' FETCH(BIC - INT IN BIC) = 0
675         'THEN'
676         'BEGIN' B:= FETCH(BIC - BLOCK IN BIC);
677           'IF' 'IF' 'IF' B<0 'THEN' 'FALSE' 'ELSE' B<FB ∨ B>LB 'THEN'
678             FETCH(BIC - NTRANS IN BIC) = 0 'ELSE' 'FALSE' 'THEN'
679           'BEGIN' 'IF' N=1 ∧ FETCH(DES - NPTRS IN DES) = 0 'THEN'
680             'BEGIN' INCR(DES - NFREEBICS IN DES,1);
681               STORE(BIC - BLOCK IN BIC,-777)
682             'END' 'ELSE'
683             'BEGIN' NAD:= FETCH REF(AD); STORE REF(AD,0);
684               SYS DELETE(AD); STORE REF(LAD,NAD);
685               INCR(DES - NBICS IN DES,-1); N:= N - 1;
686               'IF' N>0 'THEN' AD:= LAD
687             'END'
688           'END'
689         'END';
690         LAD:= AD; AD:= FETCH REF(AD); 'IF' AD 'NE' 0 'THEN' 'GOTO' NEXT
691       'END'
692     'END';
693
694     'INT' 'PROC' BLOCK OF BIC(BIC); 'VAL' BIC; 'INT' BIC;
695       BLOCK OF BIC:= FETCH(BIC - BLOCK IN BIC);
696
697     'PROC' ASSURE PRESENCE BLOCK(F,P); 'VAL' F,P; 'INT' F,P;
698       DEMAND REST(BICB(F,P));
699
700     'PROC' DEMAND REST(BIC); 'VAL' BIC; 'INT' BIC;
701     WAIT: 'IF' FETCH(BIC - NTRANS IN BIC) 'NE' 0 'THEN' 'GOTO' WAIT;
702
703     'INT' 'PROC' CLAIM BIC(F,AL); 'VAL' F; 'INT' F; 'LABEL' AL;
704     'BEGIN' 'INT' BIC;
705       EXTEND BAD(F,AL);
706       BIC:= CLAIM CORE(PI,BICL,BADB(F) - 1,NO);
707       INCR(DESB(F) - NBICS IN DES,1);
708       INCR(DESB(F) - NFREEBICS IN DES,1);
709       INITIALIZE BIC(BIC);
710       'IF' 'TRUE' 'THEN' CLAIM BIC:= BIC 'ELSE'
711     NO: 'BEGIN' BIC:= BADB(F);
712           STORE(CADB(F) - BADB IN CAD,FETCH REF(BIC)); STORE REF(BIC,0);
713           SYS DELETE(BIC); 'GOTO' AL
714         'END'
715     'END';
716
```

```
'INT' 'PROC' CLAIM FREE BIC(F); 'VAL' F; 'INT' F;
'BEGIN' 'INT' AD,BIC;
  AD:= BADB(F);
REP:  BIC:= FETCH REF(AD + 1); 'IF' FETCH(BIC + BLOCK IN BIC) > 0 'THEN'
  'BEGIN' AD:= FETCH REF(AD); 'GOTO' REP 'END';
  INCR(DESB(F) + NFREEBICS IN DES,-1);
  CLAIM FREE BIC:= BIC
'END';

'PROC' INITIALIZE BIC(BIC); 'VAL' BIC; 'INT' BIC;
'BEGIN'  STORE(BIC + MOD IN BIC, -777);
  STORE(BIC + INT IN BIC,0);
  STORE(BIC + NTRANS IN BIC, 0);
  STORE(BIC + BLOCK IN BIC, -777)
'END';

'PROC' INCR INT IN BIC(BIC); 'VAL' BIC; 'INT' BIC;
'BEGIN' INCR(BIC + INT IN BIC,1);
  SYS INCR REFCNT(BIC)
'END';

'PROC' DECR INT IN BIC(F,P); 'VAL' F,P; 'INT' F,P;
'BEGIN' 'INT' AD,INT,BIC,B;
  BIC:= BICB(F,P); AD:= BIC + INT IN BIC;
  INT:= FETCH(AD) - 1; STORE(AD,INT); SYS DECR REFCNT(BIC);
  STORE REF(PADB(F) +(P-1)*PADCELL + BICB IN PAD,0);
  'IF' INT=0 'THEN'
  'BEGIN' 'IF' FETCH(BIC + MOD IN BIC) > 0
  'THEN' BIC TO BACK(F,BIC,BACKAD OF BLOCK(F,BLOCK OF PTR(F,P)))
  'ELSE' 'IF' P=BP 'THEN' TRY BIC RELEASE(F) 'ELSE'
  'IF' P=EP 'THEN'
    'BEGIN'  B:= BLOCK OF PTR(F,P); 'IF' B>FETCH(DESB(F) + LAST
      BLOCK IN DES) 'THEN' 'BEGIN' MARK BLOCK IN CORE(F,B,'FALSE');
      TRY BIC RELEASE(F) 'END'
    'END'
  'END'
'END';

'INT' 'PROC' BACKAD OF BLOCK(F,BLOCK); 'VAL' F,BLOCK; 'INT' F,BLOCK;
  BACKAD OF BLOCK:= SYS COMPUTE BACKAD(
  FETCH(SADB(F) + REDUCED BNUMB(F,BLOCK)/NBPS*SADCELL
  +SEGMAD IN SAD), REMAINDER(BLOCK + 1,NBPS));

'INT' 'PROC' REDUCED BNUMB(F,B); 'VAL' F,B; 'INT' F,B;
'BEGIN'  'INT' DES,RB,RRB; DES:= DESB(F);
  RB:= B + FETCH(DES + OFFSET IN DES);
  RRB:= RB + FETCH(DES + NBLOCKS IN DES);
  REDUCED BNUMB:= 'IF' RRB 'GE' 0 'THEN' RRB 'ELSE' RB
'END';

'PROC' CONSIDER OFFSET(F,B); 'VAL' F,B; 'INT' F,B;
'BEGIN' 'INT' DES; DES:= DESB(F);
  'IF' REDUCED BNUMB(F,B) = 0 'THEN'
  INCR(DES + OFFSET IN DES,FETCH(DES + NBLOCKS IN DES));
'END';

'PROC' MARK BLOCK IN CORE(F,B,IN CORE); 'VAL' F,B, IN CORE;
    'INT' F,B; 'BOOL' IN CORE;
'BEGIN' 'INT' AD,W;
  B:= REDUCED BNUMB(F,B);
```

