Eidgenössische
Technische Hochschule
Zürich

Institut für Informatik
Fachgruppe
Computer-Systeme

Niklaus Wirth
Jürg Gutknecht

The Oberon System

July 1988

88

Authors address:

Institut für Informatik
ETH-Zentrum
CH-8092 Zürich / Switzerland

The Oberon System

N. Wirth and J. Gutknecht

Abstract

In this paper we describe an operating system for a workstation designed and implemented by the authors within two and a half years. It includes memory management and module loader, a file system, a viewer system, editors for text and graphics, a compiler, a server interface, and various tools. The primary motivation was to demonstrate the feasibility of a small, yet highly flexible and powerful system, a system that is a (decimal) order of magnitude smaller than commonly used operating systems. This is possible due to regularity of concepts and concentration on the essential. The benefits are not only fewer resources needed, but elegance and generality of concepts resulting in transparency and convenience of usage and increased reliability. A cornerstone of this approach is genuine extensibility, which is achieved by a new language, in particular by a facility called type extension. It allows to integrate variables (objects) of a new, extended type in structures of elements of an existing base type.

Introduction

In late 1985, the authors started a project with the goal to develop an operating environment especially tuned to personal workstations, and a language to implement the system. After 30 months of intensive programming, a highly flexible and reliable tool is operational, and this is a summarizing report on both the project and the product called Oberon.

The system is implemented on the computer Ceres, designed by H. Eberle and the first author. Its core is an NS32032 microprocessor, and it features a high-resolution display as output medium and a mouse and a keyboard as input devices. A color display of equal resolution can b- added optionally. A hard disk serves as store for non-volatile data.

Principles of Design and Operation

In the design of both hardware and software we followed a guiding principle, namely to strive for clarity and simplicity. This is not only wise in view of the tiny team and the desire to achieve a workable system within the time bounds of human patience, but simply indispensible for producing *any* system with a claim to reliability. Clarity and simplicity is best achieved through a regular and purpose-tuned structure. This in turn is possible, if the underlying model of operation is well understood, reasonably simple, and free of conflicting premises.

Our chosen model is that of a single process at the disposal of a single human operator who may be pursuing several tasks simultaneously. Such tasks usually are manifest through documents displayed on the screen. The computing process consists of a sequence of operations, each of which is initiated by a command given by the operator. Consecutive commands may be directed at different operands in the pursuit of different tasks. Hence Oberon may be called a *single process multitasking system.*

This model contrasts with that of conventional multitasking systems, where each task is served by its own process. The processor is then switched from one process to another, either as a reaction to user input or due to the system's scheduling strategy. Each suspended process retains a state (represented by its associated workspace), until it is resumed. Switching may occur between any two consecutive machine instructions, except where explicit measures have been taken to prevent it. In Oberon, switching occurs between commands only under the user's control. Hence, the unit of uninterrupted processing is very much larger: the granularity of interleaving is coarse. Complex interlocking mechanisms preventing undesirable process

switches become superfluous, and a single workspace suffices. All this contributes to the system's structural simplicity.

An operation initiated by a command does not permit a dialog with the user; commands are the atomic units of action on the part of the computer within the dialog. They are initiated by the user's input, which occurs through the keyboard, the pointing device (mouse), or remotely from another system via network. A command initiated by the mouse, by far the most frequent case, is handled by the viewer to which the mouse (represented on screen by a cursor) points. A command initiated by the keyboard is directed at the viewer in focus, i.e. the one in which the caret was placed most recently. The most noteworthy effect of this scheme is that no command leaves the system in a hidden state. If a command would permit a dialog, its request for an answer (e.g. "type file name>") would leave the system (or the process) in the requesting state, hence dictating the user a specific reaction or to remember the process' state. Typically, inputs are interpreted differently depending on the current state. This we believe to be an essential impediment to convenient computer usage, perhaps the cornerstone of user-unfriendliness; Oberon avoids hidden states categorically.

Commands take their parameters from the existing global system state, in most cases a displayed text. An essential feature of Oberon is that the output of commands is non-volatile. This implies that it is generated as a data structure that continues to exist after command termination. A visible representation can be derived from this structure (typically text or graphics) and displayed in a viewer, it can be used as input to another command, and it can be stored as a file. For example, a requested excerpt of the file directory can be used as an editable text, it can be subjected to further searching, or selected parts can be moved and inserted in other texts. As a result, the user has the entire system at his disposal in order to prepare for his next action.

The general form of a command name is M.P, where P is the name of a procedure and M the name of the module containing it. It can be typed at the keyboard or, if visible in some text on screen, it can be selected by a simple mouse click. Only a few, most frequently applied commands are built-in; they are evoked by clicking specific mouse buttons (in some cases in combination with the shift or control keys of the keyboard). For example, pressing the middle button signifies that the text identified by the cursor is to be interpreted as a command. This action therefore serves as anchor for all command activations. Or, pressing the right button of the mouse signifies that the text be selected over which the cursor moves until the button is released. Or, pressing a character key at the keyboard signifies that this character be inserted in the focus viewer at the position of the caret, the insertion point mark.

Typically, a user will have some short texts or graphics displayed which contain lists of frequently used commands (Fig. 1). We call such texts *tools*. It is noteworthy that tools are normal, editable texts, and as such can be tuned to each user's individual needs and preferences. Additional commands are listed in the heading of viewers and form so-called *menus*. These commands automatically take their viewer as parameter, i.e. are directed at this viewer. Obviously, no limits are set to fantasy for exploiting this universal scheme of command activation.

Extensibility

Another central theme in the design of Oberon was *extensibility*. On a conventional level, *function* can be added by programming further commands included in new modules. These modules may utilize the resources of the existing system by importing procedures and data types which are freely exported. For example, a compiler importing the type *Text* may acquire source text directly from visible, edited text instead of from files on backing store. Or, a module may be provided which performs some user-specific operation on a (displayed) text, a possibility that in conventional systems requires a so-called programmable editor.

A more sophisticated and more powerful extensibility applies to *data types*. This property of the Oberon system permits to define, in modules to be added, data types that extend imported types. Variables of the extended type are compatible with those of the base type, and therefore

can be integrated in already existing data structures. Operations are polymorphic in Oberon. If an operation is applicable to base type objects, it can also be applied to those of the extended types. This form of extensibility has even more far-reaching consequences, and indeed appears to be a prerequisite for genuine extensibility of entire operating systems. For example, if a certain data type in a drawing system represents graphic objects such a lines and symbols, extensions of this type may specify "semantics", i.e. data to be interpreted in some way, e.g. lines as connecting wires, and symbols as resistors, diodes, transistors, or gates.

The development of the *language* Oberon was largely motivated by the need for a facility for extending data types in the sketched sense. The chosen scheme is similar to that of classes and prefixes in object-oriented languages. The major characteristic is, however, that it is integrated into a strong typing concept, an indispensible property, as shall become evident later. Oberon emerged from Modula-2 and is described in [1]. It omits several of Modula's features and has thus become simpler to comprehend and to implement. Hence, also the language's design followes the guiding principle of clarity and simplicity.

An important and desirable consequence of system extensibility is that the borderline between a system's users and its designers (programmers) becomes obscure and even vanishes. Clearly, programmers are also users when the system for which they program also serves as their software development tool. Conversely, users (perhaps temporarily) become programmers when they discover that an additional function is needed to accomplish their task. The conventional strict distinction between users and designers may be quite appropriate for such single-purpose tools as cars or kitchen aids, but it does by no means justice to the general-purpose tool called computer, and it effectively prevents the full exploitation of the computer's inherent power.

System Structure, an Overview

The system is a collection of modules. By virtue of their import/export relationships they form a hierarchy. In essence there is no distinction between operating system modules and those added by a programmer. Each module consists of procedures (code) and global variables (data). Apart from modules, the computer's store contains a workspace – the *stack* of procedure activation records - and a space for dynamically allocated variables, the so-called *heap*. The structure of the system in terms of its module hierarchy is perhaps best explained by following the process performed when the computer is switched on.

First control passes to a boot loader. This is the only program resident in a read-only store (it is about 300 bytes long), and it loads the *boot file* resident either on track 0 of the disk or on a disk cassette. This file contains the following linked modules: *Kernel, Disk, FileDir, Files, Modules*. They form the *inner core* of Oberon.

Kernel contains all functions which either make use of privileged instructions or access protected data, i.e. must be executed by the processor being in the supervisor state. Hence, the Kernel includes routines for "memory management", i.e. address mapping, page allocation and deallocation, and for disk sector reservation.

Disk is the driver for the hard disk, *FileDir* is the handler of the file directory, and *Files* the collection of routines operating on files, which are sequences of bytes. *Modules* is the system's loader, both linking and transforming modules from object code files on disk into executable form in main store.

After initializing the page maps and the disk sector reservation table of the Kernel, control passes to the loader with the built-in request to load the module *Oberon*. The loading of Oberon causes the subsequent loading of all imported modules, resulting in a structure shown in Fig. 2, called the *outer core*. Then, the loading of module *Display* is requested, and thereby also that of all modules imported by it. A viewer is opened which typically displays a standard tool text reflecting the software configuration available on the particular computer. The resulting system structure is shown in Fig. 3. The start-up process terminates by a call to the *central loop*, in which the computer repeats ad infinitum to poll the various input devices, viz. the mouse, the keyboard, and (optionally) the network.

Let us now trace the events occuring when an input is sensed, for example the clicking of the middle mouse button. First, a routine in module Viewers, where the books are kept about the layout of the tiled viewers on the screen, identifies the viewer in which the cursor lies. Thereupon, control is relegated to the *handler* (command interpreter) of this viewer.

A handler is a procedure assigned to a variable, typically the field of a record, in this case of a record representing the viewer. The presence of a handler in a record is perhaps the most relevant criterion for letting the record be called an active *object* and its use *object-oriented* programming. Nevertheless, the call of a handler is like an ordinary procedure call, with the exception that the caller is unaware of the identity of the callee. The call has the connotation of "handle the supplied parameters in any way deemed appropriate", and hence the call's parameters are typically regarded as a *message*, and the call of the installed procedure is regarded as the sending of a message.

Let us now further assume that the addressed viewer is a text viewer. In this case the handler of text viewers is activated; it receives the message "middle button at position x,y". The respective action is to locate the text (word) displayed at this position, which occurs with the aid of appropriate procedures in modules Texts and TextFrames. If this word has the form M.P, the action "middle button" signifies to interpret that word, which implies the loading of module M (if not already loaded), and of activating its procedure P. Termination (normal or abnormal) causes the return of control into the central loop and the renewed polling of input devices.

In the module structure shown in Fig. 3, there exist three similar triples: Texts, TextFrames, TextViewers, and Graphics, GraphicFrames, GraphicViewers, and Pictures, PictureFrames, PictureViewers. They demonstrate a typical modular decomposition serving the effective separation of concerns. The base modules Texts, Graphics, and Pictures define data objects and structures reflecting the respective entities, and procedures appropriate to reorganise the structure. These procedures perform insertion of new objects, deletion, or their localization (search). The structures represent texts, graphs, or pictures as abstractions, without respect of their semantics or visualization. The latter is performed by the respective *Frames* module (e.g. TextFrames). Its operators relate a particular object to a particular frame and generate the respective visualized representation, i.e. bitmap raster. A *frame* is a rectangular area on the screen. Each viewer is a frame, and in itself usually contains two subframes, called the *menu* and the *text* (in the case of text viewers).

The respective Viewers module (e.g. TextViewers) has the task of providing operations for generating, restoring, closing a viewer, and, most important, of providing a handler. The general module Viewers is responsible for the management of the screen area, i.e. it contains routines which keep record of allocated viewers and which - relying on heuristics - determine the best place to position a new viewer.

TextFrames, GraphicFrames, and PictureFrames are of course extensions of Frames, and provide a fine example for an application of Oberon's type extension concept.

Resource management

The reader may have noticed that the word *program* is missing from the Oberon vocabulary. This may seem irrelevant; after all, it has been replaced by the word *command*. However, this impression is misleading.

Conventional systems effectively process a sequence (in the case of multiprocess systems: sequences) of program activations. The essential characteristic is that after termination of each program its used resources (store) are free to be reused by the following program. The interface between program activations consists of data generated by the preceding and consumed by the subsequent program. Typically, these data are disk files.

In contrast to this paradigm, Oberon's interface between consecutive commands are primarily data in main store, viz. global variables (or data structures anchored in global pointers). This fact not only enhances efficiency of many operations and vastly expands the system's flexibility, but it has also the consequence that there is no event (such as program

termination) which explicitly signals the opportunity for easy storage reclamation. This also holds for storage occupied by program code: the termination of a procedure does not prevent it from being requested soon again, perhaps by the very next command.

An obvious recourse in such a situation is the use of an automatic storage retrieval mechanism, also called *garbage collector*. It requires that all currently accessible objects must be identifyable. This is achieved by providing additional data in the form of type descriptors. Because the garbage collector is no safer than the descriptors guiding its actions, the descriptors must be guarded against access, particularly against inadvertant write access. Although this is easily achieved in principle, namely by not mentioning the tag's existence in the language definition, it is much more difficult in practice. For, it requires the guarantee that an implementation safely guards against all possibilities of disruption. Potential adversaries are all sorts of access via computed address, specifically those involving indexing or dereferencing. Indices must be guarded by a check of the index bounds, pointers by a NIL-test. Furthermore, pointer variables must be type safe, i.e. no assignment of any value other than an appropriately bound pointer must be permitted. The existence of a general type ADDRESS is out of the question. Type safety through type guards, range checks, and NIL tests can no longer be regarded as a convenient luxury, but becomes a simple necessity. Evidently, the design of a system is intimately coupled with (the design of) the language in which that system is programmed. The system's reliability is strongly influenced by the safety the language and its compiler provide against blunders.

