CHAPTER 1

HISTORY AND MOTIVATION

The most important thing in the programming language is the name. A language will not succeed without a good name. I have recently invented a very good name and now I am looking for a suitable language.

— Donald Knuth (19??)

How could anyone diligently concentrate on his work on 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 operating systems? 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, operating systems, 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 brain-storming 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 operating system 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 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.

---

1 N. Wirth. The programming language 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. Addison- Wesley, 1992.

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 System 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.

---

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

5 M. Reiser. The Oberon System - User Guide and Programmer's Manual. Addison-Wesley, 1991.

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

CHAPTER 2

STRUCTURE OF THE SYSTEM

2.1. INTRODUCTION

In order to warrant the sizeable effort of designing and constructing an entire operating system from scratch, a number of basic concepts need to be novel. We start this chapter with a discussion of the principal concepts underlying the Oberon System and of the dominant design decisions. On this basis, a presentation of the system's structure follows. It will be restricted to its coarsest level, namely the composition and interdependence of the largest building blocks, the modules. The chapter ends with an overview of the remainder of the book. It should help the reader to understand the role, place, and significance of the parts described in the individual chapters.

The fundamental objective of an operating system is to present the computer to the user and to the programmer at a certain level of abstraction. For example, the store is presented in terms of requestable pieces or variables of a specified data type, the disk is presented in terms of sequences of characters (or bytes) called files, the display is presented as rectangular areas called viewers, the keyboard is presented as an input stream of characters, and the mouse appears as 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 the system to implement these operations and to manage them, constrained by the available resources of the underlying computer. This is commonly called resource management.

Every abstraction inherently hides details, namely those from which it abstracts. Hiding may occur at different levels. For example, the computer may allow certain parts of the store, or certain devices to be made inaccessible according to its mode of operation (user/supervisor mode), or the programming language may make certain parts inaccessible through a hiding facility inherent in its visibility rules. The latter is of course much more flexible and powerful, and 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 modularization every hope of being able to guarantee reliability and correctness vanishes. Clearly, the Oberon System was designed with the goal of establishing a modular structure on the basis of purpose-oriented abstractions. The availability of an appropriate programming language is an indispensable prerequisite, and the importance of its choice cannot be over-emphasized.

2.2. CONCEPTS

2.2.1. 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 the human eye. This occurred with the advent of the high-resolution display in the mid 1970s, which in turn had become feasible due to faster and cheaper electronic memory components. The high-resolution display 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 high-resolution display made visual output a subject of abstraction and *resource management*. In the Oberon System, the display is partitioned into *viewers*, also called *windows*, or more precisely, into *frames*, rectangular areas of the screen(s). A viewer typically consists of two frames, a title bar containing a subject name and a menu of commands, and a main frame containing some text, graphic, picture, or other object. A viewer itself is a frame; frames can be nested, in principle to any depth.

The System provides routines for generating a frame (viewer), for moving and for closing it. It allocates a new viewer at a specified place, and upon request delivers hints as to where it might best be placed. It keeps track of the set of opened viewers. This is what is called *viewer management*, in contrast to the handling of their displayed contents.

But high interactivity requires not only a high bandwidth for visual output, it demands also flexibility of input. Surely, there is no need for an equally large 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 the so-called *mouse*, a pointing device which appeared roughly at the same time as the high-resolution display.

This was by no means just a lucky coincidence. The mouse comes to fruition only through appropriate software and the high-resolution display. 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 the human eye, no great precision is required from the mouse. For example, when the user wishes to identify a certain object on the screen, such as a letter, he moves the mouse as long as required until the mapped cursor reaches the object. This stands in marked contrast to a digitizer which is supposed to deliver exact coordinates. The Oberon System relies very much on the availability of a mouse.

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 the Oberon System basically assumes three 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 the Oberon System, the left button is primarily used for *marking* a position (setting a caret), the middle button for issuing general *commands* (see below), and the right button for *selecting* displayed objects.

Recently, it has become fashionable to use overlapping windows mirroring documents being piled up on one's desk. 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 compared to overlapping windows: tiled viewers as shown in Fig. 2.1.

2.2.2. Commands.

Position-dependent commands with fixed 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, the Oberon System offers a novel and much more flexible solution which is presented in the following paragraphs. First of all we remark that a program in the common sense of a text compiled as a unit is mostly a far too large unit of action to serve as a command. Compare it, for example, with the insertion of a piece of text through a mouse command. In Oberon, the notion of a unit of action is separated from the notion of unit of compilation. The former is a command represented by a (exported) procedure, the latter is 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` that of 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.

TODO: Fig. 2.1. Oberon display with tiled viewers

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 operating systems, these objects are *files* registered in the directory, and they act as interfaces between commands. The Oberon System 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 module *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 module 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 contained in a text viewer, or a graph contained in a graphic viewer, or any other part of the marked viewer is taken as the actual parameter depends on how the command procedure 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. Such a trap 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 the cause of the trap.

2.2.3. Tasks.

From the presentations above it follows that the Oberon System 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 module `Oberon` which may be regarded as the system's heart. The two 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 compute-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 that exclude such interference. This is a significant simplification.

The essential difference between Oberon and multiprocess-systems is that in the former task switches occur between commands only, whereas in the latter a switch may be invoked after any single instruction. Evidently, the difference is one of granularity of action. Oberon's granularity is coarse, which 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 two 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 driver of an input device, in the keyboard driver as well as the network driver. 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.

2.2.4. 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 as command 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.

2.2.5. Extensibility.

An important objective in the design of the Oberon System 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 operating system 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 language Oberon 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 at the end of Section 2.3).

2.2.6. 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.

The Oberon System 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 the Oberon System, and every module is freely reusable.

2.3. THE SYSTEM'S STRUCTURE

The largest identifiable units of the system are its modules. It is therefore most appropriate to describe a system's structure in terms of its modules. As their interfaces are explicitly declared, it is also easy to exhibit their interdependence in the form of 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 Figure 2.2. 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`.