```
777     W:= B ↓ NBPW;
778     AD:= SADB(F) + SADCELL * (B↓NBPS) + BITWRD IN SAD + W;
779     B:= B - W*NBPW;
780     STORE(AD,SET('IF' IN CORE 'THEN' 0 'ELSE' 1,B,B,FETCH(AD)));
781   'END';
782
783   'BOOL' 'PROC' BLOCK IN CORE(F,B); 'VAL' F,B; 'INT' F,B;
784   'BEGIN' 'INT' W;
785     B:= REDUCED BNUMB(F,B);
786     W:= B ↓ NBPW;
787     BLOCK IN CORE:= BIT(B - W*NBPW, FETCH(SADB(F) + SADCELL *
788                       (B↓NBPS) + BITWRD IN SAD + W)) = 0
789   'END';
790
791
792   'COMMENT' ROUTINES FOR STORAGE ADMINISTRATION **************************;
793
794
795   'INT' 'PROC' CLAIM CORE(GEN,L,REFDES,ALARM); 'VAL' GEN,L; 'LABEL' ALARM;
796   'INT' GEN,L,REFDES;
797   'BEGIN'  'INT' AD,K;
798     AD:= SYS CLAIM('IF' GEN = LAB 'THEN' 2 'ELSE' L + 1);
799     'IF' AD < 0 'THEN' 'GOTO' ALARM;
800     STORE(AD,SYS GENWRD(GEN,L,1));
801     STORE REF(REFDES,AD);
802     'IF' GEN=LAB 'THEN' STORE(AD + 1,0) 'ELSE'
803     'IF' GEN = RHO 'THEN'
804     'FOR' K:= 1 'STEP' 1 'UNTIL' L 'DO'
805     STORE(AD + K,0);
806     CLAIM CORE:= AD
807   'END';
808
809   'PROC' DELETE CORE(AD); 'VAL' AD; 'INT' AD;
810   'BEGIN' SYS DELETE(FETCH REF(AD));
811     STORE REF(AD,0)
812   'END';
813
814   'INT' 'PROC' NEW FILENUMBER(NO); 'LABEL' NO;
815   'BEGIN'  'INT' FAD,F;
816     FAD:= FADB; 'IF' FAD = 0 'THEN'
817     'BEGIN'  INITIALIZE FILESYSTEM; FAD:= FADB;
818       'IF' FAD = 0 'THEN' 'GOTO' NO
819     'END';
820     'FOR' F:= 1 'STEP' 1 'UNTIL' FMAX 'DO'
821     'IF' CADB(F) = 0 'THEN' 'GOTO' FOUND; F:= FMAX + 1;
822     EXTEND FAD(NO);
823   FOUND;  NEW FILENUMBER:= F
824   'END';
825
826   'PROC' EXTEND PAD(F,ALARM); 'VAL' F; 'INT' F; 'LABEL' ALARM;
827       EXTEND ADM(PADB(F),NEXP * PADCELL,CADB(F) + PADB IN CAD,ALARM);
828
829   'PROC' EXTEND FAD(ALARM); 'LABEL' ALARM;
830        EXTEND ADM(FADB,NEXF,SYSVAR FILEHANDLE,ALARM);
831
832   'PROC' EXTEND SAD(F,ALARM); 'VAL' F; 'INT' F; 'LABEL' ALARM;
833        EXTEND ADM(SADB(F),NEXS * SADCELL,CADB(F) + SADB IN CAD,ALARM);
834
835   'PROC' EXTEND ADM(BASE,EXTRA,REFDES,ALARM); 'VAL' BASE,EXTRA;
836       'INT' BASE,EXTRA,REFDES; 'LABEL' ALARM;
```

```
837   'BEGIN'  'INT' NEWB,K,L;
838     NEWB:= SYS EXTEND(BASE,EXTRA); 'IF' NEWB<0 'THEN' 'GOTO' ALARM;
839     STORE REF(REFDES,NEWB); L:= SYS LENGTH(NEWB);
840     'FOR' K:= 1 'STEP' 1 'UNTIL' EXTRA 'DO'
841     STORE(NEWB - L + K - 1,0)
842   'END';
843
844   'PROC' EXTEND BAD(F,ALARM); 'VAL' F; 'INT' F; 'LABEL' ALARM;
845   'BEGIN' 'INT' AD,HANDLE,H;
846     HANDLE:= CADB(F) - BADB IN CAD; H:= FETCH(HANDLE);
847     AD:= CLAIM CORE(LAB,2,HANDLE ,ALARM);
848     STORE REF(AD,H)
849   'END';
850
851   'PROC' SHRINK SAD(F); 'VAL' F; 'INT' F;
852        SYS SHRINK(SADB(F),(FETCH(DESB(F) - NSEGM IN DES) +
853            ('IF' FETCH(DESB(F) - WORK IN DES) > 0 'THEN' NEXS
854             'ELSE' 0)) * SADCELL);
855
856
857   'COMMENT' ROUTINE HANDLING 1 ELEMENT OF A FILE ************************;
858
859
860   'REAL' 'PROC' TRANS EL(F,P,KIND,EL);'VAL' F,P,KIND,EL;
861   'INT' F,P,KIND; 'REAL' EL;
862   'COMMENT' KIND = 1: WRITE EL,
863                    2: NEXT EL,
864                    3: PREV EL;
865   'BEGIN'  'INT'FAD,CAD,DES,PAD,PTR,BIC,ADP,PVAL,BPVAL,EPVAL,
866     NELPW,ELT,WRD,AD,ELM,BPEL,LOW,NFREE,ADFREE,BLOCK;
867     'BOOL' EXTEND;
868   HARD CHECK ON F;
869     FAD:= FADB; 'IF' FAD = 0 'THEN' ERROR(ER NF);
870     'IF' F < 1 ∨ F > SYS LENGTH(FAD) 'THEN' ERROR(ER WF);
871     CAD:= FETCH REF(FAD - F); 'IF' CAD = 0 'THEN' ERROR(ER WF);
872   HARD CHECK ON P;
873     PAD:= FETCH REF(CAD - PADB IN CAD);
874     'IF' P<1 ∨ P>SYS LENGTH(PAD)/PADCELL 'THEN' ERROR(ER WP);
875     ADP:= PAD - (P - 1) * PADCELL;
876     PTR:= FETCH REF(ADP - PTRB IN PAD);
877     'IF' PTR = 0 'THEN' ERROR(ER WP);
878   HARD CHECK ON WORK PERMIT IF NEEDED;
879     DES:= FETCH REF(CAD - DESB IN CAD);
880     'IF' 'IF' 'IF' KIND>1 'THEN' ('IF' P 'NE' EP 'THEN' P=BP 'ELSE'
881     'TRUE') 'ELSE' 'TRUE' 'THEN' FETCH(DES - WORK IN DES)<0
882     'ELSE' 'FALSE' 'THEN' ERROR(ER NW);
883   HARD CHECK ON PTR VALUE;
884     PVAL:= FETCH(PTR - VAL IN PTR) - ('IF' KIND = 3 'THEN' 1 'ELSE' 0);
885     BPVAL:= FETCH(DES - BP IN DES);
886     EPVAL:= FETCH(DES - EP IN DES);
887     'IF' PVAL < BPVAL 'THEN' ERROR(ER PL);
888     'IF' 'IF' PVAL 'GE' EPVAL 'THEN' P 'NE' EP 'ELSE' 'FALSE'
889     'THEN' ERROR(ER PH);
890   PRELIMINARY ACTIONS DONE;
891     SYS NOT; BIC:= FETCH REF(ADP - BICB IN PAD);
892     NELPW:= FETCH(DES - NELPW IN DES);
893     ELT:= FETCH(PTR - ELT IN PTR);
894     WRD:= FETCH(PTR - WRD IN PTR);
895     EXTEND:= 'FALSE';
896   SCATTER ON KIND;
```

