Project OBERON

the Design
of
an Operating System (OS),
a Compiler,
and
a Computer

Niklaus Wirth
Jürg Gutknecht

2013 revised
ISBN 0-201-54428-8

PO

PREFACE

0.1 1^{st} EDITION

This book presents the results of Project Oberon, namely an entire software environment for a modern workstation. The project was undertaken by the authors in 1986-89, and its primary goal was to design and implement an entire system from scratch, and to structure it in such a way that it can be described, explained, and understood as a whole. In order to become confronted with all aspects, problems, design decisions and details, the authors not only conceived but also programmed the entire system described in this book, and more.

Although there exist numerous books explaining principles and structures of OSes, there is a lack of descriptions for one actually implemented and used. We wished not only to give advice on how a system might be built, but to demonstrate how one was built. Program listings therefore play a key role in this text, because they alone contain the ultimate explanations. The choice of a suitable formalism therefore assumed great importance, and we designed the language Oberon as not only an effective vehicle for implementation, but also as a publication medium for algorithms in the spirit in which Algol 60 had been created three decades ago. Because of its structure, the language Oberon is equally well suited to exhibit global, modular structures of programmed systems.

In spite of the small number of man-years spent on realizing the Oberon, and in spite of its compactness letting whose description fit a single book, it is not an academic toy, but rather a versatile workstation system that has found many satisfied and even enthusiastic users in academia and industry. The core system described here, consisting of storage, file, display, text, and viewer managers, of program loader and device drivers, draws its major power from a suitably chosen, flexible set of basic facilities and, most importantly, of their effective extensibility in many directions and for many applications. The latter is particularly enhanced by the language Oberon on the one, and by the efficiency of its basic core on the other hand. It is rooted in the application of the object-oriented paradigm which is employed wherever extensibility appears advantageous.

In addition to the core system, we describe in full detail the compiler for language Oberon and a graphics system, which both may be regarded as applications. The former reveals how a compact compiler is designed to achieve both fast compilation and efficient, dense code. The latter stands as an example of extensible design based on object-oriented techniques, and it shows how a proper integration with an existing text system is possible. A network module is another addition to the core system allowing many workstations to be interconnected. We also show how the Oberon serves conveniently as the basis for a multi-server station, accommodating a file distribution, a printing, and an electronic-mail facility.

Compactness and regular structure, and due attention to efficient implementation of important details appear to be the key to economical software engineering. With the Oberon, we wish to refute Reiser's Law, which has been confirmed by virtually all recent releases of operating systems: In spite of great leaps forward, hardware being becoming faster is more slowly than software being slower. The Oberon has required a tiny fraction of the manpower demanded for the construction of widely-used commercial OSes, and a small fraction of their demands on computing power and storage capacity, while providing equal power and flexibility to the user, albeit without certain bells and whistles. The reader is invited to study how this was possible.

But most importantly, we hope to present a worth-while case study of a substantial piece of programming in large for the benefit of all those who are eager to learn from the experiences of others.

We wish to thank the many anonymous contributors of suggestions, advice, and encouragement. In particular we wish to thank our colleagues H. Mössenböck, B. Sanders and our associates at the Institut für Computer systeme for reading all or parts of the book draft. We are grateful to M. Brandis, R. Crelier, A. Disteli, M. Franz, and J. Templ for their work in porting the Oberon System successfully to various commercially available computers, and thus making the study of this book more worth-while for many readers. And we gratefully acknowledge the contribution of our school, ETH, for providing the environment and support which made it possible for us to pursue and complete this project.

Zürich, February 1992
N.W. and J.G.

0.2 2^{nd} EDITION

Comments about plans to prepare a second edition to this book varied widely. Some felt that this book is outdated, that nobody is interested in a system of this kind any longer. "Why bother"? Others felt that there is an urgent need for this type of text, which explains an entire system in detail rather than merely proposing strategies and approaches. "By all means"!.

Very much has changed in these last 30 years. But even without this change, it would be preposterous to propose and construct a system competing with existing, worldwide "standards". Indeed, very few people would be interested in using it. The community at large seems to be stuck with these gigantic software systems, and helpless against their complexity, peculiarities, and occasional unreliability.

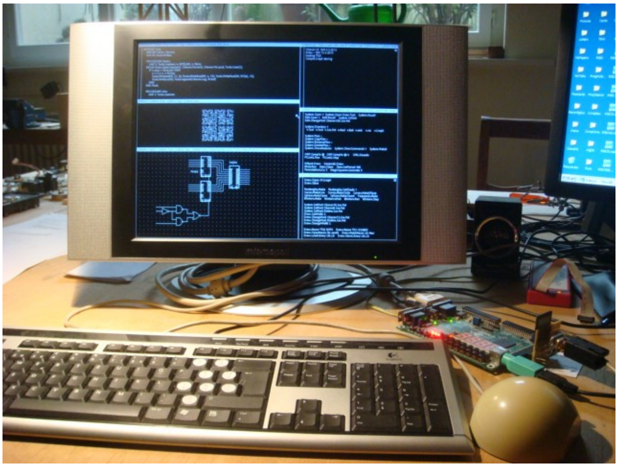
But surely new systems will emerge, perhaps for different, limited purposes, allowing for smaller systems. One wonders where their designers will study and learn their trade. There is little technical literature, and my conclusion is that understanding is generally gained by doing, that is, "on the job". However, this is a tedious and sub-optimal way to learn. Whereas sciences are governed by principles and laws to be learned and understood, in engineering experience and practice are indispensable. Does Computer Science teach laws that hold for (almost) ever? More than any other field of engineering, it would be predestined to base on rigorous mathematical principles. Yet, its core hardly is. Instead, one must rely on experience, that is, on studying sound examples.

The main purpose of and the driving force behind this project is to provide a single book that serves as an example of a system that exists, is in actual use, and is explained in all detail. This task drove home the insight that it is hard to design a powerful and reliable system, but even much harder to make it so simple and clear that it can be studied and fully understood. Above everything else, it requires a stern concentration on what is essential, and the will to leave out the rest, all the popular "bells and whistles".

Recently, a growing number of people has become interested in designing new, smaller systems. The vast complexity of popular OSes makes them not only obscure, but also provides opportunities for "back doors". They allow external agents to introduce spies and devils unnoticed by the user, making the system attack-able and corruptible. The only safe remedy is to build a safe system anew from scratch.

Turning now to a practical aspect: The largest chapter in the 1992 edition of this book dealt with the compiler translating Oberon programs into code for the NS32032 processor. Which is now neither available nor is its architecture recommendable. Instead of writing a new compiler for some other commercially available architecture, I decided to design my own in order to extend the desire for simplicity and regularity to the hardware. The ultimate benefit of this decision is not only the software, but also the hardware of Oberon is described completely and rigorously. The processor is called RISC. The hardware modules are described exclusively in the language Verilog.

The decision for a new processor was expedited by the possibility to implement it, that is, to make it concrete and available. This is due to the advent of programmable gate arrays (FPGA), allowing to turn a design into a real, functioning processor on a single chip. As a result, the described system can be realized using a low-cost development board. Which, Xilinx Spartan-3 by Digilent, features a 1MB static memory, which easily accommodates the entire Oberon, including its compiler. It is shown, together with a display, keyboard and mouse in the photo below. In the lower, right corner, is the board.



The decision to develop our own processor required that the compiler and linking loader chapters had to be completely rewritten. However, it also provided the welcome chance to improve their clarity considerably. The new processor indeed allowed to simplify and straighten out the entire compiler.

For the description of a system to be comprehensible, the key element is the notation, formalism, or language in which it is defined. Algol 60, published 50 years ago, was proposed as a publication language, as a formalism in which algorithms could be defined without reference to particular computers, or to any mechanism at all. This was a great goal, but so far it was hardly achieved. Yet, it emphasized the importance of abstraction to be achieved by the notation with a mathematically rigorous foundation. At least, Algol was the first language based on a formally defined syntax. Algol was the result of early recognition that programs must never be written just to feed computers, but always to be understood and instructive to people.

In all my past work, I have tried to design a successor to Algol, that improves its rigor and at the same time extends its applicability from numerical algorithms to software systems. From a long sequence, starting with Algol, through Pascal, Modula, and Oberon, we have come closer to this goal than ever before, and than any other language in existence. The key lay in a continued struggle for sensible simplification.

The Oberon language, defined in 1988, underwent a revision in 2007, mostly discarding features that were either duplicated or not essential. Adaptation of the system's source code to the revised language was, besides the change of processor, the second important reason for numerous, local changes in this text. We summarize the various deletions of features:

1. The data types LONGINT, SHORTINT, and LONGREAL have been discarded, and with them the concept of type inclusion.

2. The LOOP and EXIT statements (repetitions with multiple exit points) have been discarded.

3. The WITH statement (regional type guard) has been discarded.

4. The RETURN statement has been removed and is now syntactically merged with the ending of function procedure declarations.

5. Objects declared in procedures are not accessible within their local counterparts. That is, objects must be either strictly local or global in order to be accessible.

6. Assignments to imported variables are not permitted (read-only export).

7. Forward procedure declarations have been discarded.

In contrast to these removals, there is a single addition (made in 2012):

- The data type BYTE has been added. Its values are integers x satisfying $0 \leq x < 256$. This addition prevents the frequent abuse of the type CHAR. BYTE is mainly used for elements of arrays and records in low level modules in order to economize the use of memory.

In spite of these two reasons for changes – one at the highest level, language, the other at the lowest, hardware – the remainder of this book proved to be pretty stable and still valid. It has been my desire to present the system essentially as it existed 25 years ago, without embellishments. The chapters 3 - 5 on tasking, the display and text, originally written by J. Gutknecht, have been carried over virtually unchanged. Significant changes, however, were necessary mainly in the descriptions of device drivers for keyboard and mouse. They now use the PS-2 interface standard. The disk has been replaced by a single SD-card (flash memory) with a standard SPI interface. The net interface no longer uses RS-485, but is also based on the SPI standard. Chapters on the compiler and linker are completely new.

Mostly thanks to the regularity of RISC instruction set, the compiler size could be reduced significantly. It now measures less than 2900 lines of program and compiles itself in about 3 seconds, which is proof of its efficiency. The entire system compiles itself in less than 10 seconds.

Considered extravagant and hardly necessary only years ago, run-time checks are generated automatically. In particular, they cover index range checks and access to NIL-pointers. Due to their efficiency they hardly affect run-time speed, but are a great benefit to programmers.

A welcome consequence of the language and processor simplifications is the fact that all parts had been written in assembler code in 1992 – and therefore were not included in the book – have now been expressed in Oberon as well. Vindicating my perennial efforts to obtain a high-level language which is powerful, flexible, and also efficient enough to express parts such as device drivers and raster operations, this was the necessary and final step to make this book comprehensive and complete.

REFERENCES

1. Oberon 07 Report pdf
2. RISC Arch pdf

ACKNOWLEDGMENTS

I gratefully acknowledge the valuable contributions of Paul Reed. He designed the interfaces to various devices, such as the PS-2 and SPI, including the SD-card, acting as disk store. He suggested many improvements and simplifications. He originally decisively suggested a re-edition of this book of 30 year ago, and was the key impetus to do all this work. My thanks go to him.

Niklaus Wirth, September 2013

1 OVERVIEW

1.1 HISTORY & MOTIVATION

How could anyone diligently concentrate on his work in an afternoon with such warmth, splendid sunshine, and blue sky. This rhetorical question was one I asked many times while spending a sabbatical leave in California in 1985. Back home everyone would feel compelled to profit from the sunny spells to enjoy life at leisure in the country-side, wandering or engaging in one's favourite sport. But here, every day was like that, and giving in to such temptations would have meant the end of all work. And, had I not chosen this location in the world because of its inviting, enjoyable climate?

Fortunately, my work was also enticing, making it easier to buckle down. I had the privilege of sitting in front of the most advanced and powerful workstation anywhere, learning the secrets of perhaps the newest fad in our fast developing trade, pushing colored rectangles from one place of the screen to another. This all had to happen under strict observance of rules imposed by physical laws and by the newest technology. Fortunately, the advanced computer would complain immediately if such a rule was violated, it was a rule checker and acted like your big brother, preventing you from making steps towards disaster. And it did what would have been impossible for oneself, keeping track of thousands of constraints among the thousands of rectangles laid out. This was called computer-aided design. "Aided" is rather a euphemism, but the computer did not complain about the degradation of its role.

While my eyes were glued to the colorful display, and while I was confronted with the evidence of my latest inadequacy, in through the always open door stepped my colleague (JG). He also happened to spend a leave from duties at home at the same laboratory, yet his face did not exactly express happiness, but rather frustration. The chocolate bar in his hand did for him what the coffee cup or the pipe does for others, providing temporary relaxation and distraction. It was not the first time he appeared in this mood, and without words I guessed its cause. And the episode would reoccur many times.

His days were not filled with the great fun of rectangle-pushing; he had an assignment. He was charged with the design of a compiler for the same advanced computer. Therefore, he was forced to deal much more closely, if not intimately, with the underlying software system. Its rather frequent failures had to be understood in his case, for he was programming, whereas I was only using it through an application; in short, I was an end-user! These failures had to be understood not for purposes of correction, but in order to find ways to avoid them. How was the necessary insight to be obtained? I realized at this moment that I had so far avoided this question; I had limited familiarization with this novel system to the bare necessities which sufficed for the task on my mind.

It soon became clear that a study of the system was nearly impossible. Its dimensions were simply awesome, and documentation accordingly sparse. Answers to questions that were momentarily pressing could best be obtained by interviewing the system's designers, who all were in-house. In doing so, we made the shocking discovery that often we could not understand their language. Explanations were fraught with jargon and references to other parts of the system which had remained equally enigmatic to us.

So, our frustration-triggered breaks from compiler construction and chip design became devoted to attempts to identify the essence, the foundations of the system's novel aspects. What made it different from conventional OSes? Which of these concepts were essential, which ones could be improved, simplified, or even discarded? And where were they rooted? Could the system's essence be distilled and extracted, like in a chemical process?

During the ensuing discussions, the idea emerged slowly to undertake our own design. And suddenly it had become concrete. "Crazy" was my first reaction, and "impossible". The sheer amount of work appeared as overwhelming. After all, we both had to carry our share of teaching duties back home. But the thought was implanted and continued to occupy our minds.

Sometime thereafter, events back home suggested that I should take over the important course about System Software. As it was the unwritten rule that it should primarily deal with operating system principles, I hesitated. My scruples were easily justified: After all I had never designed such a system nor a part of it. And how can one teach an engineering subject without first-hand experience?

Impossible? Had we not designed compilers, OSes, and document editors in small teams? And had I not repeatedly experienced that an inadequate and frustrating program could be programmed from scratch in a fraction of source code used by the original design? Our brainstorming continued, with many intermissions, over several weeks, and certain shapes of a system structure slowly emerged through the haze. After some time, the preposterous decision was made: we would embark on the design of an OS for our workstation (which happened to be much less powerful than the one used for my rectangle-pushing) from scratch.

The primary goal, to personally obtain first-hand experience, and to reach full understanding of every detail, inherently determined our manpower: two part-time programmers. We tentatively set our time-limit for completion to three years. As it later turned out, this had been a good estimate; programming was begun in early 1986, and a first version of the system was released in the fall of 1988.

Although the search for an appropriate name for a project is usually a minor problem and often left to chance and whim of the designers, this may be the place to recount how Oberon entered the picture in our case. It happened that around the time of the beginning of our effort, the space probe Voyager made headlines with a series of spectacular pictures taken of the planet Uranus and of its moons, the largest of which is named Oberon. Since its launch I had considered the Voyager project as a singularly well-planned and successful endeavor, and as a small tribute to it I picked the name of its latest object of investigation. There are indeed very few engineering projects whose products perform way beyond expectations and beyond their anticipated lifetime; mostly they fail much earlier, particularly in the domain of software. And, last but not least, we recall that Oberon is famous as the king of elfs.

The consciously planned shortage of manpower enforced a single, but healthy, guideline: Concentrate on essential functions and omit embellishments that merely cater to established conventions and passing tastes. Of course, the essential core had first to be recognized and crystallized. But the basis had been laid. The ground rule became even more crucial when we decided that the result should be able to be used as teaching material. I remembered C.A.R. Hoare's plea that books should be written presenting actually operational systems rather than half baked, abstract principles. He had complained in the early 1970s that in our field engineers were told to constantly create new artifacts without being given the chance to study previous works that had proven their worth in the field. How right was he, even to the present day!

The emerging goal to publish the result with all its details let the choice of programming language (PL) appear in a new light: it became crucial. Modula-2 which we had planned to use, appeared as not quite satisfactory. Firstly, it lacked a facility to express extensibility in an adequate way. And we had put extensibility among the principal properties of the new system. By "adequate" we include machine-independence. Our programs should be expressed in a manner that makes no reference to machine peculiarities and low-level programming facilities, perhaps with the exception of device interfaces, where dependence is inherent.

Hence, Modula-2 was extended with a feature that is now known as type extension. We also recognized that Modula-2 contained several facilities that we would not need, that do not genuinely contribute to its power of expression, but at the same time increase the complexity of the compiler. But the compiler would not only have to be implemented, but also to be described, studied, and understood. This led to the decision to start from a clean slate also in the domain of language design, and to apply the same principle to it: concentrate on the essential, purge the rest. The new language, which still bears much resemblance to Modula-2, was given the same name as the system: Oberon [1, 2]. In contrast to its ancestor it is terser and, above all, a significant step towards expressing programs on a high level of abstraction without reference to machine-specific features.

We started designing the system in late fall 1985, and programming in early 1986. As a vehicle we used our workstation Lilith and its language Modula-2. First, a cross-compiler was developed, then followed the modules of the inner core together with the necessary testing and down-loading facilities. The development of the display and the text system proceeded simultaneously, without the possibility of testing, of course. We learned how the absence of a debugger, and even more so the absence of a compiler, can contribute to careful programming.

Thereafter followed the translation of the compiler into Oberon. This was swiftly done, because the original had been written with anticipation of the later translation. After its availability on the target computer Ceres, together with the operability of the text editing facility, the umbilical cord to Lilith could be cut off. The Oberon had become real, at least its draft version. This happened around the middle of 1987; its description was published thereafter [3], and a manual and guide followed in 1991 [5].

The system's completion took another year and concentrated on connecting the workstations in a network for file transfer [4], on a central printing facility, and on maintenance tools. The goal of completing the system within three years had been met. The system was introduced in the middle of 1988 to a wider user community, and work on applications could start. A service for electronic mail was developed, a graphics system was added, and various efforts for general document preparation systems proceeded. The display facility was extended to accommodate two screens, including color. At the same time, feedback from experience in its use was incorporated by improving existing parts. Since 1989, Oberon has replaced Modula-2 in our introductory programming courses.

1.2 OVERVIEW

This book consists of 3 main parts, excluding this preface and overview(PO) part:

1. OS
2. apps

3. hardware computer

First, OS. Implementation of a system proceeds bottom-up naturally, because higher level modules are clients of those lower level and cannot function without their imports. Description, on the contrary, should better be arranged in the reverse top-down way. This is because a system is designed with its expected applications and functions in mind. Decomposition into a hierarchy of modules is justified by the use of auxiliary functions and abstractions, or by postponing more detailed explanation later, when the need has been fully motivated. Thus, we will describe the OS part top-down essentially.

2 first explains the most important basic concepts like Viewers, Commands, Tasks and Texts, etc. which will be further elaborated in the following chapters. It ends with the system structure, to share readers a high-level overview understanding to the whole system.

Chapters 3 - 5 describe the outer core system:

3 focuses on the dynamic aspects. In particular, it introduces the fundamental operational units of task and command. Oberon's tasking model distinguishes the categories of interactive tasks and background tasks. The former are represented on the display screen by rectangular areas, so-called viewers. Yet the latter need not be connected with any displayed object. They are scheduled with low priority when

interactions are absent. A good example of a background task is the memory garbage collecting. Both of them are mapped to a single process by the task scheduler. Commands in Oberon are explicit, atomic units of interactive operations. They are realized in the form of exported parameterless procedures and replace the heavier-weight notion of program known from more conventional OSes. This chapter continues with a definition of a software toolbox as a logically connected collection of commands. It terminates with an outline of the system control toolbox.

4 explains Oberon's display system. It starts with a discussion of our choice of a hierarchical tiling strategy for the allocation of viewers. A detailed study of the exact role of Oberon viewers follows. Type Viewer is presented as an object class with an open message interface providing a conceptual basis for far-reaching extensibility. Viewers are then recognized as just a special case of so-called frames that may be nested. A category of standard viewers containing a menu frame and a frame of contents is investigated. The next topic is cursor handling. A cursor in Oberon is a marked path. Both viewer manager and cursor handler operate on an abstract logical display area rather than on individual physical monitors. This allows a unified handling of display requests, independent of number and types of monitors assigned. For example, smooth transitions of the cursor across screen boundaries are conceptually guaranteed. The chapter continues with the presentation of a

concise and complete set of raster operations that is used to place textual and graphical elements in the display area. An overview of the system display toolbox concludes the chapter.

5 introduces Text. Oberon distinguishes itself by treating Text as an abstract data type that is integrated in the central system. Numerous fundamental consequences are discussed. For example, a text can be produced by one command, edited by a user, and then consumed by a next command. Commands themselves can be represented textually in the form M.P, followed by a textual parameter list. Consequently, any command can be called directly from within a text (so called tool) simply by pointing at it with the mouse. However, the core of this chapter is a presentation of Oberon's text system as a case study in program modularization. The concerns of managing a text and displaying it are nicely separated. Both the text manager and the text display feature an abstract public interface as well as an internally hidden data structure. Finally in this chapter, Oberon's type-font management and the toolbox for editing are discussed.

Chapters 6 - 9 describe the inner core system:

6 explains the loader of modules and motivates the intro of data type *Module*. The chapter includes the management of the memory part holding program code and defines the format in which compiled modules are stored as object files. Furthermore, it discusses the problems of binding

separately compiled modules together and of referencing objects defined in other modules.

7 is devoted to the file system(FS), a part of crucial importance, because files are involved in almost every program and computation. The chapter consist of two distinct parts, the first introducing the type File and describing the structure of files, i.e. their representation on disk storage with its sequential characteristics, the second describing the directory of file names and its organisation as a B-tree for obtaining fast searches.

8 the memory management. A single, central storage management was one of the key design decisions, guaranteeing an efficient and economical use of storage. The chapter explains the store's partitioning into specific areas. Its central concern, however, is the discussion of dynamic storage management in the partition called the heap. The algorithm for allocation (corresponding to the intrinsic procedure NEW) and for retrieval (called garbage collection) are explained in detail.

9 describes the lowest level of the module hierarchy: device drivers, which contains drivers for some widely accepted interface standards. The first is PS-2, a serial transmission with synchronous clock. This is used for the keyboard and for the Mouse. The second is SPI, a standard for bi-directional, serial transmission with synchronous clock. This is used for the "disk", represented by an SDI-card (flash

memory), and for the network. And the third standard is RS-232 typically used for simple and slow data links. It is bidirectional and asynchronous.

The second part, consisting of Chapters 10 - 15, is devoted to what may be called first applications of the basic Oberon. These chapters are therefore independent to each other, only making reference to the upper Chapters 3 - 9.

Although the Oberon is well-suited for operating stand-alone workstations, a facility for connecting a set of computers should be considered as fundamental. Module *Net*, which makes transmission of files among workstations connected by a bus-like network possible, is the subject of **??**. It presents not only the problems of network access, of transmission failures and collisions, but also those of naming partners. The solutions are implemented in a surprisingly compact module which uses a network driver presented in 9.

When a set of workstations is connected in a network, the desire for a central server appears. A central facility serving as a file distribution service, as a printing station, and as a storage for electronic mail is presented in **??**. It emerges by extending the Net module of **??**, and is a convincing application of the tasking facilities explained in Section 2.2. In passing we note that the server operates on a machine that is not under observation by a user. This circumstance requires an increased degree of robustness, not only against transmission failures, but also against data that do not conform to defined formats.

The presented system of servers demonstrates that Oberon's single-thread scheme need not be restricted to single-user systems. The fact that every command or request, once accepted, is processed until completion, is acceptable if the request does not occupy the processor for too long, which is mostly the case in the presented server applications. Requests arriving when the processor is engaged are queued. Hence, the processor handles requests one at a time instead of interleaving them which, in general, results in faster overall performance due to the absence of frequent task switching.

?? describes the Oberon compiler. It translates source text in Oberon into target code, i.e. instruction sequences of some target computer. Its principles and techniques are explained in [6]. Both, source language and target architecture must be understood before studying a compiler. Both source language and the target computer's RISC architecture are presented in the Appendix.

Although here the compiler appears as an application module, it naturally plays a distinguished role, because the system (and the compiler itself) is formulated in the language which the compiler translates into code. Together with the text editor it was the principal tool in the system's development. The use of straight-forward algorithms for parsing and symbol table organization led to a reasonably compact piece of software. A main contributor to this result is the language's definition: the language is devoid of complicated structures and rarely used embellishments.

The compiler and thereby the chapter is partitioned into two main parts. The first is language specific, but does not refer to any particular target computer. It consist of the scanner and the parser. This part is therefore of most general interest to the readership. The second part is, essentially, language-independent, but is specifically tailored to the instruction set of the target computer. It is called the code generator.

Texts play a predominant role in the Oberon. Their preparation is supported by the system's major tool, the editor. In **??** we describe another one, which handles graphic objects. At first, only horizontal or vertical lines and short captions are introduced as objects. The major difference to texts lies in the fact that their coordinates in the drawing plane do not follow from those of their predecessor automatically, because they form a set rather than a sequence. Each object carries its own, independent coordinates. The influence of this seemingly small difference upon an editor are far-reaching and permeate the entire design. There exist hardly any similarities between a text and a graphics editor. Perhaps one should be mentioned: the partitioning into three parts. The bottom module defines the respective abstract data structure for texts or graphics, together with, of course, the procedures handling the structure, such as searches, insertions, and deletions. The middle module in the hierarchy defines a respective frame and contains all procedures concerned with displaying the respective objects including the frame handler defining interpretation of mouse and keyboard events. The top modules are the respective tool modules (Edit, Draw). The presented graphics editor is particularly interesting in so far as it constitutes a convincing example of Oberon's extensibility. The graphics editor is integrated into the entire system; it embeds its graphic frames into menu-viewers and uses the facilities of the text system for its caption elements. And lastly, new kinds of elements can be incorporated by the mere addition of new modules, i.e. without expanding, even without recompiling the existing ones. Two examples are shown in **??** itself: rectangles and circles.

The Draw System has been extensively used for the preparation of diagrams of electronic circuits. This application suggests a concept that is useful elsewhere too, namely a recursive definition of the notion of object. A set of objects may be regarded as an object itself and be given a name. Such an object is called a macro. It is a challenge to the designer to implement a macro facility such that it is also extensible, i.e. in no way refers to the type of its elements, not even in its input operations of files on which macros are stored.

?? presents two other tools, namely one used for installing an Oberon on a bare machine, and one used to recover from failures of the file store. Although rarely employed, the first was indispensable for the development of the system. The maintenance or recovery tools are invaluable assets when failures occur. And they do! **??** covers material that is rarely presented in the literature.

?? is devoted to tools that are not used by the Oberon presented so far, but may be essential in some applications. The first is a data link with a protocol based on the RS-232 standard shown in 9. Another is a standard set of basic mathematical functions. And the third is a tool for creating new macros for the Draw System.

The last part is a detailed description of the hardware:

?? defines the processor, for which the compiler generates code. The target computer is a truly simple and regular processor called RISC with only 14 instructions, represented not by a commercial processor, but implemented with an FPGA, a Field Programmable Gate Array. It allows its structure to be described in full detail. It is a straight-forward, von Neumann type device consisting of a register bank, an arithmetic-logic unit, including a floating-point unit. Typical optimization facilities, like pipelining and cache memory, have been omitted for the sake of transparency and simplicity. The processor circuit is described in the language Verilog.

?? describes the environment in which the processor is embedded. This environment consists of the interfaces to main memory and to all external devices.

REFERENCES

1. N. Wirth. The PL Oberon. Software - Practice and Experience 18, 7, (July 1988) 671-690.
2. M. Reiser and N. Wirth. Programming in Oberon - Steps beyond Pascal and Modula. AddisonWesley, 1992.
3. N. Wirth and J. Gutknecht. The Oberon. Software - Practice and Experience, 19, 9 (Sept. 1989), 857-893.
4. N. Wirth. Ceres-Net: A low-cost computer network. Software - Practice and Experience, 20, 1 (Jan. 1990), 13-24.
5. M. Reiser. The Oberon - User Guide and Programmer's Manual. Addison-Wesley, 1991.
6. N. Wirth. Compiler Construction. Addison-Wesley, Reading, 1996. ISBN 0-201-40353-6

Part I

OS

2

BASIC CONCEPTS & SYSTEM STRUCTURE

In order to warrant the sizeable effort designing and constructing an entire OS from scratch, basic concepts need be novel. The principal concepts underlying Oberon and the dominant design decisions are 1st discussed. Upon this, the system structure was then presented, restricted to the coarsest, the composition and interdependence of the largest building blocks, *modules*. Finally, an overview, helps understanding the place, role, and significance of each part (chapter).

The fundamental OS objective is to present the computer at a certain abstraction level to the user and programmer. For example,

STORE	requestable *variables* of a specified data type,
DISK	sequences of characters or bytes called *files*,
DISPLAY	rectangular areas called *viewers*,
KEYBOARD	an *input stream* of characters, and

MOUSE a pair of *coordinates* and a set of *key states*.

Every abstraction is characterized by certain *properties* and governed by a set of *operations*. It is the *task* of OS to implement these operations and to manage them, constrained by the available resources of the underlying computer. This is commonly called *resource management* (RM).

Every abstraction inherently hides details, from which it abstracts. Hiding occurs at different levels. For example, the

- computer may make certain store parts or devices inaccessible according to operation modes (user/supervisor);
- PL may make certain parts inaccessible through a facility inherent in visibility rules.

The latter is of course much more flexible and powerful, while the former indeed plays an almost negligible role in our system. Hiding is important because it allows maintenance of certain properties (called *invariants*) of an abstraction to be guaranteed. Abstraction is indeed the key of any modularization, and without it every hope of being able to guarantee reliability and correctness vanishes. Clearly, Oberon was designed with the goal of establishing a modular structure on the basis of purpose-oriented abstractions. The availability of an appropriate PL is an indispensable prerequisite, and the importance of prudent choice cannot be over-emphasized.

2.1 CONCEPTS

2.1.1 *Viewers*

Whereas the abstractions of individual variables representing parts of the primary store, and of files representing parts of the disk store are well established notions and have significance in every computer system, abstractions regarding input and output devices became important with the advent of high interactivity between user and computer. High interactivity requires high bandwidth, and the only channel of human users with high bandwidth is the eye. Consequently, the computer's visual output unit must be properly matched with human eyes. This occurred with the advent of the *high-resolution display* (HRD) in the mid 1970s, which in turn had become feasible due to faster and cheaper electronic memory components. The HRD marked one of the few very significant break-throughs in the history of computer development. The typical bandwidth of a modern display is in the order of 100 MHz. Primarily the HRD made visual output a subject of abstraction and RM. In Oberon, the display is partitioned into *viewers*, also called windows, or more precisely, *frames*, rectangular areas of the screen(s). A viewer typically consists of:

TITLE BAR contains a subject name and commands menu,

MAIN FRAME some text, graphic, video, or other object.

Viewers (frames) can be nested, in principle to any depth.

`System` provides routines for generating, moving, and closing a viewer. It allocates a new one at a specified place, and upon request delivers hints as to where it might best be placed. It keeps track of their opened set. These are called *viewer management*, in contrast to contents handling.

High interactivity requires not only high visual output bandwidth but also input flexibility. Surely, there is no need for an equally high bandwidth, but a **keyboard** limited by the speed of typing to about 100 Hz is not good enough. The break-through on this front was achieved by a pointing device, **mouse**, appeared roughly the same time as HRD.

This was by no means just a lucky coincidence. The mouse comes to fruition only through HRD and appropriate software. It is itself a conceptually very simple device delivering signals when moved on the table. These signals allow the computer to update the position of a mark - the cursor - on the display. Since feedback occurs through human eyes, no great precision is required. For example, when the user wishes to identify a certain object, e.g. a letter on the screen, one moves the mouse as far as required until the mapped cursor reaches the object. This stands in marked contrast to a digitizer which is supposed to deliver exact coordinates. Oberon relies highly on it.

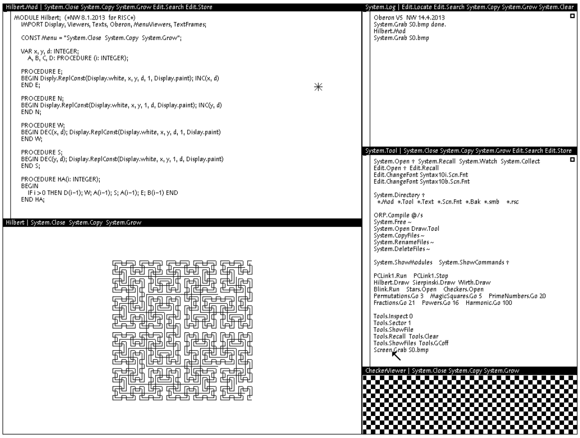
Perhaps the cleverest idea was to equip mice with buttons. By being able to signal a request with the same hand that determines the cursor position, the user obtains the direct impression of issuing position-dependent requests. Position-dependence is realized in software by delegating interpretation of the signal to a procedure - a so-called handler or interpreter - which is local to the viewer in whose area the cursor momentarily appears. A surprising flexibility of command activation can be achieved in this manner by appropriate software. Various techniques have emerged in this connection, e.g. pop-up menus, pull-down-menus, etc. which are powerful even under the presence of a single button only. For many applications, a mouse with several keys is far superior, and Oberon basically assumes 3 buttons to be available. The assignment of different functions to the keys may of course easily lead to confusion when every application prescribes different key assignment. This is, however, easily avoided by the adherence to certain "global" conventions. In Oberon, button of the:

LEFT primarily used for marking a position (setting a caret),

MIDDLE for issuing general commands (see below), and

RIGHT for selecting displayed objects.