This is old but often repressed knowledge. The relevant question here is not whether or not it is *possible to construct* a fully reliable system with a low-level tool such as assembler code or C, but whether it is *possible to convince* others (and oneself!) of the claimed reliability. Our experience gave convincing evidence that without the quality of the language our goal would have been utterly unreachable.

The garbage collector operates on collectible storage only, the *heap*. It does not apply to module storage. Modules are never released automatically, but can be freed by explicit command. This facility might be regarded as merely a concession to the reality of limited storage, but this is not so. Module release is also necessary when a module needs to be replaced by an updated, recompiled version. By convention, upon release of a module all its imports are also released unless, of course, they are imported by yet another module. The determination of releasable modules is based on the method of reference counting. Its well-known drawback is that of the persistence of circular references. However, circles should be avoided in module structures anyway, and therefore this problem was not considered as serious. Nevertheless, the problem of storage reclamation turned out to be one of the hardest nuts to be cracked in the course of the Oberon project.

It is perhaps worth mentioning that the Oberon scheme of abandoning the notion of program and of relying on automatic retrieval has become practicable through the advent of large (yet not infinite) storage capacity. Not letting conventional systems simply grow bigger and bigger, but making sensible use of large stores is the major challenge in modern system design. In the Oberon system, the loading of a client module is delayed until the first access to it occurs during program interpretation.

Examples where the concept of delayed loading bears fruit are a compiler linked into a general utility module, or a module for editing mathematical formulae linked into a general document editor. In both cases, memory space is consumed only at the instance when the respective facility is called upon.

The virtual address mapping facility (memory management) is hence utilised in Oberon for two purposes: First, for implementing delayed loading, and second, for protecting certain critical data from inadvertant modification. Apart from this, virtual addresses simplify the allocation mechanism for modules. However, Oberon does not use the concept of demand paging for hiding the differences between primary store and disk store.

Storage Layout and Module Structure

The storage layout in the Oberon system is determined by the structures of programs and data present in high-level languages, in particular the Oberon language. Programs consist of modules, which are separately compiled entities. It is therefore appropriate to let storage layout and management reflect the module structure. Data are classified into three categories according to their allocation technique: Variables declared globally are called *static*. Their space is allocated when a module is loaded, just as the space for the module's program code is allocated. Variables declared local to procedures are allocated when the procedure is called (and due to recursion may exist in several incarnations). *Dynamically* allocated variables belong to the third category. They are allocated by an explicitly programmed statement NEW(p). According to these classes, the linear address space is divided into three sections: module area, stack, and heap.

The *module area* contains the modules' code and static variables. The *stack* is the system's workspace and contains - for each called procedure - a procedure activation record which includes space for the procedure's local variables. The *heap* is the space for dynamically allocated variables. The global storage layout is shown in Fig. 4.

Although local variables might also be allocated in the heap, thereby making a separate stack superfluous, we claim that the retention of a stack is crucial for a system's efficiency. This is because a stack implies a storage reclamation free of any explicit effort (apart from resetting a stack pointer). Since local variables appear to dominate, the gain in speed obtained from the use of a workspace organised as a stack is significant.

The Module Area

The second source of influence on storage layout is the architecture of the underlying hardware. The NS32000 processor was chosen because it supports the language implementation with appropriate registers and addressing modes. Nevertheless, there are particularities that determine further details of the chosen layout. In order to understand them, it is necessary to explain the pertinent features of the processor's architecture and instruction set.

The mentioned support for language implementation offered by the processor's structure lies in its addressing modes and in the instructions for external procedure call and return. In particular, we refer to the facilities for calling procedures and accessing variables declared in other modules. These facilities effectively establish the binding of modules at the time of program execution, and make their binding at load time superfluous. In our experience, this is of significant advantage, as it avoids both module duplication and the (perhaps inadvertant) presence of different versions of the same module entering at different linking stages.

The NS32000 architecture requires that each module is represented by three (possibly disjoint) blocks of storage. These blocks contain data (static variables), code, and a link table respectively. Their base addresses are stored in a *module descriptor*. Further details are described in Appendix 1. Here it suffices to note that the architecture of the NS32000 processor represents addresses of module descriptors by 16 bits. Hence, descriptors must reside within the first 64k of memory, requiring that they be treated differently from other variables, and that they cannot be placed in collectible storage. The *module descriptor area* therefore must be considered as a fourth, separate, (write-protected) partition of memory space.

Various properties of the Oberon system require that the organization described so far be extended. Module descriptors as well as module blocks contain parts in addition to those imposed by the hardware. The areas allocated for program code, static variables, and links are made contiguous and blocks for constants and for data accessed by the loader are added. The resulting combination is called a *module block*. The extended structure of a module descriptor and its corresponding module block are shown in Fig. 5. The fourth element of the descriptor is the base address of the block. Descriptors are linked as a linear list whose anchor is a global variable of the loader, requiring a field with a link to the next descriptor in the list. In order to

understand the role of the extensions a closer look at the process of module loading is necessary.

The Module Loader

The loading of a module consists of the following steps:

1. Allocation of a module descriptor in the descriptor area.
2. Allocation of a block in the module area.
3. Transferring of the program code and of constants from the object file into the block.
4. Translating the linkage information of the file into a link table in the block.

The linkage information of an object file consists of a list of module names from which objects are imported (corresponding to the import list in the program text), a list of external procedure references, each characterized by a pair consisting of the procedure's (local) module number and its entry point number, and a list of the module's own entry points (procedure addresses).

The translation procedure first identifies (finds the addresses of) the descriptors of the imported modules. For this purpose, each descriptor must contain the module's name (and version key). Then the entry address of each referenced procedure is determined; together with the descriptor address it forms the procedure descriptor which is stored in the link table and which is referenced by the call instruction CXP. For this purpose, each module must carry the list of its entry points. In principle, it could be reread from the object file each time a client module is loaded. However, it was taken as a premise that upon loading a module all information needed to establish links later-on had to be retained in order to avoid reaccessing object files. As a consequence, a module's entry point list is copied from the object file into the descriptor. The resulting extended descriptor format is shown in Fig. 5. Modules are loaded by procedure *GetMod* in the loader module *Modules* which returns a pointer to the module's descriptor.

```
PROCEDURE GetMod(name: ARRAY OF CHAR; VAR mod: Module)
```

Should the named module already be present in the list of loaded modules, the reading of the object file is of course avoided. As each module may require the loading of further imported modules, the procedure *GetMod* is recursive.

There exist three features of the Oberon system which call for further information to be present in every module block:

1. The *command list.* Parameterless procedures can be activated by specifying a text of the form M.P, where P is the procedure's identifier and M that of the module containing it. As a consequence, once the module M is identified (and loaded), a possibility must exist to infer P's entry point from the name. This is made possible by the inclusion of a list of commands in the heading of each block. Each entry is a pair consisting of a name and an entry address. The procedure *GetProc* yields, given a name and a module, the corresponding procedure descriptor consisting of module descriptor address and entry point offset.

   ```
   TYPE Command = PROCEDURE;
   PROCEDURE GetProc(name: ARRAY OF CHAR; mod: Module; VAR P: Command)
   ```

2. The *pointer reference list.* Unused storage is to be reclaimed automatically, i.e. space occupied by variables (objects) no longer accessible must be locatable. This requires a list of the locations of all pointer variables, because they are the roots of data structures in collectible storage.
3. The *import list.* Modules, once loaded, remain in the module area unless explicitly unloaded by a call of *Free.*

   ```
   PROCEDURE Free(name: ARRAY OF CHAR; all: BOOLEAN)
   ```

If *all* is specified, unloading is transitive, i.e. also extends to the imported modules, requiring that Free be recursive. In order to identify the modules to be affected, a *list of imported modules* needs to be available.

The resulting, complete module block structure is shown in Fig. 5.

Virtual Addresses and Delayed Loading

Frames are allocated sequentially in the module area, and address space once allocated is never reused, even if released (by a call of *Free*). This strategy may appear wasteful, but it is quite realistic in view of the fact that modules are allocated in the *virtual* address space which is quite large (14 MByte for the NS32032, and almost 4 GByte for the NS32332 and NS32532). However, procedure *Free* causes the release of the assigned physical pages. A corollary is that the block size must always be rounded up to a multiple of the page size (512 Bytes for NS32032, 4096 Bytes for the other processors). Each module block wastes half of the page size in the average (internal fragmentation).

The use of virtual addresses thus prevents the occurrence of unusable "holes" in memory (external fragmentation), and hence the Oberon design takes advantage of the processor's memory management facility (MMU). But the concept of demand paging, i.e. of dislocating certain pages from main to disk store, is not used. The large memories of modern computers appear to make demand paging quite useless, at least for single-user workstations.

However, Oberon employs demand paging in a limited form which we call *delayed loading:* When a module M is loaded, its imports, if not already present, are merely allocated in the module area in virtual space. The actual loading and occupation of physical pages is delayed until genuinely needed, i.e. it happens on demand. The occurrence of such a demand, i.e. the reference to an object, is detected by the MMU through a page fault. This in turn is interpreted as a request for the actual, delayed loading of the missing module.

Care has to be taken that all information needed for linking an allocated but not yet loaded module with one of its clients be physically present. This information consists of the module name and all its entry point offsets. It is for this reason that name and entry points are contained in the module descriptor instead of the block. Even in the case of delayed loading, the descriptor is truly loaded (in the descriptor area), whereas the block is not. Hence the block must contain information only that is accessed by the module itself, but not by clients.

We regard the delayed loading facility as a key to implement potentially very large systems with moderately sized memories. Vice-versa, we regard the lack of this feature in most commercially available systems as the principal culprit for the exceedingly large memories required to compile and execute even the tiniest programs. The amount of code needed to implement delayed loading is very small.

Collectible Storage and Type Descriptors

Dynamically allocated variables, often called *objects*, play an important role in programming in Oberon. They are allocated in the heap by a call of the intrinsic procedure NEW. Its parameter is the pointer variable to which the address of the created object is assigned; the amount of storage needed is derived from the type of the object created, i.e. from the data type to which the pointer variable is bound.

At system initialization, the heap has size zero. Whenever an object is allocated and no free space is found, the heap area is enlarged by assigning an additional physical page to it. Such heap pages are never released in order to guarantee that the heap occupies a contiguous (virtual) address space.

It is in this space that the *garbage collector* operates according to the conventional mark-scan principle. During the mark phase, all data structures accessible from loaded modules are traversed and marked. In the subsequent scan phase, the heap is scanned sequentially. All unmarked objects are "retrieved", i.e. collected in lists of reassignable storage blocks, and all marks are removed. No explicit deallocation procedure is provided in Oberon, since such hints

could not be taken as absolutely trustworthy, and since any misdirected effort of the garbage collector could easily have disastrous consequences for the entire system.

All global pointer variables existing in the system are potential roots of data structures to be traversed during the *mark phase*. They are located by traversing the list of loaded modules, and for each of them traversing its pointer reference list. The latter is created upon loading, and provisions for establishing it must be made in the compiler, the object file format, and in the loader. These preparatory measures cause negligible overhead during loading.

For local pointer variables, however, the overhead is considerable, because the relevant data have to be established potentially for every procedure call. This appears as prohibitive. Another approach has been taken with the Modula-2+ system [2]. Since local variables are placed in the stack(s), the stack is simply scanned for potential pointer values, which must lie within the address range of the heap. Nevertheless, safety tags must be included in the objects themselves to prevent the inadvertant marking of a location specified by a value that lies within the range, yet happens not be a pointer.

In Oberon, we use a more conservative approach: The garbage collector is activated between two consecutive commands only. At these moments, the stack is guaranteed to be void, since no local variables exist. This scheme is not practicable, if single commands call for the allocation of very large amounts of objects that cannot be fitted into the available heap space.

Given a root of a dynamically allocated data structure, the mark phase must traverse the entire structure. Since each node may contain an arbitrary number of pointers, information must be available about the number of these pointers and their individual locations within the object (offsets). This information is contained in *type descriptors* derived by the compiler for each record and array type. Each object contains a (hidden) pointer to the descriptor of its type, a *type tag*. As the number of different types in a system is never very large, the time to establish the descriptors and the space occupied by them is quite negligible. Fig. 6 illustrates a typical structure and its associated descriptors.

The traversal of the data structures must not require further storage space for the collector's local data. The traditional method to satisfy this requirement during the traversal of arbitrarily complex structures is that of Bobrow and Deutsch [3]. It is based on the idea of inverting traversed pointers, thereby establishing the return path (the thread of Adiadne) and fitting its information within the structure itself.

During the *scan phase* of garbage collection, the collectible blocks of storage are discovered as the complement of the marked objects. Various schemes exist to collect these blocks in lists or other structures. The key issue here is to organise the set of free blocks in such a way that (1) a block of the appropriate size is quickly located upon request, and that (2) smaller blocks can be merged into larger blocks without undue overhead in order to reduce storage fragmentation.