```
897     'IF' KIND = 3 'THEN' 'GOTO' STEP PTR;
898     'IF' 'IF' WRD = 0 'THEN' ELT = 0 'ELSE' 'FALSE' 'THEN'
899     'BEGIN'  ASSURE PRESENCE BLOCK(F,P); 'IF' P = EP 'THEN'
900       STORE(DES - LAST BLOCK IN DES,FETCH(PTR - BLOCK IN PTR))
901     'END';
902   READWRITE:
903     AD:= BIC - INFO IN BIC - WRD;
904     'IF' KIND = 1 'THEN' 'GOTO' WRITE ELEMENT;
905   READ ELEMENT:
906     'IF' NELPW=0 'THEN' TRANS EL:= FFETCH(AD - 1) 'ELSE'
907     'BEGIN'  ELM:= FETCH(AD); 'IF' NELPW > 1 'THEN'
908       'BEGIN'  BPEL:= FETCH(DES -BPEL IN DES);
909         LOW:= ELT * BPEL;
910         TRANS EL:= BITSTRING(LOW + BPEL - 1,LOW,ELM)
911       'END' 'ELSE' TRANS EL:= ELM
912     'END';
913     'IF' P = BP ∨ P = EP 'THEN'
914     'BEGIN'  AD:= DES - NFREE IN DES;
915       STORE(AD,FETCH(AD) + 1)
916     'END';
917     'GOTO' 'IF' KIND = 2 'THEN' STEP PTR 'ELSE' EXIT TRANS;
918   WRITE ELEMENT:
919     'IF' P = EP 'THEN'
920     'BEGIN'  ADFREE:= DES - NFREE IN DES;
921       NFREE:= FETCH(ADFREE);
922       'IF' NFREE > 0 'THEN'
923       'BEGIN'  STORE(ADFREE,NFREE - 1);
924         EXTEND:= NFREE = 1
925       'END' 'ELSE' 'BEGIN' EXTEND:= - EXTEND FILE(F); SYS TON;
926                 'IF' - EXTEND 'THEN' 'GOTO' RETRY 'ELSE' ERROR(ERFE)
927                   'END'
928     'END' 'ELSE' 'IF' P = BP 'THEN'
929     'BEGIN'  ADFREE:= DES - NFREE IN DES;
930       STORE(ADFREE,FETCH(ADFREE) + 1); 'GOTO' STEP PTR
931     'END';
932     'IF' NELPW=0 'THEN' SSTORE(AD - 1,EL) 'ELSE'
933     'BEGIN'  ELM:= TAIL OF(EL); 'IF' NELPW > 1 'THEN'
934       'BEGIN'  BPEL:= FETCH(DES - BPEL IN DES);
935         LOW:= ELT * BPEL; STORE(AD,SET(BITSTRING(BPEL - 1,
936         0,ELM),LOW + BPEL - 1,LOW,FETCH(AD)))
937       'END' 'ELSE' STORE(AD,ELM)
938     'END'; STORE(BIC - MOD IN BIC,+777);
939   STEP PTR:
940     'IF' KIND < 3 'THEN' PVAL:= PVAL + 1;
941     STORE(PTR - VAL IN PTR,PVAL);
942     'IF' P = EP 'THEN' STORE(DES - EP IN DES,PVAL) 'ELSE'
943     'IF' P = BP 'THEN' STORE(DES - BP IN DES,PVAL);
944   STEP PTR ELT;
945     'IF' NELPW > 1 'THEN'
946     'BEGIN'  'IF' KIND < 3 'THEN'
947       'BEGIN'  ELT:= ELT - 1; 'IF' ELT 'GE' 0 'THEN'
948         'BEGIN'  STORE(PTR - ELT IN PTR,ELT);
949           'GOTO' EXIT TRANS
950         'END' 'ELSE' STORE(PTR - ELT IN PTR,NELPW - 1)
951       'END' 'ELSE'
952       'BEGIN'  ELT:= ELT + 1; 'IF' ELT < NELPW 'THEN'
953         'BEGIN'  STORE(PTR - ELT IN PTR,ELT);
954           'GOTO' READWRITE
955         'END' 'ELSE'
956         'BEGIN'  STORE(PTR - ELT IN PTR,0);
```

```
957           ELT:= 0
958         'END'
959       'END'
960     'END';
961   STEP PTR WRD:
962     'IF' KIND < 3 'THEN'
963     'BEGIN'  WRD:= WRD + ('IF' NELPW = 0 'THEN' 2 'ELSE' 1);
964       'IF' WRD < INFOL 'THEN'
965       'BEGIN'  STORE(PTR - WRD IN PTR, WRD);
966         'GOTO' EXIT TRANS
967       'END' 'ELSE' STORE(PTR - WRD IN PTR,0)
968     'END' 'ELSE'
969     'BEGIN'  WRD:= WRD - ('IF' NELPW = 0 'THEN' 2 'ELSE' 1);
970       'IF' WRD 'GE' 0 'THEN'
971       'BEGIN'  STORE(PTR - WRD IN PTR,WRD);
972         'GOTO' READWRITE
973       'END' 'ELSE'
974       'BEGIN'  WRD:= INFOL - ('IF' NELPW = 0 'THEN' 2 'ELSE' 1);
975         STORE(PTR - WRD IN PTR,WRD)
976       'END'
977     'END';
978   STEP PTR BLOCK:
979     AD:= PTR - BLOCK IN PTR;
980     BLOCK:= FETCH(AD) + ('IF' KIND = 3 'THEN' -1 'ELSE' 1);
981     'IF' P = BP 'THEN'
982     'BEGIN' MARK BLOCK IN CORE(F,BLOCK-1,'FALSE');
983       STORE(BIC - MOD IN BIC,-777);
984       CONSIDER OFFSET(F,BLOCK)
985     'END' 'ELSE' 'IF' 'IF' P=EP 'THEN' KIND=3 'ELSE' 'FALSE' 'THEN'
986     ONE BLOCK DOWN(F,BIC);
987     DECR INT IN BIC(F,P); STORE(AD,BLOCK);
988     ASK FOR BLOCK(F,P);
989     'IF' KIND = 3 'THEN'
990   RETRY:
991     'BEGIN' CAD:= FETCH REF(FADB - F);
992       DES:= FETCH REF(CAD - DESB IN CAD);
993       PAD:= FETCH REF(CAD - PADB IN CAD);
994       ADP:= PAD - (P - 1) * PADCELL;
995       PTR:= FETCH REF(ADP - PTRB IN PAD);
996       BIC:= FETCH REF(ADP - BICB IN PAD);
997       ASSURE PRESENCE BLOCK(F,P);
998       'GOTO' READWRITE
999     'END';
1000  EXIT TRANS:
1001    'IF' EXTEND 'THEN' EXTEND FILE(F);
1002    SYS TON
1003  'END';
1004
1005
1006  'COMMENT' ROUTINES FOR FILE EXTENSION ***********************************;
1007
1008
1009  'BOOL' 'PROC' EXTEND FILE(F); 'VAL' F; 'INT' F;
1010  'BEGIN' 'INT' SEGMAD,DES;
1011    'IF' FETCH(DESB(F) - NSEGM IN DES) =
1012      SYS LENGTH(SADB(F))/SADCELL 'THEN' EXTEND SAD(F,NO);
1013    DES:= DESB(F);
1014    SEGMAD:= SYS CLAIM SEGMENT(FETCH(DES - SCRATCH IN DES)>0);
1015    'IF' SEGMAD < 0 'THEN'
1016  NO:  EXTEND FILE:= 'FALSE' 'ELSE'
```

```
1017      'BEGIN' 'INT' FB,LB,PINS,N,L,K,SAD,NSAD,NS,LAD;
1018        'BOOL' FIRST;
1019        EXTEND FILE:= 'TRUE';
1020        FB:= BLOCK TO POS(F,FETCH (DES - BP IN DES));
1021        LB:= FETCH(DES - LAST BLOCK IN DES);
1022        PINS:= REMAINDER(FB - 1,NBPS);
1023        FIRST:= PINS < NBPS/2;
1024        N:= 'IF' FIRST 'THEN' PINS 'ELSE' NBPS - PINS;
1025        L:= 'IF' FIRST 'THEN' LB - N + 1 'ELSE' FB;
1026      TRANSFER OF INFO BLOCKS;
1027        INITIALIZE TRANSFER(F);
1028        'FOR' K:= 1 'STEP' 1 'UNTIL' N 'DO'
1029        TRANSFER(F,L + K - 1,
1030        SYS COMPUTE BACKAD(SEGMAD,'IF' FIRST 'THEN' K - 1
1031        'ELSE' NBPS + K - N - 1));
1032        AFTERMATH TRANSFER(F);
1033      TRANSFER OF ADM;
1034        SAD:= SADB(F); NSAD:= DES - NSEGM IN DES;
1035        NS:= FETCH(NSAD); STORE(NSAD,NS + 1);
1036        SHIFT SLICES TO RIGHT(SAD - 1,SADCELL,NS,
1037        (FB - FETCH(DES - OFFSET IN DES))1 NBPS);
1038        LAD:= SAD - NS * SADCELL;
1039        'FOR' K:= SADCELL - 1 'STEP' -1 'UNTIL' 0 'DO'
1040        STORE(LAD - 1 - K,FETCH(SAD - 1 - K));
1041        STORE(('IF' FIRST 'THEN' LAD 'ELSE' SAD) - SEGMAD IN SAD,SEGMAD);
1042        STORE(DES - OFFSET IN DES,(FB-1)'/'NBPS*NBPS+1);
1043        INCR(DES - NFREE IN DES,NBPS*FETCH(DES - NELPB IN DES));
1044        INCR(DES - NBLOCKS IN DES,NBPS);
1045        L:= FETCH(DES - NBLOCKS IN DES) + FB - 1;
1046        'FOR' K:= LB + 1 'STEP' 1 'UNTIL' L 'DO'
1047        MARK BLOCK IN CORE(F,K,'FALSE')
1048      'END'
1049    'END';
1050
1051    'PROC' INITIALIZE TRANSFER(F); 'VAL' F; 'INT' F;
1052    'BEGIN' 'INT' BIC;
1053      BIC:= FETCH REF(BADB(F) - 1);
1054      'IF' FETCH(BIC - INT IN BIC) > 0 ∧ FETCH(BIC - MOD IN BIC) > 0
1055      'THEN' BIC TO BACK(F,BIC,BACKAD OF BLOCK(F,FETCH(BIC-BLOCK IN BIC)));
1056      DEMAND REST(BIC);
1057      STORE REF(CADB(F) - TRANSB IN CAD,BIC);
1058      SYS INCR REFCNT(BIC);
1059      STORE(DESB(F) - TRANSCOR IN DES, -777)
1060    'END';
1061
1062    'PROC' AFTERMATH TRANSFER(F); 'VAL' F; 'INT' F;
1063    'BEGIN'  'INT' BIC,BLOCK,BACK;
1064      BIC:= FETCH REF(CADB(F) - TRANSB IN CAD);
1065      STORE REF(CADB(F) - TRANSB IN CAD,0);
1066      SYS DECR REFCNT(BIC);
1067      'IF' FETCH(BIC - INT IN BIC) > 0 'THEN'
1068      BIC FROM BACK(F,BIC,BACKAD OF BLOCK(F,FETCH(BIC - BLOCK IN BIC)));
1069      BACK:= FETCH(DESB(F) - TRANSCOR IN DES);
1070      'IF' BACK > 0 'THEN' BIC TO BACK(F,BIC,BACK);
1071      DEMAND REST(BIC)
1072    'END';
1073
1074    'PROC' TRANSFER(F,B,BACK); 'VAL' F,B,BACK; 'INT' F,B,BACK;
1075    'BEGIN'  'INT' BIC,AD;
1076      'IF' - BLOCK IN CORE(F,B) 'THEN'
```