Recently, it has become fashionable to use overlapping windows mirroring documents being piled up on the desktop. We have found this metaphor not entirely convincing. Partially hidden windows are typically brought to the top and made fully visible before any operation is applied to their contents. In contrast to the insignificant advantage stands the substantial effort necessary to implement this scheme. It is a good example of a case where the benefit of a complication is incommensurate with its cost. Therefore, we have chosen a solution that is much simpler to realize yet has no genuine disadvantages: *tiled viewers* as shown in Fig 1:


Hilbert.Mod | System.Close System.Copy System.Grow Edit.Search Edit.Store
Hilbert | System.Close System.Copy System.Grow
System.Log | Edit.Locate Edit.Search System.Copy System.Grow System.Clear
System.Tool | System.Close System.Copy System.Grow Edit.Search Edit.Store
CheckerViewer | System.Close System.Copy System.Grow

```
MODULE Hilbert;  (*NW 8.1.2013  for RISC*)
  IMPORT Display, Viewers, Texts, Oberon, MenuViewers, TextFrames;

  CONST Menu = "System.Close  System.Copy  System.Grow";

  VAR x, y, d: INTEGER;
    A, B, C, D: PROCEDURE (i: INTEGER);

  PROCEDURE E;
  BEGIN Disply.ReplConst(Display.white, x, y, d, 1, Display.paint); INC(x, d)
  END E;

  PROCEDURE N;
  BEGIN Display.ReplConst(Display.white, x, y, 1, d, Display.paint); INC(y, d)
  END N;

  PROCEDURE W;
  BEGIN DEC(x, d); Display.ReplConst(Display.white, x, y, d, 1, Dislay.paint)
  END W;

  PROCEDURE S;
  BEGIN DEC(y, d); Display.ReplConst(Display.white, x, y, 1, d, Display.paint)
  END S;

  PROCEDURE HA(i: INTEGER);
  BEGIN
    IF i > 0 THEN D(i-1); W; A(i-1); S; A(i-1); E; B(i-1) END
  END HA;
```

```
Oberon V5  NW 14.4.2013
System.Grab S0.bmp done.
Hilbert.Mod
System.Grab S0.bmp
```

```
System.Open ↑  System.Recall  System.Watch  System.Collect
Edit.Open ↑  Edit.Recall
Edit.ChangeFont Syntax10i.Scn.Fnt
Edit.ChangeFont Syntax10b.Scn.Fnt

System.Directory ↑
  *.Mod  *.Tool  *.Text  *.Scn.Fnt  *.Bak  *.smb   *.rsc

ORP.Compile @/s
System.Free ~
System.Open Draw.Tool
System.CopyFiles ~
System.RenameFiles ~
System.DeleteFiles ~

System.ShowModules   System.ShowCommands ↑

PCLink1.Run   PCLink1.Stop
Hilbert.Draw  Sierpinski.Draw  Wirth.Draw
Blink.Run   Stars.Open   Checkers.Open
Permutations.Go 3   MagicSquares.Go 5   PrimeNumbers.Go 20
Fractions.Go 21   Powers.Go 16   Harmonic.Go 100

Tools.Inspect 0
Tools.Sector 1
Tools.ShowFile
Tools.Recall  Tools.Clear
Tools.ShowFiles  Tools.GCoff
Screen.Grab S0.bmp
```

Figure 1: Oberon display with tiled viewers

2.1.2 *Commands*

Position-dependent commands with meaning fixed for each type of viewer must be supplemented by general commands. Conventionally, such commands are issued through the keyboard by typing the program's name that is to be executed into a special command text. In this respect, Oberon offers a novel and much more flexible solution:

1^{st} of all we remark that a program in the common sense of a text compiled as a unit is mostly a far too large action unit to serve as a command. Compare it with, for example, the insertion of a piece of text through a mouse command. In Oberon, the notion of a action unit is separated from the notion of compilation unit. The former is a command represented by a (exported) procedure, the latter a module. Hence, a module may, and typically does, define several, even many commands. Such a (general) command may be invoked at any time by pointing at its name in any text visible in any viewer on the display, and by clicking the middle mouse button. The command name has the form `M.P`, where `P` is the procedure's identifier and `M` the module in which `P` is declared. As a consequence, any command click may cause the loading of one or several modules, if `M` is not already present in main store. The next invocation of `M.P` occurs instantaneously, since `M` is already loaded. A further consequence is that modules are never (automatically) removed, because a next command may well refer to the same module.

Every command has the purpose to alter the state of some operands. Typically, they are denoted by text following the command identification, and Oberon follows this convention. Strictly speaking, commands are denoted as parameterless procedures; but the system provides a way for the procedure to identify the text position of its origin, and hence to read and interpret the text following the command, i.e. the actual parameters. Both reading and interpretation must, however, be programmed explicitly.

The parameter text must refer to objects that exist before command execution starts and are quite likely the result of a previous command interpretation. In most OSes, these objects are files registered in the directory, and they act as interfaces between commands. Oberon broadens this notion; the links between consecutive commands are not restricted to files, but can be any global variable, because modules do not disappear from storage after command termination, as mentioned above.

This tremendous flexibility seems to open Pandora's box, and indeed it does when misused. The reason is that global variables' states may completely determine and alter the effect of a command. The variables represent hidden states, hidden in the sense that the user is in general unaware of them and has no easy way to determine their value. The positive aspect of using global variables as interfaces between commands is that some of them may well be visible on the display.

All viewers - and with them also their contents - are organized in a data structure that is rooted in a global variable (in `Viewers`). Parts of this variable therefore constitute visible states, and it is highly appropriate to refer to them as command parameters.

One of the rules of what may be called the Oberon Programming Style is therefore to avoid hidden states, and to reduce the introduction of global variables. We do not, however, raise this rule to the rank of a dogma. There exist genuinely useful exceptions, even if the variables have no visible parts.

There remains the question of how to denote visible objects as command parameters. An obvious case is the use of the most recent selection as parameter. A procedure for locating that selection is provided by `Oberon`. (It is restricted to text selections). Another possibility is the use of the caret position in a text. This is used in the case of inserting new text; the pressing of a key on the keyboard is also considered to be a command, and it causes the character's insertion at the caret position.

A special facility is introduced for designating viewers as operands: the star marker. It is placed at the cursor position when the keyboard's mark key (`SETUP`) is pressed. The procedure `Oberon.MarkedViewer` identifies the viewer in whose area the star lies. Commands which take it as their parameter are typically followed by an asterisk in the text. Whether the text in a text viewer, a graph in a graphic viewer, or any other part of the marked viewer is taken as the actual parameter depends on how the command is programmed.

Finally a most welcome property of the system should not remain unmentioned. It is a direct consequence of the persistent nature of global variables and becomes manifest when a command fails. Detected failures result in a trap that should be regarded as an abnormal command termination. In the worst case, global data may be left in an inconsistent state, but they are not lost, and a next command can be initiated based on their current state. A trap opens a small viewer and lists the sequence of invoked procedures with their local variables and current values. This information helps a programmer to identify its cause.

2.1.3 *Tasks*

From the presentations above it follows that Oberon is distinguished by a highly flexible scheme of command activation. The notion of a command extends from the insertion of a single character and the setting of a marker to computations that may take hours or days. It is moreover distinguished by a highly flexible notion of operand selection not restricted to registered, named files. And most importantly, by the virtual absence of hidden states. The state of the system is practically determined by what is visible to the user.

This makes it unnecessary to remember a long history of previously activated commands, started programs, entered modes, etc. Modes are in our view the hallmark of user-unfriendly systems. It should at this point have become obvious that the system allows a user to pursue several different tasks concurrently. They are manifest in the form of viewers containing texts, graphics, or other displayable objects. The user switches between tasks implicitly when choosing a different viewer as operand for the next command. The characteristic of this concept is that task switching is under explicit control of the user, and the atomic units of action are the commands.

At the same time, we classify Oberon as a single-process (or single-thread) system. How is this apparent paradox to be understood? Perhaps it is best explained by considering the basic mode of operation. Unless engaged in the interpretation of a command, the processor is engaged in a loop continuously polling event sources. This loop is called the *central loop*; it is contained in `Oberon` which may be regarded as the system heart. The 2 fixed event sources are the mouse and the keyboard. If a keyboard event is sensed, control is dispatched to the handler installed in the so-called *focus viewer*, designated as the one holding the caret. If a mouse event (key) is sensed, control is dispatched to the handler in which the cursor currently lies. This is all possible under the paradigm of a single, uninterruptible process.

The notion of a single process implies non-interruptability, and therefore also that commands cannot interact with the user. Interaction is confined to the selection of commands before their execution. Hence, there exists no input statement in typical Oberon programs. Inputs are given by parameters supplied and designated before command invocation.

This scheme at first appears as gravely restrictive. In practice it is not, if one considers single-user operation. It is this single user who carries out a dialog with the computer. A human might be capable of engaging in simultaneous dialogs with several processes only if the commands issued are very time-consuming. We suggest that execution of time-consuming computations might better be delegated to loosely coupled servers in a distributed system.

The primary advantage of a system dealing with a single process is that task switches occur at user-defined points only, where no local process state has to be preserved until resumption. Furthermore, because the switches are user-chosen, the tasks cannot interfere in unexpected and uncontrollable ways by accessing common variables. The system designer can therefore omit all kinds of protection mechanisms excluding such interference. This is significant simplification.

The essential difference between Oberon and multiprocess-systems is that in Oberon task switches occur between commands only, whereas in the latter switches may be invoked after any single instruction. Evidently, the difference is one of action granularity. Oberon's coarse granularity is entirely acceptable for a single-user system.

The system offers the possibility to insert further polling commands in the central loop. This is necessary if additional event sources are to be introduced. The prominent example is a network, where commands may be sent from other workstations. The central loop scans a list of so-called *task descriptors*. Each descriptor refers to a command procedure. The 2 standard events are selected only if their guard permits, i.e. if either keyboard input is present, or if a mouse event occurs. Inserted tasks must provide their own guard in the beginning of the installed procedure.

The example of a network inserting commands, called requests, raises a question: what happens if the processor is engaged in the execution of another command when the request arrives? Evidently, the request would be lost unless measures are taken. The problem is easily remedied by buffering the input. This is done in every input device driver, including the keyboard as well as the network drivers. The incoming signal triggers an interrupt, and the invoked interrupt handler accepts the input and buffers it. We emphasize that such interrupt handling is confined to drivers, system components at the lowest level. An interrupt does not evoke a task selection and a task switch. Control simply returns to the point of interruption, and the interrupt remains unnoticeable to programs. There exists, as with every rule, an exception: an interrupt due to keyboard input of the abort character returns control to the central loop.

2.1.4 *Tool Texts as Configurable Menus*

Certainly, the concepts of viewers specifying their own interpretation of mouse clicks, of commands invokable from any text on the display, of any displayed object being selectable as an interface between commands, and of commands being dialog-free, uninterruptible units of action, have considerable influence on the style of programming in Oberon, and they thoroughly change the style of using the computer. The ease and flexibility in the way pieces of text can be selected, moved, copied, and designated as command and parameters, drastically reduces the need for typing. The mouse becomes the dominant input device: the keyboard merely serves to input textual data. This is accentuated by the use of so-called *tool texts*, compositions of frequently used commands, which are typically displayed in the narrower system track of viewers. One simply doesn't type commands! They are usually visible somewhere already. Typically, the user composes a tool text for every project pursued. Tool texts can be regarded as individually configurable private menus.

The rarity of issuing commands by typing them has the most agreeable benefit that their names can be meaningful words. For example, the copy operation is denoted by `Copy` instead of `cp`, rename by `Rename` instead of `rn`, the call for a file directory excerpt is named `Directory` instead of `ls`. The need for memorizing an infinite list of cryptic abbreviations, which is another hallmark of user unfriendly systems, vanishes.

But the influence of the Oberon concept is not restricted to the style in which the computer is used. It extends also to the way programs are designed to communicate with the environment. The definition of the abstract type `Text` in the system's core suggests the replacement of files by texts as carrier of input and output data in very many cases. The advantage to be gained lies in the text's immediate editability. For example, the output of the command `System.Directory` produces the desired excerpt of the file directory in the form of a (displayed) text. Parts of it or the whole may be selected and copied into other texts by regular editing commands (mouse clicks). Or, the compiler accepts texts as input. It is therefore possible to compile a text, execute the program, and to recompile the re-edited text without storing it on disk between compilations and tests. The ubiquitous editability of text together with the persistence of global data (in particular viewers) allows many steps that do not contribute to the progress of the task actually pursued to be avoided.

2.1.5 *Extensibility*

An important objective in the Oberon design was extensibility. It should be easy to extend the system with new facilities by adding modules that make use of the already existing resources. Equally important, it should also reduce the system to those facilities that are currently and actually used. For example, a document editor processing documents free of graphics should not require the loading of an extensive graphics editor, a workstation operating as a stand-alone system should not require the loading of extensive network software, and a system used for clerical purposes need include neither compiler nor assembler. Also, a system introducing a new kind of display frame should not include procedures for managing viewers containing such frames. Instead, it should make use of existing viewer management. The staggering consumption of memory space by many widely used systems is due to violation of such fundamental rules of engineering. The requirement of many megabytes of store for an OS is, albeit commonly tolerated, absurd and another hallmark of user-unfriendliness, or perhaps manufacturer friendliness. Its reason is none other than inadequate extensibility.

We do not restrict this notion to procedural extensibility, which is easy to realize. The important point is that extensions may not only add further procedures and functions, but introduce their own data types built on the basis of those provided by the system: data extensibility. For example, a graphics system should be able to define its graphics frames based on frames provided by the basic display module and by extending them with attributes appropriate for graphics.

This requires an adequate language feature. The Oberon language provides precisely this facility in the form of type extensions. The language was designed for this reason; Modula-2 would have been the choice, had it not been for the lack of a type extension feature. Its influence on system structure was profound, and the results have been most encouraging. In the meantime, many additions have been created with surprising ease. One of them is described at the end of this book. The basic system is nevertheless quite modest in its resource requirements (see Table 1).

2.1.6 *Dynamic Loading*

Activation of commands residing in modules that are not present in the store implies the loading of the modules and, of course, all their imports. Invoking the loader is, however, not restricted to command activation; it may also occur through programmed procedure calls. This facility is indispensable for a successful realization of genuine extensibility. Modules must be loadable on demand. For example, a document editor loads a graphics package when a graphic element appears in the processed document, but not otherwise.

Oberon features no separate linker. A module is linked with its imports when it is loaded, and never before. As a consequence, every module is present only once, in main store (linked) as well as on backing store (unlinked, as file). Avoiding the generation of multiple copies in different, linked object files is the key to storage economy. Prelinked mega-files do not occur in Oberon, and every module is freely reusable.

2.2 THE SYSTEM STRUCTURE

The largest identifiable units of the system are its modules. It is therefore most appropriate to describe its structure in terms of modules. As their interfaces are explicitly declared, it is also easy to exhibit their interdependence in a directed graph. The edges indicate imports.

The module graph of a system programmed in Oberon is hierarchical, i.e. has no cycles. The lowest members of the hierarchy effectively import hardware only. We refer here to modules which contain device drivers. But module `Kernel` also belongs to this class; it "imports memory" and includes the disk driver. The modules on the top of the hierarchy effectively export to the user. As the user has direct access to command procedures, we call these top members command modules or tool modules.

The hierarchy of the basic system is shown in a table of direct imports and as a graph in Fig 2. The picture is simplified by omitting direct import edges if an indirect path also leads from the source to the destination. For example, `Files` imports `Kernel`, the direct import is not shown, because a path from `Kernel` leads to `Files` via `FileDir`.

	Text-Frame	Menu-View	Ober-on	Texts	Fonts	Input	View	Disp	Modul	Files	FileDir	Kernel
System	x	x	x	x	x	x	x	x	x	x	x	x
Edit	x	x	x	x	x							
TextFrames		x	x	x	x	x	x	x	x			
MenuViewers			x			x	x	x				
Oberon				x	x	x	x	x	x	x		
Texts					x					x		
Fonts										x		
Viewers								x				
Display												
Modules										x		x
Files											x	x
FileDir												x

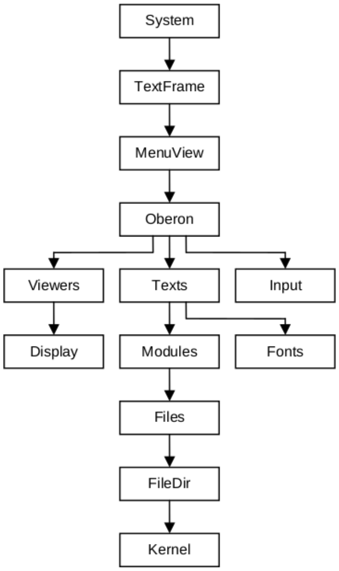


Figure 2: The structure of the Oberon core

Module names in the plural form typically indicate the definition of an abstract data type in the module. The type is exported together with the pertinent operations. Examples are `Files`, `Modules`, `Fonts`, `Texts`, `Viewers`, `MenuViewers`, and `TextFrames`. Modules whose names are in singular form typically denote a resource that the module manages, be it a global variable or a device. The variable or the device is itself hidden (not exported) and becomes accessible through the module's exported procedures. Examples are all device drivers, `Input` for keyboard and mouse, `Kernel` for memory and disk, and `Display`. Exceptions are the command modules whose name is mostly chosen according to the activity they primarily represent, like `System`, and `Edit`.

`Oberon` is, as already mentioned, the heart of the system containing the central loop to which control returns after each command interpretation, independent of whether it terminates normally or abnormally. `Oberon` exports several procedures of auxiliary nature, but primarily also the one allowing the invocation of commands (`Call`) and access to the command's parameter text through variable `Oberon.Par`. Furthermore, it contains global, exported variables: the log text. The log text typically serves to issue prompts and short failure reports of commands. The text is displayed in a log viewer that is automatically opened when module `System` is initialized. `Oberon` furthermore contains the 2 markers used globally on the display, the **mouse cursor** and the **star pointer**. It exports procedures to draw and to erase them, and allows the installation of different patterns for them.

The system shown in Fig 2 basically contains facilities for generating and editing texts, and for storing them in the FS. All other functions are performed by modules that must be added in the usual way by module loading on demand. This includes, notably, the compiler, network communication, document editors, and all sorts of programs designed by users. The high priority given in the system's conception to modularity, to avoid unnecessary frills, and to concentrate on the indispensable in the core, has resulted in a system of remarkable compactness. Although this property may be regarded as of little importance in this era of falling costs of large memories, we consider it to be highly essential. We merely should like to draw the reader's attention to the correlation between a systems' size and its reliability. Also, we do not consider it as good engineering practice to consume a resource lavishly just because it happens to be cheap. The following table lists the core's modules and the major application modules, and it indicates the size of code (in words) and static variables (in bytes) and, the number of source program lines.

REFERENCES

1. N. Wirth. The PL Oberon. Software - Practice and Experience 18, 7, (July 1988) 671-690.

module	code	data	lines
Kernel	1123	8244	263
FileDir	1963	60	352
Files	2360	148	505
Modules	1214	112	226
Input	186	32	79
Fonts	628	56	115
Display	1033	84	190
Viewers	1324	104	206
Texts	2906	204	537
Oberon	1679	288	410
MenuViewers	1513	56	208
TextFrames	5786	292	874
System	2134	72	418
Edit	1096	1104	232
	24945	10856	4615
ORS	1762	992	319
ORB	2348	408	437
ORG	6699	34976	1125
ORP	5994	144	974
	16803	36520	2855
Graphics	3484	564	685
GraphicFrames	2832	288	498
Draw	690	40	164
Rectangles	649	40	118
Curves	1765	72	241
	9420	1004	1706

Table 1: Size of Oberon Modules

2. M. Reiser and N. Wirth. Programming in Oberon - Steps beyond Pascal and Modula. AddisonWesley, 1992. ISBN 0-201-56543-9

3. N. Wirth and J. Gutknecht. The Oberon. Software - Practice and Experience, 19, 9 (Sept. 1989), 857-893.

4. N. Wirth. Ceres-Net: A low-cost computer network. Software - Practice and Experience, 20, 1 (Jan. 1990), 13-24.

5. M. Reiser. The Oberon - User Guide and Programmer's Manual. Addison-Wesley, 1991. ISBN 0-201-54422-9

3 TASK

Eventually, it is the generic ability to perform every conceivable task that turns a computing device into a versatile universal tool. Consequently, the issues of modeling and orchestrating of tasks are fundamental in the design of any OS. Of course, we cannot expect a single fixed tasking metaphor to be the ideal solution for all possible kinds of systems and modes of use. For example, different metaphors are probably appropriate in the cases of a closed mainframe system serving a large set of users in time-sharing mode, on one hand, and of a personal workstation operated by a single user at a high degree of interactivity, on the other.

In the case of Oberon, we have consciously concentrated on the domain of personal workstations. More precisely, we have directed Oberon's tasking facilities towards a single-user interactive personal workstation that is possibly integrated into a local area network.

We start in 3.1 with a clarification of the technical notion of task. In 3.2 we continue with a detailed explanation of the scheduling strategy. Then, in 3.3 we introduce the concept of command. Finally, 3.4 provides an overview of predefined system-oriented toolboxes, i.e. coherent collections of commands devoted to some specific topic, e.g. system control and diagnosis, display management, and file management.

3.1 THE CONCEPT OF TASK

In principle, we distinguish 2 categories of tasks in Oberon:

INTERACTIVE	bound to local regions on the display and interactions with their contents,
BACKGROUND	system-wide and not necessarily related to any specific displayed entity.

3.1.1 *Interactive Tasks*

Every interactive task is represented by a so-called viewer. Viewers constitute the interface to Oberon's display-system. They embody a variety of roles that are collected in an abstract data type `Viewer`. We shall give a deeper insight into the display system in 4. For the moment it suffices to know that viewers are represented graphically as rectangles on the display screen and that they are implicit carriers of interactive tasks. Fig 3 shows a typical Oberon display screen that is divided up into 6 viewers corresponding to 6 simultaneously active interactive tasks.

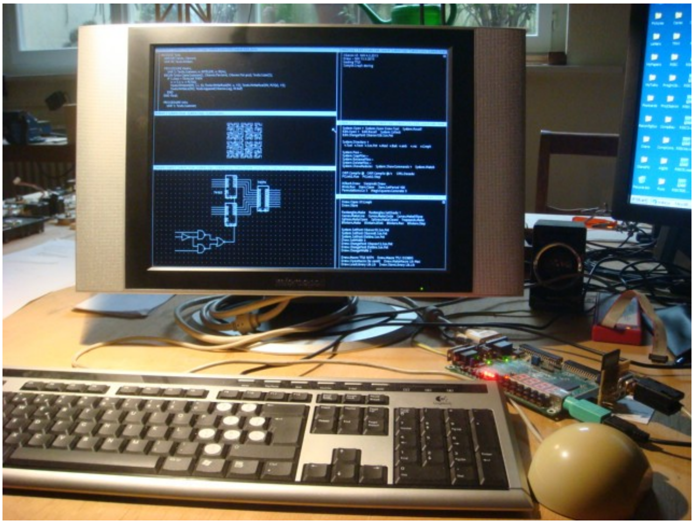


Figure 3: Typical display configuration with tool track on the right

In order to get firmer ground under our feet, we now present the programmed declaration of type `Viewer` in a slightly abstracted form:

```
Viewer = POINTER TO ViewerDesc;
ViewerDesc = RECORD X,Y,W,H, state: INT;
                    handle: Handler END;
```

`X`, `Y`, `W`, `H` define the viewer's rectangle on the screen:

> location `X`, `Y` of the lower left corner relative to the display origin,
>
> width `W` and height `H`.

`state` informs about the current state of visibility

```
visible,
closed,
covered
```

while `handle` represents the functional interface of viewers:

```
  Handler = PROC(V: Viewer; VAR M: ViewerMsg);
```

where `ViewerMsg` is some base type of messages whose exact declaration is of minor importance for now:

```
  ViewerMsg = RECORD (*basic parameter fields*) END;
```

However, we should point out the use of object-oriented (OO) terminology. It is justified because `handle` is a procedure variable (a handler) whose identity depends on the specific viewer. A call `V.handle(V, M)` can therefore be interpreted as the sending of a message `M` to be handled by the method of the receiving viewer `V`.

We recognize an important difference between the standard OO model and our handler paradigm. The standard model is closed in the sense that only a fixed set of messages is understood by a given class of objects. In contrast, the handler paradigm is open because it defines just the root (`ViewerMsg`) of a potentially unlimited tree of extending message types. For example, a concrete handler might be able to handle messages of type:

```
MyViewerMsg = RECORD (ViewerMsg)
                mypar: MyParameters
              END;
```

which is an extended type of `ViewerMsg`.

It is worth noting that our open OO model is extremely flexible. Notably, extending the set of message types that are handled by an object is a mere implementation issue, that is, it has no effect at all on the object's compile-time interface and on the system integrity. It is fair to mention though that such a high degree of extensibility does not come for free. The price to pay is the obligation of explicit message dispatching at runtime. The following chapters will capitalize on this property.

Coming back to the perspective of tasks, we note that each sending of a message to a viewer corresponds to an activation or reactivation of the interactive task that it represents.

3.1.2 *Background Tasks*

Oberon background tasks are not connected a priori with any specific aggregate in the system. Seen technically, they are instances of an abstract data type consisting of type declarations `Task` and `TaskDesc` together with intrinsic operations `NewTask`, `Install` and `Remove`:

```
Task = POINTER TO TaskDesc;
TaskDesc = RECORD state: INT; handle: PROC END;
PROC NewTask(h: PROC; period: INT): Task;
PROC Install(T: Task);
PROC Remove (T: Task);
```

The procedures `Install` and `Remove` are called explicitly in order to transfer the state of the specified task from `offline` to `idle` and from `idle` to `offline` respectively. Installed tasks take their turns in becoming `active`, that is, in being executed. The installed handlers are simple, parameterless procedures specifying their own actions and conditions for execution, with one exception: Resumption may be delayed until a certain period of time has elapsed. This period is specified in milliseconds when a task is created.

The following 2 examples of concrete background tasks may serve a better understanding of our explanations. The 1st is a system-wide garbage collector collecting unused memory. The 2nd is a network monitor accepting incoming data on a local area network. In both examples the state of the task is captured entirely by global system variables. We shall come back to these topics in **??** and **??** respectively.

We should not end without drawing an important conclusion. Transfers of control between tasks are implemented in

Oberon as ordinary calls and ordinary procedures returns (procedure variables, actually). Preemption is not possible. From that we conclude, active tasks periods are sequentially ordered and controlled by a single thread. This simplification pays well: Locks of common resources are completely dispensable and deadlocks are not a topic.

3.2 THE TASK SCHEDULER

We start from the general assumption that, at any given time, a number of well-determined tasks are ready in the system to be serviced. Remember that 2 categories of tasks exist: interactive and background ones. They differ substantially in the criteria of activation or reactivation and in the priority of dispatching. Interactive tasks are (re)activated exclusively upon interactions by the user and are dispatched with high priority. In contrast, background tasks are polled with low priority.

We already know that interactive tasks are activated by sending messages. The types of messages used for this purpose are `InputMsg` and `ControlMsg` reporting keyboard events and mouse events respectively. Slightly simplified, they are declared as

```
InputMsg = RECORD (ViewerMsg) id, X, Y: INT;
                        keys: SET; ch: CHAR END;
```

```
ControlMsg = RECORD (ViewerMsg) id,X,Y: INT END;
```

The field `id` specifies the exact request transmitted with this specific reactivation. In the case of `InputMsg` the possible requests are consume (the character specified by field `ch`) and track (mouse, starting from state given by `keys` and `X`, `Y`). In case of `ControlMsg` the choice is mark (the viewer at position `X`, `Y`) or neutralize. Mark means moving the global system pointer (typically represented as a star-shaped mark) to the current position of the mouse. Neutralizing a viewer is equivalent to removing all marks and graphical attributes from this viewer.

All tasking facilities are collected in 1 module, called `Oberon`. In particular, the module's definition exposes the declarations of the abstract data type `Task` and of the message types `InputMsg` and `ControlMsg`. The module's most important contribution, however, is the task scheduler (often referred to as "Oberon loop") that can be regarded as the system's dynamic center.

Before studying the scheduler in detail we need some more preparation. We start with the institution of the focus viewer. By definition, this is a distinguished viewer that by convention consumes subsequent keyboard input. Note that we identify the focus viewer with the focus task, hereby making use of the one-to-one correspondence between viewers and tasks.

Module `Oberon` provides the following facilities in connection with the focus viewer: A global variable `FocusViewer`, a procedure `PassFocus` for transferring the role of focus to a new viewer, and a defocus variant of `ControlMsg` for notifying the old focus viewer of such a transfer.

The implementation details of the abstract data type `Task` are hidden from the clients. It is sufficient to know that all task descriptors are organized in a ring and that a pointer points to the previously activated task. The ring is guaranteed never to be empty because the above mentioned garbage collector is installed as a permanent sentinel task at system loading time.

The following is a slightly abstracted version of the actual scheduler code operating on the task ring. It should be associated with procedure `Loop` in the module `Oberon`.

```
get mouse position and state of keys;
REPEAT
  IF keyboard input available THEN read character
    IF character is escape THEN
      broadcast neutralize message to viewers
    ELSIF character is mark THEN
      send mark message to viewer containing mouse
    ELSE send consume message to focus viewer
    END;
    get mouse position and state of keys
    ELSIF at least one key pressed THEN
      REPEAT
        send track message to viewer containing mouse;
        get mouse position and state of keys
      UNTIL all keys released
    ELSE (*no key pressed*)
      send track message to viewer containing mouse;
      take next task in ring as current task;
      call its handler
              (if specified time period has elapsed)
      get mouse position and state of keys
    END
  UNTIL FALSE
```

The system executes a sequence of uninterrupted procedures (tasks). Interactive tasks are triggered by input data being present, either from the keyboard, mouse, or other input sources. Background tasks are taken up in a round-robin manner. Interactive tasks have priority.

Having consciously excluded exceptional program behavior in our explanations so far, some comments about the way of runtime continuation in the case of a failing task or, in other words, in the case of a trap are in order here. On the (abstract) level of tasks, we can identify 3 sequential actions of recovery taken after a program failure:

```
  recovery after program failure =
BEGIN save current system state;
  call installed trap handler;
  roll back to start of task scheduler
END
```

Essentially, the system state is determined by the values of all global and local variables at a given time. The trap handler typically opens an extra viewer displaying the cause of the trap and the saved system state. Notice in the program fragment above that background tasks are removed from the ring after failing. This is an effective precaution against cascades of repeated failures. Obviously, no such precaution is necessary in the case of interactive tasks because their reactivation is under control of the user of the system.

Summarizing the essence of the Oberon tasking system:

- A multitasking system based on a 2-category model

  INTERACTIVE interfacing with the display system, high-priority scheduled upon user interactions;

  BACKGROUND stand-alone, low-priority scheduled.

- Task activations are modeled as message passing, eventually, procedures calls assigned to variables. They're sequentially ordered, controlled by a single thread.

3.3 THE CONCEPT OF COMMAND

An OS constitutes a general purpose platform on which applications can build upon. To software designers the platform appears as interface to "the system" and (in particular) to the underlying hardware. Unfortunately, interfaces defined by conventional OSes often suffer from an all too primitive access mechanism that is based solely on the concept of "software interrupt" or "supervisor call" and on files taking the role of “connecting pipes". The situation is especially ironic when compared with the development of high-level PLs towards extreme abstraction.

We have put greatest emphasis in Oberon on closing the semantic gap between applications and the system platform. The result of our efforts is a highly expressive and consistent *application programming interface* (API) in the form of an explicit hierarchy of module definitions. Perhaps the most significant and notable outcome of this approach is a collection of very powerful and system-wide abstract data types such as `Task`, `Frame`, `Viewer`, `File`, `Font`, `Text`, `Module`, `Reader`, `Scanner`, `Writer`, etc.

3.3.1 *Atomic Actions*

The most important generic function of any OS is executing programs. A clarification of the term program as it is used in Oberon comprises 2 views: a static and a dynamic one.

STATICALLY, program means a software package with an *entry point*. More formally, it is a pair `(M*, P)`, where `M` is a module, `P` is an *exported parameterless procedure* of `M`, and `M*` denotes the hierarchy consisting of `M` itself and all imported modules, directly or indirectly.

Note: 2 different hierarchies `M*` and `N*` are not generally disjoint, rather, intersect a superset of the OS.

DYNAMICALLY VIEWED, it is an *atomic action* called *command*, operating on the global system state. Here *atomic* means *no user interaction*. This definition is just a necessary consequence of our *non-preemptive task model* scheduling with the benefit of a single *carrier thread*. We can argue like this:

When a traditional interactive program requires input from the user, the current task is normally preempted in favor of another task that produces the required input data. Therefore, a traditional interactive program can be viewed as a sequence of atomic actions interrupted by actions that possibly belong to other programs. Whereas interruption in traditional systems may occur at any time, yet in Oberon it can occur only after task (command) completion.

Quintessentially, programs are represented as commands: exported parameterless procedures not interacting with the user.

Returning to programs calling and execution we now arrive at the following refined version:

```
call program(M*, P) = BEGIN
  load module hierarchy M*;
  call command P
END
```

The system interface to the command mechanism itself is again provided by `Oberon`. Its primary operation is to "call command by name and pass actual parameters":

```
PROC Call(name: ARRAY OF CHAR;
          pars: ParList; VAR res: INT);
```

`name`	the desired command name in form of `M.P`,
`pars`	the actual parameters list (APL), and
`res`	the result code.

But in fact we have separated the setting of parameters:

```
PROC SetPar(F: Display.Frame;
            T: Texts.Text; pos: INT);
```

`F`	indicates the calling viewer,
Pair `(T, pos)`	specifies the starting position of a text.

from the actual call:

```
PROC Call(name: ARRAY OF CHAR; VAR res: INT);
```

Notice the occurrence of yet another abstract data type `Text`, exported by `Texts`. We shall devote 5 to a thorough discussion of Oberon's text system. For the moment we can simply look at a text as a sequence of characters.

The APL is handed over to the command by `Oberon` as an exported global variable:

```
Par: RECORD vwr: Viewers.Viewer;
       frame: Display.Frame;
       text: Texts.Text; pos: INT
     END
```

In principle, commands operate on the entire system and can access the current global state via the system's powerful abstract modular interface, of which the APL is just one component. Another one is the so-called *system log* which is a system-wide protocol reporting on the progress of command execution and on exceptional events in chronological order. The log is represented as a global variable:

```
Log: Texts.Text;
```

It should have become clear by now that implementers of commands may rely on a rich arsenal of abstract global facilities that reflect the current system state and make it accessible. In other words, they may rely on a high degree of system integration. Therefore, Oberon features an extraordinarily broad spectrum of mutually integrated facilities. For example, the system distinguishes itself by a complete integration of the abstract data types `Viewer` (4) and `Text` (5) that we encountered above.

`Oberon` assists the integration of these types with the following conceptual features, including the 1st 2 we have just introduced:

1. Standard parameter list for commands,
2. system log,
3. generic text selection, and
4. generic copy viewer.

At this point we should add a word of clarification to our use of the term *generic*. It is synonymous with "interpretable individually by any viewer (interactive task)" and is typically used in connection with messages or orders whose receiver's exact identity is unknown.

Let us now go into a brief discussion of the generic facilities without, however, leaving the level of our current abstraction and understanding.

3.3.2 *Generic Text Selection*

Textual selections are characterized by

> a text,
> a stretch of characters within that text, and
> a time stamp.

Without further qualification "the text selection" always means "the most recent text selection". It can be obtained programmatically by calling:

```
PROC GetSelection(VAR text: Texts.Text;
                  VAR beg, end, time: LONGINT);
```

The parameters specify the desired stretch of text starting at position `beg` and ending at `end-1` as well as the associated time stamp. The procedure is implemented in form of a broadcast of a socalled selection message to all viewers. The declaration of this message is

```
SelectionMsg = RECORD (ViewerMsg) text: Texts.Text;
                        beg, end, time: INT  END;
```

3.3.3 *Generic Copy Viewer*

Generic copying is synonymous with reproducing and cloning. It is the most elementary generic operation possible. Again, a variant of type `ViewerMsg` is used for the purpose of transmitting requests of the desired type:

```
CopyMsg = RECORD (ViewerMsg) vwr:Viewers.Viewer END;
```

Receivers of a copy message typically generate a clone of themselves and return it to the sender via field `vwr`.

Let us now summarize the concept of command in Oberon:

- As an OS Oberon presents to clients a highly expressive *modular interface* that exports many powerful abstract data types like `Viewer` and `Text`. A rich arsenal of global data types and generic facilities achieve *system integration* at a high degree.

- Programs are modeled as *commands, exported parameterless procedures* not interacting with the user.

- The collection of commands provided by a module appears as its *user interface*. Parameters are passed to commands via a global parameter list, registered by the calling task in the central `Oberon`.

- Commands operate on the global state of the system.

3.4 TOOLBOXES

Modules typically appear in 3 different forms.

1st encapsulates some data, which can be accessed only through exported procedures and functions. `FileDir`, encapsulating the file directory and protecting it from disruptive access, is a good example.

2nd represent an abstract data type, exporting it and its associated operators. `Files`, `Modules`, `Viewers`, and `Texts` are typical examples.

3rd collection of procedures pertaining to the same topic, like `RS232` handling communication over a serial line.

Oberon adds a 4th form:

toolbox A pure collection of commands on the *top* of the modular hierarchy. Toolbox modules are "imported" by system users at run-time. In other words, their definitions define the user interface. Typical examples are `System` and `Edit`.