TODO: Insert Figure 2.2

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`.

Module `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. Module `Oberon` furthermore contains the two 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. 2.2. basically contains facilities for generating and editing texts, and for storing them in the file system. 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 avoiding 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.

TODO: Table here

2.4. A TOUR THROUGH THE CHAPTERS

Implementation of a system proceeds bottom-up. Naturally, because modules on higher levels are clients of those on the lower levels and cannot function without the availability of their imports. Description of a system, on the other hand, is better ordered in the top-down direction. 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 and by postponing their more detailed explanation to a later time when their need has been fully motivated. For this reason, we will proceed essentially in the top-down direction.

Chapters 3–5 describe the outer core of the system. Chapter 3 focusses on the dynamic aspects. In particular, this chapter introduces the fundamental operational units of *task* and *command*. Oberon's tasking model distinguishes the categories of *interactiv taskse* and *background tasks*. Interactive tasks are represented on the display screen by rectangular areas, so-called *viewers*. Background tasks 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 collector. Both interactive tasks and background tasks 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 operating systems. 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.

Chapter 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.

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.Po`, 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*, still in a top-down path. Chapter 6 explains the loader of program modules and motivates the introduction of the 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.

Chapter 7 is devoted to the file system, 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.

The management of memory is the subject of Chapter 8. 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.

At the lowest level of the module hierarchy we find device drivers. They are described in Chapter 9, 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 of the book, consisting of Chapters 10–15, is devoted to what may be called first applications of the basic Oberon System. These chapters are therefore independent of each other, making reference to Chapters 3–9 only.

Although the Oberon System 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 Chapter 10. 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 Chapter 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 Chapter 11. It emerges by extending the `Net` module of Chapter 10, 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.

Chapter 12 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 System. Their preparation is supported by the system's major tool, the editor. In Chapter 13 we describe another editor, one that handles graphic objects. At first, only horizontal and 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 Chapter 13 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.

Chapter 14 presents two other tools, namely one used for installing an Oberon System 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! Chapter 14 covers material that is rarely presented in the literature.

Chapter 15 is devoted to tools that are not used by the Oberon System 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 Chapter 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 third part of this book is devoted to a detailed description of the hardware. Chapter 16 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*.

Chapter 17 describes the environment in which the processor is embedded. This environment consists of the interfaces to main memory and to all external devices.

TODO: References 1. N. Wirth. The programming language 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. Addison- Wesley, 1992. ISBN 0-201-56543-9 3. N. Wirth and J. Gutknecht. The Oberon System. 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 System - User Guide and Programmer's Manual. Addison-Wesley, 1991. ISBN 0-201-54422-9 6. N. Wirth. Compiler Construction. Addison-Wesley, Reading, 1996. ISBN 0-201-40353-6

CHAPTER 3

THE TASKING SYSTEM

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 operating system. 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 the one hand, and of a personal workstation that is operated by a single user at a high degree of interactivity on the other hand.

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

3.1. THE CONCEPT OF TASK

In principle, we distinguish two categories of tasks in Oberon: *Interactive tasks* and *background tasks*. Loosely speaking, interactive tasks are bound to local regions on the display screen and to interactions with their contents while, in contrast, background tasks are 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 Chapter 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. Figure 3.1 shows a typical Oberon display screen that is divided up into seven viewers corresponding to seven simultaneously active interactive tasks.

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: INTEGER;
  handle: Handler;
  state: INTEGER;
END;
```

`X`, `Y`, `W`, `H` define the viewer's rectangle on the screen, i.e. location `X`, `Y` of the lower left corner relative to the display origin, width `W` and height `H`. The variable `state` informs about the current state of visibility (visible, closed, covered), while `handle` represents the functional interface of viewers. The type of the handler is

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

where ViewerMsg is some base type of messages whose exact declaration is of minor importance for the moment:

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

TODO: Figure 3.1 Typical Oberon display configuration with tool track on the right

However, we should point out the use of object-oriented 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 object-oriented 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`, where

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

is an extended type of `ViewerMsg`.

It is worth noting that our open object-oriented 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 objects 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: INTEGER; handle: PROCEDURE END;

PROCEDURE NewTask(h: PROCEDURE; period: INTEGER): Task;
PROCEDURE Install (T: Task);
PROCEDURE 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 two examples of concrete background tasks may serve a better understanding of our explanations. The first one is a system-wide garbage collector collecting unused memory. The second example 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 Chapters 8 and 10 respectively.

We should not end this Section without drawing an important conclusion. Transfers of control between tasks are implemented in Oberon as ordinary calls and returns of ordinary procedures (procedure variables, actually). Preemption is not possible. From that we conclude that active periods of tasks are sequentially ordered and can be controlled by a single thread of control. 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 two categories of tasks exist: Interactive tasks and background tasks. 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: INTEGER;
  X, Y: INTEGER;
  keys: SET;
  ch: CHAR
END;

ControlMsg = RECORD (ViewerMsg)
  id: INTEGER;
  X, Y: INTEGER
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 one program 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, the 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 three 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 tasking system: Oberon is a multitasking system based on a two-category model. Interactive tasks are interfacing with the display system and are scheduled with high priority upon user interactions. Background tasks are stand-alone and are scheduled with low priority. Task activations are modeled as message passing and eventually as calls of procedures assigned to variables. They are sequentially ordered and controlled by a single thread of control.

3.3. THE CONCEPT OF COMMAND

An operating system constitutes a general purpose platform on which application software packages 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 operating systems 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 programming languages towards extreme abstraction.

We have put greatest emphasis in Oberon on closing the *semantic gap* between application software packages 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 most notable outcome of this approach is a collection of very powerful and system-wide abstract data types like `Task`, `Frame`, `Viewer`, `File`, `Font`, `Text`, `Module`, `Reader`, `Scanner`, `Writer` etc..

3.3.1. Atomic actions.

The most important generic function of any operating system is executing *programs*. A clarification of the term *program* as it is used in Oberon comprises two views: a *static* one and a *dynamic* one. Statically, an Oberon program is simply a package of software together with an entry point. More formally, an Oberon program is a pair `(M*, P)`, where `M` is an arbitrary module, `P` is an exported parameterless procedure of `M`, and `M*` denotes the hierarchy consisting of `M` itself and of all directly and indirectly imported modules. Note that two hierarchies `M*` and `N*` are not generally disjoint, even if `M` and `N` are different modules. Rather, their intersection is a superset of the operating system.

Viewed dynamically, an Oberon program is defined as an atomic action (often called *command*) operating on the global system state, where *atomic* means "without user interaction". This definition is just a necessary consequence of our model of non-preemptive task 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 in traditional systems these interruptions may occur at any time, in Oberon they can occur only after the completion of a task, of a command.

Quintessentially, Oberon programs are represented in the form of *commands* that are in the form of exported parameterless procedures that do not interact with the user of the system.

Returning to the calling and execution of programs we now arrive at the following refined code 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 module `Oberon`. Its primary operation can be paraphrased as "call a command by its name and pass a list of actual parameters":