```
1077      'BEGIN'  BIC:= FETCH REF(CADB(F) - TRANSB IN CAD);
1078        BIC FROM BACK(F,BIC,BACKAD OF BLOCK(F,B))
1079      'END' 'ELSE'
1080      'BEGIN' AD:= BADB(F);
1081      LOCP:  BIC:= FETCH REF(AD - 1);
1082        'IF' FETCH(BIC - BLOCK IN BIC) 'NE' B 'THEN'
1083        'BEGIN'  AD:= FETCH REF(AD); 'GOTO' LOCP 'END';
1084        'IF' BIC = FETCH REF(CADB(F) - TRANSB IN CAD) 'THEN'
1085        'BEGIN'  STORE(DESB(F) - TRANSCOR IN DES,
1086                 BACKAD OF BLOCK(F,B)); 'GOTO' POSTPONE
1087        'END'
1088      'END';
1089      BIC TO BACK(F,BIC,BACK);
1090    POSTPONE:
1091    'END';
1092
1093    'PROC' SHIFT SLICES TO RIGHT(UPAD,SLICEL,NSLICES,SHIFT);
1094    'VAL' UPAD,SLICEL,NSLICES,SHIFT; 'INT' UPAD,SLICEL,NSLICES,SHIFT;
1095    'IF' SHIFT > 0 ∧ SHIFT < NSLICES 'THEN'
1096    'BEGIN'  'INT' N,K,L; N:= SGCD(NSLICES,SHIFT);
1097      L:= SLICEL * NSLICES; SHIFT:= SHIFT * SLICEL;
1098      'FOR' K:= 1 'STEP' 1 'UNTIL' N 'DO'
1099      ONE CYCLE TO RIGHT(UPAD - K + 1,UPAD,L,SLICEL,SHIFT)
1100    'END';
1101
1102    'PROC' ONE CYCLE TO RIGHT(START,UP,L,SL,SHIFT);
1103    'VAL' START,UP,L,SL,SHIFT; 'INT' START,UP,L,SL,SHIFT;
1104    'BEGIN' 'INT' LOW,D,K,HERE,PREV;
1105      LOW:= UP - L + 1; D:= L - SHIFT;
1106      'FOR' K:= 1 'STEP' 1 'UNTIL' SL 'DO'
1107      STORE(LOW - K,FETCH(START - K * 1)); HERE:= START;
1108    LOOP:  PREV:= HERE - D; 'IF' PREV < LOW 'THEN' PREV:= PREV + L;
1109      'IF' PREV 'NE' START 'THEN'
1110      'BEGIN'  'FOR' K:= 0 'STEP' 1 'UNTIL' SL - 1 'DO'
1111        STORE(HERE - K,FETCH(PREV - K));
1112        HERE:= PREV; 'GOTO' LOOP
1113      'END';
1114      'FOR' K:= 1 'STEP' 1 'UNTIL' SL 'DO'
1115      STORE(HERE - K + 1,FETCH(LOW - K))
1116    'END';
1117
1118    'INT' 'PROC' SGCD(A,B); 'VAL' A,B; 'INT' A,B;
1119    LOCP:  'IF' A > B 'THEN' 'BEGIN' A:= A - B; 'GOTO' LOCP 'END' 'ELSE'
1120      'IF' A < B 'THEN' 'BEGIN' B:= B - A; 'GOTO' LOCP 'END' 'ELSE'
1121      SGCD:= A;
1122
1123
1124    'COMMENT' MISCELLANEOUS ROUTINES ****************************************;
1125
1126
1127    'INT' 'PROC' WRD IN BLOCK(F,POS); 'VAL' F,POS; 'INT' F,POS;
1128    'BEGIN'  'INT' NEL; NEL:= NELPW(F);
1129      WRD IN BLOCK:= REMAINDER('IF' NEL = 0 'THEN' POS*2
1130          'ELSE' POS'/' NEL,NWPB)
1131    'END';
1132
1133    'INT' 'PROC' ELT IN WRD(F,POS); 'VAL' F,POS; 'INT' F,POS;
1134    'BEGIN'  'INT' NEL; NEL:= NELPW(F);
1135      ELT IN WRD:= REMAINDER(POS,'IF' NEL > 1 'THEN' NEL 'ELSE' POS)
1136    'END';
```