As a rule of thumb there exists a toolbox for every topic or application. As an example of a toolbox definition we quote an annotated version of `System`:

```
DEFINITION System;
  (*System management, Ch. 3 and 8*)
  PROC SetUser;  (*identification*)
  PROC SetFont;  (*for typed text*)
  PROC SetColor; (*for typed text&graphics*)
```

```
    PROC SetOffset;(*for typed text*)
    PROC Date;     (*set/get date&time*)
    PROC Collect;  (*garbage*)
    (*Display management, Ch. 4*)
    PROC Open;  (*viewer*)
    PROC Close; (*viewer*)
    PROC CloseTrack;
    PROC Recall;(*most recently closed viewer*)
    PROC Copy;  (*viewer*)
    PROC Grow;  (*viewer*)
    PROC Clear; (*log*)
    (*Module management, Ch. 6*)
    PROC Free;        (*specified modules*)
    PROC ShowCommands;(*of specified module*)
    PROC ShowModules; (*show loaded modules*)
    (*File management, Ch. 7*)
    PROC Directory;
    PROC CopyFiles;
    PROC RenameFiles;
    PROC DeleteFiles;
    (*System inspection, Ch. 8*)
    PROC Watch;(*tasks, memory&disk storage*)
  END System;
```

An important consequence of our integrated systems approach is the possibility of constructing a universal, interactive command interpreter bound to viewers of textual contents. If the text obeys the following syntax (specified in EBNF), we call it command tool:

```
CommandTool = {[Comment]CommandName[ParameterList]}
```

If present, the parameter list is made available to the called command via fields `text` and `pos` in the global variable `Par` that is exported from `Oberon`. Because this parameter list is interpreted individually by each command, its format is completely open. However, we postulate some conventions and rules for the purpose of a standardized user interface:

1. The elements of a textual parameter list (TPL) are universal syntactical tokens like name, literal string, integer, real number, and special character.

2. An arrow "^" in the TPL refers to the current text selection for continuation. In the special case of the arrow following the command name immediately, the entire parameter list is represented by the text selection.

3. An asterisk "*" in the TPL refers to the currently marked viewer. Typically, the asterisk replaces the name of a file. In such a case the contents of the viewer marked by the system pointer (star) is processed by the command interpreter instead of the contents of a file.

4. An at-character "@" in the TPL indicates that the selection marks the (beginning of the) text which is taken as operand.
5. A terminator-character "~" terminates the TPL in case of a variable number of parameters.

Because command tools are ordinary, editable texts (in contrast to menus in conventional systems), they can be customized "on the fly", which makes the system highly flexible. We refer again to Fig 3 that shows a typical Oberon screen layout consisting of 2 vertical tracks,

1. a wider user track on the left, and
   3 documents are displayed in the user track:
   a) a text,
   b) a graphic, and
   c) a picture.
2. a narrow system track on the right,
   where we find 1 logviewer displaying the system log, 2 tool-viewers making available
   a) the standard system tool, and
   b) a customized private tool respectively.

In concluding this chapter, let us exemplify the concepts of command and tool by the system control section of the `System` toolbox. Consisting of the commands

`SetUser`	installing the user's identification,
`Date`	displaying or setting the system date and time,
`SetFont`	presetting the system type-font for typed text,
`SetColor`	setting the system color, and
`Collect`	activating the garbage collector.

They are used to control system-wide facilities (detail each).

In summary,

- a toolbox is a special module, defined as a *collection of commands;*
- appearing at the top of the modular hierarchy, the entire toolboxes fix the *user interface* for the system;
- command tools are sequences of textually represented command calls, editable and customizable;
- in a typical screen layout, tools are displayed in the viewers within the *system track*.

4 DISPLAY

Of the interface a PC presents to a user, the display screen is the most important. At the 1st sight, it simply represents a rectangular output area. However, in combination with a mouse, it quickly develops into a sophisticated interactive I/O platform of almost unlimited flexibility. It is mainly this Janus-faced characteristic that makes it stand out from ordinary external devices to OS-managed core system.

In this chapter we shall give more detailed insight into why

- central position a display system takes within an OS, and
- its determining influence on entire system architecture.

In particular, we shall show that it is a natural basis or anchor for functional extensibility.

4.1 SCREEN LAYOUT MODEL

In the early 70s, Xerox PARC in California launched the Smalltalk project with the goal of conceiving and developing new more natural ways to communicate with PCs [**Goldberg**]. Perhaps the most conspicuous among several significant achievements of this endeavor is the idea of applying the desktop metaphor to the display screen. This metaphor comprises a desktop and the collection of possibly mutually overlapping pages of paper laid out on it. By projecting such a configuration onto the screen surface we get the familiar picture of Fig 4 showing a collection of partially or totally visible rectangular areas on a background, so-called *windows* or *viewers*.

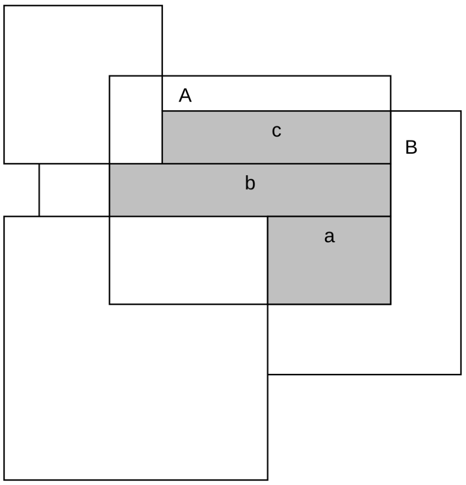


Figure 4: Desktop showing partially overlapping viewers

The desktop metaphor is used by many modern OSes and user interface shells both as

a	for	to
natural model	the system	separate displayed data belonging to different tasks, and
power -ful tool	users	organize the display screen interactively, according to individual taste and preference.

However, in this metaphor there are inherent drawbacks, primarily connected with overlapping.

1st, any efficient management of overlapping viewers must rely on

- a subordinate management of (arbitrary) sub-rectangles, and
- sophisticated clipping operations.

This is so because partially overlapped viewers must be partially restored under control of the *viewer manager*. For example, in Fig 4, rectangles *a*, *b*, and *c* in viewer *B* ought to be restored individually after closing of *A*.

2nd, there is a significant danger of covering viewers completely and losing them forever.

3^{rd}, no canonical heuristic algorithms exist for automatic allocation of screen space to newly opened viewers.

Experience has shown that partial overlapping is desirable and beneficial in rare cases only, so the additional complexity of its management is hard to justify. [**Binding**; **Wille**] Therefore, alternate strategies to structure a display screen have been looked for. An interesting class of established solutions can be titled as *tiling*. There are at least 2 variants. [**Cohen**]

Perhaps the most *unconstrained* (hence obvious) one is based on iterated horizontal or vertical splitting of existing viewers. Starting with the full screen and successively opening *A*, *B*, *C*, *D*, *E*, and *F* we get to a configuration as in Fig 5.

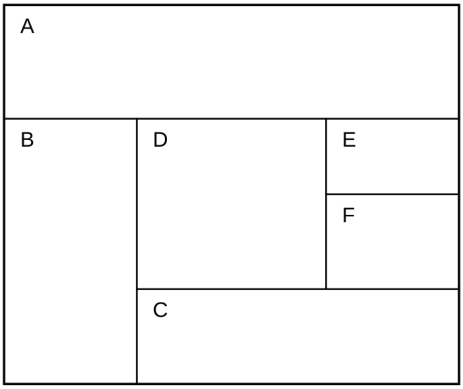


Figure 5: Viewer configuration resulting from unconstrained tiling

The 2nd variant is *hierarchic tiling*. Again, the hierarchy starts with a full screen that is now decomposed into a number of *vertical tracks*, each of which is further decomposed into a number of horizontal viewers. We decided in favor of this kind of tiling in Oberon, mainly because the algorithm of reusing the area of a closed viewer is simpler and more uniform. For example, assume that in Fig 5 viewer *F* has been closed. Then, it is straightforward to reverse the previous opening operation by extending viewer *E* at its bottom end. However, if the closed viewer is *B*, no such simple procedure exists. For example, the freed area can be shared between viewers *C* and *D* by making them extend to their left. Clearly, no such complicated situations can occur in the case of hierarchic tiling.

It is also used in Xerox PARC's Cedar system [**Teitelman**]. However, Oberon differs in:

1st, It supports quick temporary context switching by overlaying one track or any contiguous sequence of tracks with new layers. In Fig 6 a snapshot of a standard display screen is graphically represented. It suggests 2 original tracks and 2 levels of overlay, where the top layer is screen-filling.

2nd, Oberon displays do not provide reserved areas for system-wide facilities, while standard Cedar screens feature a command row at the top and an icon row at the bottom. And

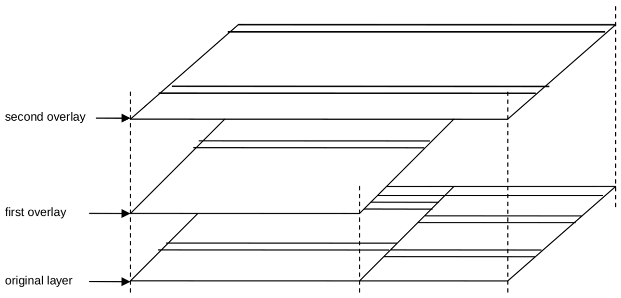


Figure 6: Overlay of tracks and sequences of tracks

3^{rd}, It is based on a different heuristic strategy for the automatic placement of new viewers. As a Cedar default invariant, the area of every track is divided up evenly among the viewers in this track. When a new viewer is to be placed, the existing viewers in the track are requested to reduce their size and move up appropriately. The newly opened is then allocated in the freed spot at the bottom. In contrast, Oberon normally splits the largest existing viewer in a given track into 2 halves of equal size. As an advantage of this latter allocation strategy we note that existing contents are kept stable.

4.2 VIEWERS AS OBJECTS

Although everybody seems to agree on the meaning of the term *viewer*, no 2 different system designers actually do. The original role of a viewer as merely a separate display area has meanwhile become heavily overloaded with additional functionality. Depending on the underlying system are viewers' individual views on a certain configuration of objects, carriers of tasks, processes, applications, etc. Therefore, we first need to define our own precise understanding of the concept of viewer.

The best guide to this aim is the abstract data type `Viewer` we introduced in 3. We recapitulate: `Viewer` serves as a template describing viewers abstractly as "black boxes" in terms of a state of visibility, a rectangle on the display screen, and a message handler. The exact functional interface provided by a given variant of viewer is determined by the set of messages accepted. This set is structured as a customized hierarchy of type extensions.

We can now obtain a more concrete specification of the role of viewer by identifying some basic categories of universal messages that are expected to be accepted by all variants of viewer. For example, we know that messages reporting about user interactions as well as messages defining a generic operation are universal. These 2 categories of universal messages document the roles of viewers as interactive tasks and as parts of an integrated system respectively. In total, there are 4 such categories. They are here listed together with the corresponding topic and message dispatchers:

Dispatcher	Topic	Message
task scheduler	dispatching tasks	report user interaction
cmd interpreter	processing command	define generic operation
view manager	organizing display area	location/size change
doc manager	operating on document	content/format change

These topics essentially define the role of viewers. In short, we may look at an viewer as a non-overlapped rectangular box on the screen both

- acting as an *integrated display area* for some objects of a document, and
- *representing an interactive task* in the form of a sensitive editing area.

Shifting emphasis a little and regarding the various message dispatchers as subsystems, we recognize immediately the role of viewers as *integrator of the different subsystems via message-based interfaces*. In this light, `Viewer` appears as a common object-oriented basis of Oberon's subsystems.

The topics listed above constitute some kind of contents backbone of the 3, 4 and 5. Task scheduling and command interpreting are already familiar to us from §3.2 and 3.3. Viewer and text management will be the topics of §**??** and 5.2, respectively. Thereby, the built-in type `Text` will serve as a prime example of a document type. The activities that a viewer performs are basically controlled by events or, more precisely, by messages representing event notices. We shall explain this in §?? and 5.3 in detail cases of an abstract class of standard viewers and a class of viewers displaying standard text, respectively.

Here is a preliminary overview of some archetypal kinds of message:

- After each stroke a keyboard message with the typed character is sent to the focus viewer and after each click a mouse message of the new state is sent to the viewer containing the mouse.

- The message often representing some generic operation is to be interpreted individually by recipients. Obvious examples are "return current textual selection", "copy-over stretch of text", and "produce a copy (clone)". Notice that generic operation is the key to extensibility.

- In a tiling viewer environment, every opening of a new viewer and every change of size or location of an existing viewer has an obvious effect on adjacent viewers. The viewer manager therefore issues a message to every affected viewer requesting to adjust its size appropriately.

- Whenever the contents or the format of a document has changed, a message notifying all visible viewers of the change is broadcast. Notice that broadcasting messages by a model (document) to the entirety of its potential views (viewers) is an interesting implementation of the famous model-view-controller (MVC) pattern that dispenses models from "knowing" (registering) their views.

4.3 FRAMES AS BASIC DISPLAY ENTITIES

When we introduced viewers in 3 and 4.2, we simplified the abstraction aim. We know already that viewers appear as elements of 2nd order in the tiling hierarchy. Having treated them as black boxes so far we have not revealed anything about the hierarchy continuation. As a matter of fact, viewers are neither elementary display entities nor atoms. They are just a special case of so-called *(display) frames*. They are arbitrary rectangles displaying a collection of objects or a document excerpt. In particular, they may recursively contain other frames, a capability that makes them an extremely powerful tool for any display organizer.

Type `Frame` is declared as

```
Frame = POINTER TO FrameDesc;
FrameDesc = RECORD next, dsc: Frame;
                  X, Y, W, H: INT;
                     handle: Handler END;
```

`next` and `dsc` are connections to further frames, whose names suggest a recursive hierarchical structure:

	points to
next	the next frame on the same level, while
dsc	the (1st) descendant, i.e. the next lower level of the nested frames hierarchy.

X, Y, W, H, and the handler handle serve the original purpose to which we introduced them. In particular, the handler allows frames to react individually on the receipt of messages:

```
Handler = PROC(F: Frame; VAR M: FrameMsg);
```

where FrameMsg represents the root of a potentially unlimited tree hierarchy of possible messages to frames:

```
FrameMsg = RECORD END;
```

Having now introduced the concept of frames, we can reveal the whole truth about viewers. As a matter of fact, Viewer is a derived type, a extension of Frame:

```
Viewer = POINTER TO ViewerDesc;
ViewerDesc = RECORD (FrameDesc) state: INT END;
```

These declarations formally express the fact that viewers are nothing but a special case (or variant or subclass) of general frames, additionally featuring a state of visibility. In particular, viewers inherit the hierarchical structure of frames. This is an extremely useful property immediately opening an unlimited spectrum of possibilities for designers of a specific subclass of viewers to organize the representing rectangular area. For example, the area of viewers of, say, class Desktop may take the role of a background being covered by an arbitrary collection of possibly mutually overlapping frames. In other words, our decision of using a tiling viewer scheme globally can easily be overwritten locally.

An even more important example of a predefined structure is provided by the abstract class, called *menu viewers*, whose shape is familiar from most snapshots taken of the standard Oberon display screen. A menu viewer consists of a thin rectangular boundary line and an interior area being vertically decomposed into a menu region at the top and a contents region at the bottom (see Fig 7).

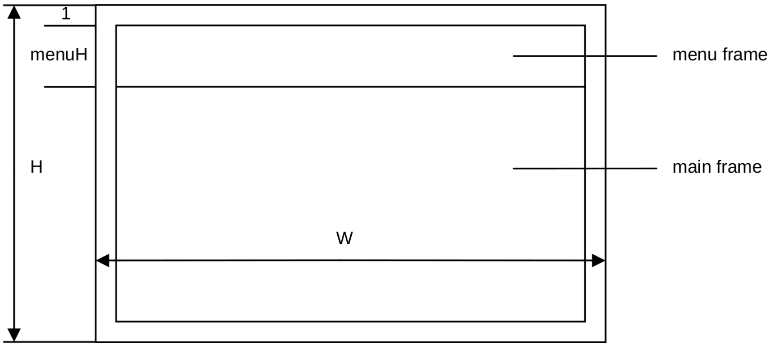


Figure 7: The compositional structure of a menu viewer

In terms of data structures, the class of menu viewers is defined as a type extension of `Viewer` with an additional component `menuH` specifying the menu frame height:

```
MenuViewer = POINTER TO MenuViewerDesc;
MenuViewerDesc = RECORD (ViewerDesc) menuH: INT END;
```

Each menu viewer `V` specifies exactly 2 descendants:

- the menu frame `V.dsc`, and
- the frame of main contents or main frame `V.dsc.next`.

Absolutely nothing is fixed about the 2 descendant frames contents. The standard menu frame is a text frame, displaying a line of commands in inverse video mode. By definition, the nature of the main frame specifies its type:

- If it is a text frame, we call the viewer a text one;
- If a graphics frame, we call it a graphics viewer, etc.

4.4 DISPLAY MANAGEMENT

Oberon's display system comprises 2 main topics:

1. *viewer management* (VM), and
2. *cursor handling*.

Let us 1st turn to the much more involved topic of 1 and postpone 2 to the end of this section. Before we can actually begin our explanations we need to introduce the concept of the *logical display area*. It is modeled as a 2-dimensional Cartesian plane housing the totality of objects to be displayed. The essential point of this abstraction is a rigorous decoupling of any aspects of physical display devices. As a matter of fact, any concrete assignment of display monitors to certain finite regions of the display area is a pure matter of configuring the system.

Being a subsystem of OS with well-defined modular structure, the display system appears in the form of a small hierarchy of modules. Its core is a linearly ordered set consisting of 3 modules:

`Display`,
`Viewers`, and
`MenuViewers`,

the latter building upon the former. Conceptually, each module contributes an associated class of display-oriented objects and a collection of related service routines.

The following is an overview of the subsystem VM.

Module	Type	Service
MenuViewer	Viewer	Message handling for menu viewers
Viewers	Viewer	Tiling VM
Display	Frame	Block-oriented raster operations

Modules on upper lines import lower ones, and
types on upper lines extend those on lower.

Inspecting the "Type" column we recognize precisely our familiar types

`Frame`,
`Viewer`, and
`MenuViewer` respectively,

where the last is an abbreviation of `MenuViewers.Viewer`. In addition to the core modules of the display system a section in `Oberon` provides a specialized API that simplifies the use of the VM package by applications in the case of standard Oberon display configurations. We shall come back to this topic in §4.6.

For this moment let us concentrate on the VM core and in particular the `Viewers` and `MenuViewers`, saving the `Display` for the next section. Typically, we start a module presentation by listing and commenting its definition, and refer to subsequent listings for its implementation.

4.4.1 *Viewers*

Focusing 1st on module `Viewers` we can roughly define the domain of its responsibility as *initializing and maintaining the global layout of the display area*. From the previous discussion we are well acquainted already with the structure of the global display space as well as its building blocks:

> The display area is hierarchically tiled with frames, where the first two levels in the frame hierarchy correspond to *tracks* and *viewers* respectively.

The formal definition:

```
DEFINITION Viewers;
  IMPORT Display;               (*message ids*)
  CONST restore = 0; modify  = 1; suspend = 2;
  TYPE Viewer = POINTER TO ViewerDesc;
       ViewerDesc = RECORD (Display.FrameDesc)
                      state: INT  END;
       ViewerMsg = RECORD (Display.FrameMsg)
                     id, X, Y, W, H: INT;
                     state: INT  END;
  VAR curW: INT; (*currently configured width*)
  PROC InitTrack(W, H: INT; Filler: Viewer);
  PROC OpenTrack(X, W: INT; Filler: Viewer);
  PROC CloseTrack(X: INT);        (*track handling*)

  PROC Open(V: Viewer; X, Y: INT);
  PROC Change(V: Viewer; Y: INT);
  PROC Close(V: Viewer);          (*viewer handling*)

  PROC This(X, Y: INT): Viewer;
  PROC Next(V: Viewer): Viewer;
  PROC Recall(VAR V: Viewer);       (*miscellaneous*)
  PROC Locate(X,H: INT; VAR fil,bot,alt,max: Viewer);
  PROC Broadcast(VAR M: Display.FrameMsg);
END Viewers.
```

The 1st 3 support the track structure of the display area.

- `InitTrack` creates a new track of width `W` and height `H` by partitioning off a vertical strip of width `W` from the display area.

  In addition, it initializes the newly created one with a 3^{rd} parameter, a filler viewer. The filler viewer essentially serves as background filling up the track at its top end. It reduces to height 0 if the track is covered completely by productive viewers.

Configuring the display area is part of system initialization. It amounts to executing a sequence of steps:

```
NEW(Filler);
Filler.handle := HandleFiller;
InitTrack(W, H, Filler)
```

where `HandleFiller` is supposed to handle messages that require modifications of size and cursor drawing.

- The global variable `curW` indicates the already configured part width of the display area. Note that configuring starts with `x = 0` and is non-reversible in sense that the grid defined by the initialized tracks cannot be refined later. However, remember that it can be coarsened at any time by overlaying a contiguous sequence of existing tracks by a single new track.

  `OpenTrack` serves exactly this purpose. The track (or sequence of tracks) to be overlaid in the display area must be spanned by the segment `[X, X + W)`.

- `CloseTrack`, inverse to `OpenTrack`, is called to

  – close the (topmost) track located at `X` in the display area, and

  – restore the previously covered track (or sequence of tracks).

The 2nd 3 are to organize viewers within individual tracks.

- `Open` allocates a viewer at given position. More precisely,

    1. locates the viewer containing point `(X, Y)`,
    2. splits it horizontally at height `Y`, and
    3. opens the new one `V` in the lower part of area.

  In the special case of `Y` coinciding with the upper boundary line of viewer in 1, it is closed automatically.

- `Change` allows to change the height of a given viewer `V` by moving its upper boundary line to a new location `Y` (within the limits of its neighbors).

- `Close` removes the given `V` from the display area.

Fig 8 makes these operations clear.

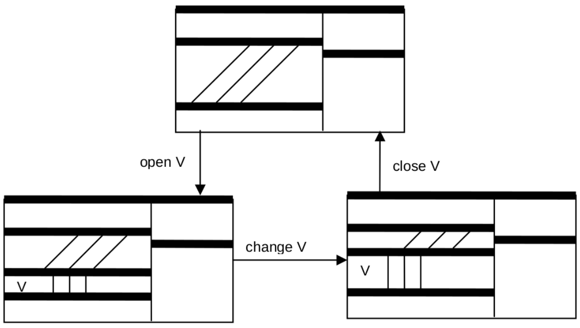


Figure 8: Basic operations on viewers

The last group provides miscellaneous services.

- `This` identifies the viewer displayed at `(X, Y)`.
- `Next` returns the next upper neighbor of a given displayed viewer `V`.
- `Recall` allows recalling and restoring the most recently closed viewer.
- `Locate` assists heuristic allocation of new viewers. For any given track and desired minimum height, it offers a choice of some distinguished viewers in this track:
  - the `fil`ler viewer,
  - the one at `bot`tom,
  - an `alt`ernative choice, and
  - the viewer of `max`imum height.
- Finally, `Broadcast` broadcasts a message to the display area, that is, sends the given message to all currently displayed viewers.

It is now a good time to throw a glance behind the scenes. Let us start with revealing `Viewer` internal data structure. Remember that according to the principle of information hiding an internal data structure is fully private to the containing module and accessible only through the module's procedural interface. Fig 9 shows a data structure view of the display snapshot taken in Fig 6. Note that the overlaid tracks and viewers are still part of the internal data structure.

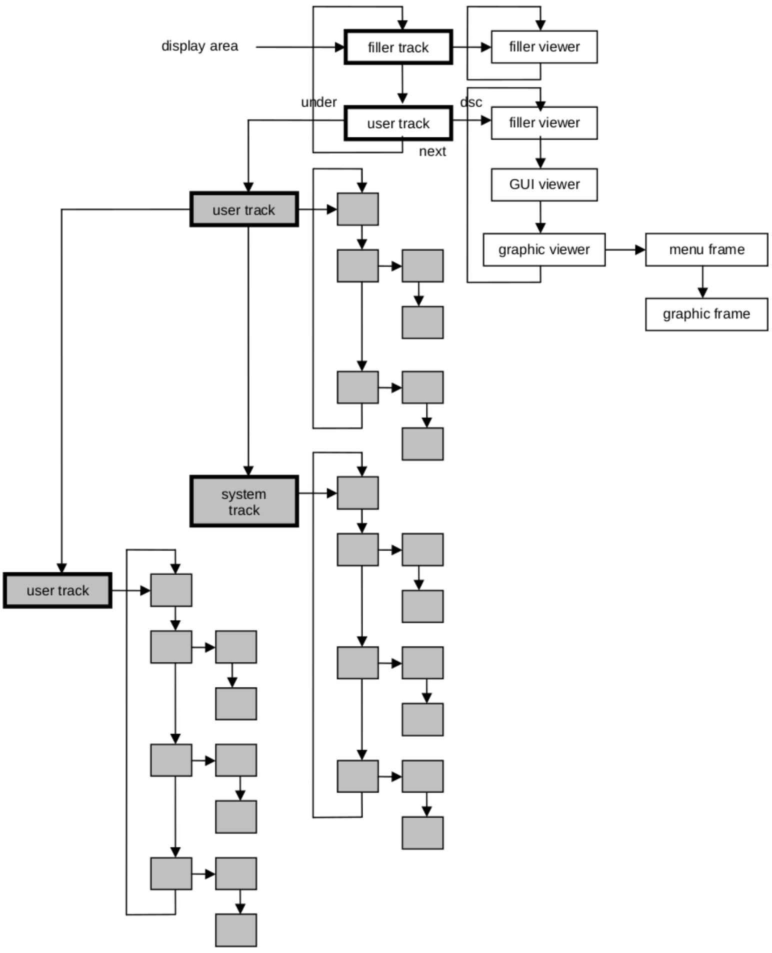


Figure 9: Internal data structure snapshot of Fig 6

In the data structure we recognize an anchor that represents the display area and points to a list of tracks, each track pointing to a list of viewers, each viewer in turn pointing to a list of arbitrary sub-frames. Both the list of tracks and the list of viewers are closed to a ring, where the filler track (filling up the display area) and the filler viewers (filling up the tracks) act as anchors. Additionally, each track points to a (possibly empty) list of tracks lying underneath. These frames are invisible on the display, and shaded in Fig 9.

Technically, the track descriptor `TrackDesc` is a private extension of the viewer descriptor `ViewerDesc`. Repeating the declarations of viewer descriptors and frame descriptors, we get to this hierarchy of types:

```
 TrackDesc = RECORD (ViewerDesc)
                   under: Display.Frame   END;

ViewerDesc = RECORD (FrameDesc) state: INT END;

 FrameDesc = RECORD  next, dsc: Frame;
                   X, Y, W, H: INT;
                       handle: Handler    END;
```

It is noteworthy that the data structure of the VM is heterogeneous with `Frame` as base type. It provides a nice example of a nested hierarchy of frames with the additional property that the 1st 2 levels correspond to the 1st 2 levels in the type hierarchy defined by `Track`, `Viewer`, and `Frame`.

In an object-oriented environment objects are autonomous entities in principle. However, they may be bound to some higher instance (other than the system) temporarily. For example, we can look at the objects belonging to a module's private data structure as bound to this module. Deciding if an object is currently bound then becomes a fundamental problem. In the case of viewers, this information is contained in an extra instance variable called `state`.

As a system invariant, we have for every viewer V

$$V \text{ is bound to module } Viewers \Leftrightarrow V.state \neq 0$$

If we call any displayed viewer *visible* and each covered by an overlaying track *suspended*, we can refine this invariant to

$$\begin{aligned} &\{V \text{ is } visible && \Leftrightarrow V.state > 0\} \text{ and} \\ &\{V \text{ is } suspended && \Leftrightarrow V.state < 0\} \end{aligned}$$

In addition, more detailed information about the kind of viewer V is given by the magnitude $|V.state|$:

$\lvert V.state \rvert$	kind of viewer
0	closed
1	filler
-1	productive

The magnitude $|V.state|$ is kept invariant by `Viewers`. It could be used, for example, to distinguish different levels of importance or preference with the aim of supporting a smarter algorithm for heuristic new viewers allocation. *state* is read-only to modules other than `Viewers`.

We are now sufficiently prepared to understand how the exported `Viewers` procedures work behind. They all operate on the internal dynamic data structure just explained.

- `This`, `Next`, `Locate`, `Change` use it as a reference only or operate on individual elements

- `InitTrack`, `OpenTrack`, `Open` add new elements, and

- `CloseTrack`, `Close` remove elements.

Most have side-effects on existing elements (*size* or *state*).

Let us now change perspective and look at `Viewers` as a general low-level viewer manager (VMer) whose exact contents are unknown to it (and whose controlling software might have been developed years later). In short, let us look at `Viewers` as a manager of black boxes. Such an abstraction immediately makes it impossible for the implementation to call fixed procedures for, say, changing a viewer's size or state. The facility needed is a *message-oriented interface*.

```
TYPE ViewerMsg = RECORD (Display.FrameMsg) id,
                                X, Y, W, H, state: INT END;
```

There're 3 variants of `Viewer` messages, discriminated by `id`:

restore contents,
modify height (extend or reduce at bottom), and
suspend (close temporarily or permanently).

The additional components of the message inform about the desired new location, size, and state. The following table lists senders, messages, and recipients of viewer messages.

Originator	Message	Recipients	
OpenTrack	suspend temporarily	viewers covered by opening track	
CloseTrack	suspend permanently	viewers in closing track	
Open	modify or suspend	upper	opening viewer
Change	modify	neighbor of	changing viewer
Close	suspend permanently		closing viewer

4.4.2 *Menu Viewers*

So far, we have treated viewers abstractly. Next step we focus on a special class called *menu viewers*. From the earlier definition we know they're characterized by a structure consisting of 2 vertically tiled *descendant* frames,

a frame of	at the
menu	top, and
contents	bottom.

Because the nature and contents of these frames are typically unknown by their "ancestor" (or "parent") viewer, a collection of abstract messages is again a postulating form of interface. As net effect, the handling of menu viewers boils down to a combination of preprocessing, transforming and forwarding messages to the descendant frames. In short, the display space is hierarchically organized and message passing within it obeys the pattern of strict parental control.

Again, we start detailed discussion with module interface:

```
DEFINITION MenuViewers;
  IMPORT Viewers, Display;        (*message ids*)
  CONST extend = 0; reduce = 1; move = 2;
  TYPE Viewer = POINTER TO ViewerDesc;
       ViewerDesc = RECORD (Viewers.ViewerDesc)
                            menuH: INT       END;
       ModifyMsg = RECORD (Display.FrameMsg)
                           id, dY, Y, H: INT  END;

  PROC Handle(       V: Display.Frame;
                 VAR M: Display.FrameMsg);

  PROC New(Menu , Main: Display.Frame;
           menuH, X, Y: INT             ): Viewer;
END MenuViewers.
```

The interface represented by this definition is conspicuously narrow. There are just 2 procedures:

a generator procedure `New`, and

Returns a newly created menu viewer displaying the 2 (arbitrary) frames passed as parameters.

a standard message handler `Handle`.

Implements the entire "behavior" of an object and in particular the above message dispatching functionality.

Message handlers in Oberon are implemented in the form of procedure variables that obviously must be initialized properly at object creation time. In other words, some concrete behavior must explicitly be bound to each object, where different instances of the same object type could potentially have a different behavior and/or the same instance could change its behavior during its lifetime. Our object model is therefore *instance-centered*.

Conceptually, the creation of an object is an atomic action consisting of 3 basic steps:

1. Allocate memory block;

2. Install message handler;

3. Initialize state variables.

In the case of a standard menu Viewer creation:

```
  NEW(V); V.handle := Handle; V.dsc      := Menu;
          V.menuH  := menuH ; V.dsc.next := Main
```

`New` equals to `create V; open V at X,Y`. Opening `V` needs `Viewers`' assistance.

Implementing `Handle` embodies the standard message handling strategy. This is a coarse-grained view:

```
IF message reports about user interaction THEN
  IF variant is mouse tracking THEN
    IF mouse is in menu region THEN
      IF mouse is in upper menu region and
         left key is pressed THEN
        handle changing of viewer
      ELSE delegate handling to menu-frame
      END
    ELSE
      IF mouse is in main-frame THEN
        delegate handling to main-frame
      END
    END
  ELSIF variant is keyboard input THEN
    delegate handling to menu frame;
    delegate handling to main frame
  END
```

```
ELSIF message defines generic operation THEN
  IF message requests copy (clone) THEN
    send copy-message to menu frame to get a copy;
    send copy-message to main frame to get a copy;
    create menu viewer clone from copies
  ELSE
    delegate handling to menu frame;
    delegate handling to main frame
  END

ELSIF message reports about change of contents THEN
  delegate handling to menu frame;
  delegate handling to main frame

ELSIF message requests change of location/size THEN
  IF operation is restore THEN
    draw viewer area and border;
    send menu frm modify-msg to make extend from H 0;
    send main frm modify-msg to make extend from H 0
  ELSIF operation is modify THEN
    IF operation is extend THEN
      extend viewer area and border;
      send modify-msg to menu frm to make it extend;
      send modify-msg to main frm to make it extend
    ELSE (*reduce*)
      send modify-msg to main frm to make it reduce;
      send modify-msg to menu frm to make it reduce;
      reduce viewer area and border
    END
  ELSIF operation is suspend THEN
    send main frm modify-msg to make reduce to H 0;
    send menu frm modify-msg to make reduce to H 0
  END
END
```

In principle, the handler acts as a message dispatcher that either processes a message directly and/or delegates its processing to the descendant frames. Note that the handler's main alternative statement discriminates precisely among the 4 basic categories of messages.

From the above outlined algorithm, handling copy messages, that is, requests for generating a copy or clone of a menu viewer, we can derive a general recursive scheme for the creation of a clone of an arbitrary frame:

1. Send copy message to each element in the list of descendants;

2. Generate copy of the original frame descriptor;

3. Attach copies of descendants to the copy of descriptor.

The essential point here is the use of new outgoing messages in order to process a given incoming message. We can regard message processing as a transformation mapping incoming messages into a set of outgoing ones, with possible side-effects. The simplest one, the input message being simply passed on to descendant(s), is called *delegation*.

As a fine point we clarify that the above algorithm is designed to create a deep copy of a composite object (a menu viewer in our case). If a shallow copy would be desired, the descendants would not have to be copied, and the original descendants instead of their copies would be attached to the copy of the composite object.

Another example of message handling is provided by mouse tracking. Assume that a mouse message is received by a menu viewer while the mouse is located in the upper part of its menu frame and the left mouse key is kept down. This means "change viewer's height by moving its top line vertically". No message to express the required transformation of the sub-frames yet exists. Consequently, module `MenuViewers` takes advantage of our open (extensible) message model and simply introduces one called:

```
ModifyMsg = RECORD (Display.FrameMsg)
              id, dY, Y, H: INT
            END;
```

Field `id` specifies one of the following 2 variants:

1. *extend*, or

Requests the frame to move by the vertical translation vector `dY` and then extend to height `H` at bottom.

2. *reduce.*

   Requests the frame to reduce to height `H` at bottom and then move by `dY`.

In both cases `Y` indicates the Y-coordinate of the new lower-left corner. Fig 10 summarizes this graphically.