```
PROCEDURE Call (name: ARRAY OF CHAR; par: ParList;
                VAR res: INTEGER);
```

`name` is the name of the desired command in the form `M.P`, `par` is the list of actual parameters, and `res` is a result code. But in fact we have separated the setting of parameters from the actual call. Parameters are set by calling

```
PROCEDURE SetPar (F: Display.Frame; T: Texts.Text;
                  pos: INTEGER);
```

and the actual call is achieved by calling

```
PROCEDURE Call (name: ARRAY OF CHAR; VAR res: INTEGER);
```

The pair `(T, pos)` specifies the starting position of a textual parameter list. `F` indicates the calling viewer. Notice the occurrence of yet another abstract data type of name `Text` that is exported by module `Texts`. We shall devote Chapter 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 list of actual parameters is handed over to the called command by module `Oberon` in the form of an exported global variable `Par`:

```
Par: RECORD vwr: Viewers.Viewer;
       frame: Display.Frame;
       text: Texts.Text;
       pos: INTEGER
     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 list of actual parameters 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 of type `Text`:

```
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` and `Text` that we encountered above. They will be the subject of Chapters 4 and 5.

Module `Oberon` assists the integration of these types with the following conceptual features, of which the first two are familiar to us already: Standard parameter list for commands, system log, generic text selection, and 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 procedure GetSelection:

```
PROCEDURE 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 so- called selection message to all viewers. The declaration of this message is

```
SelectionMsg = RECORD (ViewerMsg)
  time: INTEGER;
  text: Texts.Text;
  beg, end: INTEGER
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 this Section:. Oberon is an operating system that presents itself to its clients in the form of a highly expressive abstract modular interface that exports many powerful abstract data types like, for example, `Viewer` and `Text`. A rich arsenal of global data types and generic facilities serve the purpose of system integration at a high degree. Programs in Oberon are modeled as so-called commands, i.e. as exported parameterless procedures that do not interact 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 module `Oberon`. Commands operate on the global state of the system.

3.4. TOOLBOXES

Modules typically appear in three different forms. The first is a module that encapsulates some data, letting them be accessed only through exported procedures and functions. A good example is Module `FileDir`, encapsulating the file directory and protecting it from disruptive access. A second kind is the module representing an *abstract data type*, exporting a type and its associated operators. Typical examples are modules `Files`, `Modules`, `Viewers`, and `Texts`. A third kind is the collection of procedures pertaining to the same topic, such as module `RS-232` handling communication over a serial line.

Oberon adds a fourth form: the *toolbox*. By definition, this is a pure collection of commands in the sense of the previous section. Toolboxes distinguish themselves principally from the other forms of modules by the fact that they lie on 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 modules `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 module `System`:

```
DEFINITION System;

(*System management, Chapters 3 and 8*)
  PROCEDURE SetUser; (*identification*)
  PROCEDURE SetFont; (*for typed text*)
  PROCEDURE SetColor; (*for typed text and graphics*)
  PROCEDURE SetOffset; (*for typed text*)
  PROCEDURE Date; (*set or display time and date*)
  PROCEDURE Collect; (*garbage*)

(*Display management, Chapter 4*)
  PROCEDURE Open; (*viewer*)
  PROCEDURE Close; (*viewer*)
```

```
  PROCEDURE CloseTrack;
  PROCEDURE Recall; (*most recently closed viewer*)
  PROCEDURE Copy; (*viewer*)
  PROCEDURE Grow; (*viewer*)
  PROCEDURE Clear; (*clear log*)

(*Module management, Chapter 6*)
  PROCEDURE Free; (*specified modules*)
  PROCEDURE ShowCommands; (*of specified module*)
  PROCEDURE ShowModules; (*list loaded modules*)

(*File management, Chapter 7*)
  PROCEDURE Directory;
  PROCEDURE CopyFiles;
  PROCEDURE RenameFiles;
  PROCEDURE DeleteFiles;)

(*System inspection, Chapter 8*)
  PROCEDURE Watch; (*tasks, memory and 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 Extended Backus-Naur Form 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 module `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 are universal syntactical tokens like name, literal string, integer, real number, and special character.

2.) An arrow "^" in the textual parameter list 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 textual parameter list 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 textual parameter list indicates that the selection marks the (beginning of the) text which is taken as operand.

5.) A terminator-character "~" terminates the textual parameter list 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 Figure 3.1 that shows a typical Oberon screen layout consisting of two vertical tracks, a wider user track on the left and a narrow system track on the right. Three documents are displayed in the user track: A text, a graphic, and a picture. In the system track we find one log-viewer displaying the system log, two tool-viewers making available the standard system tool and 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`, `Date`, `SetFont`, `SetColor`, and `Collect` it is used to control system-wide facilities. In detail, their function is installing the user’s identification, displaying or setting the system date and time, presetting the system type-font for typed text, setting the system color, and activating the garbage collector.