The method chosen for Oberon is to classify objects into five categories according to size, and to include all objects of each category in a list. The sizes of categories 0 to 3 are 16, 32, 64, 128 bytes; all objects in category 4 have a size which is a multiple of 128. The size of all objects is rounded up to that of the smallest category into which the object fits. (Note that 4 bytes are always added for the tag.)

The motivation for this choice is (1) that searching for a hole of appropriate size is easy and quick, (2) that the amount of storage wasted (for smaller object, which are the vast majority) is 25% (only), and (3) that the presence of adjacent holes of category k is easy to detect and automatically leads to a hole of category k+1. This situation is similar to a so-called Buddy System.

Type descriptors serve yet another purpose. They represent the relationship of a type to its base type(s) and allow for an efficient realization of type tests (and type guards) [1]. For this purpose, an additional field (not shown in Fig. 6) is provided. It contains a pointer to the descriptor of the direct *base type*. The pointer is called *base tag*. The situation of four types T3 → T1 → T0 and T2 → T1 (T' → T is pronounced as "T' is a direct extension of T") is represented by a descriptor structure as shown in Fig. 7, together with an object x of type T3. The type test x IS T

is implemented by a simple search algorithm which typically consists of 6 to 8 machine instructions compiled as in-line code:

```
t := x.tag;
LOOP
   IF t = T THEN EXIT TRUE END;
   t := t.basetag;
   IF t = NIL THEN EXIT FALSE END
END
```

In the case of a type guard x(T), EXIT FALSE is replaced by HALT.

We note in passing that the simplicity of this solution is based on the postulation of name equivalence of types instead of structural equivalence, as it is assumed in the language Oberon. Hence, a type is identified by the address of its descriptor rather than the descriptor's value (which would have to be significantly extended).

The File System

The Oberon filing system consists of three modules, the disk driver, the file directory complex, and the actual file handling module. Only the latter is normally accessed directly by users' programs. The configuration is shown in Fig. 8.

The purpose of the module *Files* is to implement the abstraction of a sequence of bytes in terms of dispersed disk sectors presented by the disk driver. The abstraction "sequence" becomes manifest through the data type *File* and a set of procedures providing access to files.

Conventionally, a reading or writing position is associated with each file when it is opened. This position determines the next element (byte) to be read or written, and it is implicitly moved forward by each access. Upon closer inspection, this intimate connection of file and position appears as unfortunate. In a practical sense, the inadequacy becomes manifest in a system where a file may be accessed by several tasks, each performing sequential reading. If the tasks access the file alternatingly, the (single) position of reading may have to be switched back and forth many times, while each task must keep track of its own reading position. As it is common practice to associate buffers with sequential access mechanisms, the inefficiency of this scheme is inherent.

The solution lies in disentangling the notions of file structure and sequential access mechanism. The attribute *length* belongs to the file, whereas the attribute *position* evidently belongs to the access mechanism, of which several may coexist.

In Oberon, we therefore make the notion of a positioned access mechanism explicit in the form of a data type and associated operators. The type is called *Rider*, reflecting that the access point rides on the file. The opening of a file now takes two steps:

1. A file f is obtained either by a call *Old(name)* in which case the directory is searched for an entry with the specified name, or by a call *Files.New()* in which case a new, empty file is created.
2. A rider r is is connected with f by the call *Files.Open(r, f, pos)*. It places the rider at position *pos* (usually 0).

Data transfer operations then refer to the rider, not to the file. Examples are *Files.Read(r, ch)* and *Files.Write(r, ch)*. It is essential that *several* riders can be associated with the same file, i.e. step 2 can be repeated.

The following procedures are provided; the first group applies to files as a whole:

```
PROCEDURE Old(name: ARRAY OF CHAR): File;
PROCEDURE New(): File;
PROCEDURE Length(f: File): LONGINT;
PROCEDURE Register(f: File; name: ARRAY OF CHAR);
```

The last procedure serves to register a (new) file in the name directory. This operation is typically invoked after the file had been generated successfully and closed. It may be performed earlier, e.g. immediately after creating the empty file. In this case, the danger of replacing an existing file version by a new, yet incomplete one lurks. In many systems, the two functions of creating a file and of registering it are combined into a single procedure. The availability of two separate procedures has the advantage that the programmer may choose the time of registering considered most appropriate for the particular case.

The second group of procedures involve a rider. A rider is a record with fields *res* and *eof* used by file procedures to specify result conditions.

```
TYPE Rider = RECORD eof: BOOLEAN; res: INTEGER; f: File END

PROCEDURE Open(VAR r: Rider; f: File; pos: LONGINT);
PROCEDURE Read(VAR r: Rider; VAR x: BYTE);
PROCEDURE ReadBytes(VAR r: Rider; VAR x: ARRAY OF BYTE; n: INTEGER);
PROCEDURE Write(VAR r: Rider; x: BYTE);
PROCEDURE WriteBytes(VAR r: Rider; VAR x: ARRAY OF BYTE; n: INTEGER);
PROCEDURE SetPos(VAR r: Rider; pos: LONGINT);
PROCEDURE Pos(VAR r: Rider): LONGINT;
PROCEDURE Close(VAR r: Rider);
```

Implementation of Files

We first turn to the representation of files on disk. Oberon uses indexed allocation: the first sector of every file contains a *header* with a table (array) of the disk addresses of the file's sectors. This scheme allows sectors to be freely dispersed and avoids (external) storage fragmentation entirely. When the file is opened, the address table is read and copied into the *handle* (which acts as a cache) in main store. The drawback is the fixed length of such a table for all headers; in short files the header is wasteful, and for long files it constitutes an undesirable limitation. With a table size of 64, the maximum file length is 65536 bytes. In order to relax this limitation, a second table of so-called extensions is provided in each header (and consequently also in handles). Its entries are disk addresses of sectors that contain extensions of the primary index table, each adding 256 entries (sectors). The extension table has 10 entries, which determines the maximum file length as 64 +10*256 = 2624 sectors, or 2686976 bytes.

Apart from these tables and the length, the header also records the file's time and date of creation, and a protection lock.

When a file is being connected, i.e. when a file handle is established through a call of either Old or New, the file's allocation table and its length are assigned to the handle. Furthermore, a sector buffer is associated with the handle. A file may have several buffers; initially however, there is only one. When, in the second step, a rider is opened, it is associated with both the file and with a buffer. Buffers specifically belong to a file, and the data at the position of each rider are duplicated (cached) in a buffer. In principle, a buffer rides along with the rider; however, should several riders be positioned within the same sector, then only a single buffer represents that sector. This strategy makes consistency updates of duplicate buffers superfluous. New buffers are allocated only when new riders are opened or when a rider's position is changed through a SetPos operation. The data structure typically representing an opened file is shown in Fig. 9.

The key to efficiency of a filing system lies in its elementary routines for sequential reading and writing of single bytes. Even the "overhead" of book-keeping for reading or writing a sector is of secondary importance. Most important is the number of instructions executed for fetching or storing a single byte in a buffer. For, this operation may occur as many as a thousand times before a next disk access is involved. The procedure *Read* in the Oberon file system is therefore extremely short; mostly only one comparison, one assignment, and one increment are executed for each call:

```
PROCEDURE Read(VAR r: Rider; VAR x: BYTE);
BEGIN
    IF r.bytepos < r.buf.limit THEN
        x := r.buf.B[r.bytepos]; INC(r.bytepos)
    ELSIF r.secpos < r.file.secleng THEN
        INC(r.secpos); ReadPage(r); x := r.buf.B[0]; r.bytepos := 1
    ELSE x := 0X; r.eof := TRUE
    END
END Read
```

The *Write* routine is quite similar. *ReadBytes* and *WriteBytes* serve to read and write sequences of a given number of consecutive bytes, and are somewhat more complex; they make use of the processor's block move instruction, and can therefore contribute significantly to a program's efficiency.

The *file directory* module is independent of the file module. The directory establishes a mapping from file names to file addresses, i.e. the disk sector address of a file's header. Oberon does not use the conventional technique of treating the directory as a file. Instead, it is organised as a B-tree, each page allocated as an individual sector. The root page is located at a fixed position (see Fig. 10).

It is desirable to use a degree of the tree that is as large as possible. The limit is given by the sector size. Each entry in the tree consists of a key (file name), the address of a file header, and a pointer to the descendant directory page. Given a sector size of 1024 bytes and a fixed (maximum) name length of 32, each page contains at most 25 entries; this results in a B-tree of degree 12. Assuming a system with 1700 files, at most 3 directory pages need be accessed in a name search. In the luckiest case, a tree of height 3 can contain almost 14000 files. In view of this favourable performance and following the principle to keep algorithms reasonably simple and regular, we have refrained from introducing more shophisticated schemes, such as the B*-tree, which treats internal and leaf nodes differently.

The file directory module contains two recursive procedures, one for searching and inserting a name, the other for deletion. If upon deletion a directory page underflows, elements are annected from a neighbouring page; if that page underflows too, the two pages are merged. The algorithm is described in [4]. A third procedure called *Enumerate* serves to traverse the tree. It is typically used to inspect and generate excerpts of the directory. Since parameters of such search orders usually contain so-called "wild cards", this enumeration procedure accepts a prefix to the name as a parameter, and it constrains the traversal to that part of the tree whose keys start with the prefix. For each node encountered, a parametric procedure is called.

A file system must not only be reliable, but it should also be robust against malfunctions of the hardware. Sector addresses are encoded such that any single bit error is detected when checked before sector access.

The most critical part of the entire system is the directory. Should a malfunction of the disk make the directory unreadable, the file data, although unharmed, become unreachable in principle and appear to be lost. The Oberon file system, however, attaches sufficient information to each file header, so that a rescue and scavenger program might find unafflicted file headers and, at least partially, restore the directory. A header mark and the file name suffice for this purpose; they are otherwise redundant.

The *disk driver* is used directly by both modules *Files* and *FileDir*, and it presents the disk as an abstract array of blocks. It hides particularities of individual disk controllers and drives, and maps sector addresses, usually consisting of surface, cylinder, and sector numbers, onto a linear scale.

The Viewer System

Modern bitmap displays are capable of presenting data of different activities simultaneously. Typically, a rectangular area on the screen (a so-called *viewer*) is associated with each activity. The management of viewers on the screen should therefore be regarded as a central system task. Several allocation models exist, some of them are based on tiling, others on arbitrary overlapping. Oberon uses hierarchic tiling: The display is horizontally divided up into tracks, and each track is vertically subdivided into viewers (see Fig. 1). The topmost viewer in each track is a background viewer or *filler*. It has normally height zero. The standard layout features two tracks: a larger *user* track on the left for document viewers and a narrower *system* track on the right for viewers that display system oriented data, present so-called *tools* (see chapter *The Concept of Tools*), or log the progress of the currently executed command. Partial overlapping of viewers is impossible. However, Oberon's hierarchy of viewers is actually three-dimensional: Oberon allows any contiguous sequence of existing tracks to be covered by a newly created track.

The basic viewer system consists of the modules *Bitmaps* and *Viewers*. *Bitmaps* provides raster operations to copy a bit-block or a pattern from one location to another and to replicate a pattern into a bit-block. *Viewers* is the actual viewer manager. It exports procedures to open and close tracks, to open, change, and close viewers, and draw a fresh viewer. Notice that viewers are passed to the viewer manager as *input* parameters. This is in concordance with Oberon's concept of polymorphic operations: Actual parameters may well be extensions (specializations) of the base type *Viewer*. In *OpenTrack*, *X* defines the (leftmost) track and *W* the (minimum) width to be covered. In *Open*, *(X, Y)* defines a point on the top line of the new viewer.

```
PROCEDURE OpenTrack (Filler: Viewer; X, W: INTEGER);
PROCEDURE CloseTrack (X: INTEGER);
PROCEDURE Open (V: Viewer; X, Y: INTEGER);
PROCEDURE Change (V: Viewer; Y: INTEGER);
PROCEDURE Close (V: Viewer);
PROCEDURE Draw (V: Viewer)
```

The viewer manager also provides some auxiliary procedures to get various information about the constellation of viewers. *This* identifies the currently visible viewer at *(X, Y)*, *IsBottom* returns the value of the predicate *the track at X is at the bottom*, *NofViewers* returns the number of viewers in the track at *X*, and *Upper* returns the upper neighbour of *V*. The last procedure needs more explanation. Tiling viewer systems are especially predestined for automatic allocation of new viewers. *Locate* provides useful information for a heuristic decision, where a newly opened viewer should be placed. *X* specifies a track and *H* a hint on the desired height. The following viewers are located: Filler, bottom viewer, viewer of maximum height, and an alternative viewer of a height at least *H*.

```
PROCEDURE This (X, Y: INTEGER): Viewer;
PROCEDURE IsBottom (X: INTEGER): BOOLEAN;
PROCEDURE NofViewers (X: INTEGER): INTEGER;
PROCEDURE Upper (V: Viewer): Viewer;
PROCEDURE Locate (X, H: INTEGER; VAR fil, bot, alt, max: Bitmaps.Frame);
```

In reality, Oberon's display hierarchy is more unified and more general than we might have given the impression so far. In fact, the entire display, tracks, and viewers are special cases of so-called *frames*. A frame is a rectangular area in the bitmap together with a (possibly empty) sequence of subframes. For example, tracks are subframes of the display, and viewers are subframes of tracks. Even viewers themselves need not be atomic units. Usually, a viewer

consists of two subframes: a header frame containing the viewer's name and a list of commands (menu) and a main frame containing the viewer's actual contents.