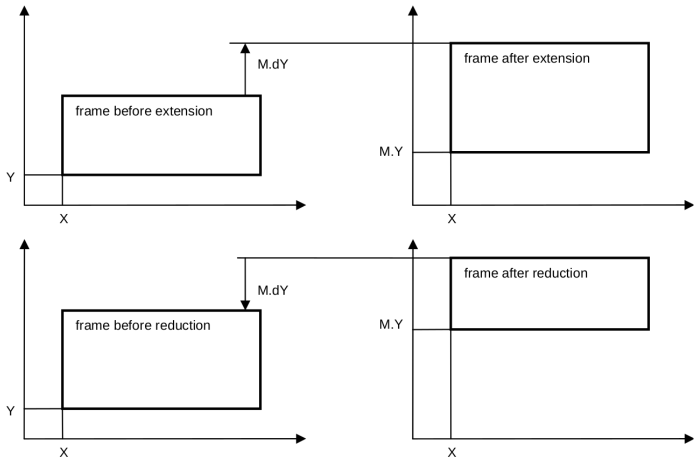


Figure 10: The modify frame operation

Messages arriving from the VMer requesting the receiving viewer to extend or reduce at its bottom are also mapped into `ModifyMsgs`. Of course, no translation, $dY = 0$.

The attentive reader might perhaps have asked why the standard handler is exported by `MenuViewers` at all. The thought behind is code reusability. For example, a message handler for a subclass of menu viewers could be implemented effectively by reusing menu viewer's standard one. After having first handled all new or differing cases it would simply (super-)call the standard handler subsequently.

4.4.3 *Cursor Management*

Traditionally, a cursor indicates and visualizes on the screen the current location of the caret in a text or, more generally, the current *focus of attention*. A small arrow or similar graphic symbol is typically used for this purpose. In Oberon, we have slightly generalized and abstracted this concept. A cursor is a path in the logical display area whose current position can be made visible by a *marker*.

The VMer and the cursor handler are 2 concurrent users of the same display area. Actually, we should imagine 2 parallel planes,

	displaying
one	viewers, and
the other	cursors.

If there's just 1 physical plane we take care of painting markers non-destructively, for example in inverse-video mode. Then, no precondition must be established before drawing a marker. However, in the case of a viewer task painting destructively in its viewer's area, the area must be locked first after turning invisible all markers in the area.

The technical support of cursor management is also contained in `Oberon`. The corresponding API:

```
DEFINITION Oberon;
  TYPE Marker = RECORD Fade,Draw: PROC(x,y: INT) END;
       Cursor = RECORD marker: Marker; X,Y: INT;
                                                on: BOOL END;
  VAR Arrow, Star   : Marker;
      Mouse, Pointer: Cursor;

  PROC OpenCursor(VAR c: Cursor);
  PROC FadeCursor(VAR c: Cursor);
  PROC DrawCursor(VAR c: Cursor;
                  VAR m: Marker; X, Y: INT);
  PROC MarkedViewer(): Viewers.Viewer;
  PROC RemoveMarks(X, Y, W, H: INT);
  ...
END Oberon.
```

The state of a cursor is given by

`on`	its mode of visibility,
`(X, Y)`	its position in the display area, and
`marker`	the current marker.

`Marker` is an abstract data type with an interface consisting of operations `Fade` and `Draw`. The main benefit of this abstraction is once more conceptual independence of the underlying hardware. For example, they can

- adapt to a given monitor hardware with built-in cursor support or, in case of absence of such support, simply
- be implemented as identical procedures (an involution) drawing the marker pattern in inverse video mode.

The functional interface to cursors consists of 3 operations:

`-Cursor`	to
`Open`	open a new cursor,
`Fade`	switch off the marker of an open cursor, and
`Draw`	extend the path of a cursor to a new position and mark it with the given marker.

We emphasize that the marker representing a given cursor can change its shape dynamically on the fly.

2 cursors are predefined:

cursor	represents	built-in marker typically
`Mouse`	the mouse	a small NW-pointing `Arrow`
`Pointer`	a global system pointer	a `Star` symbol

The pointer can be used to mark any displayed object. It serves primarily as an implicit parameter of commands.

2 assisting service procedures are added in connection with the predefined cursors:

`Marked-Viewer`	returns the viewer currently marked by the pointer, equivalent to `Viewers.This(Pointer.X, Pointer.Y)`.
`Remove-Marks`	turns invisible within a given rectangle in display area, used to lock the rectangle for its caller.

Summary the essential concept points of cursor handling:

- By virtue of the use of
  - abstract markers, and
  - the logical display area,

  any potential hardware dependence is encapsulated in system modules and is therefore hidden from the application programmer. Cursors are moving uniformly within the whole display area, even across screen boundaries.

- Cursor handling is decentralized by delegating it to the individual handlers that are installed in viewers. Typically, a handler reacts on the receipt of a mouse tracking message by drawing the mouse cursor at the indicated new position. The benefit of such individualized handling is flexibility. For example, a smart local handler might choose the shape of the visualizing marker depending on the exact location, or it might force the cursor onto a grid point.

- Even though cursor handling is decentralized, there is some intrinsic support for cursor drawing built into the `Cursor` declaration. Cursors are full value objects and, as such, can "memorize" their current state. Consequently, the interface operations `FadeCursor` and `DrawCursor` need to refer to the desired future state only.

- Looking at the VMer as one user of the display area, the cursor handler is the 2nd (and logically concurrent) user of the same resource. If there is just one physical plane implementing the display area, any region must be locked by a current user before destructive painting. Therefore, markers are usually painted non-destructively in inverse-video mode.

Let us now recapitulate the entire section.

- The central resource managed by the display subsystem is the *logical display area* whose purpose is abstraction from the underlying display monitor hardware.

  The display area is primarily used by the VMer for the accommodation of tracks and viewers, which are merely the 1st 2 levels of a potentially unlimited nested hierarchy of display frames. For example, standard menu viewers contain 2 subordinate frames:

- a menu frame, and
- a main frame of contents.

• Viewers are treated as black boxes by the VMer and are addressed via messages.

  Viewers and, more generally frames, are used as elements of *message-based interfaces* connecting the display subsystem with other subsystems like

  - the task scheduler, and
  - the various document managers.

• Finally, the display area is also the living room of cursors. In Oberon, a cursor is a marked path, 2 standard cursors `Mouse` and `Pointer` are predefined.

4.5 RASTER OPERATIONS

In **??** we introduced the display area as an abstract concept, modeled as a 2-dimensional Cartesian plane. So far, this view of the display space was sufficient because we were interested in its global structure only and ignored contents completely. However, if we are interested in what is displayed, we need to reveal more details about the model.

The Cartesian plane representing the display area is discrete. We consider points in the display area as grid points or picture elements (*pixel*), and we assume contents to be generated by assigning colors to the pixels. For the moment, the number of possible colors a pixel can attain is irrelevant. In the binary case of 2 colors we think of one representing background color and the other foreground color.

The most elementary operation generating contents in a discrete plane is "set color of pixel" or "set pixel" for short. While a few drawing algorithms directly build on this atomic operation, block-oriented functionality (traditionally called *raster operations*) plays a much more important role in practice. A *block* is a rectangular area of pixels whose bounding lines are parallel to the axes of the coordinate system.

Raster operations are based on a common principle:

- A source block of width `SW` and height `SH` is placed at a given destination point `(DX, DY)` in the display area.

- In the simplest case, the destination block `(DX, DY, SW, SH)` is plainly overwritten by the source one.

- In general, the new value of each pixel in the destination block is a combination of its old value and the value of the corresponding source pixel:

$$d := F(s, d)$$

F is sometimes called the combination mode of the raster operation. The raster is stored as an array of values of type `SET`, each set representing 32 black/white pixels. The modes of combining source and destination is implemented by the following set operations:

mode	operation
`replace`	`s`
`paint`	`s + d` (or)
`invert`	`s / d` (xor)

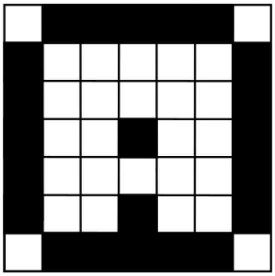
Note that `invert` is equivalent with inverse video mode if `s` is `TRUE` for all pixels. There are many different variants of raster operations. Some refer to a source block in the display area, others specify a constant pattern to be taken as source block. Some variants require replication of the source block within a given destination block `(DX, DY, DW, DH)` rather than simple placement.

The challenge when designing a raster interface is finding a unified, small and complete set of raster operations that covers all needs, in particular including the need of placing character glyphs. The amazingly compact resulting set of Oberon raster operations is exported by module `Display`:

```
DEFINITION Display;
  CONST black = 0; white = 1;         (*colors*)
        replace = 0; paint = 1; invert = 2;
                                   (*operation modes*)
  PROC Dot        (col,         x,y,     mode: INT);
  PROC ReplConst  (col,         x,y,w,h, mode: INT);
  PROC CopyBlock  (sx,sy, w,h, dx,dy,    mode: INT);
  PROC CopyPattern(col, patadr, x,y,     mode: INT);
  PROC ReplPattern(col, patadr, x,y,w,h, mode: INT);
END Display.
```

In the parameter lists of the above raster operations, `mode` is the mode of combination (`replace`, `paint`, or `invert`). `CopyBlock` copies the source block `(sx, sy, w, h)` to position `(dx, dy)` and uses `mode` to combine new contents in the destination block `(dx, dy, w, h)`. It is assumed tacitly that the numbers of colors per pixel in the source block and in the destination area are identical. It is perhaps informative to know that `CopyBlock` is essentially equivalent with the famous `BitBlt` (bit block transfer) in the SmallTalk project [**Goldberg**]. In Oberon, it is used primarily for scrolling contents within a viewer.

The remaining raster operations use a constant pattern. Patterns are implemented as arrays of bytes, and the parameter `patadr` is the address of the relevant pattern. The 1st 2 bytes indicate width `w` and height `h` of the pattern. Pattern data are given as a sequence of bytes to be placed into the destination block from left to right and from bottom to top. Each line takes an integral number of bytes. Hence, the number of data bytes is $((w + 7)/8) * h$. Fig 11 shows an example:



```
07 07 3E 49 41 49 41 41 3E
```

Figure 11: A pattern and its encoding as an array of bytes (in hex)

Some standard patterns are included in module `Display` and exported as global variables. Among them

patterns	intended to represent
arrow,	the cursor,
hook, and	the caret, and
star	the marker.

A group of predefined patterns supports drawing graphics.

- `col` in the pattern-oriented raster operations specifies the pattern's foreground color. Colors black (background) and white are predefined.

- `CopyPattern` copies the pattern to location `x`, `y` in the display area, using the given combination mode. It is probably the most frequently used operation of all because it is needed to write text.

- `ReplPattern` replicates the given pattern to the given destination block. It starts at bottom left and proceeds from left to right and from bottom to top.

- `Dot` and `ReplConst` are special cases of `CopyPattern` and `ReplPattern` respectively, taking a fixed implicit pattern consisting of a single foreground pixel.

  – `Dot` is exactly our previously mentioned "set pixel".

  – `ReplConst` is used to draw horizontal and vertical lines of various widths.

The raster operations are a prominent example of the use of Oberon's data type `SET`. Formally, variables are sets of integers between 0 and 31. Here, they are taken as sets of bits numbered from 0 to 31. We consider the replication of 1's (`mode` = `replace` or `paint`) in the rectangle with origin `x`, `y`, width `w`, and height `h`. Every line consists of 1024 pixels, or 32 words. `al`, `ar`, `a0`, `a1` are addresses.

```
VAR al, ar, a0, a1: INT;
    left, right, pixl, pixr: SET;

al := base + y*128;
ar := ((x+w-1) DIV 32)*4 + al;
al := (x DIV 32)*4 + al;
left  := {(x MOD 32) .. 31};
right := {0 .. ((x+w-1) MOD 32)};
FOR a0 := al TO al + (h-1)*128 BY 128 DO
  SYSTEM.GET(a0, pixl);
  SYSTEM.GET(ar, pixr);
  SYSTEM.PUT(a0, pixl + left);
  FOR a1 := a0+4 TO ar-4 BY 4 DO
    SYSTEM.PUT(a1, {0 .. 31});
  END
  SYSTEM.PUT(ar, pixr + right)
END
```

The definition (and even more so the implementation) of module `Display` provides support for a restricted class of possible hardware configurations only. Any number of display monitors is theoretically possible. However, they must be mapped to a regular horizontal array of predefined cells in the display area. Each cell is vertically split into 2 congruent regions, where the corresponding monitor is supposed to be able to select and display one of the 2 regions alternatively. Finally, it is assumed that all cells hosting black-and-white monitors are allocated to the left of all cells hosting color monitors. Fig 12 gives an impression of such a configuration.

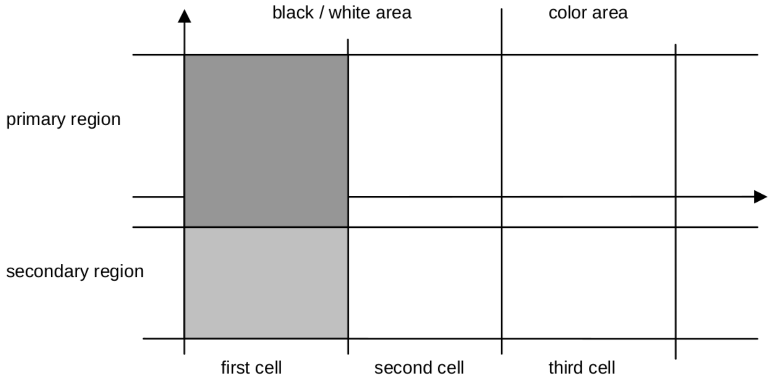


Figure 12: General, regular cell structure of display area

Under these restrictions any concrete configuration can be parameterized by the variables of the definition above. `Unit`, `Width`, and `Height` specify the extent of a displayed region, where `Width` and `Height` are width and height in pixel units, and `Unit` is the size of a pixel in units of 1/36′000 mm. This unit is a common divisor of all of the standard metric units used by the typesetting community, like mm, inch, Pica point and point size of usual printing devices. `Bottom` and `UBottom` specify the bottom y-coordinate of the primary region and the secondary region respectively. Finally, `Left` and `ColLeft` give the left x-coordinate of the area of black-and-white monitors and of color monitors respectively.

4.6 standard display configs and toolbox

Let us now take up again our earlier topic of configuring the display area. We have seen that no specific layout of the display area is distinguished by the general viewer management itself. However, some support of the familiar standard Oberon display look is provided by `Oberon`.

In the terminology of this module, a standard configuration consists of one or several horizontally adjacent displays, where a display is a pair consisting of 2 equal height tracks,

a *user track* on the left, and
a *system track* on the right.

Note that even though no reference to any physical monitor is made, a display is typically associated with a monitor in reality. This is the relevant excerpt of the definition:

```
DEFINITION Oberon;
  PROC OpenDisplay(UW, SW, H: INT);
  PROC OpenTrack(X, W: INT);
  PROC DisplayWidth (X: INT): INT;
  PROC DisplayHeight(X: INT): INT;
  PROC UserTrack    (X: INT): INT;
  PROC SystemTrack  (X: INT): INT;
  PROC AllocateUserViewer  (DX: INT; VAR X,Y: INT);
  PROC AllocateSystemViewer(DX: INT; VAR X,Y: INT);
END Oberon.
```

- `OpenDisplay` initializes and opens a new display of

parameter	dimension
`UW`	User track Width,
`SW`	System track Width,
`H`	Height.

- `OpenTrack` overlays the sequence of existing tracks spanned by the segment `[X, X + W)` by a new track.
- Both `OpenDisplay` and `OpenTrack` take from the client the burden of creating a filler viewer.
- `DisplayWidth`, `DisplayHeight`, `UserTrack` and `SystemTrack` return width or height of the respective structural entity located at position `X` in the display area.
- `AllocateUserViewer` and `AllocateSystemViewer` make proposals for the allocation of a new viewer in the desired track of the display located at `DX`.

In 1st priority, the location is determined by the system pointer that can be set manually. If the pointer is not set, a location is calculated on the basis of some heuristics whose strategies rely on different splitting fractions that are applied in the user track and in the system track respectively, with the aim of generating aesthetically satisfactory layouts.

In addition to the programming interface provided by `Oberon` for the case of standard display layouts, the display management section in the `System` toolbox provides a user interface:

```
DEFINITION System; (*Display management*)
  PROC Open;   (*viewer*)
  PROC Close;  (*viewer*)
  PROC CloseTrack;
  PROC Recall; (*most recently closed viewer*)
  PROC Copy;   (*viewer*)
  PROC Grow;   (*viewer*)
  PROC Clear;  (*system log*)
END System.
```

In turn, these commands are called to

open a text viewer in the system track,
close a viewer,
close a track,
recall (and reopen) the most recently closed viewer,
copy a viewer,
grow a viewer, and
clear the system log.

`Close`, `Copy`, `Grow`, `CloseTrack`, and `Recall` are generic commands. The 1st 3 are typically included in the title bar of a menu viewer. Their detailed implementations follow subsequently.

5 TEXT

At the beginning of the computing era, text was the only medium mediating information between users and computers. Not only was a textual notation used to denote all kinds of data and objects via names and numbers (represented by sequences of characters and digits respectively), but also for the specification of programs (based on the notions of formal language and syntax) and tasks. Actually, not even the most modern and most sophisticated computing environments have been able to make falter the dominating role of text substantially. At most, they have introduced alternative models like graphical user interfaces (GUI) as a graphical replacement for *command lines*.

There are many reasons for the popularity of text in general and in connection with computers in particular. To name but a few:

- Text containing any arbitrary amount of information can be built from a small alphabet of widely standardized elements (characters),

- their building pattern is extremely simple (lining up elements), and
- the resulting structure is most elementary (a sequence).

And perhaps most importantly, *syntactically structured text can be parsed and interpreted by a machine*.

In computing terminology, sequences of elements are called *files* and, in particular, sequences of characters are known as *text files*. Looking at their binary representation, we find text files excellently suited to be stored in computer memories and on external media. Remember that individual characters are usually encoded in 1 byte each (ASCII). We can therefore identify the binary structure of text files with sequences of bytes, matching perfectly the structure of any underlying computer storage. We should recall at this point that, with the possible exception of line-break control characters, rendering information is not part of ordinary text files. For example, the choices of character style and of paragraph formatting parameters are entirely left to the rendering interpreter.

Unfortunately, in conventional computing environments, text is merely used for input/output, and its potential is not nearly exploited optimally. Input texts are typically read from the keyboard under control of some text editor, interpreted and then discarded. Output text is volatile. Once displayed on the screen it is no longer available to any other parts of the program. The root of the problem is easily located: Conventional OSes neither feature an integrated management nor an abstract programming interface (PI) for texts.

Of course, such poor support of text on the level of programming must reflect itself on the user surface. More often than not, users are forced to retype a certain piece of text instead of simply copy/pasting it from elsewhere on the screen. Investigations have shown that, in average, up to 80% of required input text is already displayed somewhere.

Motivated by our positive experience with integrated text in the Cedar system [**Teitelman**] we decided to provide a central text management in Oberon at a sufficiently low system level. However, this is not enough. We actually need an abstract PI for text, that is, an abstract data type `Text`, together with a complete set of operations. We shall devote 5.1 to the explanation of this data type. In 5.2, we take a closer look at the basic text management, including data structures and algorithms used for the implementation of type `Text`.

Text frames are a special class of display frames. They appear typically (but not necessarily) as frames within a menu viewer (see 4.4.2). Their role is double-faced:

a) Rendering text on the display screen, and

b) interpreting interactive editing commands.

The details will be discussed in 5.3.

With the aim of exploiting the power of modern bitmap-displays and also of reusing the results of earlier projects in the field of digital font design, we decided in favor of supporting “rich texts” in Oberon, including graphical attributes and in particular font specification. In 5.4 we shall explain the font machinery, starting from an abstract level and proceeding down to the level of raster data.

5.1 TEXT AS AN ABSTRACT DATA TYPE

The concept of abstraction is arguably the most important achievement of PL development. It provides a powerful tool to create simplified views of complicated things and connections. 2 prominent examples of program abstractions

	embodying simplified views on
abstract data types	a certain kind of data, and
definitions (interfaces)	a certain piece of program.

We shall now give a precise definition of the notion of text in Oberon by presenting it as an abstract data type. It is important not to confuse this type with the far less powerful one `String` as it is often supported by advanced PLs. Here we carefully avoid revealing any implementation aspects of the abstract type `Text`. Our viewpoint is that of an application program operating on text abstractly or using it as a medium of communication.

Nevertheless, let us first use a symbolic looking glass to get a refined understanding of the concept of character in the context of rich texts. We know that each character represents a textual element of information. If displayed, it also refers to some specific graphical pattern, often called *glyph*. In Oberon, we do justice to both aspects by thinking of the ASCII as an index into a font that is into a set of glyphs of the same style. Representing characters as pairs `(font, ref)`, where `font` designates a font and `ref` the character's ASCII code and adding 2 more attributes *color* and *vertical offset*, we get to a quadruple representation `(font, ref, col, voff)` of characters. The components font, color, and vertical offset together are often referred to as *looks*. With that, we can now define a (rich) text as a *sequence of characters with looks*. We shall treat the topic of fonts and glyphs thoroughly in 5.4.

For the moment, however, let us continue our discussion of the abstract data type `Text`. Formally, we define it as

```
Text = POINTER TO TextDesc;
TextDesc = RECORD len: INT; notify: Notifier END;
```

There is only 1

- state variable `len`, and

  Represents the current length of the described text (i.e. the number of characters in the sequence).

- method `notify`.

  Occasionally called *after-method*. Notify interested clients of state changes.

By definition, each abstract data type comes with a complete set of operations. In the case of `Text`, 3 groups corresponding to 3 different topics need to be considered,

1. loading (from file), storing (to file),
2. editing, and
3. accessing (reading and writing) respectively.

5.1.1 *Loading and Storing Text*

The 1st file group introduces a pair of mutually inverse operations, *internalize* and *externalize*, respectively meaning

- load from file and build up an internal data structure, and
- serialize the internal data structure and store it in file.

There are 3 corresponding procedures:

```
PROC Open (T: Text; name: ARRAY OF CHAR);
PROC Load (T: Text; f: Files.File; pos: INT;
                               VAR len: INT);
PROC Store(T: Text; f: Files.File; pos: INT;
                               VAR len: INT);
```

Logical entities like texts are stored on external media in form of *section*, addressed by pair `(file, pos)` consisting of

`file` descriptor, and starting `pos`ition.

In general, the structure of section obeys:

```
section = identification type length contents.
```

`Open`	internalizes a named text file (consisting of a single text section)
`Load`	internalizes an arbitrary text section starting at `(f, pos)`
`Store`	externalizes a text section to `(f, pos)`
`T`	designates the internalized text
`len`	returns the length of the section

Note that in case of `Load` the identification of section must have been read and consumed before the loader is called.

5.1.2 *Editing Text*

Next group operations support text editing, comprising:

```
PROC Append     (T: Text;           B: Buffer);
PROC Insert     (T: Text; pos: INT; B: Buffer);
PROC Delete     (T: Text; beg, end: INT);
PROC ChangeLooks(T: Text; beg, end: INT; sel: SET;
                  fnt: Fonts.Font; col, voff: INT);
```

Again, we should 1st explain the types of parameters:

- `Delete` and `ChangeLooks` each take a stretch of text as an argument which, by definition, is an interval `[beg, end)` within the given text.
- In the parameter lists of `Insert` and `Append` we recognize a new data type `Buffer`.

Buffers are a facility to hold anonymous sequences of characters. `Buffer` presents itself again as an abstract data type:

```
Buffer  = POINTER TO BufDesc;
BufDesc = RECORD len: INT END;
```

`len` specifies the current length of the buffered sequence.

The following represent the intrinsic operations on buffers:

```
PROC OpenBuf(B: Buffer);
PROC Copy(SB, DB: Buffer);
PROC Save(T: Text; beg, end: INT; B: Buffer);
```

Their function, in turn:

- opening a given buffer `B`,
- copying buffer `SB` to `DB`, and
- saving a stretch `[beg, end)` of text in a given buffer, recalling the most recently deleted stretch of text and putting it into `B`.

Buffer is used as an auxiliary data type in editing.

- `Delete` d~s the given stretch `[beg, end)` within text `T`,
- `Insert` i~s the buffer's contents at `pos`ition within `T`, and
- `Append(T, B)` is a shorthand of `Insert(T, T.len, B)`.

Note that, as a side-effect of `Insert` and `Append`, the buffer involved is emptied.

Finally, `ChangeLooks` allows to change selected looks within the given stretch `[beg, end)` of `T`. `sel` is a mask selecting a subset of the looks set `{f`o`nt`, `col`or, `v`ertical `off`set`}`.

It is time now to come back to the notifier concept. Recapitulate that notify is an *after-method*. It must be installed by the client when opening the text and is called at the end of every editing operation. Its signature

```
  Notifier = PROC(T: Text; op, beg, end: INT);
```

`op`, `beg`, and `end` report

- about the `op`eration that calls the notifier, and
- on the affected stretch `[beg, end)` of the text.

There're 3 possible `op`s corresponding to different editing operations:

op	operation
`delete`	`Delete`
`insert`	`Insert(, Append)`
`replace`	`ChangeLooks`

By far the most important application of the notifier is updating the display, i.e. adjusting all affected views of the text that are currently displayed to the new state of the text (the model). We shall come back to this important matter when discussing text frames in 5.3.

In concluding this part it is worth noting that the operations just discussed have been designed to be equally useful for

- interactive text editors, as well as
- programmed text generators/manipulators.

5.1.3 *Accessing Text*

The 3rd and last group of operations on texts: Accessing, i.e. *reading* and *writing*.

According to the principle of separation of concerns, one of our guiding principles, the access mechanism operates on extra aggregates (called *readers* and *writers*) rather than on texts themselves.

Readers are used to read texts sequentially. Their type:

```
Reader = RECORD eot: BOOL; (*end of text*)
            fnt: Fonts.Font; col, voff: INT
          END;
```

A reader must 1st be opened at the desired position in the text before it can then be moved forward incrementally by reading character-by-character. Its state variables indicate end-of-text and expose the looks of the character last read. The corresponding operators are

```
PROC OpenReader(VAR R: Reader; T: Text; pos: INT);
PROC Read      (VAR R: Reader; VAR ch: CHAR);
```

- `OpenReader` sets up a reader `R` at position `pos` in text `T`.
- `Read` returns the character at the current position of `R` and makes `R` move to the next position.

The current position of `R` is returned by a call to the function:

```
PROC Pos(VAR R: Reader): INT;
```

In 3 we learned that commands plus parameter lists are often embedded in ordinary texts. When interpreting such commands, the underlying text appears as a sequence of tokens like name, number, special symbol etc. much rather than as a sequence of characters. Therefore, we have adopted the well-known concepts of syntax and scanning from the discipline of compiler construction, including functional support. The Oberon scanner recognizes tokens of some universal classes. They are name, string, integer, real, longreal, and special character. The exact syntax of universal Oberon tokens is:

```
token = name | string | integer | real | spexchar
name  = ident {.ident}
ident = letter { letter | digit }
string = '"' { char } '"'
integer = [+|-] number
real = [+|-] number . number [E [+|-] number]
number = digit { digit }
spexchar = any character except letters, digits,
                 space, tab, or carriage-return
```

Scanner is defined correspondingly as

```
Scanner = RECORD (Reader) nextCh: CHAR;
            line, class, i, len: INT;
            x: REAL; c: CHAR;
            s: ARRAY 32 OF CHAR
          END;
```

This type is actually a variant record type with class as discriminating tag. Depending on its class the value of the current token is stored in one of the fields i, x, c, or s.

- `len` gives the length of `s`,
- `nextCh` typically exposes the character terminating the current token, and
- `line` counts the number of lines scanned.

The operations on scanners:

```
PROC OpenScanner(VAR S: Scanner; T: Text; pos: INT);
PROC Scan       (VAR S: Scanner);
```

They correspond exactly to their counterparts `OpenReader` and `Read` respectively. Writers are dual to readers. They serve the purpose of creating and extending texts. However, again, they do not operate on texts directly. Rather, they act as self-contained aggregates, continuously consuming and buffering textual data.

The formal declaration of `Writer` resembles `Reader`:

```
  Writer = RECORD buf: Buffer;
             fnt: Fonts.Font; col, voff: INT
           END;
```

`buf` is an internal buffer containing the consumed data. `fnt`, `col`, and `voff` specify the current looks for the next character consumed by this writer.

The following procedures constitute the `Writer` API:

```
  PROC OpenWriter(VAR W: Writer);
  PROC SetFont   (VAR W: Writer;  fnt: Fonts.Font);
  PROC SetColor  (VAR W: Writer;  col: INT);
  PROC SetOffset (VAR W: Writer; voff: INT);
```

`OpenWriter` opens a new writer with an empty buffer. `SetFont`, `SetColor`, and `SetOffset` set the respective current look. For example, `SetFont(W, fnt)` is equivalent with `W.fnt := fnt`. These procedures are included because `fnt`, `col`, and `voff` are read-only for clients.

The question may arise how data is produced and transferred to writers. The answer is a set of writer procedures, each of them handling an individual data type:

```
PROC Write       (VAR W: Writer; ch: CHAR);
PROC WriteLn     (VAR W: Writer);
PROC WriteString (VAR W: Writer; s: ARRAY OF CHAR);
PROC WriteInt    (VAR W: Writer; x, n: INT);
PROC WriteHex    (VAR W: Writer; x: INT);
PROC WriteReal   (VAR W: Writer; x: REAL; n: INT);
PROC WriteRealFix(VAR W: Writer; x: REAL; n, k: INT);
PROC WriteClock  (VAR W: Writer; d: INT);
```

The following is schematic fragment of a client program that creates textual output:

```
  open writer; set desired font;
```

```
REPEAT
  process;
  write result to writer;
  append writer buffer to output text
UNTIL ended
```

Of course, writers can be reused. For example, a single global writer is typically shared by all of the procedures within a module. In this case, the writer needs to be opened just once at module loading time.

Typically, however, accessing aggregates are of a transient nature and are bound to a certain activity, which manifests itself in their allocation on the stack without any possibility of referencing them from the outside of the activity, in contrast to the underlying texts that are allocated on the system heap and have a much longer life time.

Let us summarize:

- `Text` in Oberon is a powerful abstract data type with intrinsic operations from 3 areas:

  1. Loading/storing,

  2. editing, and

  3. accessing (reading/writing).

- The latter 2 areas on their part introduce further abstract types `Buffer`, `Reader`, `Scanner`, and `Writer`.

  In combination they guarantee a clean separation of very different concerns. The benefits of such a rigorous decoupling are numerous. For example, it makes it possible to freely choose (and vary) the granularity at which a text and its views are updated.

- Finally, an after-method is used to allow context-dependent post-processing of editing operations. It is used primarily for preserving consistency between text models and their views.

5.2 TEXT MANAGEMENT

The art and challenge of modularization lie in finding an effective decomposition of a topic into modules with relatively thin interfaces or, in other words, into modules with a great potential for information hiding. Text systems provide a welcome opportunity of an exercise. A closer analysis immediately leads to the following separate concerns corresponding to the components Model, View and Controller of the MVC scheme: Text management, text rendering, and text editing. If we combine View and Controller and add an auxiliary font handling module Fonts, we arrive at the following three-module import hierarchy:

Module	Object type	Service
TextFrames	Frame	Text rendering and editing
Texts	Text	Text management
Fonts	Font	Font management

Note that, in contrast to the display-subsystem, the associated object types are not connected hierarchically here.

Separate Sections 5.3 and 5.4 will be devoted to modules TextFrames and Fonts respectively. In the current Section we focus on module Texts. Regarding it as a model of the abstract data type Text presented in the previous Section, its definition is congruent with the specification of the abstract data type itself, and we need not repeat it here.

The main topics of this Section are internal representation and file representation of texts. We first emphasize that the internal representation of a text is a completely private matter of module Texts that is encapsulated and hidden from clients. In particular, the representation could be changed at any time without invalidating any single client. In principle, the same is true for the file representation. However, stability is of paramount importance here because files serve the additional purposes of backing up text on external media and of porting text to other environments. Our choice of an internal representation of text was determined by a catalogue of requirements and desired properties. The wish list looks like this:

1. lean data structure
2. closed under editing operations
3. efficient editing operations
4. efficient sequential reading
5. efficient direct positioning
6. super efficient internalizing
7. preserving file representations

With the exception of 5., we found these requirements met perfectly by an adequately generalized variant of the piece list technique that was originally used for Xerox PARC's Bravo text editor and also for ETH's former document editors Dyna and Lara [Gutknecht]. The original piece list is able to describe a vanilla text without looks. It is based on two principles:

1. A text is regarded as a sequence of pieces, where a piece is a section of a text file consisting of a sequence of contiguous characters.

2. Every piece is represented by a descriptor (f, pos, len), where the components designate a file, a starting position, and a length respectively. The whole text is represented as a list of piece descriptors (in short: piece list). The editing operations operate on the piece list rather than on the pieces themselves.

If the entire life of our planet were represented by one twenty-four-hour day, intelligent life appeared within the last few seconds

Fig ?? shows a typical piece list representing (the current state of) a text. Investigating the effects of the basic editing operations delete and insert on the piece list, we end up with these algorithms:

```
delete stretch [beg, end) of text = BEGIN
  split pieces at beg and at end;
  remove p-descriptors from beg to end from the chain
```

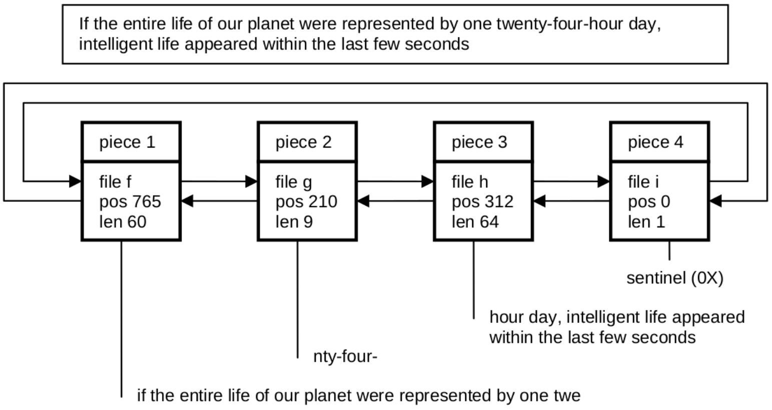

If the entire life of our planet were represented by one twenty-four-hour day, intelligent life appeared within the last few seconds
piece 1
file f
pos 765
len 60
piece 2
file g
pos 210
len 9
piece 3
file h
pos 312
len 64
piece 4
file i
pos 0
len 1
sentinel (0X)
hour day, intelligent life appeared within the last few seconds
nty-four-
if the entire life of our planet were represented by one twe

Figure 13: Piece chain representing a text

```
END


insert stretch of text at pos = BEGIN
  split piece at pos;
  insert p-descriptors representing the stretch at pos
END
```

Of course, splitting is superfluous if the desired splitting point happens to coincide with the beginning of a piece. Fig ?? and ?? show the resulting piece list after a delete and an insert operation respectively. If the entire life of our planet were represented by one day, intelligent life appeared within the last few seconds

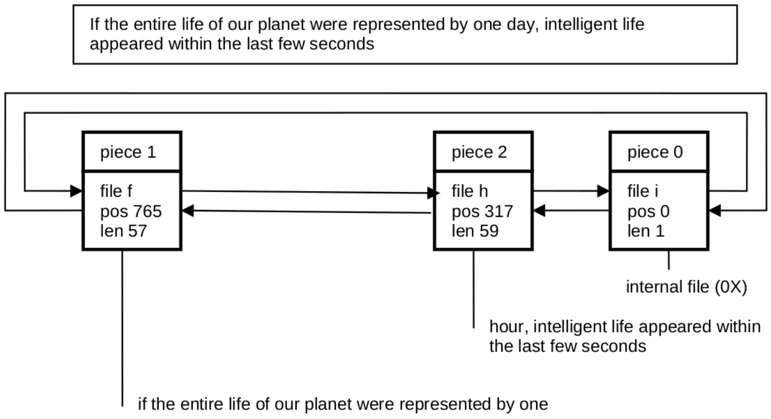

If the entire life of our planet were represented by one day, intelligent life appeared within the last few seconds
piece 1
file f
pos 765
len 57
piece 2
file h
pos 317
len 59
piece 0
file i
pos 0
len 1
internal file (0X)
hour, intelligent life appeared within the last few seconds
if the entire life of our planet were represented by one

Figure 14: Piece chain after delete operation

If the entire life of our plane, from its origin to the present moment, were represented by one day, intelligent life appeared within the last few seconds

Checking our wish list of above we immediately recognize the requirements 1.), 2.), and 3.) as met. Requirement 4.) is also met under the assumption of an efficient mechanism for direct positioning in files. Requirement 6.) can be checked off because the piece list initially consists of a single piece spanning the entire text file. Finally, requirement 7.) is met simply because the operations do not affect file representations at all.