```
1137
1138    'INT' 'PROC' BLOCK TO POS(F,POS); 'VAL' F,POS; 'INT' F,POS;
1139       BLOCK TO POS:= POS '/' NELPB(F) + 1;
1140
1141    'INT' 'PROC' NELPB(F); 'VAL' F; 'INT' F;
1142    'BEGIN'  'INT' NEL; NEL:= NELPW(F);
1143      NELPB:= ('IF' NEL = 0 'THEN' ,5 'ELSE' NEL) * NWPB
1144    'END';
1145
1146    'INT' 'PROC' NELPW(F); 'VAL' F; 'INT' F;
1147       NELPW:= FETCH(DESB(F) - NELPW IN DES);
1148
1149    'INT' 'PROC' LAST BLOCK(F); 'VAL' F; 'INT' F;
1150       LAST BLOCK:= FETCH(DESB(F) - LAST BLOCK IN DES);
1151
1152    'INT' 'PROC' PROCSBL(PROC); 'PROC' PROC;
1153       PROCSBL:= PROC;
1154
1155    'INT' 'PROC' STRINGSBL(K,STR); 'INT' K; 'STRING' STR;
1156    'BEGIN'  STRINGSBL:= STRINGSYMBOL(K,STR);
1157      K:= K + 1
1158    'END';
1159
1160    'BOOL' 'PROC' NO FILES; NO FILES:= FADB = 0;
1161
1162    'INT' 'PROC' FMAX; FMAX:= SYS LENGTH(FADB);
1163
1164    'INT' 'PROC' PMAX(F); 'VAL' F; 'INT' F;
1165      PMAX:= SYS LENGTH(PADB(F))/PADCELL;
1166
1167    'INT' 'PROC' FADB; FADB:= FETCH REF(SYSVAR FILEHANDLE);
1168
1169    'INT' 'PROC' CADB(F); CADB:= FETCH REF(FADB + F);
1170
1171    'INT' 'PROC' PADB(F); PADB:= FETCH REF(CADB(F) + PADB IN CAD);
1172
1173    'INT' 'PROC' PTRB(F,P); PTRB:= FETCH REF(PADB(F) + (P + 1) *
1174                 PADCELL + PTRB IN PAD);
1175
1176    'INT' 'PROC' DESB(F); DESB:= FETCH REF(CADB(F) + DESB IN CAD);
1177
1178    'INT' 'PROC' BICB(F,P); BICB:= FETCH REF(PADB(F) + (P + 1) *
1179                 PADCELL + BICB IN PAD);
1180
1181    'INT' 'PROC' BADB(F); BADB:= FETCH REF(CADB(F) + BADB IN CAD);
1182
1183    'INT' 'PROC' SADB(F); SADB:= FETCH REF(CADB(F) + SADB IN CAD);
1184
1185    'BOOL' 'PROC' F OK(F); 'VAL' F; 'INT' F;
1186        F OK:= 'IF' NO FILES 'THEN' 'FALSE' 'ELSE'
1187          'IF' F < 1 ∨ F > FMAX 'THEN' 'FALSE'
1188          'ELSE' CADB(F) 'NE' 0;
1189
1190    'BOOL' 'PROC' P OK(F,P); 'VAL' F,P; 'INT' F,P;
1191        P OK:= 'IF' P < 1 ∨ P > PMAX(F) 'THEN' 'FALSE'
1192          'ELSE' PTRB(F,P) 'NE' 0;
1193
1194    'PROC' HARD CHECK ON F(F); 'VAL' F; 'INT' F;
1195          'IF' ¬ F OK(F) 'THEN' ERROR(ER WF);
1196
```

```
1197   'PROC' HARD CHECK ON F AND P(F,P); 'VAL' F,P; 'INT' F,P;
1198   'BEGIN' HARD CHECK ON F(F);
1199     'IF' ¬ P OK(F,P) 'THEN' ERROR(ER WP)
1200   'END';
1201
1202   'PROC' HARD CHECK ON POS(F,POS); 'VAL' F,POS; 'INT' F,POS;
1203       'IF' POS < VAL OF PTR(F,BP) ∨
1204          POS 'GE' VAL OF PTR(F,EP) 'THEN' ERROR(ER PO);
1205
1206   'PROC' HARD CHECK ON SPECIES(SPECIES); 'VAL' SPECIES; 'INT' SPECIES;
1207       'IF' SPECIES < 0 ∨ SPECIES > 27 'THEN' ERROR(ER WS);
1208
1209   'PROC' HARD CHECK ON WORK PERMIT(F); 'VAL' F; 'INT' F;
1210       'IF' FETCH(DESB(F) ↑ WORK IN DES) < 0
1211       'THEN' ERROR(ER NW);
1212
1213   'INT' 'PROC' BPEL TO SPEC(SPEC); 'VAL' SPEC; 'INT' SPEC;
1214       BPEL TO SPEC:= 'IF' SPEC = 0 'THEN' 54 'ELSE' 27 '/' SPEC;
1215
1216   'INT' 'PROC' NELPW TO SPEC(SPEC); 'VAL' SPEC; 'INT' SPEC;
1217       NELPW TO SPEC:= SPEC;
1218
1219
1220   'COMMENT' CONSTANTS ***************************************************;
1221
1222
1223   'INT' 'PROC' ER CE; ER CE:=-1;
1224   'INT' 'PROC' ER BE; ER BE:=-2;
1225   'INT' 'PROC' ER NN; ER NN:=-3;
1226   'INT' 'PROC' ER UK; ER UK:=-4;
1227   'INT' 'PROC' ER NY; ER NY:=-5;
1228   'INT' 'PROC' ER NP; ER NP:=-6;
1229   'INT' 'PROC' ER WT; ER WT:=-7;
1230   'INT' 'PROC' ER ST; ER ST:=-8;
1231   'INT' 'PROC' ER RE; ER RE:=-9;
1232   'INT' 'PROC' ER NF; ER NF:=-10;
1233   'INT' 'PROC' ER WF; ER WF:=-11;
1234   'INT' 'PROC' ER WP; ER WP:=-12;
1235   'INT' 'PROC' ER NW; ER NW:=-13;
1236   'INT' 'PROC' ER PL; ER PL:=-14;
1237   'INT' 'PROC' ER PH; ER PH:=-15;
1238   'INT' 'PROC' ER FE; ER FE:=-16;
1239   'INT' 'PROC' ER PO; ER PO:=-17;
1240   'INT' 'PROC' ER WS; ER WS:=-18;
1241   'INT' 'PROC' ER PC; ER PC:=-19;
1242
1243   'INT' 'PROC' SYSVAR CATFIBACKAD; SYSVAR CATFIBACKAD:=1;
1244   'INT' 'PROC' SYSVAR CATFINACC; SYSVAR CATFINACC:=2;
1245   'INT' 'PROC' SYSVAR USER; SYSVAR USER:=3;
1246   'INT' 'PROC' SYSVAR FILEHANDLE; SYSVAR FILEHANDLE:=4;
1247
1248   'INT' 'PROC' DELSBL; DELSBL:=255;
1249   'INT' 'PROC' TABSBL; TABSBL:=118;
1250   'INT' 'PROC' NLCRSBL; NLCRSBL:=119;
1251   'INT' 'PROC' SPACESBL; SPACESBL:=93;
1252
1253   'INT' 'PROC' SCRATCH IDF; SCRATCH IDF:= DELSBL*512*(DELSBL*512*DELSBL);
1254   'INT' 'PROC' ALLNINC; ALLNINC:= -0;
1255
1256   'INT' 'PROC' RHO; RHO:=0;
```