In module *Bitmaps* we can find the following declarations. A frame descriptor contains references to the list of descendants and to the next frame in the own list, coordinates defining the lower left corner, and width and height of the frame. We postpone the explanation of the role of the installed handler.

```
TYPE  Frame = POINTER TO FrameDesc;

  FrameDesc = RECORD
    dsc, next: Frame; (*descendant, next*)
    X, Y, W, H: INTEGER;
    handle: Handler
  END;
```

The data types in module *Viewers* are extensions of frames. A viewer needs an additional variable describing its *state*, and a track needs another pointer to further tracks possibly lying underneath. At any given time, a viewer is in one of the following states: *displayed (state > 1)*, *filler (state = 1)*, *closed (state = 0)*, or *covered (state < 0)*. Except in closed state, a viewer is always referenced by the viewer manager.

```
TYPE  Viewer = POINTER TO ViewerDesc;

    ViewerDesc = RECORD
     (Bitmaps.FrameDesc) (*extension of Bitmaps.FrameDesc*)
     state: INTEGER
    END;

     (*state > 1: displayed
     state = 1: filler
     state = 0: closed
     state < 0: covered*)

     Track = POINTER TO TrackDesc;

     TrackDesc = RECORD
      (ViewerDesc) (*extension of ViewerDesc*)
      under: Bitmaps.Frame
     END;
```

The viewer manager maintains a private dynamic data structure reflecting the current hierarchy of visible and covered tracks and viewers. Fig. 11 shows the exact memory representation of a configuration with two tracks, the first of which is an overlay of two other tracks. Essentially, the description of the display is a linked ring of track descriptors, a track description is a linked ring of viewer descriptors, and a viewer description is a linked list of frame descriptors. This is a good example of the kind of *generic* data structures the Oberon language supports. In particular, frame descriptors can participate in the data structure even if they defined (as extensions) years after the implementation of the viewer manager.

In Oberon, frames, and in particular viewers, are not just writing and drawing areas. Most importantly, they are also *objects* in the sense of object-oriented programming. This essentially means that frames can individually react on the receipt of *messages*.

Viewer frames, for example, are obliged to handle several different categories of messages. Messages of a first category report on the state change of a viewer. For example, the viewer manager itself sends a message when a viewer becomes visible and must therefore restore its contents, or when the size of a viewer has changed because its lower neighbour was opened, changed, or closed. Messages of a second category demand updating contents after an editing operation, and messages of a third category notify a viewer of input events. The last category of messages has far-reaching consequences on the role of viewers in Oberon. It actually means

that an individual *command interpreter* is bound to every viewer. In the case of viewers displaying text the interpreter could be a text editor, in the case of graphics it could be a graphics editor, and in the case of a viewer representing a mailbox it could be a handler for electronic mail.

So far, we have not explained how objects and messages are realized in Oberon. First of all, we emphasize that we do not just mimic the conventional object-oriented programming style. In contrast to usual object-oriented programs the complete set of messages understood by an object need not be specified together with the definition of the object class. Instead, we explore a more flexible approach: Messages are typically defined in those modules which send them. For example, messages of our first category are defined in the viewer manager module *Viewers*, messages of the second category are defined in the respective editor module (*TextViewers* for text, see next section), and messages of the third category are defined in the input detector module *Oberon*. An appropriate *message handler* would handle all messages known at the time the handler was developed and simply ignore other messages.

Roughly speaking, the Oberon language supports *subclassing* (by the type extension facility), but *messages* and *message handlers* are not institutionalized. We have implemented the above outlined scheme by making use of procedure variables and by applying Oberon's record extension facility to objects *and* messages.

In module *Bitmaps* we declare the types of the handler and the (empty) base message. The base message serves as a root in the hierarchy of frame messages.

```
TYPE FrameMsg = RECORD END;
 Handler = PROCEDURE (Frame, VAR FrameMsg);
```

Module *Viewers* sends messages if a viewer has changed its state:

```
Message = RECORD
  (Bitmaps.FrameMsg) (*extension of the base message*)
  id: INTEGER; (*identification*)
  X, Y, W, H: INTEGER; (*parameters*)
  state: INTEGER (*parameter*)
END;
```

Module *TextViewers* notifies text viewers of editing operations:

```
FrameMsg = RECORD
  (Bitmaps.FrameMsg) (*extension of the base message*)
  id: INTEGER; (*identification*)
  text: Texts.Text; (*parameter*)
  beg, end: LONGINT (*parameters*)
END;
```

Module *Oberon* signals input events:

```
Message = RECORD
  (Bitmaps.FrameMsg) (*extension of the base message*)
  id: INTEGER; (*identification*)
  modes, keys: SET; (*parameter*)
  X, Y: INTEGER; (*parameters*)
  ch: CHAR (*parameter*)
END;
```

Notice that, in contrast to objects, messages are of a temporary nature and that their creation and deletion is determined by a command's control flow. It is therefore advantageous to allocate message records statically on the procedure activation stack rather than dynamically in the system heap. A garbage collector need then not be involved.

A message is sent to a specific object by simply calling its installed handler. For example, message *M* would be sent to viewer *V* by calling *V.handle(V,M)*. In addition, the viewer manager provides a procedure to broadcast a message to all currently visible viewers:

```
PROCEDURE Broadcast (VAR M: Bitmaps.FrameMsg);
```

Broadcasting is used for example after a displayed document has changed. On receipt of such a message, a viewer decides if it is affected. We note that this technique makes the management of multiple views on the same document very easy.

An individual handler must be explicitly attached to every viewer at creation time. Viewer handlers are usually implemented in higher level modules, for example in *TextViewers*, *GraphicViewers*, and *PictureViewers* dealing with viewers displaying texts, graphics, and raster pictures respectively. In the case of a text viewer *V*, creation of an instance looks like

```
NEW(V); V.handle := HandleViewer;
```

where *HandleViewer* is the procedure sketched at the end of the following section. Notice that *HandleViewer* makes extensive use of Oberon's type test (and type guard) to discriminate the different message categories.

In summarizing this section we can say that Oberon's viewer system is based on hierarchic tiling of the screen with the additional possibility of overlaying whole tracks. A viewer plays an interesting double role in Oberon: Firstly, it serves as a display area, and secondly it hosts an individual command interpreter.

The Text System

Text plays a key role in any computer system. Not only are input and output data frequently represented as text, but also objects and commands are usually identified by their name. Text is, therefore, a predefined type of document in the outer core of the Oberon system. The text system comprises three text processing modules: *Texts*, *TextFrames*, and *TextViewers*, and a base module *Fonts* for the representation of characters on screen.

Texts is the base module of the text processing triple. It exports the data type *Text* and *intrinsic* operations on texts. A text descriptor displays the length of the text and the geometry of lines (line space, ascender, and descender of characters). *Open* creates a text from a file with a given name and *Store* is the inverse of *Open*. *ReplaceFont* replaces the font in a given stretch of a text from *beg* to *end*, *Delete* deletes a stretch in a text, and *Insert* inserts the contents of a buffer at a given position in a text.

```
TYPE  Text = POINTER TO TextDesc;

  TextDesc = RECORD
    len: LONGINT; (*length*)
    lsp, asc, dsc: INTEGER; (*line space, ascender, descender*)
  END;

PROCEDURE Open (T: Text; name: ARRAY OF CHAR);
PROCEDURE Store (T: Text; name: ARRAY OF CHAR);

PROCEDURE ReplaceFont (T: Text; beg, end: LONGINT; fnt: Fonts.Font);
PROCEDURE Delete (T: Text; beg, end: LONGINT);
PROCEDURE Insert (T: Text; pos: LONGINT; B: Buffer);
```

Also contained in the definition of *Texts* are procedures to read from and to write to a text. We have devised an interesting concept for sequential reading. We use aggregates which are variants of file riders, so-called *readers* and *scanners*. A reader is used if a text is viewed as a

sequence of characters, and a scanner is used if a text is viewed as a sequence of symbols like number, name, literal string etc.

```
Reader = RECORD
  (Files.Rider) (*extension of Files.Rider*)
  fnt: Fonts.Font (*font of last char read*)
END;

Scanner = RECORD
  (Reader) (*extension of Reader*)
  nextCh: CHAR; (*next char*)
  line: INTEGER; (*line number*)
  class: INTEGER; (*token class*)
  i: LONGINT;
  x: REAL;
  y: LONGREAL;
  c: CHAR;
  len: SHORTINT;
  s: ARRAY 32 OF CHAR
END;

(*class = 0: invalid symbol,
  1: name s (length len)
  2: literal string s (length len)
  3: integer i (decimal or hexadecimal),
  4: real number x,
  5: long real number y
  6: special character c*)
```

Scanning is controlled by the following *lexicographic syntax*:

```
name = NamePart { "." NamePart }.
NamePart = letter { letter | digit }.
LiteralString = '"' { letter | digit | SpecialChar } '"' | "'" { letter | digit | SpecialChar } "'".
number = ["+" | "-"] digit { digit } ["." digit { digit } ["E" | "D" ["+" | "-"] digit { digit } ].
```

Both readers and scanners must be opened at a specified position before they can be used. Each call of procedure *Read* reads the next character, each call of *Scan* reads the next symbol.

```
PROCEDURE OpenReader (VAR R: Reader; T: Text; pos: LONGINT);
PROCEDURE Read (VAR R: Reader; VAR ch: CHAR);

PROCEDURE OpenScanner (VAR S: Scanner; T: Text; pos: LONGINT);
PROCEDURE Scan (VAR S: Scanner);
```

At first sight, writing is similar to reading. There exist so-called *writers* and procedures to open a writer, to set the current font, and to write items of different kinds, for example characters, strings, integers, and real numbers:

```
Writer = RECORD
  (Files.Rider) (*extension of Files.Rider*)
  buf: Buffer; (*associated buffer*)
  fnt: Fonts.Font (*current font*)
END;

PROCEDURE OpenWriter (VAR W: Writer);
PROCEDURE SetFont (VAR W: Writer; fnt: Fonts.Font);
PROCEDURE Write (VAR W: Writer; ch: CHAR);
```

```
PROCEDURE WriteString (VAR W: Writer; s: ARRAY OF CHAR);
PROCEDURE WriteInt (VAR W: Writer; x, n: LONGINT);
PROCEDURE WriteReal (VAR W: Writer; x: REAL; n: INTEGER);
(*right adjust to at least n positions*)
```

Notice that we have the following hierarchy of types:

```
Texts.Scanner
      ↑
Texts.Reader     Texts.Writer
      ↑              ↑
   Files.Rider
```

There is an important fine point concerning the file underlying a writer. It is created as an *anonymous* file when the writer is opened. Because no entry in the directory exists, the disk space used for this file will be reclaimed when the system is booted next time. We can therefore look at the disk space as an extension of collectible memory and at the boot loader as a garbage collector of this space. In order to avoid proliferation of files, a single writer is typically defined globally in every module that contains routines producing text output.

However, when writing to a text, another aspect should be considered. A text is normally displayed in one or more viewers. It would be rather inefficient to display the new state of the text after writing every single character. Therefore, the destination of a writer is a *buffer* rather than a text. The writer's buffer must be inserted into the text whenever a logically complete piece of text has been written. Buffers are also used for saving already existing pieces of text. The following declarations and operations on buffers are available from Texts. They include operations to open a new buffer (is implicit in *OpenWriter*), to save a stretch of a text in a buffer, to copy the contents of a source buffer into a destination buffer, to recall previously deleted text, and to insert the contents of a buffer at a given position in a text.

```
Buffer = POINTER TO BufDesc;

BufDesc = RECORD
  len: LONGINT (*buffer length*)
END;

PROCEDURE OpenBuf (B: Buffer);
PROCEDURE Save (T: Text; beg, end: LONGINT; B: Buffer);
PROCEDURE Copy (SB, DB: Buffer);
PROCEDURE Recall (VAR B: Buffer);
PROCEDURE Insert (T: Text; pos: LONGINT; B: Buffer);
```

Module *Texts* uses an interesting internal data structure to describe texts. We call any contiguous section of a text file a *piece*. Then, at any moment, a text is described by a linked list of piece-descriptors. Operations on texts are in reality operations on the describing piece list. For example, inserting a sequence of characters in a text is realized by splitting the piece containing the insert-position and then inserting the piece describing the sequence. Fig. 12 illustrates this operation. It is noteworthy that buffers are also realized as piece lists rather than actual arrays of characters. We emphasize that the piece data structure is completely encapsulated in the module *Texts*. Clients need not be aware of the existence of pieces.

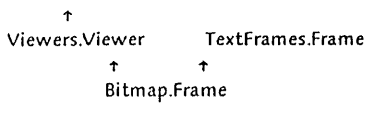
TextFrames is the next higher element in the hierarchy of text processing modules. Its main functions are displaying texts within bitmap frames (typically subframes of viewers) and interpreting frame oriented commands. Each frame descriptor contains a reference to the underlying text, the position *org* of the first displayed character, margin width, type and location of a frame mark (small crossbeam indicating viewer's position within the text or, alternatively, vertical arrow signalling busy processor), a time stamp of the current selection,

and exact locations of caret and selection (if any). The time stamp is typically used by commands to determine the most recent selection(s).

