Project Oberon

The Design of an Operating System, a Compiler, and a Computer

Revised Edition 2013

Niklaus Wirth
Jürg Gutknecht

ISBN 0-201-54428-8

Preface

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

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

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

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

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

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

We wish to thank the many anonymous contributors of suggestions, advice, and encouragement. In particular we wish to thank our colleagues H. Mössenböck and B. Sanders and our associates at the Institut für Computersysteme for reading all or parts of the draft of this book. We are grateful to M. Brandis, R. Crelier, A. Disteli, M. Franz, and J. Templ for their work in porting the Oberon

System successfully to various commercially available computers, and thus making the study of this book more worth-while for many readers. And we gratefully acknowledge the contribution of our school, ETH, for providing the environment and support which made it possible for us to pursue and complete this project.

Zürich, February 1992

N.W. and J.G.

Preface to the 2013 edition

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

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

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

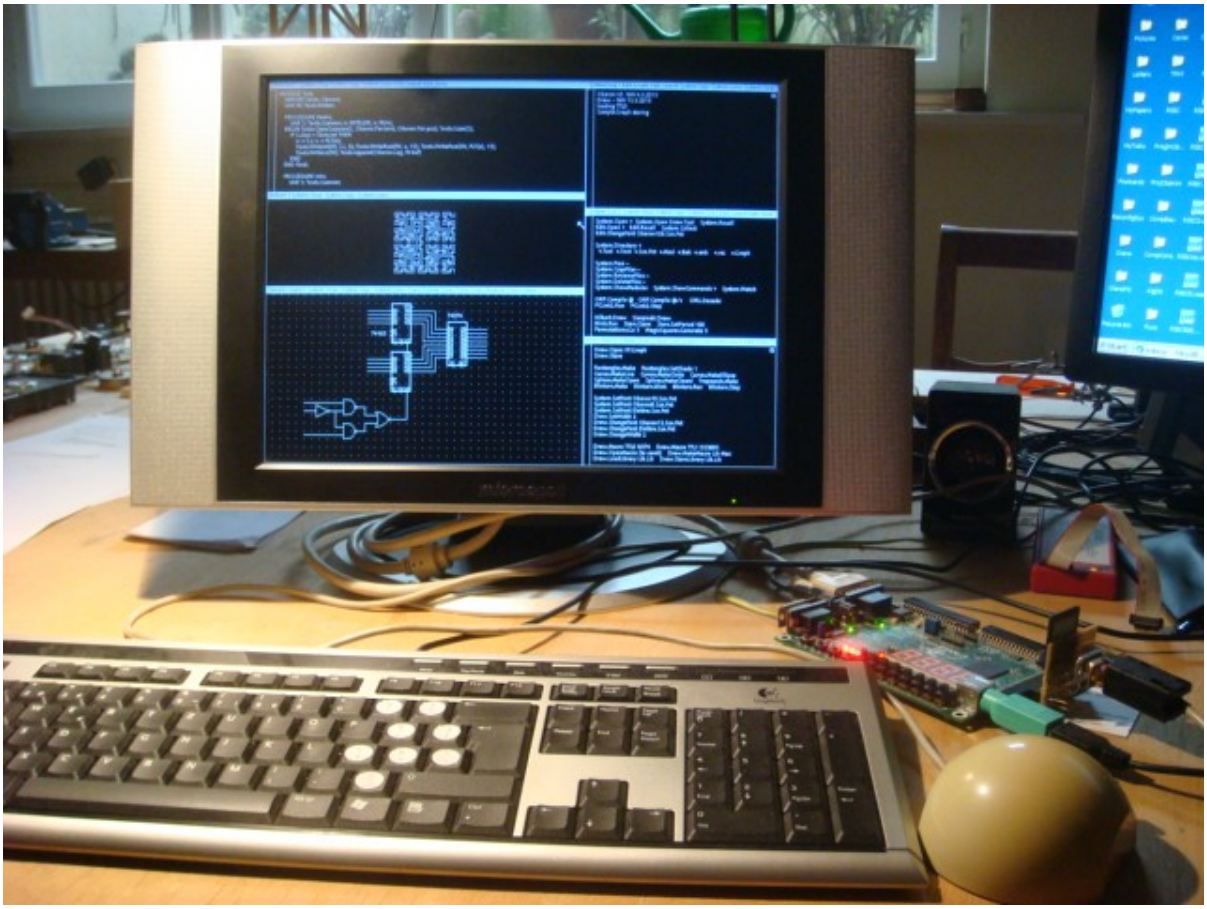
The main purpose of and the driving force behind this project is to provide a single book that serves as an example of a system that exists, is in actual use, and is explained in all detail. This task drove home the insight that it is hard to design a *powerful and reliable* system, but even much harder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

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

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

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

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



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

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

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

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

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

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

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

4. Objects declared in a procedure *P* are not accessible within a procedure *Q* that is itself local to *P*. That is, objects must be either strictly local or global in order to be accessible.