In Oberon we adapted the piece list technique to texts with looks ("rich texts"). Formally, we first define a run as a stretch of text whose characters show identical looks. Now, we require the piece list to subordinate itself to the run

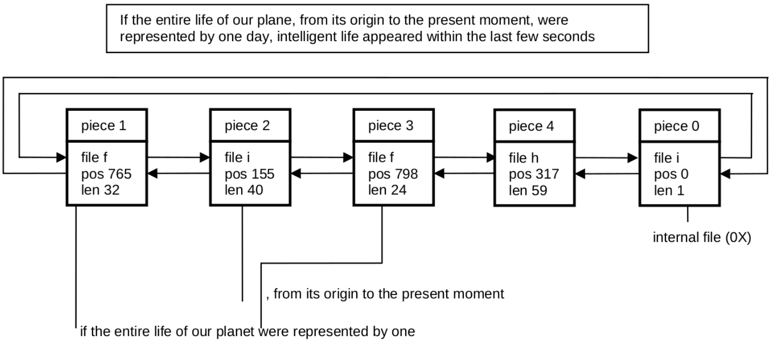
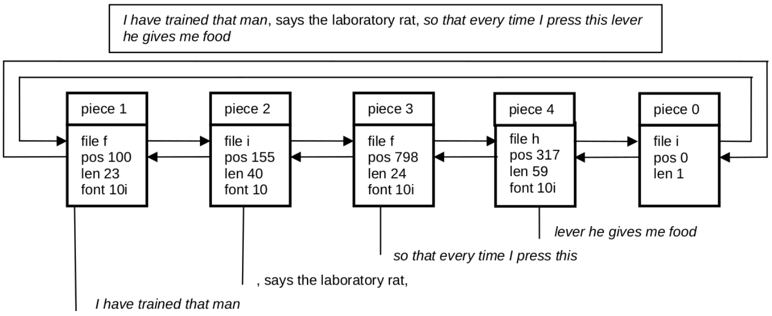

If the entire life of our plane, from its origin to the present moment, were represented by one day, intelligent life appeared within the last few seconds
piece 1
file f
pos 765
len 32
piece 2
file i
pos 155
len 40
piece 3
file f
pos 798
len 24
piece 4
file h
pos 317
len 59
piece 0
file i
pos 0
len 1
internal file (0X)
, from its origin to the present moment
if the entire life of our planet were represented by one

Figure 15: Piece chain after insert operation

structure. This obviously means that every piece needs to be contained within a single run. Figure 5.4 visualizes such a compliant piece list representing a text with varying looks. There are only two new aspects compared to the original version of the piece list discussed above: An additional operation to change looks and the initial state of the piece chain.

```
change looks in a stretch [beg, end) of text = BEGIN
split pieces at beg and at end;
change looks in piece descriptors from beg to end in the chain
END
```

This shows that requirements 2.) and 3.) in the wish list are still satisfied. I have trained that man, says the laboratory rat, so that every time I press this lever he gives me food



I have trained that man, says the laboratory rat, *so that every time I press this lever he gives me food*
piece 1
file f
pos 100
len 23
font 10i
piece 2
file i
pos 155
len 40
font 10
piece 3
file f
pos 798
len 24
font 10i
piece 4
file h
pos 317
len 59
font 10i
piece 0
file i
pos 0
len 1
lever he gives me food
so that every time I press this
, says the laboratory rat,
I have trained that man

Figure 16: Generalized piece chain representing a text with looks

Initially, the pieces are identical with runs, and the number of elements in the piece list is equal to the number of runs. Because this number is typically small in comparison with the total number of characters in a text requirement 6.) is still met.

We conclude that the new aspects do not invalidate the positive rating given above to the piece technique with regard to requirements 1.), 2.), 3.), 4.), 6.), and 7.) in our wish list. However, the requirement of efficient direct positioning remains. The problem is the necessity to scan through the piece list sequentially in order to locate the piece that contains the desired position. We investigated different solutions of this efficiency problem. They are based on different data structures connecting the piece descriptors, among them a piece tree and a variant of the piece list featuring an additional long-distance link like in a skip-list.

Eventually, we decided in favor of a simpler solution that we can easily justify by pointing out that the typical editing scenario is zooming into a local region of text, i.e. positioning at an arbitrary location once and subsequently positioning at locations in its immediate neighborhood many times. Therefore, an appropriate solution is caching the most recently calculated values (pos, piece) of the translation map. Of course, this does not solve the problem of cache misses. Notice, however, that this problem is acute only in the case of extremely long piece lists that do not occur in ordinary texts and editing sessions.

We shall now illustrate the piece technique in detail at the example of two important but basic operations: Insert and read. Let us start with an overview of the data types involved. Apart from some auxiliary private variables marked with an arrow, the types Text, Buffer, and Reader are already familiar to us from the previous Section. Type Piece is completely private and hidden from the clients.

```
Text = POINTER TO TextDesc;
Notifier = PROC (T: Text; op, beg, end: INT);

TextDesc = RECORD
len: INT;
notify: Notifier;
--> trailer: Piece;
--> org: INT;
--> pce: Piece
END;

Buffer = POINTER TO BufDesc;
BufDesc = RECORD
len: INT;
--> header, last: Piece
END;

Reader = RECORD
eot: BOOL;
fnt: Fonts.Font;
col, voff: INT;
--> ref: Piece;
--> org, off: INT;
rider: Files.Rider
END;

--> Piece = POINTER TO PieceDesc;
--> PieceDesc = RECORD
f: Files.File;
off, len: INT;
fnt: Fonts.Font;
col, voff: INT;
prev, next: Piece
END;
```

As depicted in Figure 5.1, the piece list is implemented as a doubly linked list with a sentinel piece closing it to a ring. The field trailer in type TextDesc points to the sentinel piece. Fields org and pce implement a translation cache consisting of merely one entry (org, pce). It links a position org with a piece pce. The fields header and last in type Buffer refer to the implementation of buffers as piece lists. They point to the first and last piece descriptors respectively. Finally, the fields ref, org, and off in type Reader memorize the current piece, its origin, and the current offset within this piece. The fields f, off, and len in type Piece specify the underlying file, starting position in the file, and length of the piece. fnt, col, and voff are its looks. Finally prev and next are pointers to the previous piece and to the next piece in the list respectively.

FindPiece and SplitPiece are auxiliary procedures that are used by almost all piece-oriented operations.

```
PROC FindPiece (T: Text; pos: INT; VAR org: INT; VAR p: Piece);
VAR p: Piece; porg: INT;
BEGIN p := T.pce; porg := T.org;
1) IF pos >= porg THEN
WHILE pos >= porg + p.len DO INC(porg, p.len); p := p.next END
2) ELSE p := p.prev; DEC(porg, p.len);
WHILE p < porg DO p := p.prev; DEC(porg, p.len) END
END;
3) T.pce := p; R.org := porg; (*update cache*)
```

```
pce := p; org := porg
END FindPiece;
```

Explanations (referring to the line numbers in the above code excerpt) 1) search to the right (next) 2) search to the left (prev) 3) update cache if more than 50 pieces traversed

```
1) PROC SplitPiece (p: Piece; off: INT; VAR pr: Piece);
VAR q: Piece;
BEGIN
2) IF off > 0 THEN NEW(q);
q.fnt := p.fnt; q.col := p.col; q.voff := p.voff;
q.len := p.len - off;
q.f := p.f; q.off := p.off + off;
p.len := off;
q.next := p.next; p.next := q;
q.prev := p; q.next.prev := q;
pr := q
ELSE pr := p
END
END SplitPiece;

3)
4)
```

Explanations: 1) 2) 3) 4)

return right part piece pr after split generate new piece only if remaining length > 0 insert new piece in forward chain insert new piece in backward chain

Procedure Insert handles text insertion. It operates on a buffer that contains the stretch of text to be inserted:

```
PROC Insert (T: Text; pos: INT; B: Buffer);
VAR pl, pr, p, qb, qe: Piece; org, end: INT;
BEGIN
1) FindPiece(T, pos, org, p); SplitPiece(p, pos - org, pr);
2) IF T.org >= org THEN
T.org := org - p.prev.len; T.pce := p.prev
END;
pl := pr.prev; qb := B.header.next;
3) IF (qb # NIL) & (qb.f = pl.f) & (qb.off = pl.off + pl.len)
& (qb.fnt = pl.fnt) & (qb.col = pl.col) & (qb.voff = pl.voff) THEN
pl.len := pl.len + qb.len; qb := qb.next
END;
IF qb # NIL THEN qe := B.last;
4)
qb.prev := pl; pl.next := qb; qe.next := pr; pr.prev := qe
END;
5) T.len := T.len + B.len; end := pos + B.len;
6) B.last := B.header; B.last.next := NIL; B.len := 0;
7) T.notify(T, insert, pos, end)
END Insert;
```

Explanations: 1) 2) 3) 4) 5) 6) 7)

split piece to isolate point of insertion adjust cache if necessary merge pieces if possible insert buffer update text length empty buffer notify

Procedure Read implements sequential reading of characters in texts. It operates on a text reader:

```
PROC Read (VAR R: Reader; VAR ch: CHAR);
BEGIN
1) Files.Read(R.rider, ch); R.fnt := R.ref.fnt; R.col := R.
INC(R.off);
2) IF R.off = R.ref.len THEN
3) IF R.ref.f = WFile THEN R.eot := TRUE END;
R.org := R.org + R.off; R.off := 0;
4) R.ref := R.ref.next; R.org := R.org + R.off; R.off := 0;
5) Files.Set(R.rider, R.ref.f, R.ref.off)
END
END Read;
```

Explanations: 1) 2) 3) 4) 5)

read character from file and update looks in reader if piece boundary reached check if sentinel piece reached move reader to next piece position file rider

Procedure Read is typically used as a primitive by text scanners and in particular by the built-in scanner Scan for the recognition of universal tokens, as they were defined in the previous section. Scanning is a rather complex operation that, for example, includes the conversion of a sequence of digits into an internal floating-point representation and vice-versa. Scanning a real number involves recognizing m and d, and computing x = m*10d. This is done using procedure Ten(d) computing 10d by repeated multiplication maintaining the invariant t * pn = 10n0, where n0 is the initial value of n.

```
PROC Ten(n: INT): REAL;
VAR t, p: REAL;
BEGIN t := 1.0; p := 10.0;
WHILE n > 0 DO
IF ODD(n) THEN t := p * t END ;
p := p*p; n := n DIV 2
END ;
RETURN t
END Ten;
```

Writing a real number in decimal form is more complicated. It involves the computation of m and d from x = n*2e so that x = m*10d with `1.0 \leq m < 10.0`. First, e is obtained with the standard function UNPK(x, e), then d is computed (from the relationship 10k = 2k*log(10)) as d = e/log2(10). In order to avoid a real division for obtaining d, we use the approximation 1.0 / log2(10) = 77 DIV 256, and then compute x := x / Ten(e) or x := x * Ten(-e). Further details are to be taken from the listings of WriteReal and WriteRealFix.

In spite of its apparent simplicity the piece list technique interoperates with other system components in quite a subtle way. For example, after a while of editing, there are typically numerous cross references between the documents involved. In other words, pieces of one document may point to foreign files that is to files that were originally associated with other documents. As a consequence, the FS must either employ some smart garbage collection algorithm or not recycle file pages at all, even if a new version of a file of the same name has been created in the meantime.

A problem of another kind, again affecting the FS, arises if, say, a single text line is composed of several small pieces. Then, reading this line sequentially may necessitate several jumps to different positions in different files at a high pace. Depending on the quality of the file buffering mechanism, this may lead to significantly hesitant mouse tracking.

And finally, typed characters that are supposed to be inserted into a text need to be stored on the so-called keyboard file. For this (continuously growing) file, several readers and one writer must be allowed to coexist concurrently.

As a consequence, the following qualities of the underlying FS are mandatory for the piece technique to work properly:

1. Once a file page is allocated it must not be reused (until system restart).
2. A versatile file buffering mechanism supporting multiple buffers per file is required.
3. Files must be allowed to be open in read mode and in write mode simultaneously.

The format of text sections in files obeys a set of syntactical rules (productions) that can easily be specified in EBNF-notation:

```
TextSection = ident header {char}.
header = type offset run {run} null length.
run = font [name] color offset length.
```

In the TextSection production ident is an identifier for text blocks. In the header production type is a type-discriminator, offset is the offset to the character part, run is a run-descriptor, null is a nullcharacter, and length is the length of the character sequence. In the run production font, color, and offset are specifications of looks, and length is the run-length. In order to save space, font names are coded as ordinal numbers within a text section. If and only if a font appears for the first time in a text block it is followed by the actual font name.

Let us conclude this Section with two side-remarks and a summary. Remarks: For compatibility reasons, plain ASCII-files are accepted as text files as well. They are mapped to texts consisting of a single run with standard looks.

Internalizing a text section from a file is extremely efficient because it is obviously sufficient to read the header and translate it into the initial state of the piece list.

Summary: The mechanism used for the implementation of the abstract data type Text is completely hidden from clients. It is a generalized version of the original piece list technique, adapted to texts with looks. The piece list technique in turn is based on the principle of indirection: Operations operate on descriptors of texts rather than on texts themselves. The benefits are efficiency and non-destructive operations. However, the technique works properly only in combination with an efficient (and reliable) garbage collector and a suitable FS.

5.3 TEXT FRAMES

The tasks of text frames are text rendering and user interaction. A text frame represents a text view and a controller in the form of an interactive text editor. Technically, text frames are a subclass of display frames and, as such, are objects with an open message interface of the kind explained in Chapter 4.

The geometric layout of text frames is determined by two areas: A rectangle of contents and a vertical scroll-bar along the left borderline. The type of text frames is a direct extension of type Display.Frame:

```
Frame = POINTER TO FrameDesc;
FrameDesc = RECORD (Display.FrameDesc)
text: Texts.Text;
org, col, lsp: INT;
left, right, top, bot: INT;
markH, time: INT;
hasCar, hasSel, hasMark: BOOL;
carloc: Location;
selbeg, selend: Location
END;
```

Fields text and org specify the text part to be displayed, the former referring to the underlying text and the latter designating the starting position of the displayed part. Fields col and lsp are rendering parameters. They specify the frame's background color and the line spacing. Fields left, right, top, and bot are margins. They determine the rectangle of contents. mark indicates whether there is a position marker, which is a small horizontal bar indicating the position of the displayed part relative to the whole text. markH represents its location within the text frame.

Caret and selection are two important features associated with a text frame. The caret indicates a focus, and it serves as an implicit "point of insertion" for placing consumed characters (for example from the keyboard). The selection is a stretch of displayed text. Additionally it serves as a parameter for various operations and commands, among them delete and change looks. The state and location of the caret is given by the variables car and carloc respectively. Analogously, the state of the selection and its begin and end are reflected by the fields sel, selbeg, and selend in the frame descriptor. Field time is a time stamp on the current selection.

In principle, caret and selection could be regarded as ingredients of the underlying text (the model) equally well. However, we deliberately decided to associate these features with frames (views) in order to get increased flexibility. For example, two different selections in adjacent viewers displaying the same text are normally interpreted as one extensive selection across their span. The auxiliary type

Location summarizes information about a location in a text frame. Its definition is:

```
Location = RECORD
org, pos, dx, x, y: INT
END;
```

x, y specify the envisioned location relative to the text frame's origin, and dx is the width of the character at this location. pos is the corresponding position in the text and org is the origin position of the corresponding text line.

The following is a simplified version of the message handler employed by text frames. It fully determines the behavior and capabilities of text frames.

```
1)
2)

3)
4)
5)
7)
8)
9)

PROC Handle* (F: Display.Frame; VAR M: Display.FrameMsg);
```

```
VAR F1: Frame; buf: Texts.Buffer;
BEGIN
CASE M OF
Oberon.InputMsg:
IF M.id = Oberon.track THEN Edit(F(Frame), M.X, M.Y, M.keys
ELSIF M.id = Oberon.consume THEN
IF F(Frame).hasCar THEN Write(F(Frame), M.ch, M.fnt, M.col,
END |
Oberon.ControlMsg:
IF M.id = Oberon.defocus THEN Defocus(F(Frame))
ELSIF M.id = Oberon.neutralize THEN Neutralize(F(Frame))
END |
Oberon.SelectionMsg:
GetSelection(F(Frame), M.text, M.beg, M.end, M.time) |
Oberon.CopyMsg: Copy(F(Frame), F1); M.F := F1 |
MenuViewers.ModifyMsg: Modify(F(Frame), M.id, M.dY, M.Y, M.
CopyOverMsg: CopyOver(F(Frame), M.text, M.beg, M.end) |
UpdateMsg: IF F(Frame).text = M.text THEN Update(F(Frame),
END
END Handle;
```

Explanations: 1) 2) 3) 4) 5) 6) 7) 8) 9)

Mouse tracking message: Call built-in editor immediately Consume message: In case of valid caret insert character Defocus message: Remove caret Neutralize message: Remove caret and selection Selection message: Return current selection with time stamp Copyover message: Copy given stretch of text to caret Copy message: Create a copy (clone) Modify message: Translate and change size Update message: If text was changed then update display

We recognize again our categories of universal messages introduced in Chapter 4, Table 4.6: Messages in lines 1) and 2) report about user interactions. Messages in 3), 4), 5), 6), and 7) specify generic operations. Messages in 8) require a change of location or size. Messages of the latter kind arrive from the ancestor menu viewer via delegation. They are generated by the interaction handler and preprocessed by the original viewer message handler. Finally, messages in line 9) report about changes of contents.

The text frame handler is encapsulated in a module called TextFrames. This module exports the above introduced types Frame (text frame) and Location, as well as the procedure Handle. Furthermore, it exports type UpdateMsg to report on changes made to a displayable text.

```
UpdateMsg = RECORD (Display.FrameMsg)
id: INT;
text: Texts.Text;
beg, end: INT
END;
```

Field id names one of the operators replace, insert, or delete. The remaining fields text, beg, and end restrict the change to a range. Additional procedures generate a new standard menu text frame and contents text frame respectively:

```
PROC NewMenu (name, commands: ARRAY OF CHAR): Frame;
PROC NewText (text: Texts.Text; pos: INT): Frame;
```

This completes the minimum definition of module TextFrames. In addition, this module exports a set of useful service procedures supporting the composition of custom handlers from elements of the standard handler:

```
PROC Edit (F: Frame; X, Y: INT; Keys: SET);
PROC Write (F: Frame; ch: CHAR; fnt: Fonts.Font; col, voff:
PROC Defocus (F: Frame);
PROC Neutralize (F: Frame);
PROC GetSelection (F: Frame; VAR text: Texts.Text;
VAR beg, end, time: INT);
PROC CopyOver (F: Frame; text: Texts.Text; beg, end: INT);
PROC Copy (F: Frame; VAR F1: Frame);
PROC Modify (F: Frame; id, dY, Y, H: INT);
PROC Update (F: Frame; VAR M: UpdateMsg);
```

The module also supports mouse tracking inside text frames:

```
PROC TrackCaret (F: Frame; X, Y: INT; VAR keysum: SET);
PROC TrackSelection (F: Frame; X, Y: INT; VAR keysum: SET);
PROC TrackLine (F: Frame; X, Y: INT; VAR org: INT; VAR keys
PROC TrackWord (F: Frame; X, Y: INT; VAR pos: INT; VAR keys
```

Let us now take a closer look at the implementation of some selected operations. For this purpose, we must first explain the notion of line descriptor that is used to optimize the operation of locating positions within text frames.

```
Line = POINTER TO LineDesc;
LineDesc = RECORD
len, wid: INT;
eot: BOOL;
next: Line
END;
```

Each line descriptor provides detailed information about a single line of text that is currently displayed: len is the number of characters on the line, wid is the line width, eot indicates terminating line, and next points to the next line descriptor.

Text frames maintain a private data structure called line list that describes the list of text lines displayed:

```
Frame = POINTER TO FrameDesc;
FrameDesc = RECORD (Display.FrameDesc)
text: Texts.Text;
org, col, lsp: INT;
left, right, top, bot: INT;
markH, time: INT;
hasChar, hasSel, hasMark: BOOL;
```

```
carloc, selbeg, selend: Location;
--> trailer: Line
END;
```

Field trailer represents a sentinel element that closes the line list to a ring. The line list contains useful summary information about the current contents of the text frame. It can be used beneficially by some related data types, for example by type Location that was introduced earlier:

```
Location = RECORD
org, pos, dx, x, y: INT;
--> lin: Line
END;
```

The built-in editor procedure Edit is a worthwhile part to look at in more detail. It is called by the task scheduler to handle mouse events within a text frame. The following code excerpt shows nicely how the different components of the text system interoperate.

```
PROC Edit* (F: Frame; X, Y: INT; Keys: SET);
VAR M: CopyOverMsg;
text: Texts.Text;
buf: Texts.Buffer;
v: Viewers.Viewer;
loc0, loc1: Location;
```

```
beg, end, time, pos: INT;
keysum: SET;
fnt: Fonts.Font;
col, voff: INT;
BEGIN
IF X < F.X + Min(F.left, barW) THEN (*cursor is in scroll bar*)
Oberon.DrawMouse(ScrollMarker, X, Y); keysum := Keys;
IF Keys = {2} THEN (*ML, scroll up*)
TrackLine(F, X, Y, pos, keysum);
IF (pos >= 0) & (keysum = {2}) THEN
RemoveMarks(F); Oberon.RemoveMarks(F.X, F.Y, F.W, F.H);
Show(F, pos)
END
ELSIF Keys = {1} THEN (*MM*) keysum := Keys;
REPEAT Input.Mouse(Keys, X, Y); keysum := keysum + Keys;
Oberon.DrawMouse(ScrollMarker, X, Y)
UNTIL Keys = {};
IF ~(keysum = {0, 1, 2}) THEN
IF 0 IN keysum THEN pos := 0
ELSIF 2 IN keysum THEN pos := F.text.len - 100
ELSE pos := (F.Y + F.H - Y) * (F.text.len) DIV F.H
END ;
RemoveMarks(F); Oberon.RemoveMarks(F.X, F.Y, F.W, F.H);
Show(F, pos)
END
ELSIF Keys = {0} THEN (*MR, scroll down*)
TrackLine(F, X, Y, pos, keysum);
IF keysum = {0} THEN
```

```
LocateLine(F, Y, loc0); LocateLine(F, F.Y, loc1);
pos := F.org - loc1.org + loc0.org;
IF pos < 0 THEN pos := 0 END ;
RemoveMarks(F); Oberon.RemoveMarks(F.X, F.Y, F.W, F.H);
Show(F, pos)
END
END
ELSE (*cursor is in text area*)
Oberon.DrawMouseArrow(X, Y);
IF 0 IN Keys THEN (*MR: select*)
TrackSelection(F, X, Y, keysum);
IF F.hasSel THEN
IF keysum = {0, 2} THEN (*MR, ML: delete text*)
Oberon.GetSelection(text, beg, end, time);
Texts.Delete(text, beg, end, TBuf);
Oberon.PassFocus(Viewers.This(F.X, F.Y)); SetCaret(F, beg)
ELSIF keysum = {0, 1} THEN (*MR, MM: copy to caret*)
Oberon.GetSelection(text, beg, end, time);
M.text := text; M.beg := beg; M.end := end;
Oberon.FocusViewer.handle(Oberon.FocusViewer, M)
END
END
ELSIF 1 IN Keys THEN (*MM: call*)
TrackWord(F, X, Y, pos, keysum);
IF (pos >= 0) & ~(0 IN keysum) THEN Call(F, pos, 2 IN keysu
ELSIF 2 IN Keys THEN (*ML: set caret*)
Oberon.PassFocus(Viewers.This(F.X, F.Y));
TrackCaret(F, X, Y, keysum);
```

```
IF keysum = {2, 1} THEN (*ML, MM: copy from selection to caret*)
Oberon.GetSelection(text, beg, end, time);
IF time >= 0 THEN
NEW(TBuf); Texts.OpenBuf(TBuf);
Texts.Save(text, beg, end, TBuf); Texts.Insert(F.text, F.carloc.pos
SetSelection(F, F.carloc.pos, F.carloc.pos + (end - beg));
SetCaret(F, F.carloc.pos + (end - beg))
ELSIF TBuf # NIL THEN
NEW(buf); Texts.OpenBuf(buf);
Texts.Copy(TBuf, buf); Texts.Insert(F.text, F.carloc.pos, buf);
SetCaret(F, F.carloc.pos + buf.len)
END
ELSIF keysum = {2, 0} THEN (*ML, MR: copy looks*)
Oberon.GetSelection(text, beg, end, time);
IF time >= 0 THEN
Texts.Attributes(F.text, F.carloc.pos, fnt, col, voff);
IF fnt # NIL THEN Texts.ChangeLooks(text, beg, end, {0,1,2}, fnt, c
END
END
END
END
END Edit;
```

We see that the editing operation is determined by the first key pressed (primary key) and can then be varied by “interclicking” that is, by clicking a secondary key while holding down the primary key. As a convention, (inter)clicking all keys means cancelling the operation. Mouse clicks and subsequent actions can now be summarized as follows: 1. In the scroll bar primary key

secondary key

action

ML MM MM MM

ML MR

scroll designated line to the top scroll proportional to mouse position scroll to the end of the text scroll to the beginning of the text

primary key

secondary key

action

ML ML ML MM MR MR MR

MM MT ML MM

set caret copy selection to caret copy looks call selected procedure select delete selection copy selection to caret

2. In the text area In the text area the keys are interpreted according to their generic semantics: left key = middle key = right key =

point key execute key select key

Let us "zoom into" one of the editing operations, for example into TrackCaret.

```
PROC TrackCaret (F: Frame; X, Y: INT; VAR keysum: SET);
VAR loc: Location; keys: SET;
BEGIN
1) IF F.trailer.next # F.trailer THEN
2)
LocateChar(F, X - F.X, Y - F.Y, F.carloc);
3)
FlipCaret(F);
4)
keysum := {};
REPEAT
Input.Mouse(keys, X, Y); keysum := keysum + keys;
Oberon.DrawMouseArrow(X, Y);
LocateChar(F, X - F.X, Y - F.Y, loc);
IF loc.pos # F.carloc.pos THEN FlipCaret(F); F.carloc := loc; FlipCa
5)
UNTIL keys = {};
6)
F.hascar := TRUE
END
END TrackCaret;
```

Explanations: 1) guard guarantees non-empty line list 2) locates the character pointed at 3) drags caret to new location 4) - 5) tracks mouse and drags caret accordingly 6) set caret state TrackCaret makes use of two auxiliary procedures FlipCaret and LocateChar. FlipCaret is used to turn off or on the pattern of the caret. LocateChar is an important operation that is used to locate the character at a given Cartesian position (x, y) within the frame.

```
PROC FlipCaret (F: Frame);
BEGIN
1) IF F.carloc.x < F.W THEN
2)
IF (F.carloc.y >= 10) & (F.carloc.x + 12 < F.W) THEN
3)
Display.CopyPattern(Display.white, Display.hook,
F.X + F.carloc.x, F.Y + F.carloc.y - 8, Display.invert)
END
END
END FlipCaret;
```

Explanations: 1) - 2) if there is room for drawing the caret 3) copy standard hook-shaped pattern to caret location in inverse video mode

```
PROC LocateChar (F: Frame; x, y: INT; VAR loc: Location);
VAR R: Texts.Reader;
patadr, pos, lim: INT;
```

```
ox, dx, u, v, w, h: INT;
1) BEGIN LocateLine(F, y, loc);
2) lim := loc.org + loc.lin.len - 1;
3) pos := loc.org; ox := F.left; dx := eolW;
4) Texts.OpenReader(R, F.text, loc.org);
5) WHILE pos # lim DO
6)
Texts.Read(R, nextCh);
7)
Fonts.GetPat(R.fnt.raster, nextCh, dx, u, v, w, h, patadr);
IF ox + dx <= x THEN
INC(pos); ox := ox + dx;
IF pos = lim THEN dx := eolW END
ELSE lim := pos
END
END ;
8) loc.pos := pos; loc.dx := dx; loc.x := ox
END LocateChar;
```

Explanations: 1) locate text line corresponding to at y 2) set limit to the last actual character on this line 3) start locating loop with first character on this line 4) setup reader and read first character of this line 5) - 7) scan through characters of this line until limit or x is reached 6) get character width dx of current character 8) return location found Note that the need to read characters from the text (again) in LocateChar has its roots in the socalled proportional fonts in which our rich texts are represented. We found that keeping character widths is an unnecessary optimization thanks to the buffering capabilities of the underlying file system. In the case of fixed-pitch fonts a simple division by the character width would be sufficient, of course. Finally, procedure LocateLine uses the line list to locate the desired text line without reading text at all.

```
PROC LocateLine (F: Frame; y: INT; VAR loc: Location);
VAR L: Line; org, cury: INT;
BEGIN org := F.org;
1) org := F.org; L := F.trailer.next; cury := F.H - F.top -
2) WHILE (L.next # F.trailer) & (cury > y + dsr) DO
org := org + L.len; L := L.next; cury := cury - lsp
3) END;
4) loc.org := org; loc.lin := L; loc.y := cury
END LocateLine;
```

Explanations: 1) start with first line in the frame 2) - 3) traverse line chain until last line or y is reached 4) return found line After text editing text, rendering is our next topic. Let us pursue the case in that a user pressed the point-key and then interclicked the middle key, corresponding to line 56) in procedure Edit. Remember that notifier is called at the end of every editing operation and in particular at the end of Texts.Insert. In case of standard text frames, the notifier simply broadcasts an update message into the display space:

```
PROC NotifyDisplay (T: Texts.Text; op, beg, end: INT);
```

```
VAR M: UpdateMsg;
BEGIN M.id := op; M.text := T; M.beg := beg; M.end := end; Viewers.
END NotifyDisplay;
```

Let us now take the perspective of a text frame receiving an update message. Looking at line 9) in the text frame handler, we see that procedure Update is called, which in turn calls procedure Insert in TextFrames:

```
PROC Insert (F: Frame; beg, end: INT);
VAR R: Texts.Reader; L, L0, l: Line;
org, len, curY, botY, Y0, Y1, Y2, dY, wid: INT;
BEGIN
IF beg < F.org THEN F.org := F.org + (end - beg)
ELSE
1)
org := F.org; L := F.trailer.next; curY := F.Y + F.H - F.top - asr;
WHILE (L # F.trailer) & (org + L.len <= beg) DO
org := org + L.len; L := L.next; curY := curY - lsp
2)
END;
3)
IF L # F.trailer THEN
botY := F.Y + F.bot + dsr;
4)
Texts.OpenReader(R, F.text, org); Texts.Read(R, nextCh);
5)
len := beg - org; wid := Width(R, len);
```

```
6)
ReplConst (F.col, F, F.X + F.left + wid, curY - dsr, L.wid
7)
DisplayLine(F, L, R, F.X + F.left + wid, curY, len);
8)
org := org + L.len; curY := curY - lsp;
Y0 := curY; L0 := L.next;
WHILE (org <= end) & (curY >= botY) DO
NEW(l);
Display.ReplConst(F.col, F.X + F.left, curY - dsr, F.W - F.
DisplayLine(F, l, R, F.X + F.left, curY, 0);
L.next := l; L := l;
org := org + L.len; curY := curY - lsp
9)
END;
10)
IF L0 # L.next THEN Y1 := curY;
11)
L.next := L0;
WHILE (L.next # F.trailer) & (curY >= botY) DO
L := L.next; curY := curY - lsp
12)
END;
L.next := F.trailer;
dY := Y0 - Y1;
IF Y1 > curY + dY THEN
13)
Display.CopyBlock
```

```
(F.X + F.left, curY + dY + lsp - dsr, F.W - F.left, Y1 - curY - dY,
F.X + F.left, curY + lsp - dsr,
0);
Y2 := Y1 - dY
ELSE Y2 := curY
END;
14)
curY := Y1; L := L0;
WHILE curY # Y2 DO
Display.ReplConst(F.col, F.X + F.left, curY - dsr, F.W - F.left, ls
DisplayLine(F, L, R, F.X + F.left, curY, 0);
L := L.next; curY := curY - lsp
15)
END
END
END
END;
16) UpdateMark(F)
END Insert;
```

Some explanations: 1) - 2) search line where inserted part starts 3) if it is displayed in this viewer 4) setup reader on this line 5) get width of unaffected part of line (avoid touching it) 6) clear remaining part of line 7) display new remaining part of line 8) - 9) display newly inserted text lines 10) if it was not a one line update 11) - 12) skip overwritten text lines 13) use fast block move to adjust reusable lines 14)

- 15) redisplay previously overwritten text lines 16) adjust position marker

Special care needs to be exercised in the implementation to avoid "flickering" and to minimize processing time. Concretely, the following measures are taken for this purpose: 1.) Avoid writing the same data again. 2.) Keep the number of newly rendered text lines at a minimum. 3.) Use raster operations (block moves) to adjust reusable displayed lines.

Of course, the rules governing the rendering and formatting process crucially influence the complexity of procedures like Insert. For text frames we have consciously chosen the simplest possible set of formatting rules. They can be summarized as: 1.) For a given text frame the distance between lines is constant. 2.) There are no implicit line breaks.

It is exactly this set of rules that makes it possible to display a text line in one pass. Two passes are inevitable if line distances have to adjust to font sizes or if lines must be broken implicitly. Updating algorithms make use of the following one-pass rendering procedures Width and DisplayLine:

```
PROC Width (VAR R: Texts.Reader; len: INT): INT;
VAR patadr, pos, ox, dx, x, y, w, h: INT;
1) BEGIN pos := 0; ox := 0;
WHILE pos < len DO
Fonts.GetPat(R.fnt, nextCh, dx, x, y, w, h, pat);
```

```
ox := ox + dx; INC(pos); Texts.Read(R, nextCh)
2) END;
3) RETURN ox
END Width;
```

Explanations: 1) - 2) scan through len characters of this line 3) return accumulated width Procedures Width and LocateChar are similar. Therefore the above comment about relying on the buffering capabilities of the underlying FS applies to procedure Width equally well.

```
PROC DisplayLine (F: Frame; L: Line;
VAR R: Texts.Reader; X, Y, len : INT);
VAR patadr; NX, Xlim, dx, x, y, w, h: INT;
1) BEGIN NX := F.X + F.W; Xlim := NX - 40;
2) WHILE (nextCh # CR) & ((nextCh > " ") OR (X < Xlim)) & (R.fnt #
3)
Fonts.GetPat(R.fnt, nextCh, dx, x, y, w, h, patadr);
4)
IF (X + x + w <= NX) & (h # 0) THEN
5)
Display.CopyPattern(R.col, patadr, X + x, Y + y, Display.invert)
6)
END;
7)
X := X + dx; INC(len); Texts.Read(R, nextCh)
8) END;
9) L.len := len + 1; L.wid := X + eolW - (F.X + F.left);
```

```
10) L.eot := R.fnt = NIL; Texts.Read(R, nextCh)
END DisplayLine;
```

Explanations: 1) set right margin 2) - 8) display characters of this line 3) get width dx, box x, y, w, h, and pattern address of next character 4) if there is enough space in the rectangle of contents 5) display pattern 7) jump to location of next character; read next character 9) - 10) setup line descriptor

Procedure DisplayLine is again similar to LocateChar, and the comment about relying on the file system's buffering capabilities applies once more. The principal difference between LocateChar and Width on one hand and DisplayLine on the other hand is the fact that the latter accesses the display screen physically. Therefore, possession of the screen lock is a tacit precondition for calling DisplayLine.