TextFrames exports operations to open a new text frame, restore an open frame, show a certain text position in an open frame, update the display of a replaced, deleted, or inserted stretch of text, and track the mouse to set the caret, select a piece of text, or point at a text word.

```
Frame = POINTER TO FrameDesc;

FrameDesc = RECORD
  (Bitmaps.FrameDesc)
  text: Texts.Text;
  org: LONGINT;
  margW, markH: INTEGER;
  time: LONGINT;
  mark, car, sel: INTEGER;
  carloc: Location;
  selbeg, selend: Location
END;

(*mark < 0: arrow mark
  mark = 0: no mark
  mark > 0: position mark*)

PROCEDURE Open (F: Frame; T: Texts.Text; org: LONGINT; margW: INTEGER);
PROCEDURE Restore (F: Frame);
PROCEDURE Show (F: Frame; pos: LONGINT);

PROCEDURE Replace (F: Frame; beg, end: LONGINT);
PROCEDURE Delete (F: Frame; beg, end: LONGINT);
PROCEDURE Insert (F: Frame; beg, end: LONGINT);

PROCEDURE TrackCaret (F: Frame; X, Y: INTEGER);
PROCEDURE TrackSelection (F: Frame; X, Y: INTEGER; VAR modes: SET);
PROCEDURE TrackWord (F: Frame; X, Y: INTEGER; VAR pos: LONGINT; VAR modes: SET);
```

Notice that the type *Frame* is another extension of the base type *Bitmaps.Frame*. We thus have the following hierarchy of types:



The internal data structure of a text frame is a simple linked list of line descriptors. By summarizing visible lines line descriptors optimize the time-critical process of locating a character at a position specified by the mouse.

In the section on viewers we have seen that a handler or command interpreter is associated with every Oberon viewer. Module *TextViewers* elaborates one such command interpreter for text viewers. A text viewer contains two text subframes, a *menu* frame as a header and a *main text* frame, and the command interpreter is actually an editor operating on these two subframes. TextViewers exports a procedure to create and initialize a new text viewer:

```
PROCEDURE NewViewer (menu, text: Texts.Text; org: LONGINT;
  X, Y: INTEGER): Viewers.Viewer;
```

Therefore, the command interpreter as a whole need not be exported. However, *TextViewers* exports some of its elements, for example procedures to edit a menu or a main text frame, to update a text frame in reaction to a change of the underlying text, and to write a character at the caret's location. The idea behind exporting elements is that implementors of similar command interpreters can make use of these elements. For example, an implementor of an interpreter for graphic viewers consisting of a menu frame and a graphic frame can use *EditMenu* instead of reprogramming it.

```
PROCEDURE EditMenu (V: Viewers.Viewer; F: TextFrames.Frame;
  X, Y: INTEGER; Modes, Keys: SET);
PROCEDURE EditText (V: Viewers.Viewer; F: TextFrames.Frame;
  X, Y: INTEGER; Modes, Keys: SET);
PROCEDURE Update (F: TextFrames.Frame; VAR M: FrameMsg);
PROCEDURE Write (F: TextFrames.Frame; ch: CHAR);
```

As can be seen from the following refinement of the command interpreter, editing in the menu frame is done by a different routine than editing in the main text frame. Essentially, menu operations are restricted (by the command interpreter) to non-destructive manipulations.

```
1   PROCEDURE HandleViewer(V: Bitmaps.Frame; VAR M: Bitmaps.FrameMsg);
2   VAR Menu, Text: TextFrames.Frame;
3   BEGIN
4   WITH V: Viewers.Viewer DO
5     Menu := V.dsc(TextFrames.Frame); Text := V.dsc.next(TextFrames.Frame);
6     IF M IS Oberon.Message THEN
7       WITH M: Oberon.Message DO
8         IF M.id = Oberon.defocus THEN Defocus(V)
9         ELSIF M.id = Oberon.neutralize THEN Neutralize(V)
10        ELSIF M.id = Oberon.consume THEN
11          IF Text.car # 0 THEN
12            IF M.ch = ENTER THEN Call(V, Text, Text.carloc.org)
13              ELSE Write(Text, M.ch)
14            END
15          END
16        ELSIF M.id = Oberon.track THEN
17          IF (M.X >= V.X) & (M.X < V.X + V.W) & (V.Y <= M.Y) THEN
18            IF M.Y < V.Y + V.H - menuH THEN
19              EditText(V, Text, M.X, M.Y, M.modes, M.keys)
20            ELSIF M.Y < V.Y + V.H - barH THEN
21              EditMenu(V, Menu, M.X, M.Y, M.modes, M.keys)
22            ELSIF M.Y < V.Y + V.H THEN
23              IF left IN M.keys THEN Change(V, M.X, M.Y, M.modes, M.keys)
24                ELSE EditMenu(V, Menu, M.X, M.Y, M.modes, M.keys)
25              END
26            END
27          END
28        END
29      END
30    ELSIF M IS Viewers.Message THEN
31      WITH M: Viewers.Message DO
32        IF M.id = Viewers.restore THEN Restore(V)
33          ELSIF M.id = Viewers.modify THEN Modify(V, M.Y)
34          ELSIF M.id = Viewers.suspend THEN Suspend(V)
35        END
```

```
36          END
37        ELSIF M IS FrameMsg THEN
38          WITH M: FrameMsg DO
39            IF Text.text = M.text THEN Update(Text, M) END
40          END
41        END
42      END
43    END HandleViewer;
```

Explanations:

```
1     procedure of type Bitmaps.Handler
2     local variables Menu and Text point to subframes of handled viewer
4     type guard; V must be a viewer frame
5     initialize subframe pointers
6     type test; is message an Oberon (input) message?
7     type guard
8     if message demands defocussing then defocus viewer
9     if message demands neutralizing then neutralize viewer
10    if message demands consuming then consume a character
11    if caret is active in main text frame
12    if character is ENTER then call command specified by caret's text line
13    if not, echo the character to caret's location
16    if message demands mouse tracking
17    if the mouse is located within viewer
18    if the mouse is located within main text frame
19    edit main text (pass initial values as parameters)
20    if the mouse is located within menu frame
21    edit menu text
22    if the mouse is located in the title-bar
23    if left mouse key is depressed then change viewer size
24    if not, edit menu text
30    type test; is message a Viewers message?
31    type guard
32    if message demands restoring then restore viewer
33    if message demands modification then modify viewer
34    if message demands suspending then suspend viewer
37    type test; is message a TextViewers.FrameMsg (update message)?
38    type guard
39    update frame if the viewer's main text is concerned
```

Although the command interpreter is the most important ingredient of *TextViewers*, this module has a second face that can best be characterized by *ViewerTexts*. It exports procedures which operate on texts that are displayed in viewers:

```
PROCEDURE ReplaceFont (T: Texts.Text; beg, end: LONGINT; fnt: Fonts.Font);
PROCEDURE Delete (T: Texts.Text; beg, end: LONGINT);
PROCEDURE Insert (T: Texts.Text; pos: LONGINT; buf: Texts.Buffer);
```

These procedures differ from their counterparts in module *Texts* in the fact that they automatically reestablish consistency of text and display by sending appropriate broadcast messages to all viewers.

Oberon texts are sequences of characters rather than merely sequences of ASCII-codes. In particular, a *font* is associated with each character. The modules *Fonts* and *Bitmaps* support the management of fonts and the display of characters in diverse fonts. *Fonts* exports the data type

Font and procedures to (load and) get a font from its name and to get the default font respectively. A font descriptor contains the name and the height of the described font and the extent of a hypothetical bounding box spanned by the set of all characters in the font. Most importantly, font descriptors also contain a pointer to an array of character descriptors. Each character descriptor contains information about the character's bounding box, the distance to the next character, and the raster data. This information can be accessed by calling procedure *GetChar* from module *Bitmaps*.

in module *Fonts*:

```
TYPE Font = POINTER TO FontDesc;

FontDesc = RECORD
  name: Name;
  height, minX, maxX, minY, maxY: INTEGER;
  raster: Bitmaps.Font
END;

PROCEDURE This (name: ARRAY OF CHAR): Font;
PROCEDURE Default (): Font;
```

in module *Bitmaps*:

```
PROCEDURE GetChar (f: Font; ch: CHAR; VAR dx, x, y, w, h: INTEGER; VAR p: Pattern);
```

The Concept of Tools

A *tool package*, or *tool* for short, in Oberon is a package of logically connected commands. It is quite essential that the set of tools is not fixed once and for all. In order to tailor a system to specific needs, an expert user would perhaps create new tools or modify existing tools. In the hierarchy of modules, tools are located at the top (see Figure 3). As is shown in Appendix 2, programmers of tools usually make extensive use of Oberon's system base and library modules.

There are some basic standard tools in Oberon. The *Display* tool, for example, offers commands to manipulate tracks and viewers on the screen. The *Edit* tool provides commands to edit and print texts that are displayed in text viewers. The *System* tool contains commands to get system-oriented information and to specify system-parameters. The *Compiler* tool features the Oberon compiler, and the *Diskette* tool handles backup to and restore from diskette.

An Oberon station is typically part of a local area network. Consequently, there are commands to access remote servers. For example, the *print* command in the Edit tool sends the text to a laser printer server. In addition, there is a special *Server* tool supporting the handling of electronic mail and the transfer of files from and to remote stations. It is perhaps noteworthy that Oberon's electronic mail server is smoothly integrated in the text system. The server tool represents received messages as plain text that is immediately available for arbitrary editing operations. Conversely, the tool allows to send as a message any text obeying the syntax of a message (text with a header).

As mentioned in the chapter *Principles of Design and Operation* the term *tool* has another, however related meaning. It also designates a text form that contains a list of command names, possibly together with predefined parameters. Normally, a separate tool form exists for each tool package. A user typically selects commands from several tool forms that are simultaneously displayed in the system track.

Because tool forms are ordinary texts, an Oberon user can easily setup individual variants of tool forms. For example, the diskette tool might look as follows. The exclamation marks are guards which prevent unintentional execution of destructive commands, and the symbol ~ terminates the argument lists. Notice that the last command in this example tool may be regarded as a backup server for a specific project.

```
Diskette.Read ~
Diskette.Write ~
Diskette.Write
  Texts.Def Texts.Mod
  TextFrames.Def TextFrames.Mod
  TextViewers.Def TextViewers.Mod ~
!Diskette.Format
!Diskette.Initialize
```

Symmary and Conclusions

The system Oberon deviates from conventional operating systems in several respects:

- The notion of program is absent; instead of a program activation, the procedure call is the unit of action specified by the computer's operator.
- Each procedure call (command) is an atomic action within the dialog between the operator and the computer: the switch from one task to another occurs between the user's commands rather than between two arbitrary machine instructions.
- Commands take their input from texts and from other kinds of documents rather than from the keyboard. Instead of writing directly onto the screen, commands generate non-volatile output in the form of (displayed) data structures.
- The interface between two consecutive actions consists of abstract data structures (texts, graphics) in main store, rather than of files on disk. When displayed in viewers, they are editable.
- Oberon provides distributed command interpretation. Viewers are regarded as rectangular areas on the screen that are capable of interpreting commands individually. To that purpose, the object-oriented programming paradigm is used. A message is sent to a viewer whenever an input event refers to it.
- Oberon features a simple and extremely efficient file system. The disk directory is organised as a B-tree. A clear distinction is made between a file and aggregates to access it, which are called riders.
- Modules are loaded under Oberon only when they are actually used. Delayed loading is important because packages may statically consist of dozens of modules, of which only a few are used for every specific application. Delayed loading is controlled by page faults, which are caused by the virtual address mechanism.
- A garbage collector is built into the Oberon Kernel. Instead of running as a separate process, the garbage collector is explicitly activated between commands under the precondition of a void stack. This precondition simplifies and accelerates the algorithm significantly.
- The system and the user packages are implemented in a language offering data type extension and polymorphic operations with guaranteed type safety. Full type safety is mandatory for a system relying on automatic storage retrieval.
- Oberon can be extended (possibly years later) by declaring new data types which are extensions of existing, imported types. Objects of extended types are compatible with objects of their base type, and therefore can be integrated in existing data structures.
- There is no deep ditch in Oberon separating users from programmers. Having a powerful module basis at their disposal, users can extend the system or adapt it to their needs by programming new tools.

The design of both system and language was guided by the desire to free the computer's user from artificial constraints imposed by traditional operating systems. This required the courage to experiment, and it was made feasible by the ample storage and computing resources offered by modern hardware. The main challenge lies in making innovative use of them, rather than letting systems of the conventional flavour simply adapt their size to that of available memory.

Although the design of Oberon was influenced by Cedar [5] in several respects, the two systems differ in their size by orders of magnitude. As shown in Fig. 13, the Oberon system, including the compiler, is specified by 15000 lines of source program. The compiled code consists of 150 kilobytes, and compilation takes about five minutes. Both in source form and in object form, the whole system fits easily on a single double-sided 3.5 inch diskette.

References

1. N. Wirth.
The Programming Language Oberon.
Software - Practice and Experience, 18 (1988).

2. P. Rovner, R. Levin, J. Wick.
On Extending Modula-2 for Building Large, Integrated Systems.
DEC-SRC Techn. Rep. Palo Alto, Jan. 1985.

3. H. Schorr and W. Waite.
An efficient machine-independent procedure for garbage collection in various list structures.
Comm. ACM 10, 8 (Aug. 1967), 501 - 505.