In summary, a toolbox is a special form of an `Oberon` module. It is defined as a collection of commands. Appearing at the top of the modular hierarchy the toolboxes in their entirety fix the systems user interface. Command tools are sequences of textually represented command calls. They are editable and customizable. In a typical Oberon screen layout the tools are displayed in viewers within the system track.

CHAPTER 4

THE DISPLAY SYSTEM

The display screen is the most important part of the interface presented by a personal workstation to its users. At first sight, it simply represents a rectangular output area. However, in combination with the mouse, it quickly develops into a sophisticated interactive input/output platform of almost unlimited flexibility. It is mainly its Janus-faced characteristic that makes the display screen stand out from ordinary external devices to be managed by the operating system. In the current chapter we shall give more detailed insight into the reasons for the central position the display system takes within the operating system, and for its determining influence on the entire system architecture. In particular, we shall show that the display system is a natural basis or anchor for functional extensibility.

4.1. THE SCREEN LAYOUT MODEL

In the early seventies, Xerox PARC in California launched the Smalltalk-project with the goal of conceiving and developing new and more natural ways to communicate with personal computers 1. 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 a collection of possibly mutually overlapping pages of paper that are laid out on the desktop. By projecting such a configuration onto the surface of a screen we get the familiar picture of Figure 4.1 showing a collection of partially or totally visible rectangular areas on a background, so-called *windows* or *viewers*.

TODO: Figure 4.1 Desktop showing partially overlapping viewers

The desktop metaphor is used by many modern operating systems and user interface shells both as a natural model for the system to separate displayed data belonging to different tasks, and as a powerful tool for users to organize the display screen interactively, according to individual taste and preference. However, there are inherent drawbacks in the metaphor. They are primarily connected with overlapping. Firstly, any efficient management of overlapping viewers must rely on a subordinate management of (arbitrary) sub-rectangles and on sophisticated clipping operations. This is so because partially overlapped viewers must be partially restored under control of the viewer manager. For example, in Figure 4.1, rectangles a, b, and c in viewer B ought to be restored individually after closing of viewer A. Secondly, there is a significant danger of covering viewers completely and losing them forever. And thirdly, no canonical

1 [Goldberg] A. Goldberg, Smalltalk-80: The Interactive Programming Environment, Addison-Wesley 1984.

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, and so the additional complexity of its management 1 2 is hard to justify. 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 several variants of tiling 3. Perhaps the most obvious one (because the most unconstrained one) is based on iterated horizontal or vertical splitting of existing viewers. Starting with the full screen and successively opening viewers A, B, C, D, E, and F we get to a configuration as in Figure 4.2.

TODO: Figure 4.2 Viewer configuration resulting from unconstrained tiling

A second 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 Figure 4.2 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.

Hierarchic tiling is also used in Xerox PARC's Cedar system 4. However, the Oberon variant differs from the Cedar variant in some respects. Firstly, Oberon supports quick temporary context switching by overlaying one track or any contiguous sequence of tracks with new layers. In Figure 4.3 a snapshot of a standard Oberon display screen is graphically represented. It suggests two original tracks and two levels of overlay, where the top layer is screen-filling. Secondly, unlike Cedar display screens, Oberon displays do not provide reserved areas for system-wide facilities, Standard Cedar screens feature a command row at the top and an icon row at the bottom. And thirdly, Oberon 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 viewer is then allocated in the freed spot at the bottom. In contrast, Oberon normally splits the largest existing viewer in a given track into two

1 C. Binding, User Interface Components based on a Multiple Window Package, University of Washington, Seattle, Technical Report 85-08-07.

2 M. Wille, Overview: Entwurf und Realisierung eines Fenstersystems fr Arbeitsplatzrechner, Diss. ETH Nr. 8771, 1988.

3 [Cohen] E.S. Cohen, E.T. Smith, L.A. Iverson, Constraint-Based Tiled Windows, IEEE, 1985

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

halves of equal size. As an advantage of this latter allocation strategy we note that existing contents are kept stable.

TODO: Figure 4.3 Overlay of tracks and sequences of tracks

4.2. VIEWERS AS OBJECTS

Although everybody seems to agree on the meaning of the term viewer, no two 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` that we introduced in Chapter 3. We recapitulate: Type `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 two categories of universal messages document the roles of viewers as interactive tasks and as parts of an integrated system respectively.

In total, there are four such categories. They are here listed together with the corresponding topics and message dispatchers:

TODO: Table Dispatcher Task scheduler Command interpreter Viewer manager Document manager Topic dispatching of task processing of command organizing display area operating on document Message reports user interaction defines generic operation change of location or size change of contents or format

These topics essentially define the role of Oberon viewers. In short, we may look at an Oberon 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 integrators of the different subsystems via message-based interfaces. In this light type Viewer appears as a common object-oriented basis of Oberon's subsystems.

The topics listed above constitute some kind of backbone of the contents of the Chapters 3, 4 and 5. Task scheduling and command interpreting are already familiar to us from Sections 3.2 and 3.3. Viewer management and text management will be the topics of Sections 4.4 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 detail in Sections 4.4 and 5.3 in the 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:

TODO: List bullets After each key stroke a keyboard message containing the typed character is sent to the current focus viewer and after each mouse click a mouse message reporting the new state of the mouse is sent to the viewer containing the current mouse position.

A message often represents some generic operation that is expected to be interpreted individually by its recipients. Obvious examples in our context are "return current textual selection", "copy-over stretch of text", and "produce a copy (clone)". Notice that generic operations are 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 for every affected viewer requesting it 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 MVC (model-view-controller) pattern that dispenses models from "knowing" (registering) their views.