```
1257    'INT' 'PROC' PI; PI:=1;
1258    'INT' 'PROC' LAB; LAB:=2;
1259
1260    'INT' 'PROC' SPEC IN DES; SPEC IN DES:=1;
1261    'INT' 'PROC' BP IN DES; BP IN DES:=2;
1262    'INT' 'PROC' EP IN DES; EP IN DES:=3;
1263    'INT' 'PROC' NSEGM IN DES; NSEGM IN DES:=4;
1264    'INT' 'PROC' OFFSET IN DES; OFFSET IN DES:=5;
1265    'INT' 'PROC' BACKAD IN DES; BACKAD IN DES:=6;
1266    'INT' 'PROC' NEW IN DES; NEW IN DES:=7;
1267    'INT' 'PROC' SCRATCH IN DES; SCRATCH IN DES:=8;
1268    'INT' 'PROC' WORK IN DES; WORK IN DES:=9;
1269    'INT' 'PROC' NBICS IN DES; NBICS IN DES:= 10;
1270    'INT' 'PROC' NPTRS IN DES; NPTRS IN DES:=11;
1271    'INT' 'PROC' NELPW IN DES; NELPW IN DES:= 12;
1272    'INT' 'PROC' BPEL IN DES; BPEL IN DES:=13;
1273    'INT' 'PROC' NBLOCKS IN DES; NBLOCKS IN DES:= 14;
1274    'INT' 'PROC' NELPB IN DES; NELPB IN DES:=15;
1275    'INT' 'PROC' NFREE IN DES; NFREE IN DES:=16;
1276    'INT' 'PROC' LAST BLOCK IN DES; LAST BLOCK IN DES:=17;
1277    'INT' 'PROC' CATPOS IN DES; CATPOS IN DES:=18;
1278    'INT' 'PROC' TRANSCOR IN DES; TRANSCOR IN DES:=19;
1279    'INT' 'PROC' NFREEBICS IN DES; NFREEBICS IN DES:= 20;
1280    'INT' 'PROC' IDF IN DES; IDF IN DES:= 21;
1281
1282    'INT' 'PROC' SPEC IN CAT; SPEC IN CAT:=1;
1283    'INT' 'PROC' BP IN CAT; BP IN CAT:=2;
1284    'INT' 'PROC' EP IN CAT; EP IN CAT:=3;
1285    'INT' 'PROC' NSEGM IN CAT; NSEGM IN CAT:=4;
1286    'INT' 'PROC' OFFSET IN CAT; OFFSET IN CAT:=5;
1287    'INT' 'PROC' SEGMAD IN CAT; SEGMAD IN CAT:= 6;
1288
1289    'INT' 'PROC' BLOCK IN BIC; BLOCK IN BIC:=1;
1290    'INT' 'PROC' MOD IN BIC; MOD IN BIC:=2;
1291    'INT' 'PROC' INT IN BIC; INT IN BIC:=3;
1292    'INT' 'PROC' NTRANS IN BIC; NTRANS IN BIC:= 4;
1293    'INT' 'PROC' INFO IN BIC; INFO IN BIC:= 5;
1294
1295    'INT' 'PROC' PTRB IN PAD; PTRB IN PAD:=1;
1296    'INT' 'PROC' BICB IN PAD; BICB IN PAD:=2;
1297
1298    'INT' 'PROC' SEGMAD IN SAD; SEGMAD IN SAD:=1;
1299    'INT' 'PROC' BITWRD IN SAD; BITWRD IN SAD:=2;
1300
1301    'INT' 'PROC' DESB IN CAD; DESB IN CAD:=1;
1302    'INT' 'PROC' SADB IN CAD; SADB IN CAD:=2;
1303    'INT' 'PROC' PADB IN CAD; PADB IN CAD:=3;
1304    'INT' 'PROC' BADB IN CAD; BADB IN CAD:=4;
1305    'INT' 'PROC' TRANSB IN CAD; TRANSB IN CAD:=5;
1306
1307    'INT' 'PROC' VAL IN PTR; VAL IN PTR:=1;
1308    'INT' 'PROC' WRD IN PTR; WRD IN PTR:=2;
1309    'INT' 'PROC' ELT IN PTR; ELT IN PTR:=3;
1310    'INT' 'PROC' BLOCK IN PTR; BLOCK IN PTR:=4;
1311
1312    'INT' 'PROC' BP; BP:=1;
1313    'INT' 'PROC' EP; EP:=2;
1314    'INT' 'PROC' WP; WP:=3;
1315    'INT' 'PROC' FFP; FFP:=4;
1316    'INT' 'PROC' PADCELL; PADCELL:=BICB IN PAD;
```

```
1317    'INT' 'PROC' PADL; PADL:=10*PADCELL;
1318    'INT' 'PROC' NEXP; NEXP:=10;
1319
1320    'INT' 'PROC' IDFL; IDFL:=4;
1321    'INT' 'PROC' DESL; DESL:=IDF IN DES + IDFL - 1;
1322
1323    'INT' 'PROC' NBPW; NBPW:=27;
1324    'INT' 'PROC' NBPS; NBPS:=4;
1325    'INT' 'PROC' NBITWRDS; NBITWRDS:=(NBPS - 1)*NBPW + 1;
1326    'INT' 'PROC' NWPB; NWPB:=20;
1327
1328    'INT' 'PROC' NEXS; NEXS:= 3;
1329    'INT' 'PROC' MAXS; MAXS:= 4*NEXS;
1330    'INT' 'PROC' SADCELL; SADCELL:=BITWRD IN SAD + NBITWRDS - 1;
1331    'INT' 'PROC' SADL; SADL:=MAXS*SADCELL;
1332
1333    'INT' 'PROC' CADL; CADL:=TRANSB IN CAD;
1334    'INT' 'PROC' PTRL; PTRL:=BLOCK IN PTR;
1335    'INT' 'PROC' INFOL; INFOL:=NWPB;
1336    'INT' 'PROC' BICL; BICL:=INFOL + INFO IN BIC - 1;
1337
1338    'INT' 'PROC' NEXF; NEXF:=10;
1339
1340    'INT' 'PROC' DESCRIPTORL; DESCRIPTORL:= IDFL + 3;
1341
1342
1343    'COMMENT' INTERFACE ROUTINES ***********************************************;
1344
1345    'INT' 'PROC' LENGTH(A); LENGTH:= SYS LENGTH(A);
1346
1347    'PROC' ERROR(E); 'VAL' E; 'INT' E;
1348    'BEGIN' 'INT' K;
1349      NLCR; NLCR; PRINTTEXT("ERROR");
1350      PRINT(E); DUMP(0); EXIT
1351    'END';
1352
1353    'PROC' DUMP(D); 'VAL' D; 'INT' D;
1354    'BEGIN' 'INT' MP,GEN,L,K;
1355      MP:= MEMEND; NLCR;
1356      NLCR; PRINTTEXT("DUMP"); PRINT(D); NLCR;
1357      SPACE(20); PRINTTEXT("HANDLE ="); PRINT(FETCH(SYSVAR FILEHANDLE));
1358      NLCR; NLCR;
1359      'FOR' GEN:= MEM[MP] 'WHILE' MP > FREEPTR 'DO'
1360      'BEGIN' NLCR; ABSFIXT(5,0,MP); PRINTTEXT(":  ");
1361        L:= 'IF' GEN>0 'THEN' 2 'ELSE' SYS LENGTH(MP) + 1;
1362    LINE:
1363        'FOR' K:= 1 'STEP' 1 'UNTIL' 10 'DO'
1364        'BEGIN' FIXT(8,0,MEM[MP]); MP:= MP - 1;
1365          L:= L -1; 'IF' L=0 'THEN' K:= 10
1366        'END';
1367        NLCR; 'IF' L>0 'THEN' 'BEGIN' SPACE(10); 'GO TO' LINE 'END'
1368      'END'
1369    'END';
1370
1371    'PROC' SYS NOT;;
1372    'PROC' SYS TON;;
1373    'PROC' SYS EL;;
1374    'PROC' SYS LE;;
1375    'PROC' SYS SWAP;;
1376
```

```
1377  'BOOL' 'PROC' SYS IS STRING(P); SYS IS STRING:= 'TRUE';
1378  'BOOL' 'PROC' SYS IS INT PROC(P); SYS IS INT PROC:= 'FALSE';
1379
1380  'PROC' SYS FANCY IDF(IDF); IDF[1]:= IDF[1]+1;
1381  'PROC' SYS IDF FANCY(F,OLD IDF,IDF);;
1382
1383  'PROC' STORE(AD,W); 'VAL' AD,W; 'INT' AD,W;
1384  'BEGIN' TEST AD(AD,6); MEM[AD]:= W 'END';
1385
1386  'INT' 'PROC' FETCH(AD); 'VAL' AD; 'INT' AD;
1387  'BEGIN' TEST AD(AD,7); FETCH:= MEM[AD] 'END';
1388
1389  'PROC' STORE REF(AD,REF); 'VAL' AD,REF; 'INT' AD,REF;
1390  'BEGIN' TEST AD(AD,8);
1391    'IF' REF < 0 'OR' REF > 262143 'THEN' ERROR(2)
1392    'ELSE' MEM[AD]:= SET(REF,17,0,MEM[AD])
1393  'END';
1394
1395  'INT' 'PROC' FETCH REF(AD); 'VAL' AD; 'INT' AD;
1396  'BEGIN' 'INT' W; TEST AD(AD,9); W:= MEM[AD];
1397    'IF' W<0 'THEN' ERROR(3)
1398    'ELSE' FETCH REF:= BITSTRING(17,0,W)
1399  'END';
1400
1401  'PROC' SSTORE(AD,REAL); 'VAL' AD,REAL; 'INT' AD; 'REAL' REAL;
1402  'BEGIN' TEST AD(AD,10); MEM[AD]:= HEAD OF(REAL);
1403    MEM[AD + 1]:= TAIL OF(REAL)
1404  'END';
1405
1406  'REAL' 'PROC' FFETCH(AD); 'VAL' AD; 'INT' AD;
1407  'BEGIN' TEST AD(AD,11);
1408    FFETCH:= COMPOSE(MEM[AD],MEM[AD + 1]);
1409  'END';
1410
1411  'PROC' INCR(AD,INC); 'VAL' AD,INC; 'INT' AD,INC;
1412  'BEGIN' TEST AD(AD,12); MEM[AD]:= MEM[AD] + INC
1413  'END';
1414
1415  'PROC' TEST AD(AD,N); 'VAL' AD,N; 'INT' AD,N;
1416  'IF' AD < 0 ∨ AD > MEMEND 'THEN' ERROR(N);
1417
1418  'BOOL' 'PROC' SYS NO LONGER SCRATCH(A);
1419    SYS NO LONGER SCRATCH:= 'TRUE';
1420
1421  'PROC' SYS SCRATCH NOW(A);;
1422
1423  'PROC' SYS INCR REFCNT(AD); 'VAL' AD; 'INT' AD;
1424    STORE(AD,¬(¬FETCH(AD) + T19));
1425
1426  'PROC' SYS DECR REFCNT(AD); 'VAL' AD; 'INT' AD;
1427  'BEGIN' 'INT' W; W:= ¬FETCH(AD); STORE(AD, ¬(W-T19));
1428          'IF' BITSTRING(25,19,W)=1 'THEN' SYS DELETE(AD)
1429  'END';
1430
1431  'INT' 'PROC' SYS GENWRD(GEN,L,RC); 'VAL' GEN,L,RC; 'INT' GEN,L,RC;
1432    SYS GENWRD:= ¬SET(RC,25,19,'IF' GEN=LAB 'THEN' T26 'ELSE'
1433      L + ('IF' GEN=P 'THEN' T18 'ELSE' 0));
1434
1435  'INT' 'PROC' SYS LENGTH(AD); 'VAL' AD; 'INT' AD;
1436    SYS LENGTH:= BITSTRING(17,0,¬FETCH(AD));
```