4. D. Comer.
The ubiquitous B-tree.
ACM Comp. Surveys, 11, 2 (June 1979), 121 - 137.

5. W. Teitelman.
A Tour Through Cedar.
IEEE Software, 1, 2 (April 1984), 44-73.

Appendix 1: Architectural Support for Module Structure

The NS 32000 processor contains a few registers, an addressing mode, and special call and return instructions which support the realization of a module structure without the need for a linking phase. Each module is represented by three blocks of storage holding data, code, and a link table respectively. Their base addresses are contained in a descriptor, and the address of the descriptor of the module currently under execution is stored in the processor's MOD register. The address of the data block in the SB register (static base) can be regarded as an optimization feature, accelerating access to global variables.

The presence of these registers requires that they be updated each time control switches from a module A to a module B. Fortunately, there exist instructions for procedure call (CXP) and return (RXP) which include these updates in addition to affecting the program counter PC. The effect of the instructions CXP k and RXP n is explained below and additionally illustrated in Fig. 14. M denotes memory as an array of bytes. A module descriptor is denoted as a record with three fields named *data*, *link*, and *code*. The link table is an array of procedure descriptors, each identifying an external procedure, and each being a record with two fields named *mod* and *pc* representing the module to which the procedure belongs and the offset of its entry point in the code block. k is the index of the referenced descriptor. In Fig. 14, the register values *after* the call are shaded.

```
ModuleDescriptor = RECORD
        SB, LB, PB: LONGINT
    END ;

ProcedureDescriptor = RECORD
        mod, pc: INTEGER
    END ;

CXP k:  SB := M[MOD].data;
        pd := M[M[MOD].link + 4*k];
        SP := SP-4; M[SP] := MOD;
        SP := SP-4; M[SP] := PC;
        PC := M[MOD].code + pd.pc;
        MOD := pd.mod

RXP n:  PC := M[SP]; SP := SP+4;
        MOD := M[SP]; SP := SP+4;
        SB := M[MOD].data; SP := SP+n
```

It may be noted that the CXP and RXP instructions require 6 and 3 memory accesses respectively. This is the price to be paid for the late binding of modules and for the elimination of an explicit module linker. The "overhead" is noticeable, and it is important that calls within one module use the simpler BSR (branch subroutine) and RTS instructions, each requiring a single memory reference only.

```
BRS n:  SP := SP-4; M[SP] := PC;
        PC := PC+n

RTS n:  PC := M[SP]; SP := SP+4;
        SP := SP+n
```

Variable access is supported by three addressing modes and an additional base register (FP) pointing at the most recent procedure activation record in the stack. Hence, local variables are addressed with FP as base, global variables (of the module currently holding control) with SB as base, and static variables in other modules with the external mode. The latter requires in addition to the offset n the specification of the module, given in terms of a link table index k.

$$EXT(k, n) = M[M[MOD].link] + 4*k] + n$$

The formula shows that access of an external variable requires two memory references in addition to the one for the variable's value. This overhead is quite acceptable in view of the infrequency of external references.

Appendix 2: Examples of Applications of the System Library

In this section we demonstrate how implementors of applications or tool packages can use the Oberon system's powerful module base. Notice a stylistic peculiarity: Applications typically import numerous rather abstract data types and profit from library functions operating on objects of these types. Examples of frequently imported types are *Oberon.ParList, Viewers.Viewer, Fonts.Font, Texts.Text, Texts.Reader, Texts.Scanner, Texts.Buffer,* and *TextFrames.Frame.*

The first two examples show how an applicatibn can open a viewer and display data. The third example shows how to enlarge an existing viewer, and the next example explains how a command determines its operands generically. The last four examples show how to implement an edit tool. They are included in this collection to demonstrate the adaptability of the Oberon system to specific requirements.

Example 1. Open a viewer in the system track, generate, and display text data.

```
PROCEDURE Directory;
VAR Text: TextFrames.Frame; V: Viewers.Viewer; f: File;
BEGIN
V := TextViewers.NewViewer(
      NewMenu("Diskette.Directory"),
      NewText(""), 0,
      Oberon.SX,
      Oberon.SY());
  Text := V.dsc.next(TextFrames.Frame); (*access main text frame*)
  T := Text.text; (*text of main text frame*)
  TextFrames.Mark(Text, -1); (*setup vertical arrow mark*)
  (*read directory and first entry to f*)
  WHILE (*f is a file*) DO
    Texts.WriteString(W, f.file.name);
    Texts.Write(W, " "); Texts.WriteInt(W, f.file.size, 1);
    Texts.WriteLn(W);
    (*assign next entry to f*)
  END;
  TextViewers.Insert(T, T.len, W.buf); (*insert at T's end and display written text*)
  TextFrames.Mark(Text, 1) (*restore position mark*)
END Directory;
```

where

```
VAR T: Texts.Text; W: Texts.Writer;
```

are globally defined, and *W* is globally initialized by *Texts.OpenWriter(W)*, and *NewMenu* and *NewText* are functions creating and initializing a menu text and the main text respectively:

```
PROCEDURE NewText (name: ARRAY OF CHAR): Texts.Text;
  VAR T: Texts.Text;
BEGIN NEW(T); Texts.Open(T, name); RETURN T
END NewText;
```

```
PROCEDURE NewMenu (name: ARRAY OF CHAR): Texts.Text;
  VAR T: Texts.Text;
BEGIN T := NewText("");
  Texts.WriteString(W, name); Texts.WriteLn(W); Texts.WriteLn(W);
  Texts.WriteString(W, "Display.Close Edit.Copy Edit.Grow Edit.Locate Edit.Save");
  Texts.WriteLn(W);
  TextViewers.Insert(T, 0, W.buf);
  RETURN T
END NewMenu;
```

Remarks:

1. Normally, one writer per module is sufficient.
2. The above program generates its whole output text before displaying it. Alternatively, if the statement *TextViewers.Insert(T, T.len, W.buf)* is moved into the WHILE-loop, every generated line is displayed immediately.
3. If a specific part of the output text is to appear in a new font, for example in bold face *Syntax10b.Scn.Fnt* or italics *Syntax10i.Scn.Fnt*, call *Texts.SetFont(W, Fonts.This(Syntax10b.Scn.Fnt))* before writing the part and *Texts.SetFont(W, Fonts.Default())* before continuing to write ordinary text.
4. *Oberon.SX* specifies the start of the system track. *Oberon.SY()* is a standard proposal for the placing of a new system viewer containing output data that was produced by a command. Of course, individual algorithms are possible as well. For example, if the star-shaped marker is allowed to override standard placing, the algorithm is

```
PROCEDURE SY (): INTEGER;
BEGIN
  IF Oberon.Marker.on & (Oberon.Marker.X >= Oberon.SX) THEN
    RETURN Oberon.Marker.Y + Viewers.barH DIV 2
  END;
  RETURN Oberon.SY()
END SY;
```

5. Oberon provides another standard allocation procedure *Oberon.LY()* for system viewers. It is used to allocate so-called *log-viewers*, i.e. viewers reporting on the progress in the execution of a command.

Example 2. Open a viewer in the user track and display an existing text specified by a parameter.

```
PROCEDURE OpenText;
  VAR par: Oberon.ParList;
    Text: TextFrames.Frame;
    S: Texts.Scanner;
BEGIN
  par := Oberon.Par(); (*access parameters*)
  Text := par.frame(TextFrames.Frame); (*calling frame*)
  TextFrames.Mark(Text, -1); (*arrow mark*)
```

```
    Texts.OpenScanner(S, par.text, par.pos); (*open scanner at position of parameter list*)
    Texts.Scan(S); (*get first symbol*)
    IF S.class = 1 THEN (*if symbol is a string*)
      V := TextViewers.NewViewer(
        NewMenu(S.s),
        NewText(S.s), 0,
        Oberon.UX,
        Oberon.UY())
    END;
    TextFrames.Mark(Text, 1) (*restore position mark*)
  END OpenText;
```

Remark:

Oberon.UX specifies the start of the user track. *Oberon.UY()* is a standard proposal for the placing of a new viewer in the user track. Again, individual algorithms (respecting perhaps the marker's state) are possible as well.

Example 3. Enlarge a text viewer.

```
  PROCEDURE EnlargeViewer;
    VAR par: Oberon.ParList;
      Menu, Text: TextFrames.Frame;
      V, newV: Viewers.Viewer;
      X, W: INTEGER;
  BEGIN
    par := Oberon.Par(); (*access parameters*)
    V := par.vwr;
    IF (V.dsc # NIL) & (V.dsc.next IS TextFrames.Frame) THEN (*if text viewer*)
      IF V.H < Viewers.DH THEN X := V.X; W := V.W (*grow to size of track*)
      ELSE X := 0; W := Viewers.DW (*grow to size of display*)
      END;
      Menu := V.dsc(TextFrames.Frame); Text := V.dsc.next(TextFrames.Frame);
      Oberon.OpenTrack(X, W); (*open new standard overlaying track*)
      newV := TextViewers.NewViewer(Menu.text, Text.text, Text.org, X, Viewers.DH) (*open
      copy*)
    END
  END EnlargeViewer;
```

Remark:

Actually, a copy of the original viewer is opened in the new track. When this track is later closed, the original viewer will reappear.

Example 4. Process a viewer text or a sequence of texts, depending on command's location.

```
  PROCEDURE ProcessText;
    VAR par: Oberon.ParList;
      Text: TextFrames.Frame;
      S: Texts.Scanner;
```

```
    T: Texts.Text;
BEGIN
  par := Oberon.Par(); (*access parameters*)
  IF par.frame = par.vwr.dsc THEN (*command in menu frame*)
    IF par.vwr.dsc.next IS TextFrames.Frame THEN
      Text := par.vwr.dsc.next(TextFrames.Frame); (*main text frame*)
      TextFrames.Mark(Text, -1) (*set arrow mark*)
      Process(Text.text); (*process displayed text*)
      TextFrames.Mark(Text, 1) (*restore position mark*)
    END
  ELSE (*command in main text frame*)
    Text := par.frame(TextFrames.Frame);
    TextFrames.Mark(Text, -1) (*set arrow mark*)
    Texts.OpenScanner(S, par.text, par.pos); (*open scanner at position of parameter list*)
    Texts.Scan(S); (*get first symbol*)
    WHILE S.class = 1 THEN (*while symbol is a string*)
      Texts.Open(T, S.s); (*open text from file*)
      (*process text T*)
      Texts.Scan(S) (*get next symbol*)
    END;
    TextFrames.Mark(Text, 1) (*restore position mark*)
  END;
END ProcessText;
```

Example 5. Delete a selected part of text in the marked viewer.

```
PROCEDURE Delete;
  VAR Text: TextFrames.Frame; v: Viewers.Viewer;
BEGIN
  v := Viewers.This(Oberon.Marker.X, Oberon.Marker.Y); (*get marked viewer*)
  Text := v.dsc.next(TextFrames.Frame); (*main text frame of marked viewer*)
  IF Text.sel > 0 THEN (*if there exists a selection*)
    TextViewers.Delete(Text.text, Text.selbeg.pos, Text.selend.pos) (*delete text*)
  END
END Delete;
```

Example 6. Copy the most recently selected text part to the caret's position.

```
PROCEDURE CopyText;
  VAR Text: TextFrames.Frame; buf: Texts.Buffer; v: Viewers.Viewer;
BEGIN
  FindSelection(T, beg, end); (*find most recent selection*)
  IF T # NIL THEN (*if found*)
    Texts.OpenBuffer(buf);
    Save(T, beg, end, buf); (*save text in buffer*)
    v := Oberon.Focus; (*get focus viewer*)
    IF (v.dsc # NIL) & (v.dsc.next IS TextFrames.Frame) THEN (*if text viewer*)
      Text := v.dsc.next(TextFrames.Frame); (*main text frame*)
```

```
      IF Text.car > 0 THEN (*if caret set*)
        TextViewers.Insert(Text.text, Text.carloc.pos, buf) (*insert text at caret's position*)
      END
    END
  END
END CopyText;
```

where

```
PROCEDURE FindSelection (VAR T: Texts.Text, VAR beg, end: LONGINT);
  VAR F: Bitmaps.Frame; time: LONGINT; X: INTEGER;
BEGIN
  T := NIL; time := -1; X := 0;
  WHILE X # Viewers.DW DO (*another track*)
    V := Viewers.This(X, 0); (*get bottom viewer*)
    WHILE V.state > 1 DO (*if not filler*)
      F := V.dsc; (*first frame in list*)
      WHILE (F # NIL) & (F IS TextFrames.Frame) DO
        WITH F: TextFrames.Frame DO
          IF (F.sel > 0) & (F.time > time) THEN (*newer selection*)
            T := F.text; beg := F.selbeg.pos; end := F.selend.pos;
            time := F.time
          END
        END;
        F := F.next (*next frame in list*)
      END;
      V := Viewers.Upper(V) (*get upper neighbour*)
    END;
    X := X + V.W (*move to next track*)
  END
END FindSelection;
```