4.3. FRAMES AS BASIC DISPLAY ENTITIES

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

The type `Frame` is declared as

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

The components `next` and `dsc` are connections to further frames. Their names suggest a multi-level recursive hierarchical structure: `next` points to the next frame on the same level, while `dsc` points to the (first) descendant, i.e. to the next lower level of the hierarchy of nested frames. `X`, `Y`, `W`, `H`, and the handler handle serve the original purpose to that we introduced them. In particular, the handler allows frames to react individually on the receipt of messages. Its type is

```
Handler = PROCEDURE (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, type `Viewer` is a derived type, it is a type extension of `Frame`:

```
Viewer = POINTER TO ViewerDesc;
ViewerDesc = RECORD (FrameDesc)
  state: INTEGER
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 of so-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 Figure 4.4).

TODO: Figure 4.4 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 height of the menu frame:

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

Each menu viewer `V` specifies exactly two descendants: The menu frame `V.dsc` and the frame of main contents or main frame `V.dsc.next`. Absolutely nothing is fixed about the contents of the two descendant frames. In the standard case, however, the menu frame is a text frame, displaying a line of commands in inverse video mode. By definition, the nature of the main frame specifies the type of the viewer. If it is a text frame as well, then we call the viewer a *text viewer*, if it a graphics frame, we call it a *graphics viewer* etc.

4.4. DISPLAY MANAGEMENT

Oberon's display system comprises two main topics: Viewer management and cursor handling. Let us first turn to the much more involved topic of viewer management and postpone cursor handling 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 two-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 a system with a 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 three modules: `Display`, `Viewers`, and `MenuViewers`, the latter building upon the formers. 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 viewer management. Modules on upper lines import modules on lower lines and types on upper lines extend types on lower lines.

TODO: table Module MenuViewer Viewers Display Type Viewer Viewer Frame Service Message handling for menu viewers Tiling viewer management Block-oriented raster operations

Inspecting the column titled Type we recognize precisely our familiar types `Frame`, `Viewer`, and `MenuViewer` respectively, where the latter is an abbreviation of `MenuViewers.Viewer`.

In addition to the core modules of the display system a section in module `Oberon` provides a specialized application programming interface (API) that simplifies the use of the viewer management package by applications in the case of standard Oberon display configurations. We shall come back to this topic in Section 4.6.

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

4.4.1. Viewers.

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

This is the formal definition:

```
DEFINITION Viewers;
  IMPORT Display;
  CONST restore = 0; modify = 1; suspend = 2; (*message ids*)
  TYPE Viewer = POINTER TO ViewerDesc;
       ViewerDesc = RECORD (Display.FrameDesc)
         state: INTEGER
       END;
       ViewerMsg = RECORD (Display.FrameMsg)
         id: INTEGER;
         X, Y, W, H: INTEGER;
         state: INTEGER
       END;

  VAR curW: INTEGER;

  (*track handling*)
  PROCEDURE InitTrack (W, H: INTEGER; Filler: Viewer);
  PROCEDURE OpenTrack (X, W: INTEGER; Filler: Viewer);
  PROCEDURE CloseTrack (X: INTEGER);

  (*viewer handling*)
  PROCEDURE Open (V: Viewer; X, Y: INTEGER);
  PROCEDURE Change (V: Viewer; Y: INTEGER);
  PROCEDURE Close (V: Viewer);

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

Some comments: A first group of procedures consisting of `InitTrack`, `OpenTrack`, and `CloseTrack` supports 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, `InitTrack` initializes the newly created track with a filler viewer that is supplied as a parameter. 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 after startup. It amounts to executing a sequence of steps of the form

```
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 width of the already configured part of the display area. Note that configuring starts with `x = 0` and is non-reversible in the 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.

Procedure `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)`. Procedure `CloseTrack` is inverse to `OpenTrack`. It is called to close the (topmost) track located at `X` in the display area, and to restore the previously covered track (or sequence of tracks).

The next three procedures are used to organize viewers within individual tracks. Procedure `Open` allocates a given viewer at a given position. More precisely, `Open` locates the viewer containing the point `(X, Y)`, splits it horizontally at height `Y`, and opens the viewer `V` in the lower part of the area. In the special case of `Y` coinciding with the upper boundary line of the located viewer this is closed automatically. Procedure `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). Procedure `Close` removes the given viewer `V` from the display area. Figure 4.5 makes these operations clear.

TODO: Figure 4.5 Basic operations on viewers

The last group of procedures provides miscellaneous services. Procedure `This` identifies the viewer displayed at `(X, Y)`. Procedure `Next` returns the next upper neighbor of a given displayed viewer `V`. Procedure `Recall` allows recalling and restoring the most recently closed viewer. `Locate` is a procedure that assists heuristic allocation of new viewers. For any given track and desired minimum height, procedure `Locate` offers a choice of some distinguished viewers in the track: the filler viewer, the viewer at the bottom, an alternative choice, and the viewer of maximum height. Finally, procedure `Broadcast` broadcasts a message to the display area, that is, sends the given message to all viewers that are currently displayed.

It is now a good time to throw a glance behind the scenes. Let us start with revealing module `Viewers` 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 through the modules procedural interface only. Figure 4.6 shows a data structure view of the display snapshot taken in Figure 4.4. Note that the overlaid tracks and viewers are still part of the internal data structure.

In the data structure we recognize an anchor that represents the display area and points to a list of tracks, each of them in turn pointing to a list of viewers, each of them 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 Figure 4.6.

