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..