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

6. Forward procedure declarations have been discarded.

In contrast to these removals, there is a single addition (made in 2012): The data type BYTE has been added. Its values are integers *x* satisfying $0 \leq x < 256$. This addition prevents the frequent abuse of the type CHAR. The type BYTE is mainly used for elements of arrays and records in low-level modules in order to economise the use of memory.

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

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

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

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

References

http://www.inf.ethz.ch/personal/wirth/Oberon/Oberon07.Report.pdf

http://www.inf.ethz.ch/personal/wirth/FPGA-relatedWork/RISC.Arch.pdf

Acknowledgements

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

Niklaus Wirth, September 2013

Project Oberon
The Design of an Operating System, a Compiler, and a Computer

Preface (1992)

Preface to the revised edition (2013)

1. History and motivation

2. Basic concepts and structure of the system

2.1. Introduction
2.2. Concepts
  2.2.1. Viewers
  2.2.2. Commands
  2.2.3. Tasks
  2.2.4. Tool texts as configurable menus
  2.2.5. Extensibility
  2.2.6. Dynamic loading
2.3. The system's structure
2.4. A tour through the chapters

3. The tasking system

3.1. The concept of task
  3.1.1. Interactive tasks
  3.1.2. Background tasks
3.2. The task scheduler
3.3. The concept of command
  3.3.1. Generic actions
  3.3.2. Generic text selection
  3.3.3. Generic copy viewer
3.4. Toolboxes

4. The display system

4.1. Screen layout model
4.2. Viewers as objects
4.3. Frames as basic display entities
4.4. Display management
  4.4.1. Viewers
  4.4.2. Menu viewers
  4.4.3. Cursor management
4.5. Raster Operations
4.6. Standard display configurations

5. The text system

5.1. Text as an abstract data type
  5.1.1. Loading and storing
  5.1.2. Editing text
  5.1.3. Accessing text
5.2. Text management
5.3. Text frames
5.4. The font machinery
5.5. The edit toolbox

6. The module loader

6.1. Linking and loading
6.2. Module representation
6.3. The linking loader
6.4. The toolbox of the loader
6.5. The object file format

7. The file system

7.1. Files
7.2. Implementation of files on a random-access store
7.3. Implementation of files on a disk
7.4. The file directory
7.5. The toolbox of file utilities

8. Storage layout and management

8.1. Storage layout and run-time organization
8.2. Management of dynamic storage
8.3. The kernel
8.4. The storage management's toolbox

9. Device drivers

9.1. Overview
9.2. Keyboard and mouse
9.3. Serial peripheral interface (SPI)
9.4. Serial asynchronous interface (RS 232)

10. The network

10.1. Introduction
10.2. The protocol
10.3. Station addressing
10.4. The implementation

11. A dedicated file-distribution and mail server

11.1 Concept and structure
11.2. Electronic mail service
11.3. Printing service
11.4. Miscellaneous services
11.5. User administration

12. The compiler

12.1. Introduction
12.2. Code patterns
12.3. Internal data structures and module interfaces
  12.3.1. Data structures
  12.3.2. Module interfaces
12.4. The parser
12.5. The scanner
12.6. Searching the symbol table, and handling symbol files
  12.6.1. The structure of the symbol table
  12.6.2. Symbol files
12.7. The code generator
  12.7.1. Expressions
  12.7.2. Relations
  12.7.3. Set operations
  12.7.4. Assignments

12.7.5. Conditional and repetitive statements
12.7.6. Boolean expressions
12.7.7. Procedures
12.7.8. Type extension
12.7.9. Import and export, global variables
12.7.10. Traps

13. A graphics editor

13.1. History and goal
13.2. A brief user guide
13.2.1. Basic commands
13.2.2. Menu commands
13.2.3. Further commands
13.2.4. Macros
13.2.5. Rectangles
13.2.6. Oblique lines and circles
13.2.7. Spline curves
13.2.8. Constructing new macros
13.3. The core and its structure
13.4. Displaying graphics
13.5. The user interface
13.6. Macros
13.7. Object classes
13.8. The implementation
13.8.1. Module Draw
13.8.2. Module GraphicFrames
13.8.3. Module Graphics
13.9. Rectangles and Curves
13.9.1. Rectangles
13.9.2. Oblique lines and circles