TODO: Figure 4.6 A snapshot of the internal data structure corresponding to Figure 4.3

Technically, the track descriptor type `TrackDesc` is a private extension of the viewer descriptor type `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: INTEGER
END;

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

It is noteworthy that the data structure of the viewer manager is heterogeneous with `Frame` as base type. It provides a nice example of a nested hierarchy of frames with the additional property that the first two levels correspond to the first two 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 is then 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` is bound to module `Viewers` $\Leftrightarrow$`V.state` $\neq$0

If we call visible any displayed viewer and suspended any viewer that is covered by an overlaying track we can refine this invariant to

(`V` is visible $\Leftrightarrow$`V.state` > 0) and (`V` is suspended $\Leftrightarrow$`V.state` < 0)

TODO: Clean this math shit up

In addition, more detailed information about the kind of viewer `V` is given by the magnitude TODO: absolute value `V.state` :

TODO: table V.state kind of viewer 0 closed 1 filler -1 productive

The magnitude $|V.state|$ is kept invariant by module `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 allocation of new viewers. The variable state is treated as read-only by every module other than `Viewers`.

We are now sufficiently prepared to understand how the exported procedures of module `Viewers` work behind the scenes. All of them operate on the internal dynamic data structure just explained. Some use the structure as a reference only or operate on individual elements (procedures `This`, `Next`, `Locate`, `Change`), others add new elements to the structure (procedures `InitTrack`, `OpenTrack`, `Open`), and even others remove elements (procedures `CloseTrack`, `Close`). Most procedures have side-effects on the size or state of existing elements.

Let us now change perspective and look at module `Viewers` as a general low-level manager of viewers whose exact contents are unknown to it (and whose controlling software might have been developed years later). In short, let us look at module `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: INTEGER;
  X, Y, W, H: INTEGER;
  state: INTEGER
END;
```

There exist three variants of `Viewer` messages, discriminated by the field `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.

TODO: table Originator OpenTrack CloseTrack Open Change Close Message Suspend temporarily Suspend permanently Modify or suspend Modify Suspend permanently Recipients Viewers covered by opening track Viewers in closing track Upper neighbor of opening viewer Upper neighbor of changing viewer Closing viewer

4.4.2. Menu Viewers.

So far, we have treated viewers abstractly as black boxes. Our next step is now to focus on a special class of viewers called *menu viewers*. Remembering the definition given earlier we know that a menu viewer is characterized by a structure consisting of two vertically tiled "descendant" frames, a *menu frame* at the top and a *frame of contents* at the 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 in Oberon is hierarchically organized and message passing within the display space obeys the pattern of strict *parental control*.

Again, we start our more detailed discussion with a module interface definition:

```
DEFINITION MenuViewers;
  IMPORT Viewers, Display;
  CONST extend = 0; reduce = 1; move = 2; (*message ids*)

  TYPE
    Viewer = POINTER TO ViewerDesc;
    ViewerDesc = RECORD (Viewers.ViewerDesc)
      menuH: INTEGER
    END;
    ModifyMsg = RECORD (Display.FrameMsg)
      id: INTEGER;
      dY, Y, H: INTEGER
   END;
   PROCEDURE Handle (V: Display.Frame; VAR M: Display.FrameMsg);
   PROCEDURE New (Menu, Main: Display.Frame;
                  menuH, X, Y: INTEGER): Viewer;
END MenuViewers.
```

The interface represented by this definition is conspicuously narrow. There are just two procedures: A generator procedure `New` and a standard message handler `Handle`. The generator returns a newly created menu viewer displaying the two (arbitrary) frames passed as parameters. The message handler implements the entire "behavior" of an object and in particular the above mentioned 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 three basic steps:

allocate memory block; install message handler; initialize state variables

In the case of a standard menu viewer `V` this can be expressed as

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

With that, calling `New` is equivalent with

```
create V; open V at X, Y
```

where opening `V` needs assistance by module `Viewers`.

The implementation of procedure `Handle` embodies the standard strategy of message handling by menu viewers. The following code is a coarse-grained view of it.

```
Message handler for menu viewers

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 (clone);
    send copy message to main frame to get a copy (clone);
    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 or size THEN
  IF operation is restore THEN
    draw viewer area and border;
    send modify message to menu frame to make it extend from height 0;
    send modify message to main frame to make it extend from height 0
  ELSIF operation is modify THEN
    IF operation is extend THEN
      extend viewer area and border;
      send modify message to menu frame to make it extend;
      send modify message to main frame to make it extend
```

```
      ELSE (*reduce*)
        send modify message to main frame to make it reduce;
        send modify message to menu frame to make it reduce;
        reduce viewer area and border
      END
    ELSIF operation is suspend THEN
      send modify message to main frame to make it reduce to height 0;
      send modify message to menu frame to make it reduce to height 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 four 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:

send *copy* message to each element in the list of descendants;

generate copy of the original frame descriptor;

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 that maps incoming messages into a set of outgoing messages, with possible side-effects. The simplest case of such a transformation is known as *delegation*. In this case, the input message is simply passed on to the descendant(s).

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 an appropriate message type called `ModifyMsg`:

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

The field `id` specifies one of two variants: `extend` or `reduce`. The first variant of the message requests the receiving frame to move by the vertical translation vector `dY` and then to extend to height `H` at bottom. The second variant requests the frame to reduce to height `H` at bottom and then to move by `dY`. In both cases `Y` indicates the Y-coordinate of the new lower-left corner. Figure 4.7 summarizes this graphically.

Messages arriving from the viewer manager and requesting the receiving viewer to extend or reduce at its bottom are also mapped into messages of type `ModifyMsg`. Of course, no translation is needed in these cases, and `dY` is 0.

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

TODO: Figure 4.7 The modify frame operation

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 viewer manager and the cursor handler are two concurrent users of the same display area. Actually, we should imagine two parallel planes, one displaying viewers and the other displaying cursors. If there is just one 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 again contained in module `Oberon`. The corresponding application programming interface is

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

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

The state of a cursor is given by its mode of visibility (on), its position (X, Y) in the display area, and the current marker. `Marker` is an abstract data type with an interface consisting of two operations `Fade` and `Draw`. The main benefit we can draw from this abstraction is once more conceptual independence of the underlying hardware. For example, `Fade` and `Draw` can adapt to a given monitor hardware with built-in cursor support or, in case of absence of such support, can simply be implemented as identical procedures (an involution) drawing the marker pattern in inverse video mode.