Example 7. Copy font from visibly marked position to the most recent text selection.

```
PROCEDURE CopyFont;
  VAR F: TextFrames.Frame;
    T: Texts.Text;
    v: Viewers.Viewer;
    beg, end: LONGINT;
    X, Y: INTEGER;
    ch: CHAR;
BEGIN
  FindSelection(T, beg, end); (*find most recent selection*)
  IF (T # NIL) & Oberon.Marker.on THEN (*if found and marker visible*)
    X := Oberon.Marker.X; Y := Oberon.Marker.Y;
    v := Viewers.This(X, Y); (*marked viewer*)
    IF (v.dsc # NIL) & (v.dsc.next IS TextFrames.Frame) THEN
      F := v.dsc.next(TextFrames.Frame);
      IF (X >= Text.X) & (X < F.X + F.W) & (Y >= F.Y) & (Y < F.Y + F.H) THEN
```

```
        Texts.OpenReader(R, F.text, TextFrames.Pos(F, X, Y)); (*position reader*)
        Texts.Read(R, ch); (*read marked char*)
        TextViewers.Change(T, beg, end, R.fnt) (*change font*)
      END
    END
  END
END CopyFont;
```

Example 8. Move caret to the next character written in italics.

```
PROCEDURE SearchItalics;
  VAR Text: TextFrames.Frame;
    R: Texts.Reader;
    italic: Fonts.Font;
    v: Viewers.Viewer;
    pos: LONGINT;
    ch: CHAR;
BEGIN
  italic := Fonts.This("Syntax10.Scn.Fnt");
  v := Oberon.Focus; (*get focus viewer*)
  IF (v.dsc # NIL) & (v.dsc.next IS TextFrames.Frame) THEN (*if text viewer*)
    Text := v.dsc.next(TextFrames.Frame); (*main text frame*)
    IF Text.car > 0 THEN (*if caret set*)
      Texts.OpenReader(R, Text.text, Text.carloc.pos); (*open reader at caret's position*)
      Texts.Read(R, ch);
      WHILE (ch # 0X) & (R.fnt # italic) DO Texts.Read(R, ch) END; (*read char stream*)
      IF ch # 0X THEN (*not end of text*)
        pos := Texts.Pos(R); (*reader's position*)
        TextFrames.RemoveSelection(Text); (*remove all marks*)
        TextFrames.RemoveCaret(Text);
        Oberon.RemoveMarks(Text.X, Text.Y, Text.W, Text.H);
        TextFrames.Show(Text, Max(0, pos - 200)); (*show text at pos*)
        Oberon.SetFocus(v); (*define v as focus viewer*)
        TextFrames.SetCaret(Text, pos) (*set caret to new position*)
      END
    END
  END
END SearchItalics;
```

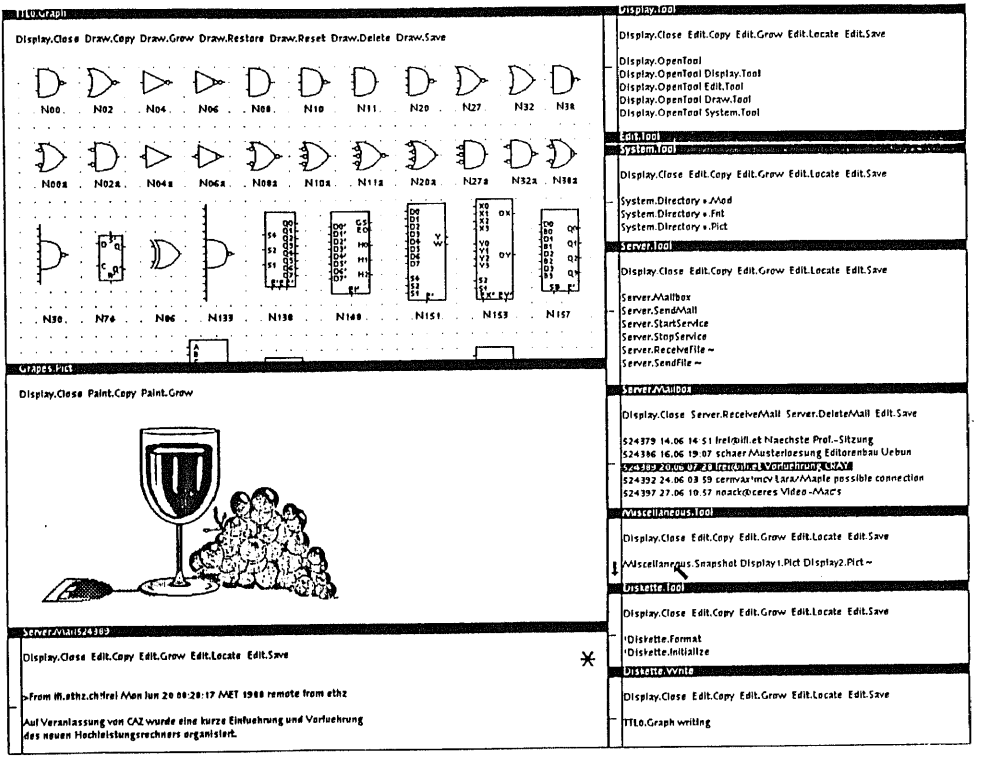


Fig. 1. Typical Oberon Screen Layout

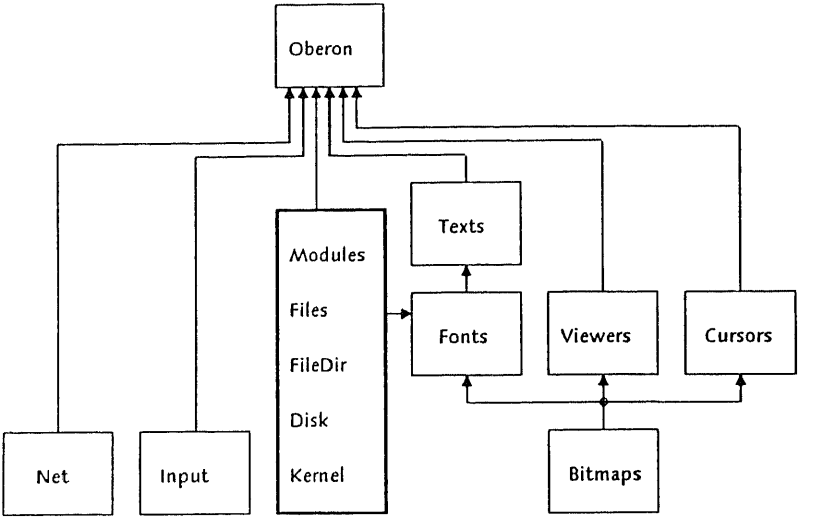


Fig. 2. The Outer Core

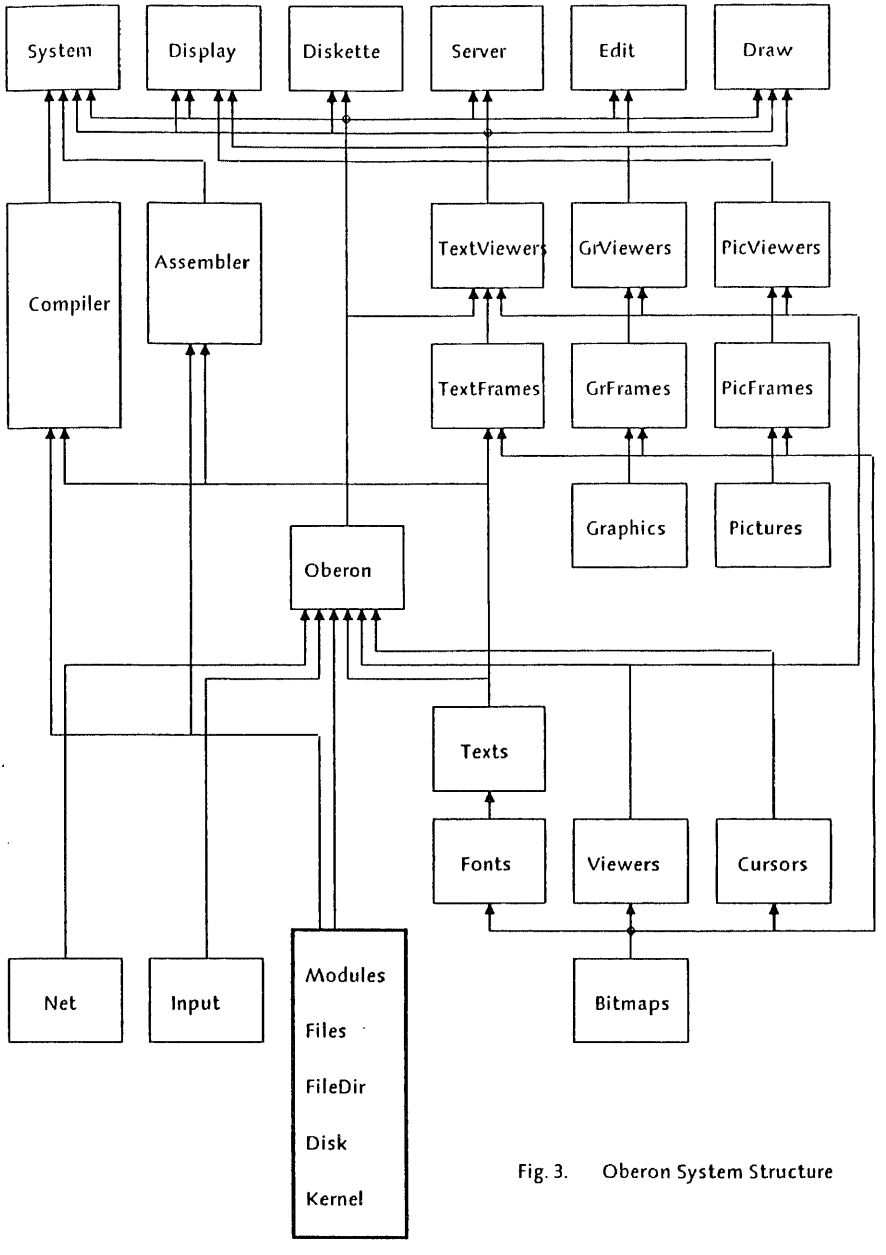


Fig. 3. Oberon System Structure

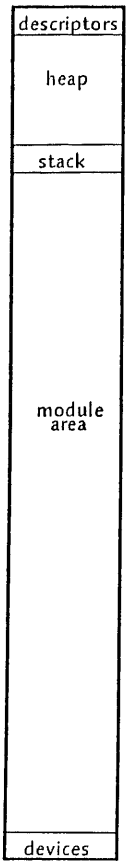


Fig. 4

Virtual Address Space

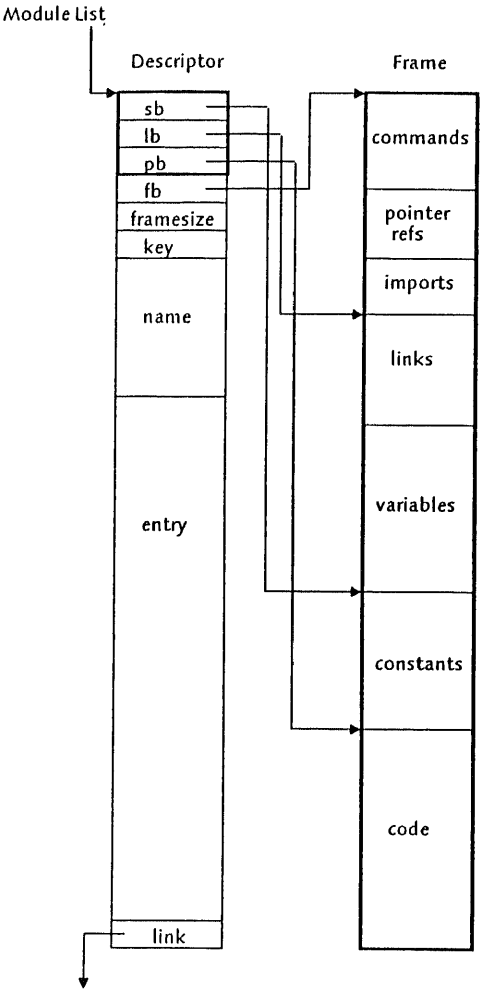


Fig. 5.