```
1437
1438  'INT' 'PROC' SYS EXTEND(AD,EXTRA); 'VAL' AD,EXTRA; 'INT' AD,EXTRA;
1439  'BEGIN' 'INT' BASE,OL,K; OL:= LENGTH(AD);
1440    BASE:= SYS CLAIM(OL + EXTRA + 1);
1441    'IF' BASE>0 'THEN'
1442    'BEGIN' STORE(BASE,FETCH(AD)-EXTRA);
1443      STORE(AD,-SET(0,25,19,-FETCH(AD)));
1444      'FOR' K:=1 'STEP' 1 'UNTIL' OL 'DO' STORE(BASE-K,FETCH(AD-K));
1445      'FOR' K:=1 'STEP' 1 'UNTIL' EXTRA 'DO' STORE(BASE-OL-K,0)
1446    'END';
1447    SYS EXTEND:= BASE
1448  'END';
1449
1450  'PROC' SYS SHRINK(AD,L); 'VAL' AD,L; 'INT' AD,L;
1451  'BEGIN' 'INT' W,OL; OL:= SYS LENGTH(AD); 'IF' OL-L>1 'THEN'
1452    'BEGIN' W:= FETCH(AD);
1453      STORE(AD,-SET(L,17,0,-W));
1454      STORE(AD - L - 1, -(T18 + OL - L - 1))
1455    'END'
1456  'END';
1457
1458  'INT' 'PROC' SYS CLAIM(L); 'VAL' L; 'INT' L;
1459       'BEGIN' 'INT' AD,GEN,LH; 'BOOL' F;  F:= 'TRUE';
1460  AGAIN:  AD:= MEMEND;
1461    'FOR' GEN:= FETCH(AD) 'WHILE' AD>FREEPTR 'DO'
1462    'BEGIN' LH:= 'IF' GEN>0 'THEN' 2 'ELSE' SYS LENGTH(AD)+1;
1463      'IF' 'IF' BITSTRING(25,19,-GEN) 'NE' 0 'THEN' 'FALSE'
1464        'ELSE' L=LH ∨ LH=L>1
1465      'THEN' 'BEGIN' 'IF' L'NE' LH 'THEN'
1466          STORE(AD-L,-(T18+LH-L-1)); 'GOTO' FND 'END'
1467      'ELSE' AD:= AD - LH
1468    'END';
1469    'IF' FREEPTR - L +1 < 5 'THEN'
1470    'BEGIN' 'IF' F 'THEN'
1471      'BEGIN' F:= 'FALSE'; FREE SEMIFREE; 'GOTO' AGAIN 'END'
1472      'ELSE' AD:= -1
1473    'END' 'ELSE'
1474    'BEGIN' AD:= FREEPTR; FREEPTR:= FREEPTR - L 'END';
1475  FND:  SYS CLAIM:= AD
1476  'END';
1477
1478  'PROC' SYS DELETE(AD); 'VAL' AD; 'INT' AD;
1479  'BEGIN' 'INT' GEN;
1480    GEN:= FETCH(AD); 'IF' GEN > 0 'THEN'
1481    'BEGIN' SYS DECR REFCNT(FETCH REF (AD-1));
1482      SYS DECR REFCNT(FETCH REF(AD))
1483    'END' 'ELSE'
1484    'IF' BIT(18,GEN) = 1 'THEN'
1485    'BEGIN' 'INT' L,NAD;
1486      L:= SYS LENGTH(AD); 'FOR' NAD:= AD-L 'STEP' 1 'UNTIL' AD-1 'DO'
1487      'IF' FETCH REF(NAD) 'NE' 0 'THEN'
1488      SYS DECR REFCNT(FETCH REF(NAD))
1489    'END';
1490    STORE(AD,-SET(0,25,19,-GEN))
1491  'END';
1492
1493  'PROC' SYS TO DISK(COREAD,LH,BACKAD,CNT); 'VAL' COREAD,LH,BACKAD,CNT;
1494       'INT' COREAD,LH,BACKAD,CNT;
1495  'BEGIN' 'INT' 'ARRAY' A[1:LH]; 'INT' K;
1496    'FOR' K:=1 'STEP' 1 'UNTIL' LH 'DO' A[K]:= MEM[COREAD+K-1];
```

```
1497      TO DRUM(A,BACKAD); K:= A[1];
1498      SYS EL; MEM[CNT]:= MEM[CNT] - 1; SYS LE
1499    'END';
1500
1501    'PROC' SYS FROM DISK(COREAD,LH,BACKAD,CNT);
1502      'VAL' COREAD,LH,BACKAD,CNT; 'INT' COREAD,LH,BACKAD,CNT;
1503    'BEGIN' 'INT' 'ARRAY' A[1:LH]; 'INT' K;
1504      FROM DRUM(A,BACKAD);
1505      'FOR' K:= 1 'STEP' 1 'UNTIL' LH 'DO' MEM[COREAD+K-1]:= A[K];
1506      SYS EL; MEM[CNT]:= MEM[CNT] - 1; SYS LE
1507    'END';
1508
1509    'INT' 'PROC' SYS COMPUTE BACKAD(SEGMAD,B IN S);
1510        'VAL' SEGMAD,B IN S; 'INT' SEGMAD, B IN S;
1511      SYS COMPUTE BACKAD:= SEGMAD + B IN S*INFOL;
1512
1513    'INT' 'PROC' SYS CLAIM BLOCK;
1514      'IF' NBBFREE=0 'THEN' SYS CLAIM BLOCK:= -1 'ELSE'
1515      'BEGIN' 'INT' K;
1516        'FOR' K:= 1 'STEP' 1 'UNTIL' NBB 'DO'
1517        'IF' BBFREE[K] 'THEN' 'GO TO' FND;
1518    FND:BBFREE[K]:= 'FALSE'; NBBFREE:= NBBFREE - 1;
1519        SYS CLAIM BLOCK:= BBOFFSET + (K-1)*INFOL
1520      'END';
1521
1522    'PROC' SYS DELETE BLOCK(BAD); 'VAL' BAD; 'INT' BAD;
1523    'BEGIN' 'INT' B;
1524      B:= (BAD - BBOFFSET)/INFOL + 1;
1525      BBFREE[B]:= 'TRUE'; NBBFREE:= NBBFREE + 1
1526    'END';
1527
1528    'INT' 'PROC' SYS CLAIM SEGMENT(SCRATCH);
1529      'VAL' SCRATCH; 'BOOL' SCRATCH;
1530      'IF' NBSFREE=0 'THEN' SYS CLAIM SEGMENT:=-1 'ELSE'
1531      'BEGIN' 'INT' K; 'FOR' K:= 1 'STEP' 1 'UNTIL' NBS 'DO'
1532        'IF' BSFREE[K] 'THEN' 'GO TO' FND;
1533    FND:BSFREE[K]:= 'FALSE'; NBSFREE:= NBSFREE - 1;
1534        SYS CLAIM SEGMENT:= BSOFFSET + (K-1)*NBPS*INFOL
1535      'END';
1536
1537    'PROC' SYS DELETE SEGMENT(SAD,SCRATCH);
1538      'VAL' SAD,SCRATCH; 'INT' SAD; 'BOOL' SCRATCH;
1539    'BEGIN' 'INT' S; S:= (SAD - BSOFFSET)/NBPS/INFOL + 1;
1540      NBSFREE:= NBSFREE + 1; BSFREE[S]:= 'TRUE'
1541    'END';
1542
1543    'PROC' INITIALIZE FILESYSTEM;
1544    'BEGIN' 'INT' FAD;
1545      FAD:= CLAIM CORE(RHO,NEXF,SYSVAR FILEHANDLE,NO);
1546    NO:
1547    'END';
1548
1549    'COMMENT' TEST ROUTINES **********************************************;
1550
1551
1552      'PROC' INITEST;
1553      'BEGIN' 'INT' K;
1554        'FOR' K:= 1 'STEP' 1 'UNTIL' NBB 'DO' BBFREE[K]:= 'TRUE';
1555        'FOR' K:= 1 'STEP' 1 'UNTIL' NBS 'DO' BSFREE[K]:= 'TRUE';
1556        'FOR' K:= 0 'STEP' 1 'UNTIL' MEMEND 'DO' MEM[K]:= 10000000 + K;
```