The functional interface to cursors consists of three operations: `OpenCursor` to open a new cursor, `FadeCursor` to switch off the marker of an open cursor, and `DrawCursor` to 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.

Two cursors, `Mouse` and `Pointer` are predefined. They represent the mouse and an interactively controlled global system pointer respectively. Typically (but not necessarily) these cursors are visualized by the built-in markers `Arrow` (a small arrow pointing to north-west) and `Star` (a star symbol) respectively. The pointer can be used to mark any displayed object. It serves primarily as an implicit parameter of commands.

Two assisting service procedures `MarkedViewer` and `RemoveMarks` are added in connection with the predefined cursors. `MarkedViewer` returns the viewer that is currently marked by the pointer. Its resulting value is equivalent to `Viewers.This(Pointer.X, Pointer.Y)`. `RemoveMarks` turns invisible the predefined cursors within a given rectangle in the display area. This procedure is used to lock the rectangle for its caller.

Summary of the essential points and characteristics of Oberon's concept of cursor handling:

1.) By virtue of the use of abstract markers and of 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.

2.) 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.

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

4.) Looking at the viewer manager as one user of the display area, the cursor handler is a second (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 viewer manager for the accommodation of tracks and viewers, which are merely the first two levels of a potentially unlimited nested hierarchy of display frames. For example, standard menu viewers contain two subordinate frames: A menu frame and a main frame of contents. Viewers are treated as black boxes by the viewer manager 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. Two standard cursors Mouse and Pointer are predefined.

4.5. RASTER OPERATIONS

In Section 4.4 we introduced the display area as an abstract concept, modeled as a two-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 the displayed contents, 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 (pixels), 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 two colors we think of one color representing background and the other color representing foreground.

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. By a block we mean 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 block of width `SW` and height `SH` of source pixels is placed at a given point of destination `(DX, DY)` in the display area. In the simplest case, the destination block `(DX, DY, SW,` SH) is plainly overwritten by the source block. In general, the new value of a 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 *mode* of combination 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:

TODO: table mode replace paint invert operation s s + d (or) 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*)
  PROCEDURE Dot (col, x, y, mode: INTEGER);
  PROCEDURE ReplConst (col, x, y, w, h, mode: INTEGER);
  PROCEDURE CopyPattern (col, patadr, x, y, mode: INTEGER);
  PROCEDURE CopyBlock (sx, sy, w, h, dx, dy, mode: INTEGER);
  PROCEDURE ReplPattern (col, patadr, x, y, w, h, mode: INTEGER);
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 1. In Oberon, `CopyBlock` 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 first two 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.

1 [Goldberg] A. Goldberg, Smalltalk-80: The Interactive Programming Environment, Addison-Wesley 1984.

Hence, the number of data bytes is $((w + 7) \div 8) \times h$. An example is shown in Figure 4.8.

TODO: Figure 4.8 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 are patterns arrow, hook, and star intended to represent the cursor, the caret, and the marker. A second group of predefined patterns supports drawing graphics.

The parameter col in the pattern-oriented raster operations specifies the pattern's foreground color. Colors black (background) and white are predefined. Procedure `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. Procedure `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. Procedures `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: INTEGER;
    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 two congruent regions, where the corresponding monitor is supposed to be able to select and display one of the two 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. Figure 4.9 gives an impression of such a configuration.

TODO: Figure 4.9 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 1/36000 mm. 1/36000 mm 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 CONFIGURATIONS 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 module `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 two tracks of equal height, 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;
  PROCEDURE OpenDisplay (UW, SW, H: INTEGER);
  PROCEDURE OpenTrack (X, W: INTEGER);
  PROCEDURE DisplayWidth (X: INTEGER): INTEGER;
  PROCEDURE DisplayHeight (X: INTEGER): INTEGER;
  PROCEDURE UserTrack (X: INTEGER): INTEGER;
  PROCEDURE SystemTrack (X: INTEGER): INTEGER;
  PROCEDURE AllocateUserViewer (DX: INTEGER; VAR X, Y: INTEGER);
  PROCEDURE AllocateSystemViewer (DX: INTEGER; VAR X, Y: INTEGER);
END Oberon.
```

Procedure `OpenDisplay` initializes and opens a new display of the dimensions `H` (height), `UW` (width of user track), and `SW` (width of system track). Procedure `OpenTrack` overlays the sequence of existing tracks spanned by the segment $[X, X+W)$ by a new track. Both procedure `OpenDisplay` and `OpenTrack` take from the client the burden of creating a filler viewer.

The next group of procedures `DisplayWidth`, `DisplayHeight`, `UserTrack` and `SystemTrack` return width or height of the respective structural entity located at position X in the display area.

Procedures `AllocateUserViewer` and `AllocateSystemViewer` make proposals for the allocation of a new viewer in the desired track of the display located at `DX`. In first 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 module `Oberon` for the case of standard display layouts, the display management section in the `System` toolbox provides a user interface:

```
DEFINITION System; (*Display management*)
  PROCEDURE Open; (*viewer*)
  PROCEDURE Close; (*viewer*)
  PROCEDURE CloseTrack;
  PROCEDURE Recall; (*most recently closed viewer*)
  PROCEDURE Copy; (*viewer*)
  PROCEDURE Grow; (*viewer*)
  PROCEDURE Clear; (*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, and grow a viewer. The commands `Close`, `CloseTrack`, `Recall`, `Copy`, and `Grow` are generic. `Close`, `Copy`, and `Grow` are typically included in the title bar of a menu viewer. Their detailed implementations follow subsequently.

CHAPTER 5

THE TEXT SYSTEM

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 one byte each (ASCII-code). 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 operating systems neither feature an integrated management nor an abstract programming interface (API) 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 programming interface (API) for text that is, an abstract data type Text, together with a complete set of operations. We shall devote Section 5.1 to the explanation of this data type. In Section 5.2, we take a closer look at the basic text management in Oberon, 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 Section 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 Section 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 Section 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 programming language development. It provides a powerful tool to create simplified views of complicated things and connections. Two prominent examples of program abstractions are definitions (interfaces) and abstract data types, embodying simplified views on a certain piece of program and on a certain kind of data respectively.