A quick look at an auxiliary procedure that updates the position marker concludes our tour behind the scenes of the text system:

```
PROC UpdateMark (F: Frame);
VAR oldH: INT;
BEGIN
1) oldH := F.markH; F.markH := F.org * F.H DIV (F.text.len 
IF (F.mark > 0) & (F.left >= barW) & (F.markH # oldH) THEN
2)
Display.ReplConst(Display.white, F.X + 1, F.Y + F.H - 1 - o
3)
```

```
Display.ReplConst(Display.white, F.X + 1, F.Y + F.H - 1 - F.markH, 
END
END UpdateMark;
```

Explanations 1) shows how the marker's position is calculated. Loosely spoken, the invariant is distance from top of frame / frame height = text position of first character in frame / text length 2) erase the old marker 3) draw the new marker

And this in turn concludes our Section on text frames. Recapitulating the most important points: The tasks of text editing (input oriented) and text rendering (output oriented) are combined in the concept of text frames. Text frames constitute a subclass of display frames, and they are implemented in a separate module called TextFrames. The implementation of TextFrames accesses the displayed text exclusively via the "official" abstract interface of module Texts discussed in Section 5.2. It maintains a private data structure of line lists to accelerate locating requests. Text frames use simple formatting rules that allow super-efficient rendering of text in a single pass. In particular, line spacing is fixed for every text frame. Therefore, different styles of a base font are possible within a given text frame while different sizes are not.

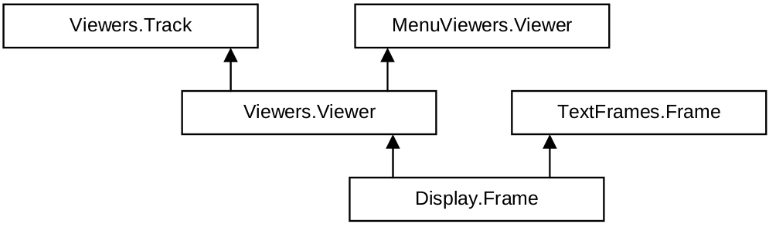


Figure 17: Extensions of type *Display.Frame*

Putting into relation the different extensions of type Display.Frame that we came across in Chapters 4 and 5, we obtain the type hierarchy as shown in Fig ??.

5.4 THE FONT MACHINERY

We saw in the previous Sections that Oberon texts support attribute specifications ("looks") for characters. Three different attributes are supported: font, color, and vertical offset. Let us first focus on the font attribute. A font can be regarded as a style the standard character set is designed in. Typically, an entire text is typeset in a single style, that is, there is one font per text. However, sometimes, an author wants to emphasize titles or words by changing the size of the font or by varying it to bold face or italics. In special texts, special characters like mathematical symbols or other kinds of icons may occur. In even more complex documents, mathematical or chemical formulae might flow within the text.

This generalized view leads us to a different interpretation of the notion of font. We can regard a font as an indexed library of (graphical) objects, mostly but not necessarily glyphs. In the case of ordinary characters it is natural to use the ASCII-code as an index, ending up with an interpretation of text as sequence of pairs (library, index). Note that this is a very general view indeed that, in principle, is equivalent with defining text as sequence of arbitrary objects.

The imaging model of characters provides two levels of abstraction. On the first level, characters are black boxes specified by a set of metric data x, y, w, h, and dx. (x, y) is a vector from the current point of reference on the base line to

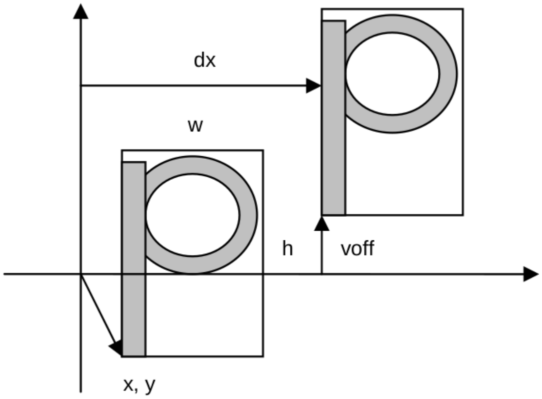


Figure 18: The geometric character model

the origin of the box. w and h are width and height of the box, and dx is the distance to the point of reference of the next character on the same base line. On the second level of abstraction, a character is defined by a digital pattern or glyph that is to be rendered into the box. Fig ?? visualizes this model of characters.

The additional two character attributes color and vertical offset appear now as parameters for the character model. The vertical offset allows translating the glyph vertically and the color attribute specifies the foreground color of the pattern.

Good examples of procedures operating on the first level of abstraction are procedures LocateChar and Width that we discussed in the previous Section, as well as text formatters for a remote printer. In contrast, procedure DisplayLine operates on the second level.

The representation of characters as digital patterns is merely the last step in a complex font design and rendering process. At the beginning is a generic description of the shape of each character in the form of outlines and hints. Outlines are typically composed of straight lines and spline-curves. Hints are included to assist the digitizer in its effort to faithfully map the filled character outlines into the device raster. For example, hinting can guarantee consistency of serif shapes and stem widths across an entire font within a text, independent of the relative positions of the characters with respect to the grid lines. Automatic digitization produces digital patterns of sufficiently high quality for printing media resolutions. For screen resolutions, however, we prefer to add a hand-tuning step. This is the reason why digital patterns are not produced "on the fly" in Oberon.

Oberon's font management is encapsulated in module Fonts, with a low-level extension into the module Display that we already know from Chapter 4. The interface to module Fonts is very simple and narrow:

```
MODULE Fonts;
TYPE Font = POINTER TO FontDesc;
FontDesc = RECORD
name: ARRAY 32 OF CHAR;;
```

```
height, minX, maxX, minY, maxY: INT;
next: Font
END;
VAR Default: Font;
PROC GetPat(fnt: Font; ch: CHAR; VAR dx, x, y, w, h, patadr
PROC This (name: ARRAY OF CHAR): Font;
PROC Free;
END Fonts.
```

Variable name in type Font is the name of the underlying file. The variables height, minX, maxX, minY, and maxY designate line height and summary metric data. Default is a system-wide default font. It is installed at system loading time. GetPat delivers the geometric data for a given character in a given font (see Figure 5.5). This is a procedure to internalize (load) a font from a file given by its name. Free releases from storage fonts that are no longer needed.

Type Font should again be regarded as an abstract data type with two intrinsic operations This and GetPat Thinking of the immutable nature of fonts, multiple internal copies of the same font are certainly undesirable. Therefore, internalized fonts are cached in a private list that manifests itself in a private field next in type FontDesc. The cache is maintained by the internalizing procedure This according to the following scheme:

```
search font in cache;
```

```
IF found THEN return cached internalization
ELSE internalize font; cache it
END
```

The implementation of type Font did not raise many challenges. One, however, is an undesirable side-effect of caching. The problem arises if a font is used for a limited time only. Because it is referenced by the cache it will never be collected by the system's garbage collector. Two possible solutions offer themselves: a) provide an explicit freeing operation and b) enforce some special handling by the garbage collector based on a concept of "weak" pointers.

We conclude this Section with a formal specification of the font file format. Note that on the one hand, the file format is completely private to the managing Fonts module and on the other hand, it should be ultimately stable because it is probably used for long-term backup and for wide-range data exchange across multi-system platforms.

This is an EBNF specification of the Oberon font file format:

```
FontFile = ident header contents.
header = abstraction family variant height minX maxX minY maxY.
contents = nofRuns { beg end } { dx x y w h } { rasterByte }.
```

ident, abstraction, family, and variant are one-byte values indicating file identification, abstraction (first level without raster bytes, second level with raster bytes), font family (Times Roman, Oberon, etc.), and variant (bold face, italics etc.). The values height, minX, maxX, minY and maxY are two bytes long each. They define in turn line height, minimum x-coordinate (of a box), maximum xcoordinate, minimum y-coordinate, and maximum y-coordinate. All values in production contents are two bytes long. nofRuns specifies the number of runs within the ASCII-code range (intervals occupied without gaps) and every pair [beg, end) describes one run. The tuples (dx, x, y, w, h) are the metric data of the corresponding characters (in their ASCII-code order), and the sequence of rasterByte gives the total of raster information.

In summary, fonts in Oberon are indexed libraries of objects. The objects are descriptions of character images in two levels of abstraction: As metric data of black boxes and as binary patterns (glyphs). Type Font is an abstract data type with intrinsic operations to internalize and to get character object data. Internalized fonts are cached in a private list.

5.5 THE EDIT TOOLBOX

We have seen that every text frame integrates an interactive text editor that we can regard as an interpreter of a set of built-in commands (intrinsic commands). Of course, we would like to be able to extend this set by custom editing commands (extrinsic commands). Adding additional editing commands was indeed a worthwhile stress test for the underlying texts API. Module Edit is the result of this effort. It is a toolbox of consisting of some standard extrinsic editing commands.

```
DEFINITION Edit;
PROC Open; (*text viewer*)
PROC Show; (*text*)
PROC Locate; (*position*)
PROC Search; (*pattern*)
PROC Store; (*text*)
PROC Recall; (*deleted text*)
PROC CopyFont;
PROC ChangeFont;
PROC ChangeColor;
PROC ChangeOffset;
END Edit.
```

The first group of commands in Edit is used to display, locate, and store texts or parts of texts. In turn they open a text file and display it, open a program text and show the declaration of a given object, locate a given position in a displayed text (main application: locating an error found by the compiler), search a pattern, and store the current state of a displayed text. Commands in the next group are related with editing. They allow restoring of the previously deleted part of text, copying a font attribute to the current text selection, and change attributes of the current text selection. Note that the commands CopyFont, ChangeFont, ChangeColor, and ChangeOffset are extrinsic variations of the intrinsic copy-look operation. The implementations of the toolbox commands are given in the Appendix.

REFERENCES

GUTKNECHT J. Gutknecht, "Concept of the Text Editor Lara", Communications of the ACM, Sept. 1985, Vol.28, No.9.

TEITELMAN W. Teitelman, "A tour through Cedar", IEEE Software, 1, (2), 44-73 (1984).

6 MODULE LOADER

6.1 LINKING AND LOADING

When the execution of a command `M.P` is requested, module `M` containing procedure `P` must be loaded, unless it

- already loaded for its another command had earlier been executed, or
- had been imported by another one before.

Modules are available in the form of so-called *object files*, generated by the compiler. The term *loading* refers to the transfer of module code from file into main memory, from where processors fetch individual instructions. It involves also a certain amount of transformation as required by

- the object file format on the one hand, and
- the storage layout, on the other.

A system typically consists of many modules, and hence loading modules also involves linking them together, in particular linking with already loaded ones. Before loading,

- references to another module's objects are relative to the base address of this one;
- the linking or binding process converts them into absolute addresses.

This process may require a significant amount of address computations. But they're simple enough and can be executed very swiftly, if the data are organized in an appropriate way. Nevertheless and surprisingly, in many OSes it needs more time than compilation. The remedy, which system designers offer, is to separate linking from loading:

- A set of compiled modules is 1st linked into a object file with absolute addresses.
- The loader then merely transfers it into main store.

We consider it an unfortunate solution. Instead of trying to cure it with aid of an additional processing stage and tool, it is wiser to remove the malady at its core, namely to speed up linking itself.

Indeed, there is no separate linker. In Oberon, they are integrated and fast enough to avoid any desire for pre-linking. Furthermore, the system extensibility crucially depends on the possibility to link additional modules to already loaded ones by calls from any module. This is called *dynamic loading*, with prelinked object files which is impossible.

- Newly loaded simply refer to already loaded, whereas
- prelinked files lead to multiple copies of same module.

Evidently, the essence of linking is the conversion of

- relative addresses as generated by the compiler for all external references, into
- absolute ones as required during program execution.

To proceed, we must consider an additional complication.

Assume that a module `M1` is to be compiled which is a client of (that is, it imports) `M0`. The interface of `M1` - in the form of a symbol file - does not specify the entry addresses of its exported procedures, but merely specifies a unique number (`pno`) for each one of them. The reason are:

- In this way the implementation of `M0` may be modified, causing a change of entry addresses without affecting its interface specification. And
- a crucial property of the scheme of separate modules compilation: Changes of the implementation of `M0` must not necessitate the recompilation of clients (`M1`).

The consequence is

- the binding of entry addresses to `pnos` must be performed by the linker. To make this possible,
- the object file must contain a list (table) of its entry addresses, one for each `pno` used as index to the table.

Similarly, the object file must contain a table of imported modules, containing their names. An external reference in the program code then appears in the form of a pair consisting of

- a module number (`mno`) - used as index to the import table (of modules), and
- a procedure number (`pno`) - used as index to the entry table of this module.

Certain linkage information must not only be provided in each object file, but also be present along with each loaded module's program code, because a module to be loaded must be linkable with modules loaded at any earlier time without reading their object files again.

6.2 MODULE REPRESENTATION

The primary requirement is

a system must be represented in a form that allows to add new modules quickly.

What is a sensible representation for this purpose? The simplest solution that comes to mind is a list of module blocks containing sections for

- the global data,
- the program code, and perhaps
- meta data for the linking process.

The list is rooted in a variable global to the loader module, here called `Modules`.

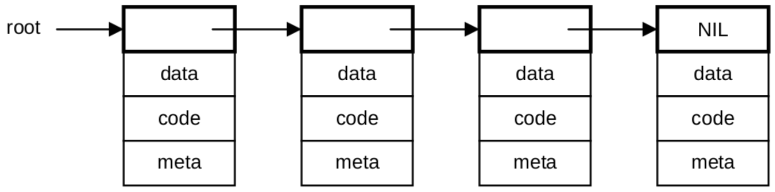


Figure 19: System of 4 modules

The 1st part, containing the link to the next module, is called *module descriptor*. In Oberon, it contains further links to the various sections of a module. The type `Module` is defined as:

```
TYPE Module = POINTER TO ModDesc;
     ModuleName = ARRAY 32 OF CHAR;
     ModDesc = RECORD name: ModuleName;
```

```
    next: Module;
    key, num, size, refcnt, data, code,
    imp, cmd, ent, ptr: INT (*addresses*)
  END;
```

key the module's key used for version consistency checking. Changes, if and only if, the module's interface and thereby its symbol file changes.

num the module's number, index of the module's entry in a global module table, referenced by the processor's `MT` register. The invariant relationship is

```
ModTable[mod.mno] = mod.data
```

for all `mod` in the module list.

size the entire module block's size excluding the descriptor, and

refcnt the number of other modules importing this module. Used to check whether a module can be released by procedure `Modules.Free`.

The section with meta data follows the `data` and `code` areas and consists of several parts.

- Imports is an array of the modules imported by this module, each entry being the address of the respective module descriptor.

- Commands is a sequence of procedure identifiers followed by their offset in the code section. This section is used when activating a command.

- Entries is an array of offsets of all exported entities (including commands). This section is used by the loader itself for linking.

- Pointer refs is an array of offsets of global pointer variables in the data section. These are used by the garbage collector as the roots of graphs of heap objects in use.

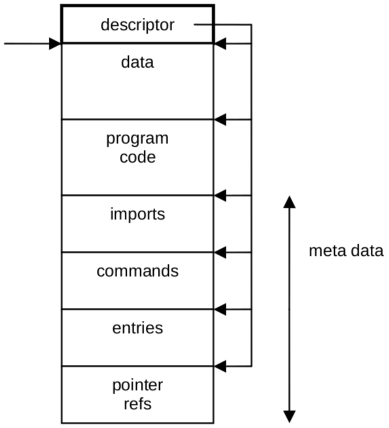


Figure 20: Module block headed by descriptor

6.3 THE LINKING LOADER

The purpose of the loader is to read object files, and to transform the file representation of modules into their internal image.

The loader is represented by procedure `Load` in module `Modules`. It accepts a name and returns a pointer to the specified module's descriptor. It 1st scans the list searching for the named module. Only if it is not present, the module is loaded and added to the list. Duplications therefore cannot occur.

```
mod := root;
WHILE (mod # NIL) & (name # mod.name) DO
  mod := mod.next
END;

IF mod = NIL THEN (*load*)
  F := ThisFile(name);
  Files.Set(R, F, 0);
  ...
```

First, the header of the respective object file is read. It specifies the required size of the block which is allocated in the module area at the position indicated by the global variable `AllocPtr`. Then the list of imports of the module being imported is read, and these module are imported.

Evidently procedure `Load` is used recursively. Because cyclic imports are excluded, recursion always terminates.

```
  Files.ReadString(R, impname); (*imports*)
  WHILE (impname[0] # 0X) & (res = 0) DO
    Files.ReadInt(R, impkey);
    Load(impname, impmod);
    import[nofimps] := impmod;
    importing := name1;
    IF res = 0 THEN
      IF impmod.key = impkey THEN
        INC(impmod.refcnt);
        INC(nofimps)
      ELSE
        error(3, name);
        imported := impname
      END
    END;
    Files.ReadString(R, impname)
  END
```

The loading process stops, if a key mismatch is detected (`err = 3`). After successful loading of all imports, the loading of the actual module proceeds by allocating a descriptor and then reading the remaining sections of the file. The data is allocated (and cleared) and the code section is read in a straight-forward way without alteration.

At the very end of the file 3 integers called `fixorgP`, `fixorgD`, and `fixorgT` are read. They are the anchors of linked lists in the program code of instructions that need fixups. These fixups are performed only after the entire file had been read. Traversing the P-list, the pairs mno-pno are replaced by computed offsets in `BL` instructions (procedure calls). Traversing the D-list, addresses of `LDR` instructions and instruction pairs are fixed up, and traversing the T-list, addresses of type descriptors are computed and inserted. This low-level piece of code is shown below for call instructions (`BL`). Those for the D-List and the T-list are analogous.

```
adr := mod.code + fixorgP*4;
WHILE adr # mod.code DO
  SYSTEM.GET(adr, inst);
  mno := inst DIV 100000H MOD 10H; (*decompose*)
  pno := inst DIV 1000H MOD 100H;
  disp := inst MOD 1000H;
  SYSTEM.GET(mod.imp + (mno-1)*4, impmod);
  SYSTEM.GET(impmod.ent + pno*4, dest);
  dest := dest + impmod.code;
  offset := (dest - adr - 4) DIV 4;    (*compose*)
  SYSTEM.PUT(adr,(offset MOD 1000000H)+0F7000000H);
  adr := adr - disp*4
END;
```

After the module has been loaded successfully, its initialization body is executed. Apart from `Load`, module `Modules` also contains the procedures

```
PROC ThisCommand(mod: Module; name: ARRAY OF CHAR):
                                              Command;
PROC Free       (             name: ARRAY OF CHAR);
```

- The former yields the procedure named `name` from module `mod`. It is used in `TextFrames.Call` for activating command procedures.
- The latter unloads the named module, i.e. removes it from the list of loaded modules.

The frequent use of the low-level procedures `SYSTEM.GET` and `SYSTEM.PUT` is easily justified in base modules such as the loader or device drivers. After all, here data are transferred into untyped main storage.

6.4 THE TOOLBOX OF THE LOADER

User commands directed to the loader are contained in module `System`. The toolbox offers the following 3 commands:

```
System.ShowModules
System.ShowCommands modname
System.Free {modname}
```

- The 1st command opens a viewer and provides a list of all loaded modules. The list indicates the block length and the number of clients importing a module (the reference count).
- `ShowCommands` opens a viewer and lists the commands provided by the specified module. The commands are prefixed by the module name, and hence can immediately be activated by a mouse click.
- `Free` is called in order to remove modules either to regain storage space or to replace a module by a newly compiled version. A module can be dispensed only if
  1. it has no clients, and
  2. does not declare any record types which are extensions of imported types.

6.5 THE OBERON OBJECT FILE FORMAT

The name extension of object files is `.rsc`. Their syntax:

```
CodeFile = name key version size
imports typedesc varsize strings code commands
        entries ptrrefs fixP fixD fixT body "O".
imports = {modname key} 0X.
typedesc = nof {byte}.
strings = nof {char}.
```

```
code = nof {word}.
commands = {comname offset} 0X.
entries = nof {word}.
ptrrefs = {word} 0.
```

- `fixP`, `fixD`, `fixT` are the origins of chains of instructions to be updated (fixed up).
- `body` is the entry point offset of the module `body`.

7 FILE SYSTEM(FS)

7.1 FILES

It is essential that a computer system has a facility for storing data over longer periods of time and for retrieving the stored data. Such a facility is called a FS. Evidently, a FS cannot accommodate all possible data types and structures that will be programmed in the future. Hence, it is necessary to provide a simple, yet flexible enough base structure that allows any data structure to be mapped onto this base structure (and vice-versa) in a reasonably straight-forward and efficient way. This base structure, called file, is a sequence of bytes. As a consequence, any given structure to be transformed into a file must be sequentialized. The notion of sequence is indeed fundamental, and it requires no further explanation and theory. We recall that texts are sequences of characters, and that characters are typically represented as bytes.

The sequence is also the natural abstraction of all physically moving storage media. Among them are magnetic tapes and disks. Magnetic media have the welcome property of non-volatility and are therefore the primary choices for storing data over longer periods of time, especially over periods where the equipment is switched off. Sequential access is also necessary for media that allow access only by large blocks, such as flash-RAMs and SD-cards.

A further advantage of the sequence is that its transmission between media is simple too. The reason is that its structural information is inherent and need not be encoded and transmitted in addition to the actual data. This implicitness of structural information is particularly convenient in the case of moving storage media, because they impose strict timing constraints on transmission of consecutive elements. Therefore, the process which generates (or consumes) the data must be effectively decoupled from the transmission process that observes the timing constraints. In the case of sequences, this decoupling is simple to achieve by dividing a sequence into subsequences which are buffered. A sequence is output to the storage medium by alternately generating data (and filling the buffer holding the current subsequence) and transmitting data (fetching elements from the buffer and transmitting them). The size of the subsequences (and the buffer) depends on the storage medium under consideration: there must be no timing constraints between accesses to consecutive subsequences.

The file is not a static data structure like the array or the record, because the length may increase dynamically, i.e. during program execution. On the other hand, the sequence is less flexible than general dynamic structures, because it cannot change its form, but only its length, since elements can only be appended but not inserted. It might therefore be called a semi-dynamic structure. The discipline of purely sequential access to a file is enforced by restricting access to calls of specific procedures, typically read and write procedures for scanning and generating a file. In the jargon of data processing, a file must be opened before reading or writing is possible. The opening implies the initialization of a reading and writing mechanism, and in particular the fixing of its initial position. Hence each (opened) file not only has a value and a length, but also a position attributed to it. If reading must occur from several positions (still sequentially) alternately, the file is "multiply opened"; it implies that the same file is represented by several variables, each denoting a different position.

This widespread view of files is conceptually unappealing, and the Oberon FS therefore departs from it by introducing the notion of a rider. A file simply has a value, the sequence of bytes, and a length, the number of bytes in the sequence. Reading and writing occurs through a rider, which denotes a position. "Multiple opening" is achieved by simply using several riders riding on the same file. Thereby the two concepts of data structure (file) and access mechanism (rider) are clearly distinct and properly disentangled.

Given a file f, a rider r is placed on a file by the call Files.Set (r, f, pos), where pos indicates the position from which reading or writing is to start. Calls of Files.Read (r, x) and Files.Write (r, x) implicitly increment the position beyond the element read or written, and the file is implicitly denoted via the explicit parameter r, which denotes a rider. The rider has two (visible) attributes, namely r.eof and r.res. The former is set to FALSE by Files.Set, and to TRUE when a read operation could not be performed, because the end of the file had been reached. r.res serves as a result variable in procedures ReadBytes and WriteBytes allowing one to check for correct termination.

A FS must not only provide the concept of a sequence with its accessing mechanism, but also a registry. This implies that files be identified, that they can be given a name by which they are registered and retrieved. The registry or collection of registered names is called the file system's directory. Here we wish to emphasize that the concepts of files as data structure with associated access facilities on the one hand, and the concept of file naming and directory management on the other hand must also be considered separately and as independent notions. In fact, in the Oberon their implementation underscores this separation by the existence of two modules: Files and FileDir. The following procedures are available. They are summarized by the interface specification (definition) of module Files.

```
DEFINITION Files;
TYPE File = POINTER TO FileDesc;
FileDesc = RECORD END ;
Rider = RECORD eof: BOOL; res: INT END ;
PROC Old(name: ARRAY OF CHAR): File;
PROC New(name: ARRAY OF CHAR): File;
PROC Register(f: File);
PROC Close(f: File);
PROC Purge(f: File);
PROC Length(f: File): INT;
PROC Date(f: File): INT);
PROC Set(VAR r: Rider; f: File; pos: INT);
PROC ReadByte(VAR r: Rider; VAR x: BYTE);
PROC ReadBytes(VAR r: Rider; VAR x: ARRAY OF BYTE; n: INT);
PROC Read(VAR r: Rider; VAR ch: CHAR);
PROC ReadInt(VAR r: Rider; VAR n: INT);
PROC ReadSet(VAR r: Rider; VAR s: SET);
PROC ReadReal(VAR r: Rider; VAR x: REAL);
PROC ReadString(VAR r: Rider; VAR s: ARRAY OF CHAR);
PROC ReadNum(VAR r: Rider; VAR n: INT);
PROC WriteByte(VAR r: Rider; x: BYTE);
PROC WriteBytes(VAR r: Rider; x: ARRAY OF BYTE; n: INT);
PROC WriteInt(VAR r: Rider; n: INT);
PROC WriteSet(VAR r: Rider; s: SET);
PROC WriteReal(VAR r: Rider; x: REAL);
PROC WriteString(VAR r: Rider; x: ARRAY OF CHAR);
PROC WriteNum(VAR r: Rider; n: INT);
PROC Pos(VAR r: Rider): INT;
PROC Base(VAR r: Rider): File;
PROC Rename(old, new: ARRAY OF CHAR; VAR res: INT);
PROC Delete(name: ARRAY OF CHAR; VAR res: INT);
END Files.
```

New(name) yields a new (empty) file without registering it in the directory. Old(name) retrieves the file with the specified name, or yields NIL, if it is not found in the directory. Register(f) inserts the name of f (specified in the call of New) in the directory. An already existing entry with this name is replaced. Close(f) must be called after writing is completed and the file is not to be registered. Close actually stands for "close buffers", and is implied in the procedure Register. Procedure Purge will be explained at the end of section 7.2.

The sequential scan of a file f (reading characters) is programmed as shown in the following template:

```
VAR f: Files.File; r: Files.Rider;
f := Files.Old(name);
IF f $\neq$ NIL THEN
Files.Set (r, f, 0); Files.Read (r, x);
WHILE ~ r.eof DO ... x ...; Files.Read(r, x) END
END
```

The analogous template for a purely sequential writing is:

```
f := Files.New(name); Files.Set(r, f, 0);
WHILE ... DO Files.Write (r, x); ... END
Files.Register(f)
```

There exist two further procedures; they do not change any files, but only affect the directory. Delete(name, res) causes the removal of the named entry from the directory. Rename(old, new, res) causes the replacement of the directory entry old by new.

It may surprise the reader that these two procedures, which affect the directory only, are exported from module Files instead of FileDir. The reason is that the presence of the two modules, together forming the FS, is also used for separating the interface into a public and a private (or semi-public) part. The definition (in the form of a symbol file) of FileDir is not intended to be freely available, but restricted to use by system programmers. This allows the export of certain sensitive data, (such as file headers) and sensitive procedures (such as Enumerate) without the danger of misuse by inadvertent users.

Module Files constitutes a most important interface whose stability is utterly essential, because it is used by almost every module programmed. During the entire time span of development of the Oberon, this interface had changed only once. We also note that this interface is very terse, a factor contributing to its stability. Yet, the offered facilities have in practice over years proved to be both necessary and sufficient.

7.2 IMPLEMENTATION ON A RANDOM-ACCESS STORE

A file cannot be allocated as a block of contiguous storage locations, because its length is not fixed. Neither can it be represented as a linked list of individual elements, because this would lead to inefficient use of storage - more might be used for the links than the elements themselves. The solution generally adopted is a compromise between the two extremes: files are represented as lists of blocks (subsequently called sectors) of fixed length. A block is appended when the last one is filled. On the average, each file therefore wastes half of a sector. Typical sector sizes are 0.5, 1, 2, or 4 Kbytes, depending on the device used as store.

It immediately follows that access to an element is not as simple as in the case of an array. The primary concern in the design of a FS and access scheme must be the efficiency of access to individual elements while scanning the sequence, at least in the case when the next element lies within the same sector. This access must be no more complicated than a comparison of two variables followed by an indexed access to the file element and the incrementing of an address pointing to the element's successor. If the successor lies in another sector, the procedure may be more involved, as transitions to the next sector occur much less frequently.

The second most crucial design decision concerns the data structure in which sectors are organized; it determines how a succeeding sector is located. The simplest solution is to link sectors in a list. This is acceptable if access is to be restricted to purely sequential scans. Although this would be sufficient for most applications, it is unnecessarily restrictive for media other than purely sequential ones (tapes). After all, it is sometimes practical to position a rider at an arbitrary point in the file rather than always at its beginning. This is made possible by the use of an indexed sector table, typically stored as a header in the file. The table is an array of the addresses of the file's data sectors. Unfortunately, the length of the table needed is unknown. Choosing a fixed length for all files is controversial, because it inevitably leads to either a limitation of file length (when chosen too small) that is unacceptable in some applications, or to a large waste of file space (when chosen too large). Experience shows that in practice most files are quite short, i.e. in the order of a few thousand bytes. The dilemma is avoided by a two-level table, i.e. by using a table of tables.

The scheme chosen in Oberon is slightly more complex in order to favor short files (< 64 K bytes): Each file header contains a table of 64 entries, each pointing to a 1K byte sector. Additionally, it contains a table of 12 entries, the so-called extensions, each pointing to an index sector containing 256 further sector pointers. The file length is thereby limited to 64 + 12*256 sectors, or 3'211'264 bytes (minus the length of the header). The chosen structure is illustrated in Fig. ??. sec[0] always points to the sector containing the file header.

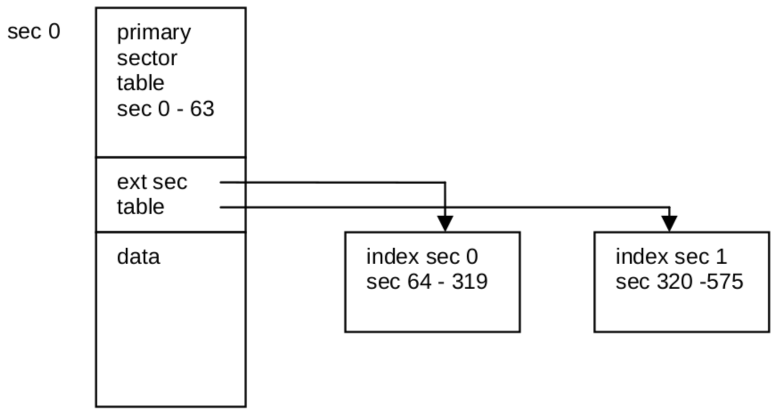


Figure 21: File header and extension sectors

The header contains some additional data, namely the length of the file (in bytes), its name, and date and time of its creation. The size of the header is 352 bytes; the remaining 672 bytes of the first sector are used for data. Hence, truly short files occupy a single sector only. The declaration of the file header is contained in the definition of module FileDir. An abbreviated version containing the fields relevant so far is:

```
FileHeader = RECORD
leng: INT;
ext: ARRAY 12 OF SectorPointer;
sec: ARRAY 64 OF SectorPointer
END
```

We now turn our attention to the implementation of file access, and first present a system that uses main storage for the file data instead of a disk and therefore avoids the problems introduced by sector buffering. The key data structure in this connection is the Rider, represented as a record.

```
Rider = RECORD
eof: BOOL; res, pos, adr: INT;
file: File
END
```

A rider is initialised by a call Set(r, f, pos), which places the rider r on file f at position pos. From this it is clear that the rider record must contain fields denoting the attached file and the rider's position on it. We note that they are not exported. However, their values can be obtained by the function procedures Pos(r) and Base(r). This allows a (hidden) representation most appropriate for an efficient implementation of Read and Write without being unsafe.

Consider now the call Read(r, x); its task is to assign the value of the byte designated by the rider's position to x and to advance the position to the next byte. Considering the structure by which file data are represented, we easily obtain the following program, assuming that the position is legal, i.e. non-negative and less than the file's length. a, b, c are local variables, HS is the size of the header (in sector 0), SS is the sector size, typically a power of 2 in order to make division efficient.

```
a := (r.pos + HS) DIV SS; b := (r.pos + HS) MOD SS;
IF a < 64 THEN c := r.file.sec[a]
ELSE c := r.file.ext[(a - 64) DIV 256].sec[(a - 64) MOD 256]
END ;
SYSTEM.GET(c + b, x) ; INC (r.pos)
```

In order to gain efficiency, we use the low-level procedure GET that assigns the value at address c+b to x. This program is reasonably short, but involves considerable address computations at every access, and in particular at positions larger than 64 * SS. Fortunately, there exists an easy remedy, namely that of caching the address of the current position. This explains the presence of the field adr in the rider record. The resulting program is shown below; note that in order to avoid the addition of HS, pos is defined to denote the genuine position, i.e. the abstract position augmented by HS.

```
SYSTEM.GET(r.adr, x); INC(r.adr); INC(r.pos);
IF r.pos MOD SS = 0 THEN
m := r.pos DIV SS;
IF m < 64 THEN r.adr := r.file.sec[m]
ELSE r.adr := r.file.ext[(m - 64) DIV 256].sec[(m - 64) MOD 256]
END
END
```

We emphasize that in all but one out of 1024 cases only three instructions and a single test are to be executed. This improvement therefore is crucial to the efficiency of file access, and to that of the entire Oberon. We now present the entire file module (for files on a random-access store).

```
MODULE MFiles; (*NW 24.8.90 / 12.10.90 / 20.6.2013*)
IMPORT SYSTEM, Kernel, FileDir;
(*A file consists of a sequence of sectors. The first sector
Part of the header is the sector table, an array of addresse
A file is referenced through riders each of which indicates
CONST
HS = FileDir.HeaderSize;
SS = FileDir.SectorSize;
STS = FileDir.SecTabSize;
XS = FileDir.IndexSize;
TYPE File* = POINTER TO FileDesc;
Index = POINTER TO IndexRecord;
IndexRecord = RECORD sec: FileDir.IndexSector END ;
Rider* =
RECORD eof*: BOOL;
res*, pos, adr: INT;
file: File
END ;
FileDesc =
RECORD mark: INT;
name: FileDir.FileName;
len, date: INT;
```

```
ext: ARRAY FileDir.ExTabSize OF Index;
sec: FileDir.SectorTable
END ;
PROC Old*(name: ARRAY OF CHAR): File;
VAR head: INT;
namebuf: FileDir.FileName;
BEGIN
FileDir.Search(name, head); RETURN SYSTEM.VAL(File, head)
END Old;
PROC New*(name: ARRAY OF CHAR): File;
VAR f: File; head: INT;
BEGIN f := NIL; Kernel.AllocSector(0, head);
IF head $\neq$ 0 THEN
f := SYSTEM.VAL(File, head); f.mark := FileDir.HeaderMark;
f.len := HS; f.name := name;
f.date := Kernel.Clock(); f.sec[0] := head
END ;
RETURN f
END New;
PROC Register*(f: File);
BEGIN
IF (f $\neq$ NIL) & (f.name[0] > 0X) THEN FileDir.Insert(f.name, f.
END Register;
PROC Length*(f: File): INT;
BEGIN RETURN f.len - HS
END Length;
PROC Date*(f: File): INT;
BEGIN RETURN f.date
```

```
END Date;
PROC Set∗(VAR r: Rider; f: File; pos: LONGINT);
VAR m, n: INT;
BEGIN r.eof := FALSE; r.res := 0;
IF f $\neq$ NIL THEN
IF pos < 0 THEN r.pos := HS
ELSIF pos > f.len - HS THEN r.pos := f.len
ELSE r.pos := pos + HS
END ;
r.file := f; m := r.pos DIV SS; n := r.pos MOD SS;
IF m < STS THEN r.adr := f.sec[m] + n
ELSE r.adr := f.ext[(m-STS) DIV XS].sec[(m-STS) MOD XS] + n
END
END
END Set;
PROC ReadByte∗(VAR r: Rider; VAR x: BYTE);
VAR m: INT;
BEGIN
IF r.pos < r.file.len THEN
SYSTEM.GET(r.adr, x); INC(r.adr); INC(r.pos);
IF r.adr MOD SS = 0 THEN
m := r.pos DIV SS;
IF m < STS THEN r.adr := r.file.sec[m]
ELSE r.adr := r.file.ext[(m-STS) DIV XS].sec[(m-STS) MOD XS
END
END
ELSE x := 0; r.eof := TRUE
END
```

```
END ReadByte;
PROC WriteByte∗(VAR r: Rider; x: BYTE);
VAR k, m, n, ix: INT;
BEGIN
IF r.pos < r.file.len THEN
m := r.pos DIV SS; INC(r.pos);
IF m < STS THEN r.adr := r.file.sec[m]
ELSE r.adr := r.file.ext[(m-STS) DIV XS].sec[(m-STS) MOD XS]
END
ELSE
IF r.adr MOD SS = 0 THEN
m := r.pos DIV SS;
IF m < STS THEN Kernel.AllocSector(0, r.adr); r.file.sec[m] := r.ad
ELSE n := (m - STS) DIV XS; k := (m - STS) MOD XS;
IF k = 0 THEN (∗new index∗)
Kernel.AllocSector(0, ix); r.file.ext[n] := SYSTEM.VAL(Index, ix)
END ;
Kernel.AllocSector(0, r.adr); r.file.ext[n].sec[k] := r.adr
END
END ;
INC(r.pos); r.file.len := r.pos
END ;
SYSTEM.PUT(r.adr, x); INC(r.adr)
END WriteByte;
PROC Pos∗(VAR r: Rider): INT;
BEGIN RETURN r.pos - HS
END Pos;
PROC Base∗(VAR r: Rider): File;
```

```
BEGIN RETURN r.file
END Base;
END MFiles.
```

Allocation of a new sector occurs upon creating a file (Files.New), and when writing at the end of a file after the current sector had been filled. Procedure AllocSector yields the address of the allocated sector. It is determined by a search in the sector reservation table for a free sector. In this table, every sector is represented by a single bit indicating whether or not the sector is allocated. Although conceptually belonging to the FS, this table resides within module Kernel.