```
1557        NBBFREE:= NBB;
1558        NBSFREE:= NBS;
1559        FREEPTR:= MEMEND;
1560        BBOFFSET:= 0;
1561        BSOFFSET:= BBOFFSET + NBB*INPOL;
1562        MEM[SYSVAR FILEHANDLE]:= 0;
1563        MEM[SYSVAR USER]:= 696515;
1564        MEM[SYSVAR CATFINACC]:= -777;
1565      FOUND CATALOGUE
1566      'END';
1567
1568    'PROC' FOUND CATALOGUE;
1569    'BEGIN' 'INT' F,B;
1570      F:= NEW FILE(1);
1571      STORE(SYSVAR CATFIBACKAD,SYS CLAIM BLOCK);
1572      MAKE ADM BLOCK(F,'TRUE'); DELETE CORE(FADB-F)
1573    'END';
1574
1575    'PROC' TEST(T); 'VAL' T; 'INT' T;
1576    'BEGIN' PRINTTEXT("TEST"); FIXT(4,0,T); NLCR 'END';
1577
1578
1579    'COMMENT' TEST PROGRAM ************************************************;
1580
1581    'BEGIN' 'INT' A,B,F,G,K,M,N,P,Q,LETTER P,PERIOD,DELETION,COLON;
1582            'REAL' SQRTN;
1583      INITEST;
1584      LETTER P:= 25; PERIOD:= 88; DELETION:= 255; COLON:= 90;
1585      N:= 140; SQRTN:= SQRT(N);
1586      F:= NEW FILE(1); G:= NEW FILE(1);
1587      'FOR' K:= 2 'STEP' 1 'UNTIL' N 'DO' WRITE EL(F,EP,K);
1588      STANDARD PTR(F,BP);
1589      'FOR' P:= NEXT EL(F,BP) 'WHILE' P 'LE' SQRTN 'DO'
1590      'BEGIN' WRITE EL(G,EP,P);
1591        M:= VALUE OF EP(F) -1;
1592        'FOR' K:= VALUE OF BP(F) 'STEP' 1 'UNTIL' M 'DO'
1593        'BEGIN' Q:= NEXT EL(F,BP);
1594          'IF' Q '/' P * P 'NE' Q 'THEN' WRITE EL(F,EP,Q)
1595        'END'
1596      'END';
1597      WRITE EL(G,EP,P);
1598      M:= VALUE OF EP(F)-1;
1599      'FOR' K:= VALUE OF BP(F) 'STEP' 1 'UNTIL' M 'DO'
1600        WRITE EL(G,EP,NEXT EL(F,BP));
1601      CLOSE FILE(F);
1602      NEW IDF(G,"PRIMES"); CLOSE FILE(G);
1603
1604      B:= NEW FILE(27);
1605      A:= OLD FILE("PRIMES");
1606      P:= NEXT EL(A,WP);
1607
1608      'FOR' K:= 1 'STEP' 1 'UNTIL' N 'DO'
1609      'BEGIN' WRITE EL(B,EP,'IF' P=K 'THEN' 0 'ELSE' 1);
1610        P:= 'IF' P 'NE' K 'THEN' P 'ELSE'
1611             'IF' VALUE OF EP(A) > VALUE OF PTR(A,WP) 'THEN' NEXT EL(A,WP)
1612             'ELSE' N+1
1613      'END';
1614      P:= NEW PTR(B,VALUE OF BP(B));
1615      'FOR' K:= 0,K+1 'WHILE' K>0 'DO'
1616      'BEGIN' Q:= IDF SYM(K,A);
```

```
1617        'IF' Q=DELETION 'THEN'
1618        'BEGIN' K:= -777; Q:= COLON 'END';
1619        PRSYM(Q)
1620      'END';
1621      NLCR;
1622      'FOR' K:= 1 'STEP' 1 'UNTIL' N 'DO'
1623      'BEGIN' 'IF' K = K'/'70*70 = 1 'THEN' NLCR;
1624        PRSYM('IF' NEXT EL(B,P)=0 'THEN' LETTER P 'ELSE' PERIOD)
1625      'END'
1626    'END' TEST PROGRAM;
1627
1628
1629    'END';
1630
1631
1632    END OF TEST ENVELOPE;
1633    'END'
```

PRIMES:

```
.PP.P.P...P.P...P.P...P.....P.P.....P...P.P...P.....P.....P.P.....P...
P.P.....P...P.....P.......P...P.P...P.P...P.............P...P.....P.P.
```

REFERENCES

[0] E.W. Dijkstra, "Cooperating sequential processes", in "Programming Languages" (ed. F. Genuys), Academic Press London, New York (1968).

[1] L.J.M. Geurts, L.G.L.Th. Meertens and H.W. Roos Lindgreen, "Files, Voorstel in de vorm van een beschrijving voor gebruikers,...", mimeograph for internal documentation, Mathematical Centre (1971).

[2] J.V.M. van der Grinten, P.J.W. ten Hagen and F.E.J. Kruseman Aretz, "Sequential access files voor de EL X8, deel 1: Voorstel voor de atomaire routines", NR 7, Mathematical Centre (1969).

[3] D. Grune, "Handleiding Milli-systeem voor de EL X8", LR 1.1., Mathematical Centre (1970).

[4] C.H. Lindsey and S.G. van der Meulen, "Informal Introduction to ALGOL 68", North-Holland Publishing Company (1971).

[5] S.E. Madnick, "Design strategies for file systems", MAC TR-78 (1970)

[6] L.G.L.Th. Meertens and H.W. Roos Lindgreen, "A dynamic storage allocation scheme, coded for the EL X8", unpublished.

[7] H.L. Oudshoorn, "Bitmanipulatie-procedures", LR 1.2., Mathematical Centre (1971).

[8] S. Rosen, "Programming systems and languages", McGraw-Hill Book Company (1967).

[9] W.J. Waghorn, "Shared files", in "File organization", File 68 / IAG Conference (1968).

[10] A. van Wijngaarden (Editor), "Report on the Algorithmic Language ALGOL 68", Numerische Mathematik, 14 (1969).

[11] - "Vocabulary for Information Processing", ANSI X3.12-1970, American National Standards Institute, Inc. (1970).