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 type String as it is often supported by advanced programming languages. In this Section 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-code 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 two 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 Section 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: INTEGER;
  notify: Notifier
END;
```

There is only one state variable and one method. The variable len represents the current length of the described text (i.e. the number of characters in the sequence). The procedure variable notify is included as a method (occasionally called after-method) to notify interested clients of state changes.

By definition, each abstract data type comes with a complete set of operations. In the case of `Text`, three different groups corresponding to three different topics need to be considered, loading (from file), storing (to file), editing, and accessing (reading and writing) respectively.

5.1.1. Loading and Storing Text.

Let us start with the file group. We first introduce a pair of mutually inverse operations called internalize and externalize. Their meaning is "load from file and build up an internal data structure" and "serialize the internal data structure and store it on file" respectively. There are three corresponding procedures:

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

Logical entities like texts are stored in Oberon on external media in the form of sections. A section is addressed by a pair (file, pos) consisting of a file descriptor and a starting position. In general, the structure of sections obeys the following syntax:

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

Procedure `Open` internalizes a named text file (consisting of a single text section), procedure `Load` internalizes an arbitrary text section starting at (f, pos), and procedure `Store` externalizes a text section to (f, pos). The parameter `T` designates the internalized text. len returns the length of the section. Note that in case of Load the identification of the section must have been read and consumed before the loader is called.

5.1.2. Editing Text.

Our next group of operations supports text editing. It comprises four procedures:

```
PROCEDURE Delete (T: Text; beg, end: INTEGER);
```

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

Again, we should first explain the types of parameters. Procedures `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. Type `Buffer` presents itself again as an abstract data type:

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

`len` specifies the current length of the buffered sequence. The following procedures represent the intrinsic operations on buffers:

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

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

`Buffer` is used as an auxiliary data type in editing procedures. Procedure `Delete` deletes the given stretch [beg, end) within text `T`, Insert inserts the buffer's contents at position pos within text `T`, and `Append(T, B)` is a shorthand form for `Insert(T, T.len, B)`. Note that, as a side-effect of `Insert` and `Append`, the buffer involved is emptied. Finally, procedure `ChangeLooks` allows to change selected looks within the given stretch [beg, end) of text `T`. `sel` is a mask selecting a subset of the set of looks `font, color, vertical offset` .

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 is

```
Notifier = PROCEDURE (T: Text; op, beg, end: INTEGER);
```

The parameters `op`, `beg`, and `end` report about the operation (`op`) that calls the notifier and on the affected stretch [beg, end) of the text. There are three different possible variants of `op` corresponding to the three different editing operations: op = delete, insert, replace correspond to procedures `Delete`, `Insert` (and `Append`), and `ChangeLooks` respectively.

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 Section 5.3.

In concluding this Section it is worth noting that the groups of operations just discussed have been designed to be equally useful for interactive text editors as for programmed text generators/manipulators.

5.1.3. Accessing Text.

Let us now turn to the third and last group of operations on texts: Accessing that is 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 is declared as

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

A reader must first 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

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

Procedure `OpenReader` sets up a reader `R` at position `pos` in text `T`. Procedure `Read` returns the character at the current position of `R` and makes `R` move to the next position.
The current position of reader `R` is returned by a call to the function `Pos`:

```
PROCEDURE Pos (VAR R: Reader): INTEGER;
```

In Chapter 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, and carriage-return.
```

Type `Scanner` is defined correspondingly as

```
Scanner = RECORD (Reader)
  nextCh: CHAR;
  line: INTEGER;
  class: INTEGER;
  i: INTEGER;
  x: REAL;
  c: CHAR;
  len: INTEGER;
  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, ors`. `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 are

```
PROCEDURE OpenScanner (VAR S: Scanner; T: Text; pos: INTEGER);
PROCEDURE 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 type `Writer` resembles that of type `Reader`:

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

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

Procedure `OpenWriter` opens a new writer with an empty buffer. Procedures `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:

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

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 three areas: Loading/storing, editing, and accessing (reading/writing). The latter two areas on their part introduce further abstract types called `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:

TODO: table Module TextFrames Texts Fonts Object type Frame Text Font Service Text rendering and editing Text management 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.

TODO: Figure 5.1 Piece chain representing a text

Figure 5.1 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 piece descriptors from beg to end from the chain
END

insert stretch of text at pos = BEGIN
  split piece at pos;
  insert piece 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. Figures 5.3 and 5.4 show the resulting piece list after a delete and an insert-operation respectively.

TODO: Figure 5.2 Piece chain after delete operation TODO: Figure 5.3 Piece chain after insert operation

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 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.

TODO: Figure 5.4 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 = PROCEDURE (T: Text; op, beg, end: INTEGER);

TextDesc = RECORD
  len: INTEGER;
  notify: Notifier;
   trailer: Piece;
   org: INTEGER;
   pce: Piece
END;

Buffer = POINTER TO BufDesc;
BufDesc = RECORD
  len: INTEGER;
   header, last: Piece
END;

Reader = RECORD
  eot: BOOLEAN;
  fnt: Fonts.Font;
```

```
  col, voff: INTEGER;
   ref: Piece;
   org, off: INTEGER;
  rider: Files.Rider
END;

 Piece = POINTER TO PieceDesc;
 PieceDesc = RECORD
  f: Files.File;
  off, len: INTEGER;
  fnt: Fonts.Font;
  col, voff: INTEGER;
  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.

```
PROCEDURE FindPiece (T: Text; pos: INTEGER;
                     VAR org: INTEGER; VAR p: Piece);
  VAR p: Piece;
      porg: INTEGER;
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