Deallocation of a file's sectors could occur as soon as the file is no longer accessible, neither through a variable of any loaded module nor from the file directory. However, this moment is difficult to determine. Therefore, the method of garbage collection is used in Oberon for the deallocation of file space. In consideration of the fact that file space is large and the collection of unused sectors relatively time-consuming, we confine this process to system initialization. It is represented by procedure FileDir.Init. At that time, the only referenced files are those registered in the directory. Init therefore scans the entire directory and records the sectors referenced in each file in the sector reservation table (see Sect. 7.4).

For applications where system startup and initialization is supposed to occur very infrequently, such as for server systems, a procedure Files.Purge is provided. Its effect is to return the sectors used by the specified file to the pool of free sectors. Evidently, the programmer then bears the responsibility to guarantee that no references to the purged file continue to exist. This may be possible in a closed server system, but files should not be purged under normal circumstances, as a violation of said precondition will lead to unpredictable disaster. The following procedures used for allocating, deallocating, and marking sectors in the sector reservation table are defined in module Kernel:

```
PROC AllocSector(hint: INT; VAR sec: INT); (*used in WriteByte*)
PROC MarkSector(sec: INT); (*used in Init*)
PROC FreeSector(sec: INT); (*used in Purge*)
```

7.3 IMPLEMENTATION ON A DISK

First we recall that the organization of files as sets of individually allocated blocks (sectors) is inherently required by the allocation considerations of dynamically growing sequences. However, if the storage medium is a tape, a disk, or a flash-RAM, there exists an additional reason for the use of blocks. They constitute the subsequences to be individually buffered for transmission in order to overcome the timing constraints imposed by the medium. If an adequate space utilization is to be achieved, the blocks must not be too long. A typical size is 1, 2, or 4K bytes. This necessity of buffering has a profound influence on the implementation of file access. The complication arises because the abstraction of the sequence of individual bytes needs to be maintained. The increase in complexity of file access is considerable, as can be seen by comparing the program listings of the two respective implementations.

The first, obvious measure is to copy the file's sector table into primary store when a file is "opened" through a call of New() or Old(). The record holding this copy is the file descriptor, and the value f denoting the file points to this handle (instead of the actual header on disk). The descriptor also contains the remaining information stored in the header, in particular the file's length.

If a file is read (or written) in purely sequential manner, a single buffer is appropriate for the transfer of data. For reading, the buffer is filled by reading a sector from the disk, and bytes are picked up individually from the buffer. For writing, bytes are deposited individually, and the buffer is written onto disk as a whole when full. The buffer is associated with the file, and a pointer to it is contained in the descriptor.

However, we recall that several riders may be placed on a file and be moved independently. It might be appealing to associate a buffer with each rider. But this proposal must quickly be rejected when we realize that several riders may be active at neighbouring positions. If these positions refer to the same sector, which is duplicated in the riders' distinct buffers, the buffers may easily become inconsistent. Obviously, buffers must not be associated with riders, but with the file itself. The descriptor therefore contains the head of a list of linked buffers. Each buffer is identified by its position in the file. An invariant of the system is that no two buffers represent the same sector.

Even with the presence of a single rider, the possibility of having several buffers associated with a file can be advantageous, if a rider is frequently repositioned. It becomes a question of strategy and heuristics when to allocate a new buffer. In the Oberon, we have adopted the following solution:

1. The first buffer is created when the file is opened (New, Old).

2. Additional buffers may be allocated when a rider is placed (or repositioned) on the file.

3. At most four buffers are connected to the same file.

4. Purely sequential movements of riders do not cause allocation of buffers.

5. Separate buffers are generated when extensions of the file's sector table need be accessed (rider position > 64K). Each buffers the 256 sector addresses of the respective index sector.

The outlined scheme requires and is based upon the following data structures and types:

```
File =
Buffer =
Index =

POINTER TO FileDesc;
POINTER TO BufferRecord;
POINTER TO IndexRecord;

FileDesc = RECORD next: File;
aleng, bleng: INT;
(*file length*)
nofbufs: INT; (*no. of buffers allocated*)
modH, registered: BOOL; (*header has been modified*)
```

```
firstbuf: Buffer:
(*head of buffer chain*)
sechint: DiskAdr;
(*sector hint*)
name: FileDir.FileName;
date: INT;
ext: ARRAY FileDir.ExTabSize OF Index;
sec: ARRAY 64 OF DiskAdr
END;
BufferRecord = RECORD apos, lim: INT; (*lim = no. of bytes*)
mod: BOOL;
(*buffer has been modified*)
next: Buffer;
(*buffer chain*)
data: FileDir.DataSector
END;
IndexRecord = RECORD adr: DiskAdr;
mod: BOOL;
(*index record has been modified*)
sec: FileDir.IndexSector
END;
Rider =

RECORD eof: BOOL;
res: INT;
file: File;
apos, bpos: INT;
buf: Buffer
```

```
END ;

(*end of file reached*)
(*no. of unread bytes*)
(*position*)
(*hint: likely buffer*)
```

In order to increase efficiency of access, riders have been provided with a field containing the address of the element of the rider's position. From the conditions stated above for the allocation of buffers, it is evident that the value of this field can be a hint only. This implies that there can be no reliance on its information. Whenever it is used, its validity has to be checked. The check consists in a comparison of the riders' position r.apos with the hinted buffer's actual position r.buf.apos. If they differ, a buffer with the desired position must be searched and, if not present, allocated. The advantage of the hint lies in the fact that the hint is correct with a very high probability. The check is included in procedures Read, ReadByte, Write, and WriteByte. Some fields of the record types require additional explanations:

1. The length is stored in a "preprocessed" form, namely by the two integers aleng and bleng such that aleng is a sector number and

   ```
   length = (aleng * SS) + bleng - HS
   aleng = (length + HS) DIV SS
   ```

```
bleng = (length + HS) MOD SS
```

The same holds for the form of the position in riders (apos, bpos).

2. The field nofbufs indicates the number of buffers in the list headed by firstbuf:

   ```
   1 <= nofbufs <= Maxbufs.
   ```

3. Whenever data are written into a buffer, the file becomes inconsistent, i.e. the data on the disk are outdated. The file is updated, i.e. the buffer is copied into the corresponding disk sector, whenever the buffer is reallocated, e.g. during sequential writing after the buffer is full and is "advanced". During sequential reading, a buffer is also advanced and reused, but needs not be copied onto disk, because it is still consistent. Whether a buffer is consistent or not is indicated by its state variable mod (modified). Similarly, the field modH in the file descriptor indicates whether or not the header had been modified.

4. The field sechint records the number of the last sector allocated to the file and serves as a hint to the kernel's allocation procedure, which allocates a next sector with an address larger than the hint. This is a measure to gain speed in sequential scans.

5. The buffer's position is specified by its field apos. Used as index in the file header's sector table, it yields the sector corresponding to the current buffer contents. The field lim specifies the number of bytes s stored in the buffer. Reading cannot proceed beyond this limiting index; writing beyond it implies an increase in the file's length. All buffers except the one for the last sector are filled and specify lim = SS.

6. The hidden rider field buf is merely a hint to speed up localization of the concerned buffer. A hint is likely, but not guaranteed to be correct. Its validity must be checked before use. The buffer hint is invalidated when a buffer is reallocated and/or a rider is repositioned. The structure of riders remains practically the same as for files using main store. The hidden field adr is merely replaced by a pointer to the buffer covering the rider's position. A configuration of a file f with two riders is shown in Fig ??.

Some comments concerning module Files follow.

1. After the writing of a file has been completed, its name is usually registered in the directory. Register invokes procedure Unbuffer. It inspects the associated buffers and copies those onto disk which had been modified. During this process, new index sectors may have to be transferred as well. If a file is to remain anonymous and local to a module or command, i.e. is not to be

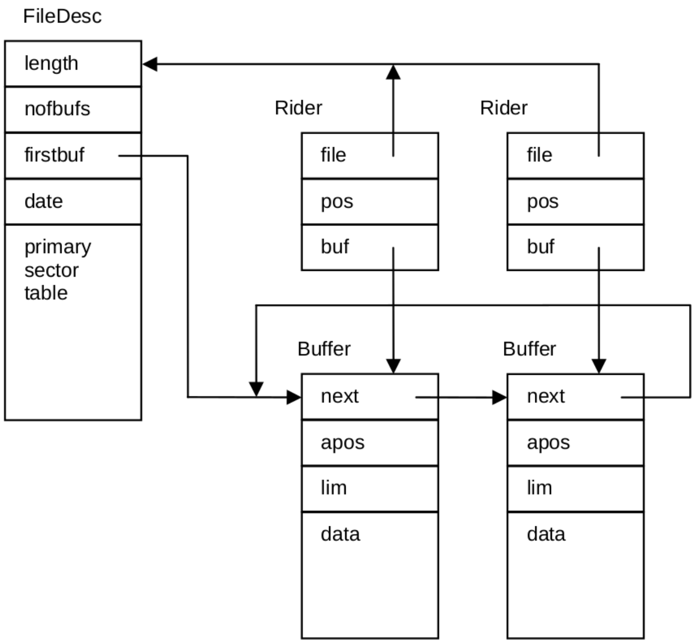


Figure 22: File f with two riders and two buffers

registered, but merely to be read, the release of buffers must be specified by an explicit call to Close (meaning "close buffers"), which also invokes Unbuffer.

2. Procedure Old (and for reasons of consistency also New) deviates from the general Oberon programming rule that an object be allocated by the calling (instead of the called) module. This rule would suggest the statements

   $$New(f); Files.Open(f, name)$$

   instead of

   $$f := Files.Old(name).$$

   The justification for the rule is that any extension of the type of f could be allocated, providing for more flexibility. And the reason for our deviation in the case of files is that, upon closer inspection, not a new file, but only a new descriptor is to be allocated. The distinction becomes evident when we consider that several statements $f := Files.Old(name)$ with different f and identical name may occur, probably in different modules. In this case, it is necessary that the same descriptor is referenced by the delivered pointers in order to avoid file inconsistency. Each (opened) file must have exactly one descriptor. When a file is opened, the first action is therefore to inspect whether a descriptor of this file already exists. For this purpose, all descriptors are linked together in a list anchored by the global variable root and linked by the descriptor field next. This measure may seem to solve the problem of avoiding inconsistencies smoothly. However, there exists a pitfall that is easily overlooked: All opened files would permanently remain accessible via root, and the garbage collector could never remove a file descriptor nor its associated buffers. This would be unacceptable. In order to hide this list from the garbage collector, it is represented by integers (addresses) instead of pointers.

3. Sector pointers are represented by sector numbers of type *INT*. Actually, we use the numbers multiplied by 29. This implies that any single-bit error leads to a number which is not a multiple of 29, and hence can easily be detected. Thereby the crucial sector addresses are software parity checked and are safe (against single-bit errors) even on computers without hardware parity check. The check is performed by procedures *Kernel.GetSector* and *Kernel.PutSector*.

7.4 THE FILE DIRECTORY

A directory is a set of pairs, each pair consisting of a name (key) and an object (here: file). It serves to retrieve objects by their name. If efficiency matters, the directory is organized as an ordered set, ordered according to the keys. The most frequently used structures for ordered sets are trees and hash tables. The latter have disadvantages when the size of the set is unknown, particularly when its order of magnitude is unknown, and when deletions occur. The Oberon system therefore uses a tree structure for its file directory, more specifically a B-tree, which was developed especially for cases where not individual pairs, but only sets of pairs as a whole (placed on a disk sector) can be accessed.

For a thorough study of B-trees we refer the reader to the literature [1, 2]. Here it must suffice to specify the B-tree's principal characteristics:

1. In a B-tree of order N, each node (called page) contains m elements (pairs), where $N \leq m \leq 2N$, except the root, where $0 \leq m \leq 2N$.

2. A page with m elements has either 0 descendants, in which case it is called a leaf page, or $m + 1$ descendants.

3. All leaf pages are on the same (bottom) level.

height	minimum	maximum
1	0	24
2	25	624
3	625	15624
4	15625	390624

From 3, it follows that the B-tree is a balanced tree. Its height, and with it the longest path's length, has an upper bound of, roughly, $2 * \log k$, where k is the number of elements and the logarithm is taken to the base N and rounded up to the next larger integer. Its minimal height is $\log k$ taken to the base $2N$.

On each page, space must be available for $2N$ elements and for $2N + 1$ references to descendants. Hence, N is immediately determined by the size of a page and the size of elements. In the case of Oberon, names are limited to 32 characters (bytes), and the object is a reference to the associated file (4 bytes). Each descendant pointer takes 4 bytes, and the page size is given by the sector size (1024) minus the number of bytes needed to store m (2 bytes). Hence

```
N = ((1024 - 2 - 4) DIV (32 + 4 + 4)) DIV 2 = 12
```

A B-tree of height h and order 12 may contain the following minimal and maximal number of elements:

It follows that the height of the B-tree will never be larger than 4, if the disk has a capacity of less than about 400 Mbyte, and assuming that each file occupies a single 1K sector. It is rarely larger than 3 in practice.

The definition of module FileDir shows the available directory operations. Apart from the procedures Search, Insert, Delete, and Enumerate, it contains some data definitions, and it should be considered as the non-public part of the FS's interface.

```
DEFINITION FileDir;
IMPORT SYSTEM, Kernel;
CONST
FnLength = 32;
(*max length of file name*)
SecTabSize = 64; (*no. of entries in primary table*)
ExTabSize = 12;
SectorSize = 1024;
IndexSize = SectorSize DIV 4;
(*no. of entries in index sector*)
HeaderSize = 352;
DirRootAdr = 29;
DirPgSize = 24;
(*max no. of elements on page*)
TYPE DiskAdr = INT;
FileName = ARRAY FnLength OF CHAR;
SectorTable = ARRAY SecTabSize OF DiskAsr;
```

```
ExtensionTable = ARRAY ExTabSize OF DiskAdr;
EntryHandler = PROC (name: FileName; sec: DiskAdr; VAR continue: BOO
FileHeader = RECORD (*first page of each file on disk*)
mark: INT;
name: FileName;
aleng, bleng, date: INT;
ext: ExtensionTable;
sec: SectorTable
END ;
IndexSector = RECORD (Kernel.Sector)
x: ARRAY IndexSize OF LONGINT;
END ;
DataSector = ARRAY SectorSize OF BYTE;
DirEntry = RECORD
name: FileName;
adr, p: DiskAdr
END ;
DirPage = RECORD
mark: INT;
m: INT; (*no. of elements on page*)
p0: DiskAdr;
e: ARRAY DirPgSize OF DirEntry;
END ;
PROC Search(name: FileName; VAR fad: DiskAdr);
PROC Insert(name: FileName; fad: DiskAdr);
PROC Delete(name: FileName; VAR fad: DiskAdr);
PROC Enumerate(prefix: ARRAY OF CHAR; proc: EntryHandler);
END FileDir.
```

Procedures Search, Insert, and Delete represent the typical operations performed on a directory. Efficiency of the first operation is of primary importance. But the B-tree structure also guarantees efficient insertion and deletion, although the code for these operations is complex. Procedure Enumerate is used to obtain excerpts of the directory. The programmer must guarantee that no directory changes are performed by the parametric procedure of Enumerate.

As in the presentation of module Files, we first discuss a version that uses main storage rather than a disk for the directory. This allows us to concentrate on the algorithms for handling the directory, leaving out the additional complications due to the necessity to read pages (sectors) into main store for selective updating and of restoring them onto disk. In particular, we point out the definitions of the data types for B-tree nodes, called DirPage, and elements, called DirEntry. The component E.p of an entry E points to the page in which all elements (with index k) have keys E.p.e[k].name > E.name. The pointer p.po points to a page in which all elements have keys p.po.e[k].name < p.e[0].name. We can visualize these conditions by Fig. ??, where names have been replaced by integers as keys.

Procedure *Search* starts by inspecting the root page. It performs a binary search among its elements, according to the following algorithm. Let $e[0 \ldots m-1]$ be the ordered keys and x the search argument.

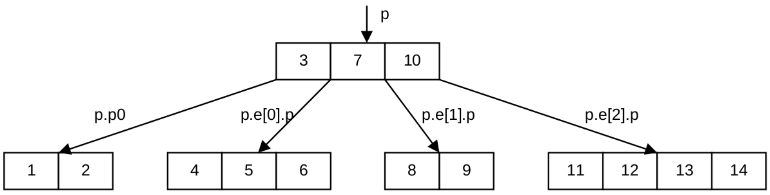


Figure 23: Example of a B-tree of order 2

```
L := 0; R := m;
WHILE L < R DO
i := (L+R) DIV 2;
IF x <= e[i] THEN R := i ELSE L := i + 1 END
END;
IF (R < m) & (x = e[R]) THEN found END
```

The invariant is

$$e[L-1] < x \leq e[R]$$

If the desired element is not found, the search continues on the appropriate descendant page, if there is one. Otherwise the element is not contained in the tree.

Procedures insert and delete use the same algorithm for searching an element within a page. However, they use recursion instead of iteration to proceed along the search path of pages. We recall that the depth of recursion is at most four. The reason for the use of recursion is that it facilitates the formulation of structural changes, which are performed during the "unwinding" of recursion, i.e. on the return path. First, the insertion point (respectively the position of the element to be deleted) is searched, and then the element is inserted (deleted).

Upon insertion, the number of elements on the insertion page may become larger than $2N$, violating B-tree condition 1. This situation is called page overflow. The invariant must be reestablished immediately. It could be achieved by moving one element from either end of the array e onto a neighbouring page. However, we choose not to do this, and instead to split the overflowing page into two pages immediately. The process of a page split is visualized by Fig **??**, in which we distinguish between three cases, namely $R < N$, $R = N$, and $R > N$, where R marks the insertion point. a denotes the overflowing, b the new page, and u the inserted element. The $2N + 1$ elements ($2N$ from the full page a, plus the one element u to be inserted) are equally distributed onto pages a and b. One element v is pushed up in the tree. It must be inserted in the ancestor page of a. Since that page obtains an additional descendant, it must also obtain an additional element in order to maintain B-tree rule 2.

A page split may thus propagate, because the insertion of element v in the ancestor page may require a split once again. If the root page is full, it is split too, and the emerging element v is inserted in a new root page containing a single

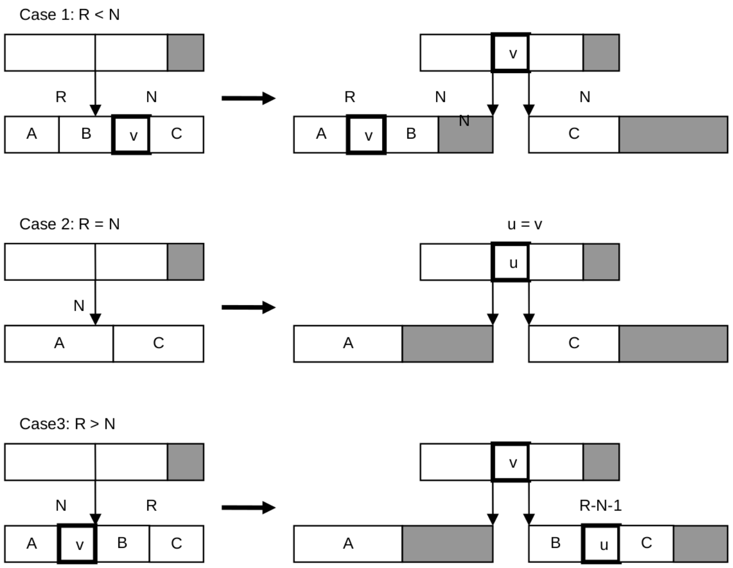


Figure 24: Page split when inserting element u

element. This is the only way in which the height of a B-tree can increase.

When an element is to be deleted, it cannot simply be removed, if it resides on an internal page. In this case, it is first replaced by another element, namely one of the two neighbouring elements on a leaf page, i.e. the next smaller (or next larger) element, which is always on a leaf page. In the presented solution, the replacing element is the largest on the left subtree (see procedure del). Hence, the actual deletion always occurs on a leaf page.

Upon deletion, the number of elements in a page may become less than N, violating invariant 1. This event is called page underflow. Since restructuring the tree is a relatively complicated operation, we first try to reestablish the invariant by borrowing an element from a neighbouring page. In fact, it is reasonable to borrow several elements, and thereby to decrease the likelihood of an underflow on the same page upon further deletions. The number of elements that could be taken from the neighbouring page b is $b.m - N$. Hence we will borrow

```
k = (b.m - N + 1) DIV 2
```

elements. The process of page balancing then distributes the elements of the underflowing and its neighbouring page equally to both pages (see procedure underflow). However, if (and only if) the neighbouring page has no elements to spare, the two pages can and must be united. This action, called page merge, places the $N - 1$ elements from the underflowing page, the N elements from the neighbouring page, plus one element from the ancestor page onto a single page of size $2N$. One element must be taken from the ancestor page, because that page loses one descendant and invariant rule 2 must be maintained. The events of page balancing and merging are illustrated in Fig ??. a is the underflowing page, b its neighbouring page, and c their ancestor; s is the position in the ancestor page of (the pointer to the) underflowing page a. Two cases are distinguished,

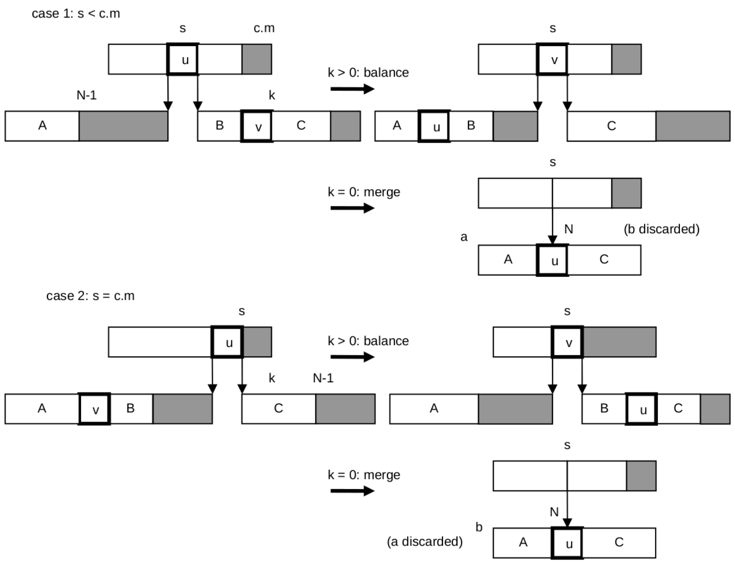


Figure 25: Page balancing and merging when deleting element

namely whether the underflowing page is the rightmost element ($s = c.m$) or not (see procedure underflow).

Similarly to the splitting process, merging may propagate, because the removal of an element from the ancestor page may again cause an underflow, and perhaps a merge. The root page underflows only if its last element is removed. This is the only way in which the B-tree's height can decrease.

```
MODULE BTree;
IMPORT Texts, Oberon;
CONST N = 3;
TYPE Page = POINTER TO PageRec;
```

```
Entry = RECORD
key, data: INT;
p: Page
END ;
PageRec = RECORD
m: INT; (*no. of entries on page*)
p0: Page;
e: ARRAY 2*N OF Entry
END ;
VAR root: Page; W: Texts.Writer;
PROC search(x: INT; VAR p: Page; VAR k: INT);
VAR i, L, R: INT; found: BOOL; a: Page;
BEGIN a := root; found := FALSE;
WHILE (a $\neq$ NIL) & ~found DO
L := 0; R := a.m; (*binary search*)
WHILE L < R DO
i := (L+R) DIV 2;
IF x <= a.e[i].key THEN R := i ELSE L := i+1 END
END ;
IF (R < a.m) & (a.e[R].key = x) THEN found := TRUE
ELSIF R = 0 THEN a := a.p0 ELSE a := a.e[R-1].p
END
END ;
p := a; k := R
END search;
PROC insert(x: INT; a: Page; VAR h: BOOL; VAR v: Entry);
(*a $\neq$ NIL. Search key x in B-tree with root a; if found
Otherwise insert new item with key x. If an entry is to be p
```

```
assign it to v. h := "tree has become higher"∗)
VAR i, L, R: INT;
b: Page; u: Entry;
BEGIN (∗a $\neq$ NIL & ~h∗)
L := 0; R := a.m; (∗binary search∗)
WHILE L < R DO
i := (L+R) DIV 2;
IF x <= a.e[i].key THEN R := i ELSE L := i+1 END
END ;
IF (R < a.m) & (a.e[R].key = x) THEN (∗found∗) INC(a.e[R].data)
ELSE (∗item not on this page∗)
IF R = 0 THEN b := a.p0 ELSE b := a.e[R-1].p END ;
IF b = NIL THEN (∗not in tree, insert∗)
u.p := NIL; h := TRUE; u.key := x
ELSE insert(x, b, h, u)
END ;
IF h THEN (∗insert u to the left of a.e[R]∗)
IF a.m < 2∗N THEN
h := FALSE; i := a.m;
WHILE i > R DO DEC(i); a.e[i+1] := a.e[i] END ;
a.e[R] := u; INC(a.m)
ELSE NEW(b); (∗overflow; split a into a,b and assign the middle ent
IF R < N THEN (∗insert in left page a∗)
i := N-1; v := a.e[i];
WHILE i > R DO DEC(i); a.e[i+1] := a.e[i] END ;
a.e[R] := u; i := 0;
WHILE i < N DO b.e[i] := a.e[i+N]; INC(i) END
ELSE (∗insert in right page b∗)
```

```
DEC(R, N); i := 0;
IF R = 0 THEN v := u
ELSE v := a.e[N];
WHILE i < R-1 DO b.e[i] := a.e[i+N+1]; INC(i) END ;
b.e[i] := u; INC(i)
END ;
WHILE i < N DO b.e[i] := a.e[i+N]; INC(i) END
END ;
a.m := N; b.m := N; b.p0 := v.p; v.p := b
END
END
END
END insert;
PROC underflow(c, a: Page; s: INT; VAR h: BOOL);
(∗a = underflowing page, c = ancestor page,
s = index of deleted entry in c∗)
VAR b: Page;
i, k: INT;
BEGIN (∗h & (a.m = N-1) & (c.e[s-1].p = a) ∗)
IF s < c.m THEN (∗b := page to the right of a∗)
b := c.e[s].p; k := (b.m-N+1) DIV 2; (∗k = nof items availal
a.e[N-1] := c.e[s]; a.e[N-1].p := b.p0;
IF k > 0 THEN (∗balance by moving k-1 items from b to a∗) i
WHILE i < k-1 DO a.e[i+N] := b.e[i]; INC(i) END ;
c.e[s] := b.e[k-1]; b.p0 := c.e[s].p;
c.e[s].p := b; DEC(b.m, k); i := 0;
WHILE i < b.m DO b.e[i] := b.e[i+k]; INC(i) END ;
a.m := N-1+k; h := FALSE
```

```
ELSE (∗merge pages a and b, discard b∗) i := 0;
WHILE i < N DO a.e[i+N] := b.e[i]; INC(i) END ;
i := s; DEC(c.m);
WHILE i < c.m DO c.e[i] := c.e[i+1]; INC(i) END ;
a.m := 2∗N; h := c.m < N
END
ELSE (∗b := page to the left of a∗) DEC(s);
IF s = 0 THEN b := c.p0 ELSE b := c.e[s-1].p END ;
k := (b.m-N+1) DIV 2; (∗k = nof items available on page b∗)
IF k > 0 THEN i := N-1;
WHILE i > 0 DO DEC(i); a.e[i+k] := a.e[i] END ;
i := k-1; a.e[i] := c.e[s]; a.e[i].p := a.p0;
(∗move k-1 items from b to a, one to c∗) DEC(b.m, k);
WHILE i > 0 DO DEC(i); a.e[i] := b.e[i+b.m+1] END ;
c.e[s] := b.e[b.m]; a.p0 := c.e[s].p;
c.e[s].p := a; a.m := N-1+k; h := FALSE
ELSE (∗merge pages a and b, discard a∗)
c.e[s].p := a.p0; b.e[N] := c.e[s]; i := 0;
WHILE i < N-1 DO b.e[i+N+1] := a.e[i]; INC(i) END ;
b.m := 2∗N; DEC(c.m); h := c.m < N
END
END
END underflow;
PROC delete(x: INT; a: Page; VAR h: BOOL);
(∗search and delete key x in B-tree a; if a page underflow arises,
balance with adjacent page or merge; h := "page a is undersize"∗)
VAR i, L, R: INT; q: Page;
PROC del(p: Page; VAR h: BOOL);
```

```
VAR k: INT; q: Page; (*global a, R*)
BEGIN k := p.m-1; q := p.e[k].p;
IF q $\neq$ NIL THEN del(q, h);
IF h THEN underflow(p, q, p.m, h) END
ELSE p.e[k].p := a.e[R].p; a.e[R] := p.e[k];
DEC(p.m); h := p.m < N
END
END del;
BEGIN (*a $\neq$ NIL*)
L := 0; R := a.m; (*binary search*)
WHILE L < R DO
i := (L+R) DIV 2;
IF x <= a.e[i].key THEN R := i ELSE L := i+1 END
END ;
IF R = 0 THEN q := a.p0 ELSE q := a.e[R-1].p END ;
IF (R < a.m) & (a.e[R].key = x) THEN (*found*)
IF q = NIL THEN (*a is leaf page*)
DEC(a.m); h := a.m < N; i := R;
WHILE i < a.m DO a.e[i] := a.e[i+1]; INC(i) END
ELSE del(q, h);
IF h THEN underflow(a, q, R, h) END
END
ELSE delete(x, q, h);
IF h THEN underflow(a, q, R, h) END
END
END delete;
PROC Search*(key: INT; VAR data: INT);
BEGIN search(key, root, data)
```

```
END Search;
PROC Insert∗(key: INT; VAR data: INT);
VAR h: BOOL; u: Entry; q: Page;
BEGIN h := FALSE; u.data := data; insert(key, root, h, u);
IF h THEN (∗insert new base page∗)
q := root; NEW(root);
root.m := 1; root.p0 := q; root.e[0] := u
END
END Insert;
PROC Delete∗(key: INT);
VAR h: BOOL;
BEGIN h := FALSE; delete(key, root, h);
IF h THEN (∗base page size underflow∗)
IF root.m = 0 THEN root := root.p0 END
END
END Delete;
BEGIN NEW(root); root.m := 0
END BTree.
```

The B-tree is also a highly appropriate structure for enumerating its elements, because during the traversal of the tree each page is visited exactly once, and hence needs to be read (from disk) exactly once too. The traversal is programmed by the procedure Enumerate and uses recursion. It calls the parametric procedure proc for each element of the tree. The type of proc specifies as parameters the name and the (address of) the enumerated element. The third parameter continue is a Boolean VAR-parameter. If the procedure sets it to FALSE, the process of enumeration will be aborted.

Enumerate is used for obtaining listings of the names of registered files. For this purpose, the actual procedure substituted for proc merely enters the given name in a text and ignores the address (sector number) of the file, unless it requires special file information such as the file's size or creation date.

The set of visited elements can be restricted by specifying a string which is to be a prefix to all enumerated names. The least name with the specified prefix is directly searched and is the name (key) of the first element enumerated. The process then proceeds up to the first element whose name does not have the given prefix. Thereby, the process of obtaining all elements whose key has a given prefix avoids traversal of the whole tree, resulting in a significant speedup. If the prefix is the empty string, the entire tree is traversed.

The principle behind procedure Enumerate is shown by the following sketch, where we abstract from the B-tree structure and omit consideration of prefixes:

```
PROC Enumerate(
proc: PROC (name: FileName; adr: INT; VAR continue: BOOL));
VAR continue: BOOL; this: DirEntry;
BEGIN continue := TRUE; this := FirstElement;
WHILE continue & (this $\neq$ NIL) DO
```

```
proc(this.name, this.adr, continue); this := NextEntry(this)
END
END Enumerate
```

From this sketch we may conclude that during the process of traversal the tree structure must not change, because the function NextEntry quite evidently relies on the structural information stored in the elements of structure itself. Hence, the actions of the parametric procedure must not affect the tree structure. Enumeration must not be used, for example, to delete a given set of files. In order to prevent the misuse of the indispensible facility of element enumeration, the interface of FileDir is not available to users in general.

The handling of the directory stored on disk follows exactly the same algorithms. The accessed pages are fetched from the disk as a whole (each page fits onto a single disk sector) and stored in buffers of type DirPage, from where individual elements can be accessed. In principle, these buffers can be local to procedures insert and delete. A single buffer is allocated globally, namely the one used by procedure Search. The reason for this exception is not only that iterative searching requires one buffer only, but because procedure Files.Old and in turn Search may be called when the processor is in the supervisor mode and hence uses the system- (instead of the user-) stack, which is small and would not accommodate sector buffers.

Naturally, an updated page needs to be stored back onto disk. Omission of sector restoration is a programming error that is very hard to diagnose, because some parts of the program are executed very rarely, and hence the error may look sporadic and mistakenly be attributed to malfunctioning hardware.