Module Descriptor and Frame Structures

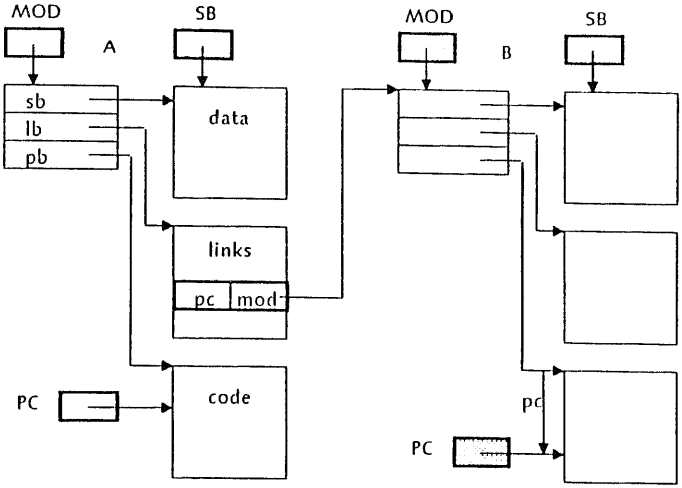


Fig. 14. Module Descriptors and Frames, and Transfer of Control

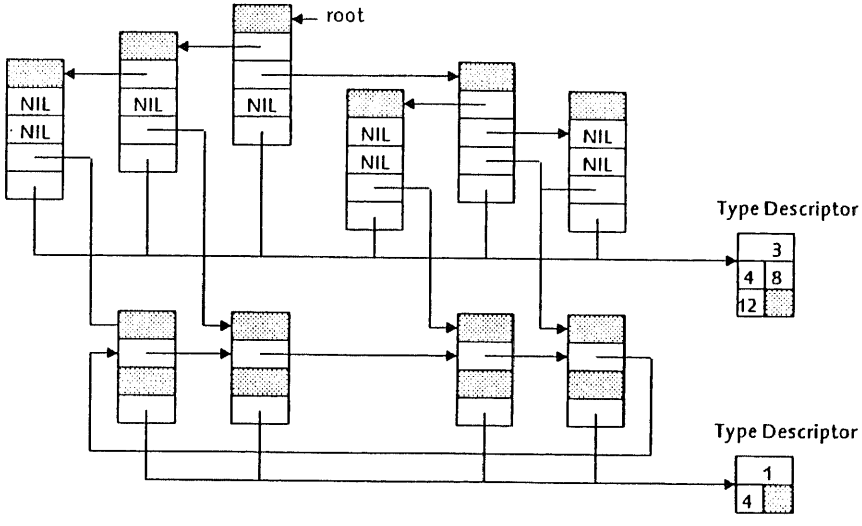


Fig. 6. Data Structure and Type Descriptors

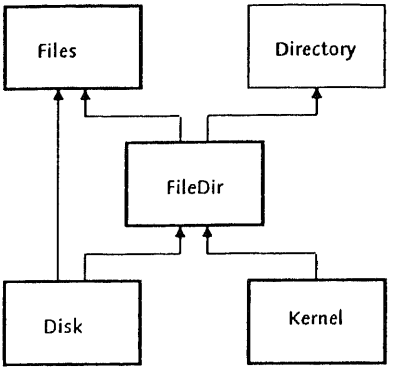


Fig. 8. Configuration of File System

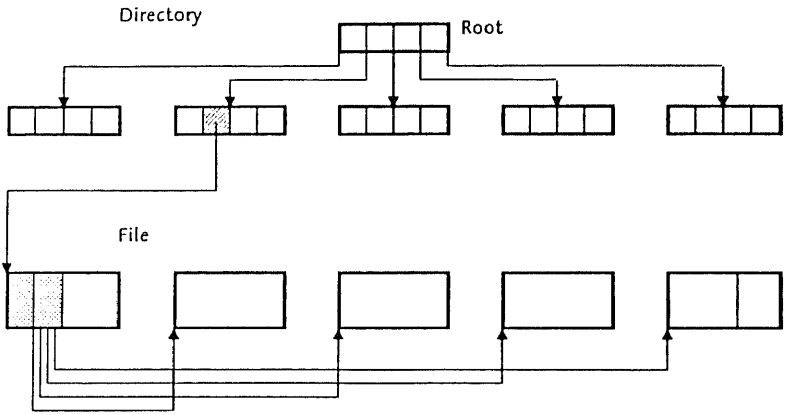


Fig. 10. File and Directory on Disk

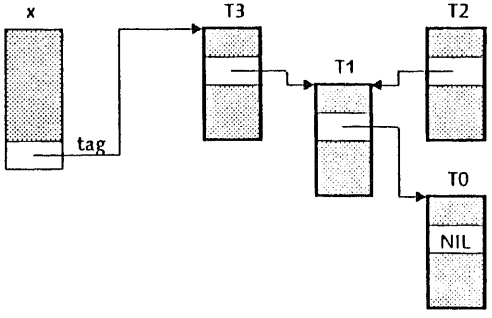


Fig. 7. Type Descriptors for T0 and its Extensions

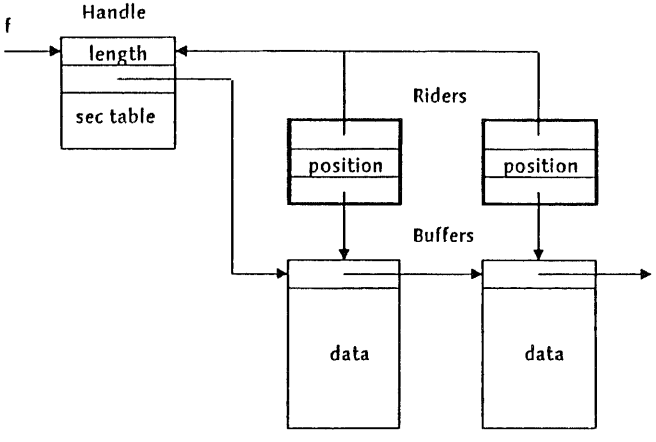


Fig. 9. File Data in Main Store

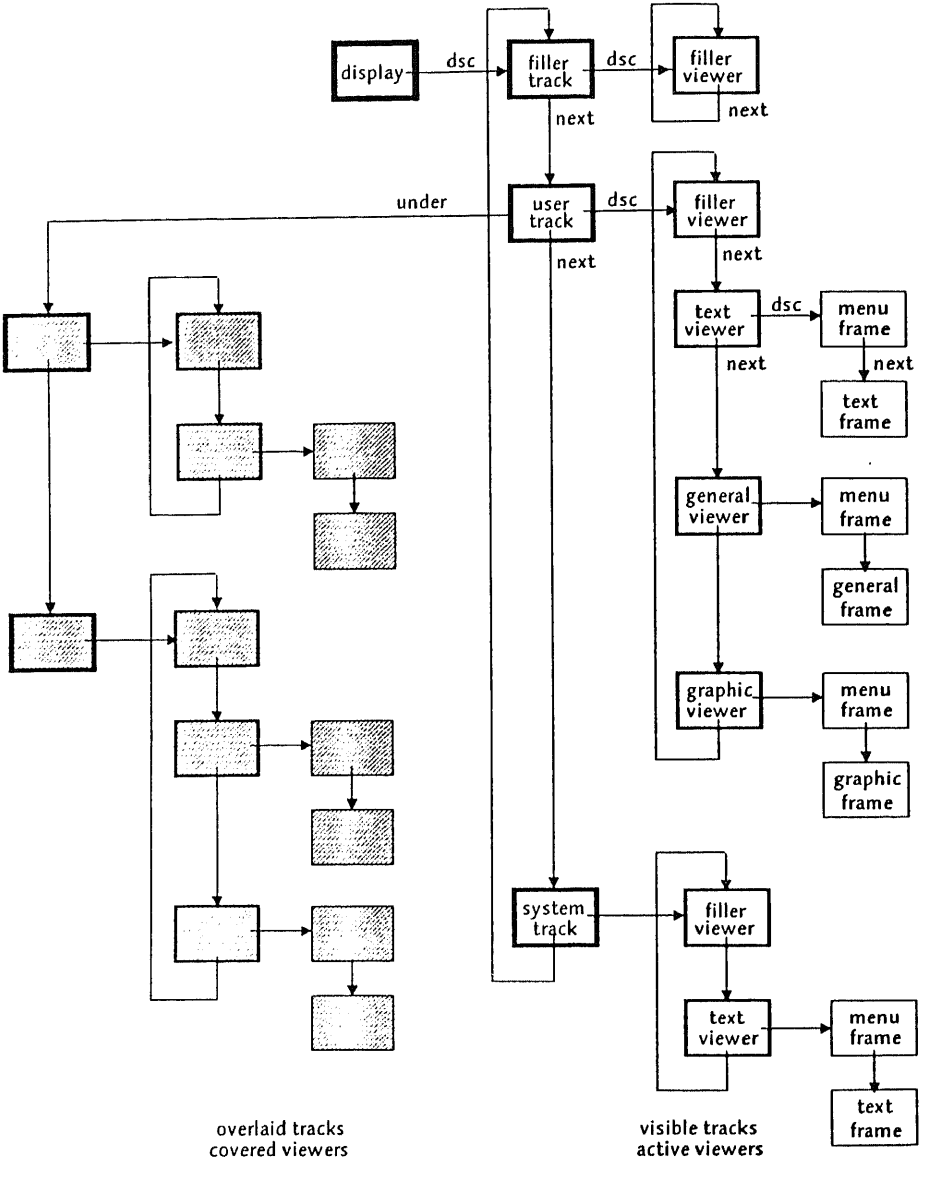


Fig. 11. Internal Data Structure of the Viewer Manager

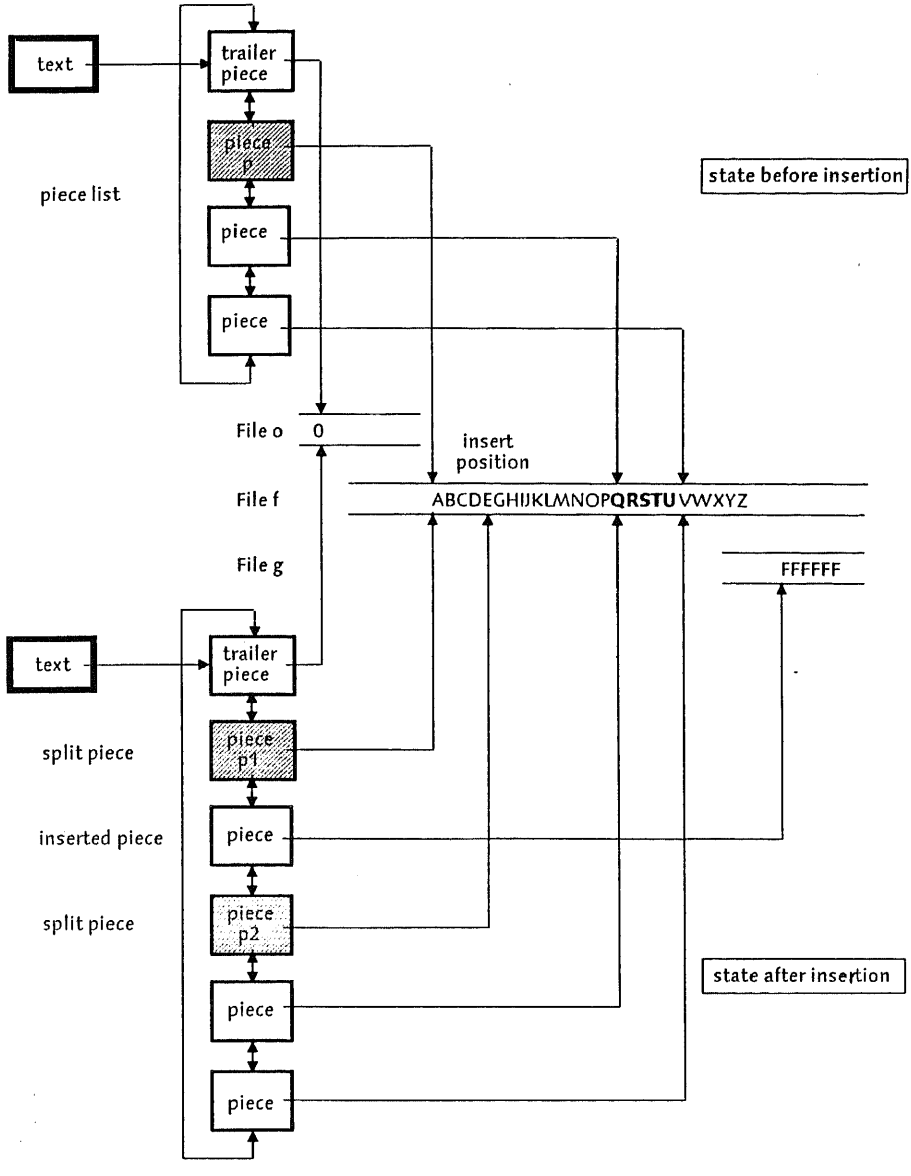
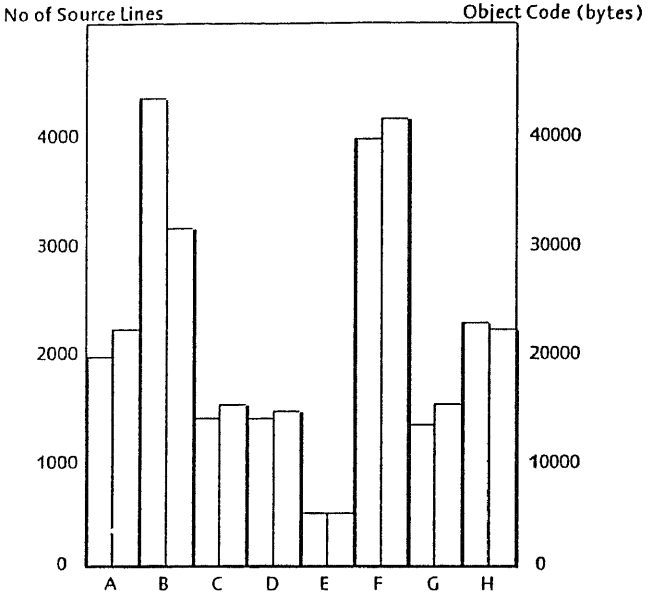


Fig. 12. Internal Data Structure of Texts



A: Inner Core	Kernel, Disk, FileDir, Files, Modules
B: Outer Core	Input, SCC, Printer, Bitmaps, Cursor, Viewers, Fonts, Texts, Oberon
C: Text System	TextFrames, TextViewers, Edit
D: Graphics System	Graphics, GraphicFrames, GraphicViewers, Draw
E: Picture System	Pictures, PictureFrames, PictureViewers, Paint
F: Compiler	
G: Assembler	
H: Tools	Display, System, Mailer, FileTransfer, Diskette, V24

Fig. 13. The Size of the Oberon System