Oberon's file directory represents a single, ordered set of name-file pairs. It is therefore also called a flat directory. Its internal tree structure is not visible to the outside. In contrast, some file systems use a directory with a visible tree structure, notably UNIX. In a search, the name (key) guides the search path; the name itself displays structure, in fact, it is a sequence of names (usually separated by slashes or periods). The first name is then searched in the root directory, whose descendants are not files but subdirectories. The process is repeated, until the last name in the sequence has been used (and hopefully denotes a file).

Since the search path length in a tree increases with the logarithm of the number of elements, any subdivision of the tree inherently decreases performance since $\log(m + n) < \log(m) + \log(n)$ for any $m, n > 1$. It is justified only if there exist sets of elements with common properties. If these property values are stored once, namely in the subdirectory referencing all elements with common property values, instead of in every element, not only a gain in storage economy results, but possibly also in accesses which depend on those properties. The common properties are typically an owner's name, a password, and access rights (read or write protection), properties that primarily have significance in a multi-user environment. Since Oberon was conceived explicitly as a single-user system, there is little need for such facilities, and hence a flat directory offers the best performance with a simple implementation.

Every directory operation starts with an access to the root page. An obvious measure for improving efficiency is to store the root page "permanently" in main store. We have chosen not to do this for four reasons:

1. If the hardware fails, or if the computer is switched off before the root page is copied to disk, the file directory will be inconsistent with severe consequences.

2. The root page has to be treated differently from other pages, making the program more complex.

3. Directory accesses do not dominate the computing process; hence, any improvement would hardly be noticeable in overall system performance. The payoff for the added complexity would be small.

4. Procedure *Init* is called upon system initialization in order to construct the sector reservation table. Therefore, this procedure (and the module) must be allowed to refer to the structure of a file's sector table(s), which

is achieved by placing its definitions into the module *FileDir* (instead of *Files*). Unlike *Enumerate*, *Init* traverses the entire B-tree. The sector numbers of files delivered by *TraverseDir* are entered into a buffer. When full, the entries are sorted, whereafter each file's head sector is read and the sectors indicated in its sector table are marked as reserved. The sorting speeds up the reading of the header sectors considerably. Nevertheless, the initialization of the sector reservation table clearly dominates the start-up time of the computer. For a file system with 10,000 files it takes in the order of 15s to record all files.

7.5 THE TOOLBOX OF FILE UTILITIES

We conclude this chapter with a presentation of the commands which constitute the toolbox for file handling. These commands are contained in the tool module *System*, and they serve to copy, rename, and delete files, and to obtain excerpts of the file directory. Procedures *CopyFiles*, *RenameFiles*, and *DeleteFiles* all follow the same pattern. The parameter text is scanned for file names, and for each operation a corresponding procedure is called. If the parameter text contains an arrow, it is interpreted as a pointer to the most recent text selection which indicates the file name. In the cases of *CopyFiles* and *RenameFiles* which require two names for a single action, the names are separated by "$\Rightarrow$" indicating the direction of the copy or rename actions.

Procedure *Directory* serves to obtain excerpts of the file directory. It makes use of procedure *FileDir.Enumerate*. The parametric procedure *List* tests whether or not the delivered name matches the pattern specified by the parameter of the directory command. If it matches, the name is listed in the text of the viewer opened in the system track. Since the pattern may contain one or several asterisks (wild cards), the test consists of a sequence of searches of the pattern parts (separated by the asterisks) in the file name. In order to reduce the number of calls of *List*, *Enumerate* is called with the first part of the pattern as parameter prefix. Enumeration then starts with the least name having the specified prefix, and terminates as soon as all names with this prefix have been scanned.

If the specified pattern is followed by an option directive "!", then not only file names are listed, but also the listed files' creation date and length. This requires that not only the directory sectors on the disk are traversed, but that additionally for each listed file its header sector must be read. The two procedures use the global variables *pat* and *diroption*.

REFERENCES

1. R. Bayer and E. M. McCreight. Organization and maintenance of large ordered indexes. Acta Informatica, 1, 3, (1972), 173-189.
2. D. Comer. The ubiquitous B-tree. ACM Comp Surveys, 11, 2, (June 1979), 121-137.

8 STORAGE LAYOUT AND MANAGEMENT

Centralized RM (CRM) is a crucial property of Oberon.

ITS ADVANTAGE is that replication of management algorithms and a premature partitioning of resources are avoided.

THE DISADVANTAGE is that management algorithms are fixed once and forever and remain the same for all applications.

The success of a CRM therefore depends crucially on its flexibility and its efficient implementation. This chapter presents the scheme and the algorithms governing main storage in Oberon.

8.1 LAYOUT AND RUN-TIME ORGANIZATION

The storage layout of Oberon is determined by the structure of code and data typical in the use of modular, high-level PLs, and in particular of the language Oberon. It suggests the subdivision of storage into 3 areas:

MODULE SPACE Each module specifies procedures (code) and global (static) variables. Its initialization can be regarded as a procedure implicitly called upon loading. Space must be allocated for code and data in *blocks* by the loader. Typically, modules contain very few global variables, hence the space size is primarily determined by code.

WORKSPACE (STACK) Execution of every command invokes a sequence of procedures, each of which uses (possibly zero) parameters and local variables. Since procedure calls and completions follow a strict first-in last-out order, the stack is the uniquely suited strategy for local storage allocation. Deallocation upon completion of a procedure is achieved by merely resetting the pointer identifying the top of the stack. Since this operation is performed by a single instruction, it costs virtually no time. Because Oberon is a single-process system, a single stack suffices. Furthermore, after completion of a command, the stack is empty. This fact will be important in simplifying the scheme for reclamation of dynamically allocated space.

DYNAMIC SPACE (HEAP) Apart from global (static) variables, and local (stack-allocated) variables, a program may refer to anonymous variables referenced through pointers. Such variables are allocated truly dynamically through calls of an explicit `NEW` operation. These variables are allocated in the so-called *heap*. Their deallocation is "automatic", when free storage is needed and they are no longer referenced from any of the loaded modules. This process is called *garbage collection* (GC). Every record allocated in the heap contains a (hidden) pointer to the type descriptor, called *type tag*, used by GC.

Unfortunately, the number of distinct spaces is larger than 2. If it were 2, no arbitrary size limitation would be necessary; merely the sum of their sizes would be inherently limited by the size of the store. In the case of 3 spaces, arbitrarily determined size limits are unavoidable. Address mapping hardware can alleviate (and delegate) this problem using a virtual address space which is so large that limits will hardly ever be reached.

Such a scheme is implemented by tables mapping virtual into physical addresses, requiring multiple memory accesses for every reference. Of course, the need for a double or a triple access for every memory reference is avoided by a translation cache in the (hardware) unit. Nevertheless, a decrease in performance is unavoidable for each cache miss. Furthermore, an additional subcycle is required for every access in order to look up the cached translation table. Without a virtual address scheme, each module block must consist of an integral number of physically adjacent pages. Holes generated by the release of modules must be reused. We employ the simple scheme of marking the released space as a hole, and of allocating a new block in the 1st hole encountered that is large enough (1st-fit strategy). Considering the relative infrequency of module releases, efforts to improve the strategy are not worth the resulting added complexity.

It is remarkable that the code for module allocation and release without virtual addressing is only marginally more complicated than with it. The only remaining advantages of an MMU are a better storage utilization, because no holes occur (a negligible advantage), and that inadvertent references to unloaded modules, e.g. via installed procedures, lead to an invalid address trap.

It is worth recalling that the concept of address mapping was introduced as a requirement for virtual memory implemented with disks as backing store, where pages could be moved into the background in order to obtain space for newly required pages, and could then be retrieved from disk on demand, i.e. when access was requested. This scheme is called *demand paging*. It is not used in Oberon, and one may fairly state that demand paging has lost its significance with the availability of large, primary stores.

Experience in the use of the RISC predecessor Ceres leads to the conclusion that whereas address translation through an MMU was an essential feature for multi-user OSes, it constitutes a dispensible overkill for single-user workstations. The fact that modern semiconductor technology made it possible to integrate the entire translation and caching scheme into a single chip, or even into the processor itself, led to the hiding (and ignoring) of the scheme's considerable complexity. Its side effects on execution speed are essentially unpredictable. This makes systems with MMU virtually unusable for applications with tight real-time constraints. The RISC processor does indeed not feature an address mapping unit.

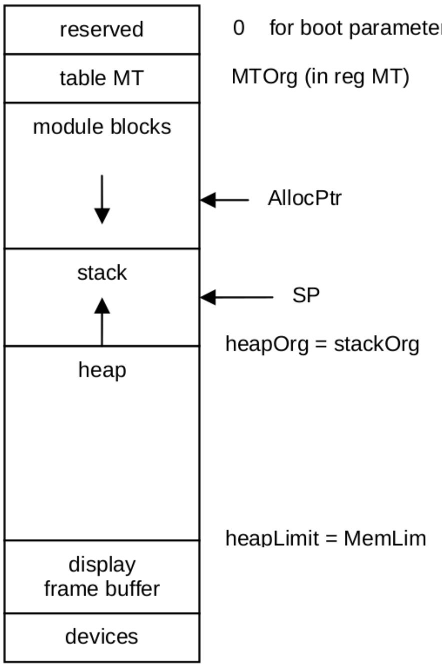


Figure 26: Storage layout

The RISC processor features 16 registers (of 32 bits). R0 - R11 are for expression evaluation. R12 - R15 have fixed, system-wide usage:

R12	address of the module table MT (typically constant)
R13	base address for variables in the current module SB (static base)
R14	stack pointer SP
R15	return address LNK (fixed by RISC's BL instruction)

The used memory layout is shown in Figure ??.

8.2 HEAP MANAGEMENT

The term *dynamic storage* is used here for all variables that are allocated neither statically (global variables) nor on the stack (local variables), but through invocation of the intrinsic procedure `NEW` in the heap. Such variables are anonymous and are referenced exclusively via pointers.

The space allocated to such dynamic variables becomes free and reusable as soon as the last reference to it vanishes. This event is hard, and in multiprocess systems even impossible to detect. The usual remedy is to ignore it and instead to determine the accessibility of all allocated variables (records, objects) only at the time when more storage space is needed. This process is then called *garbage collection*.

Oberon does not provide an explicit deallocation procedure allowing the programmer to signal that a variable will no longer be referenced. The 1st reason for this omission is that usually a programmer would not know when to call for deallocation. And 2ndly, this "hint" could not be taken as trustworthy. An erroneous deallocation, i.e. one occurring when there still exist references to the object in question, could lead to a multiple allocation of the same space with disastrous consequences. Hence, it appears wise to fully rely on system management to determine which areas of the store are truly reusable.

Before discussing the scheme for storage reclamation, which is the primary subject of heap management, we turn our attention to the problem of allocation, i.e. the implementation of procedure `NEW`. The simplest solution is to maintain a list of free blocks and to pick the 1st one large enough. This strategy leads to a relatively large fragmentation of space and produces many small elements, particularly in the 1st part of the list. We therefore employ a somewhat more refined scheme and maintain 4 lists of available space. 3 of them contain pieces of fixed size, namely 32, 64, and 128 bytes. The 4th list contains pieces whose size is any multiple of 256. We note that the choice of the values permits the merging of any 2 contiguous elements into an element of the next list. This scheme keeps fragmentation, i.e. the emergence of small pieces in large numbers, reasonably low with minimal effort. The body of procedure `NEW` consists of relatively few instructions, and typically only a small fraction of them needs to be executed.

The statement `NEW(p)` is compiled into an instruction sequence assigning the address of pointer variable *p* to a fixed register (`R0`) and the type tag to another register (`R1`). The type tag is a pointer to a type descriptor containing information required by GC. This includes the size of the space occupied and now to be allocated. The effect of `NEW` is the assignment of the address of the allocated block to *p*, and the assignment of the tag to a prefix of the block. (see Fig. ??)

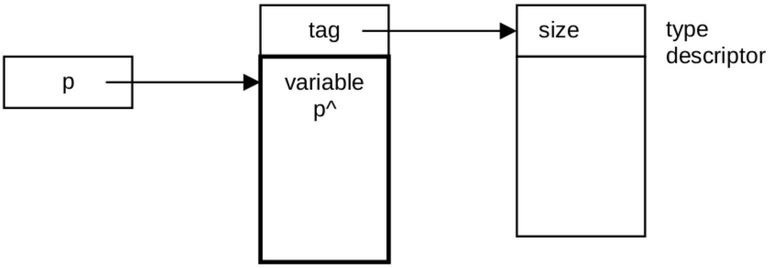


Figure 27: Heap allocation of dynamic variable $p\hat{}$ by NEW(p)

In conclusion, we emphasize that this scheme makes the allocation of an object very efficient. Nevertheless, it is considerably more costly than that of a variable explicitly declared and therefore allocated either globally or on the stack. We now turn to the problem of storage reclamation or GC. There exist 2 essentially different schemes:

1^{st}, the reference counting, and

2^{nd}, the mark-scan schemes.

In the former, every object carries a (hidden) *reference count*, indicating the number of existing references to it. The scheme works as follows:

1. `NEW(p)` initializes the reference count of *p^* to 1.
2. An assignment `q := p` decrements the reference count of *q^* by 1, performs the assignment, then increments the reference count of *p^* by 1.

When a reference count reaches zero, the element is linked into the free list. There are 2 disadvantages inherent in this approach:

1. the non-negligible overhead in pointer assignments.
2. circular data structures never become recognized as free, even if no external references point to their elements.

Oberon employs the 2nd scheme which involves no hidden operations like the 1st one, but relies on a process initiated when free storage has become scarce and more is needed. It consists of 2 phases:

MARK PHASE all referenced and therefore still accessible elements are marked.

SCAN PHASE their unmarked complement is released.

Its primary disadvantage is that the process may be started at moments unpredictable to the system's user. During the process, the computer then appears to be blocked. It follows that an interactive system using mark-scan GC must guarantee that the process is sufficiently fast in order to be hardly noticeable. Modern processors make this possible, even with large main stores. Nevertheless, finding all accessible nodes in an entire computer system within, say, a second appears to be a formidable feat.

We recognize that the mark phase essentially is a tree traversal, or rather a forest traversal. The roots of the trees are all named pointer variables in existence. We shall postpone the question of how these roots are to be found, and 1st present a quick tutorial about tree traversal. In general, nodes of the traversed structure may contain many pointers (branches). We shall, however, 1st restrict our attention to a binary tree, because the essential problem and its solution can be explained better in this way.

The essential problem alluded to is that of storage utilization by the traversal algorithm itself. Typically, information about the nodes already visited must be retained, be it explicitly, or implicitly as in the case of use of recursion. Such a strategy is plainly unacceptable, because the amount of storage needed is unpredictable and may become very large, and because GC is typically initiated just when more storage is unavailable. The task may seem impossible, yet a solution lies in the idea of inverting pointers along the path traversed, thus keeping the return path open. It is embodied in the following procedure, whose task is to traverse the tree given by the parameter root, and to mark every node. Mark values are assumed to be initially 0. Let the data structure be defined by the types

```
Ptr = POINTER TO Node;
Node = RECORD m: INT; L, R: Ptr END;
```

and the algorithm by the procedure

```
PROC traverse(root: Ptr);
  VAR p, q, r: Ptr;
BEGIN p := root; q := root;
  REPEAT (*p # NIL*) INC(p.m); (*mark*)
    IF p.L # NIL THEN (*pointer rotation*)
      r := p.L; p.L := p.R; p.R := q; q := p; p := r
    ELSE
      p.L := p.R; p.R := q; q := NIL
    END
  UNTIL p = q
END traverse
```

We note that only 3 local variables are required, independent of the size of the tree to be traversed. The 3rd, *r*, is in fact merely an auxiliary variable to perform the rotation of values

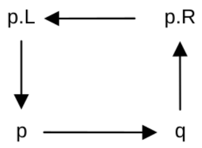


Figure 28: Rotation of four pointers

$p.L$, $p.R$, q, and p as shown in Fig. ??. A snapshot of a tree traversal is shown in Fig. ??.

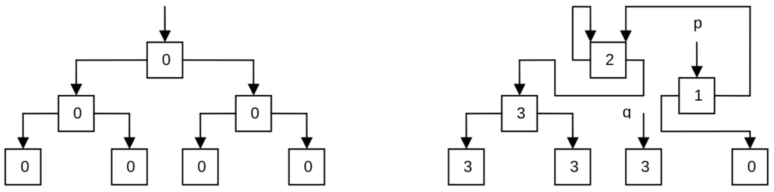


Figure 29: Tree traversal (original at left, snapshot at right)

The pair p, q of pointers marks the position of the process. The algorithm traverses the tree in a left to right, depth 1st fashion. When it returns to the root, all nodes have been marked. How are these claims convincingly supported? The best way is by analyzing the algorithm at an arbitrary node. We start with the hypothesis H that, given the initial state P, the algorithm will reach state Q, (see Fig ??). State Q differs from P by the node and its descendants B and C having been marked, and by an exchange of p and q. We now apply the algorithm to state P, assuming that B and C are not empty. The process is illustrated in Fig ??. P0 stands for P in Fig. ??. Transitions $P0 \rightarrow P1$, $P2 \rightarrow P3$, and $P4 \rightarrow P5$ are the direct

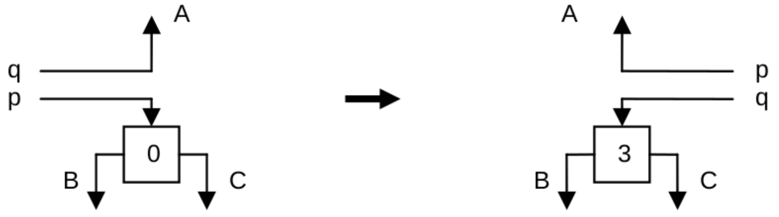


Figure 30: Transition from state P to Q

results of applying the pointer rotation as specified by the sequence of five assignments in the algorithm. Transitions $P1 \rightarrow P2$ and $P3 \rightarrow P4$ follow from the hypothesis H being applied to the states P1 and P3: subtrees are marked and p, q interchanged. We note in passing that the node is visited 3 times. Progress is recorded by the mark value which is incremented from 0 to 3.

Fig. ??. demonstrates that, if H holds for steps $P1 \rightarrow P2$ and $P3 \rightarrow P4$, then it also holds for step $P1 \rightarrow P5$, which visits the subtree p. Hence, it also holds for the step root $\rightarrow$ root, which traverses the entire tree.

This proof by recursion relies on the algorithm performing correct transitions also in the case of p.L being NIL, i.e. B being the empty tree. In this case, state P1 is skipped; the 1st transition is $P0 \rightarrow P2$ (see Fig ??). If p.L is again NIL, i.e. also C is empty, the next transition is $P2 \rightarrow P4$. This concludes the demonstration of the algorithm's correctness.

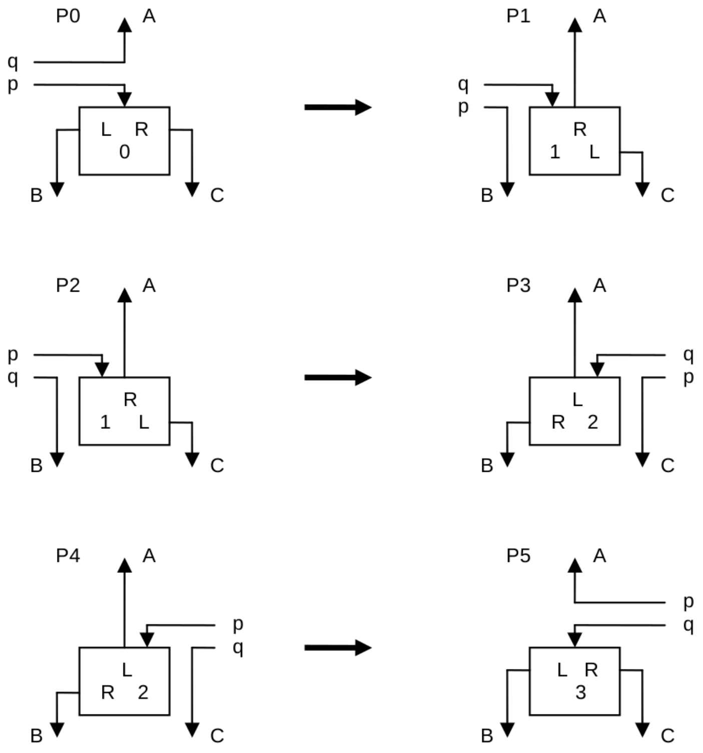


Figure 31: Transitions from P0 to P5, visiting nodes 3 times

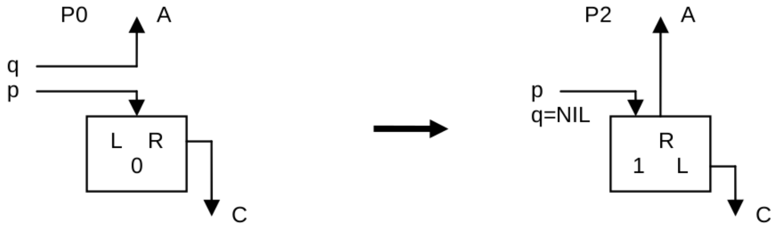


Figure 32: Direct transition from P0 to P2, if $p.L = NIL$

We now modify the algorithm of tree traversal to the case where the structure is not confined to a binary tree, but may be a tree of any degree, i.e. each node may have any number n of descendants. For practical purposes, however, we restrict n to be in the range $0 \leq n \leq N$, and therefore may represent all nodes by the type

```
Node = RECORD m, n: INT;
          dsc: ARRAY N OF Node
       END
```

In principle, the binary tree traversal algorithm might be adopted almost without change, merely extending the rotation of pointers from p.L, p.R, q, p to p.dsc[0], ... , p.dsc[n-1], q, p. However, this would be an unnecessarily inefficient solution. The following is a more effective variant. Moreover, it caters for the case of inhomogeneous graphs, where different nodes have different numbers of descendants. The key lies in associating with every node, in addition to the tag, a 2nd private field mk. It serves 2 purposes. The 1st is as a mark, with mk > 0 indicating that the node had been visited. The 2nd is to store the address of the next descendant to be visited. The underlying data structure is shown in Fig ??. Type descriptors consist of the following fields:

size	in bytes, of the described record type,
base	a table of pointers to the base types descriptors(3 elements only)
offsets	of the descendant pointers in the described type(1 word each)

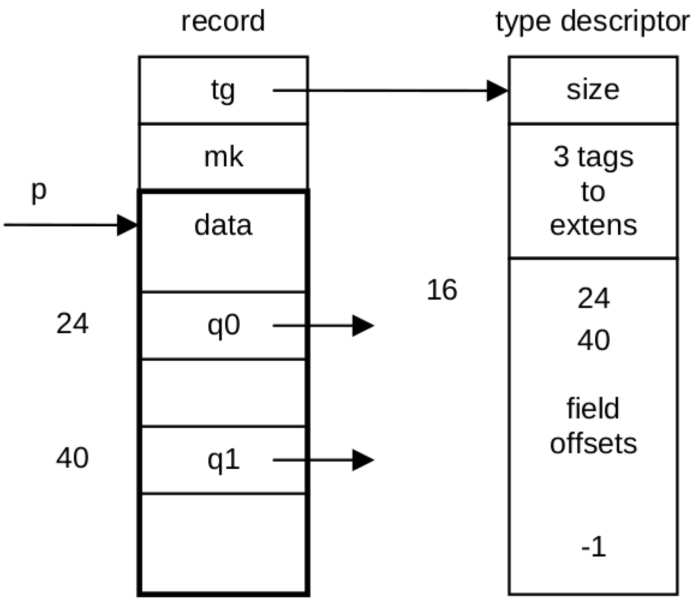


Figure 33: Record and its type descriptor

We note that the mark value, starting with zero (unmarked), is used as a counter of descendants already traversed, and hence as an index to the descendant field to be processed next. The algorithm can be applied not only to trees, but to arbitrary structures, including circular ones, if the continuation condition p # 0 (actually p >= heapOrg) is extended to (p >= heapOrg) & (offadr = 0). This causes a descendant that is already marked to be skipped. Here the array M stands for the entire memory.

```
PROC traverse(root: Ptr);
  VAR offadr, offset: INT; p, q, r: Ptr;
```

```
BEGIN p := root; q := root;
  REPEAT (*p # NIL*) offadr := p.mk; (*mark*)
    IF offadr = 0 THEN
      tag := p.tg; offadr := tag + 16
    ELSE INC(offadr, 4) END;
    p.mk := offadr; offset := M[offadr];
    IF offset # -1 THEN (*move down*)
      r := M[p+offset]; offadr := M[r-4];
      IF offadr = 0 THEN
        M[p+offset] := q; q := p; p := r
      END
    ELSE (*move up*)
      offadr := M[q-4]; offset := M[offadr]
      IF p # q THEN
        r := M[q+offset]; M[q+offset] := p;
        p := q; q := r
      END
    END
  UNTIL (p = q) & (offset = -1)
END traverse.
```

The mark is included in each record's hidden prefix. The prefix takes 2 words only; the first is used for the tag. The other is reserved for the garbage collector and used as mark and offset address. The end of the list of descendant pointers is marked by an entry with value -1. And finally, assignments involving M are expressed as

assignment	for
`SYSTEM.GET(a, x)`	$x := M[a]$
`SYSTEM.PUT(a, x)`	$M[a] := x$

The scan phase is performed by a relatively straight-forward algorithm. The heap, i.e. the storage area between HeapOrg and HeapLimit (the latter is a variable), is scanned element by element, starting at HeapOrg. Elements marked are unmarked, and unmarked elements are freed by linking them into the appropriate list of available space. As the heap may always contain free elements, the scan phase must be able to recognize them in order to skip them or merge them with an adjacent free element. For this purpose, the free elements are also considered as prefixed. The prefix serves to determine the element's size and to recognize it as free due to a special (negative) mark value. The encountered mark values and the action to be taken are:

mk value	state	action
$= 0$	unmarked	collect, mark free
> 0	marked	unmark
< 0	free	skip or merge

8.3 KERNEL

The kernel lies at the bottom of the module hierarchy. It contains the procedures for dynamic storage allocation and retrieval as described before. The procedures are `New`, `Mark`, and `Scan`. `Kernel` also contains the driver routines for the disk. They are used by modules `FileDir` and `Files`. The "disk" is actually an SD-card, a high-volume flash-RAM. It is accessed purely sequentially, byte-wise, by a standard, serial peripheral interface (SPI). Within `Kernel` a table called `SectorMap` is allocated keeping track of blocks (sectors) occupied by files. A single bit indicates, whether a sector is allocated or not. This table is accessed by the procedures `AllocSector`, `MarkSector`, and `FreeSector`. Reading and writing is done sector-wise by procedures `GetSector` and `PutSector`. Sector numbers are always a multiple of 29 for the purpose of redundancy checks.

Furthermore, the kernel contains a timer counting milliseconds and, perhaps, a real time clock, showing date and time. Clock data are packed into a single word as follows:

6	4	5	5	6	6
year	month	day	hour	minute	second

Figure 34: Encoding of date and time (year starting with 2000)

```
DEFINITION Kernel;(*NW/PR 11.4.86/27.12.95/15.5.2013*)
  CONST SectorLength = 1024;
  TYPE Sector = ARRAY SectorLength OF BYTE;
  VAR allocated, NofSectors, heapOrg, heapLim,
      stackOrg, MemSize: INT;
```

```
    PROC New(VAR ptr: INT; tag: INT);
    PROC Mark(pref: INT);
    PROC Scan;
    PROC ResetDisk;
    PROC MarkSector(sec: INT);
    PROC FreeSector(sec: INT);
    PROC AllocSector(hint: INT; VAR sec: INT);
    PROC GetSector(src: INT; VAR dest: Sector);
    PROC PutSector(dest: INT; VAR src: Sector);
    PROC Time(): INT; (*milliseconds*)
    PROC Clock(): INT;
    PROC SetClock(dt: INT);
    PROC Install(adr, procadr: INT);
    PROC Init;
  END Kernel.
```

8.4 THE STORAGE MANAGEMENT TOOLBOX

The user can obtain information about the system state and resources through its toolbox, a set of commands contained in the too module `System`. These commands are:

```
  PROC Watch;
  PROC Collect;  / n
  PROC SetClock; / year,month,day,hour,minute,second
```

Command `Watch` shows the amount of storage occupied in the heap, the number of disk sectors allocated on the disk, and the number of tasks installed. The command `Collect` allows to control the frequency of GC. The number *n* indicates how many commands are executed before the next GC.

9

DEVICE DRIVER(DD)

9.1 OVERVIEW

DDs are collections of procedures that constitute the immediate interface between hardware and software. They refer to those parts of the computer hardware that are usually called *peripheral*. Computers typically contain a system bus which transmits data among its different parts. Processor and memory are considered as its internal parts; the remaining parts, such as disk, keyboard, display, etc, are considered as external or peripheral, notwithstanding the fact that they are often contained in the same cabinet or board.

Such peripheral devices are connected to the system bus via special registers (data buffers) and transceivers (switches, buffers in the sense of digital electronics). These registers and transceivers are addressed by the processor in the same way as memory locations - they are said to be *memory-mapped* - and they constitute the hardware interface between processor bus and device. References to them are typically confined to specific driver procedures which constitute the software interface.

Drivers are inherently hardware specific, and the justification of their existence is precisely that they encapsulate these specifics and present to their clients an appropriate abstraction of the device. Evidently, this abstraction must still reflect the essential characteristics of the device, but not the details (e.g. the addresses of its interface registers).

Our justification to present the drivers connecting the Oberon with the RISC computer in detail is on the one hand the desire for completeness. But on the other hand it is also in recognition of the fact that their design represents an essential part of the engineering task in building a system. This part may look trivial from a conceptual point of view; it certainly is not so in practice. In order to reduce the number of interface types, standards have been established. The RISC computer also uses such interface standards, and we will concentrate on them in the following presentations.

The following devices are presented:

KEYBOARD Considered as a serial device delivering one byte of input data per key stroke. It is connected by a serial line according to the PS/2 and ASCII standards. The software is contained in module `Input` (9.2), and the hardware is explained in **??**.

MOUSE A pointing device delivering coordinates as well as key states. The software is also part of `Input` (9.2).

DISPLAY The interface to it is an area of memory contains the displayed information, exactly one bit per pixel for a monochrome display. This area is called *frame buffer* or bitmap. Here the size of the display area is 768 lines and 1024 dots per line, representing a raster. The software is module `Display`, which primarily consists of operations to draw frequently occurring patterns (called *raster-ops*, explained in 4.5). The actual display requires a hardware interface called a *display controller*, the connection between which and the display follows the VGA-Standard (see **??**).

DISK Our RISC computer does not use a magnetic, rotating disk to store non-volatile data. Instead, it uses an SD-card (flash-RAM). The driver is contained in module `Kernel` (8.3). The hardware is discussed in **??**.

NET In the original text, a network was presented consisting of a bus connecting many computers, based on the RS-485 standard. It was implemented by the serial communications controller Zilog 8530, operating at a frequency of 230 Kb/s. The name SCC has been retained as a generic interface, behind which the packet transport has now been re-implemented as a simple wireless network (Nordic nRF24L01 controller) in the regulation-free 2.4GHz industrial/scientific/medical (ISM) frequency band.

In all driver modules the implementation-dependent procedures `SYSTEM.PUT`, `SYSTEM.GET`, and `SYSTEM.BIT` are used to access the registers of the device interface. Their 1st parameter is the address of the register, the 2nd an expression or variable.

9.2 KEYBOARD AND MOUSE

Their driver procedures are in `Input`. `Available() > 0` signals that a character has been typed on the keyboard. The character is read by calling `Read(c)`. `Input` is restricting the data to the ASCII character set Latin-1, i.e. the values lie in the range $0X \leq c < 80X$ (7-bit values). `Mouse(keys, x, y)` yields the current keys state and coordinates of the mouse.

```
MODULE Input;
  PROC Available(): INT;
  PROC Read(VAR c: CHAR);
  PROC Mouse(VAR keys: SET; VAR x, y: INT);
  PROC SetMouseLimits(w, h: INT);
  PROC Init;
END Input.
```

The driver software accesses the keyboard via the Standard PS/2 interface represented by an 8-bit register for the received data kbdCode, and a single-bit flag indicating whether a byte had been received.

The keyboard codes received from the keyboard via a PS/2 line are not identical with the character values delivered to the Read procedure. A conversion is necessary. This is so, because modern keyboards treat all keys in the same way, including the ones for upper case, control, alternative, etc. Separate codes are sent to signal the pushing down and the release of a key, followed by another code identifying which key had been pressed or released. This requires, besides a translation table from codes to characters, a set of state variables. They are the global, boolean variables Recd, Up, Shift, Ctrl, and Ext. Procedure Peek determines whether an actual character is present, or merely a code signalling a key shift. Peek controls the state.

Procedure Mouse fetches a word from the mouse interface register and decomposes it into its components (keys state and coordinates). (kb is the bit indicating whether a code had been received from the keyboard).

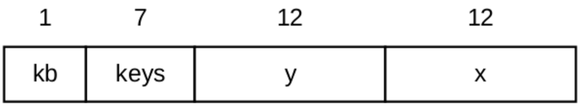


Figure 35: Format of the mouse register

9.3 THE SD-CARD (DISK)

SD-card are high-volume memory devices based on flash-store technology. They are typically organized as individually accessible blocks of 1K bytes. The driver for the SD-card is contained in module `Kernel`, which also handles allocation and reservation of blocks, here in analogy to rotating disks still called *sectors*.

```
TYPE Sector = ARRAY SectorLength OF BYTE;
PROC GetSector(src: INT; VAR dst: Sector);
PROC PutSector(dst: INT; VAR src: Sector);
PROC AllocSector(hint: INT; VAR sec: INT);
PROC MarkSector(sec: INT);
PROC FreeSector(sec: INT);
```

Data transfer is sequential and handled by procedures `ReadSD` and `WriteSD` by issuing commands. These are for transmitting a block address, for receiving, and for sending a block of data. Synchronous transmission of sequences of words follows the SPI standard, which uses 3 lines, one for data input, one for data output, and one for the clock (see also **??**). The hardware interface contains a 32-bit register. The bit-rate is 8.3 MB/s.

9.4 SERIAL ASYNCHRONOUS INTERFACE (RS 232)

The RS-232 standard serves to transmit sequences of bytes over a data line asynchronously. This implies that there is no separate clock line (see also **??**). The hardware interface contains a 10-bit register for the transmitter and one for the receiver. The data rate used here is 19,200 bit/s. A byte is sent and received over the line by the following programs.

```
CONST data = -56;
      stat = -52; (*device register addresses*)
PROC Send(x: BYTE);
BEGIN
  REPEAT UNTIL SYSTEM.BIT(stat, 1);
  SYSTEM.PUT(data, x)
END Send;

PROC Rec(VAR x: BYTE);
BEGIN
  REPEAT UNTIL SYSTEM.BIT(stat, 0);
  SYSTEM.GET(data, x)
END Rec;
```

These procedures are used in the driver module RS232 presented in **??**. This module itself is not used in the Oberon core, but it was instrumental in building the System on a host computer and downloading it. It is characterized by a very simple interface.

9.5 SERIAL COMMUNICATIONS CONTROLLER (SCC)

The interface of the driver for the network was taken over from the original design using a serial communocations controller Zilog 8530. The implementation changed totally. It was designed by Paul Reed for the wireless controller Nordic nRF24L01.

```
DEFINITION SCC;
  TYPE Header = RECORD valid: BOOL;
                  dadr, sadr, typ: BYTE;
                  len: INT; (*of data following header*)
                END;
  PROC Start(filter: BOOL);
  PROC Send(VAR head: Header; buf: ARRAY OF BYTE);
  PROC Available(): INT;
  PROC ReceiveHead(VAR head: Header);
  PROC Receive(VAR x: BYTE);
  PROC Skip(m: INT);
  PROC Stop;
END SCC.
```