14. Building and maintenance tools

14.1. The startup process
14.2. Building tools
14.3. Maintenance tools

15. Tool and Service Modules

15.1. Basic mathematical functions
15.2. A data link
15.3. A generator of graphic macros

16. Implementation of the RISC processor

16.1. Introduction
16.2. The Arithmetic and Logic Unit (ALU)
16.2.1 Shifters
16.2.2. Multiplication
16.2.3. Division
16.3. Floating-point arithmetic
16.3.1. Floating-point addition
16.3.2. Floating-point multiplication
16.3.3. Floating-point division

16.4. The Control Unit

17. The Processor's Environment

17.1. The SRAM memory
17.2. Peripheral Interfaces
  17.2.1. The PS-2 interface
  17.2.2. The SPI interface
  17.2.3. The RS-232 interface
  17.2.4. The display controller
  17.2.5. The Mouse interface

1 History and motivation

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.

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.

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.

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.

2 Basic concepts and 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.

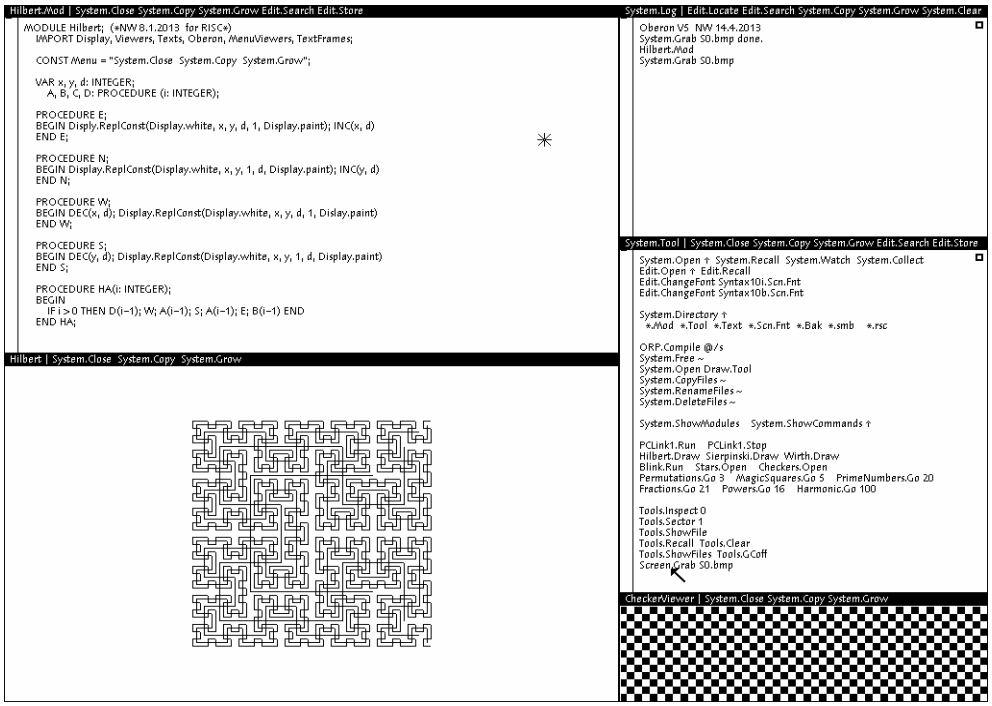


Fig. 2.1. Oberon display with tiled viewers

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*.

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

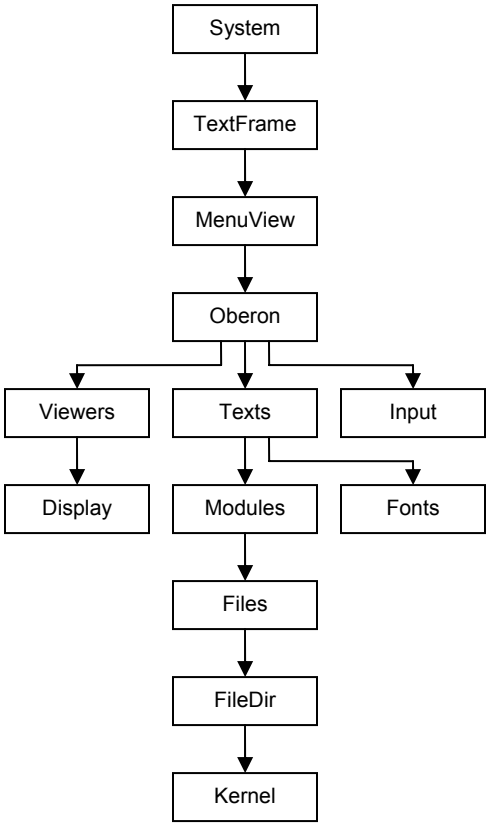


Fig. 2.2. The structure of the Oberon core

Module names in the plural form typically indicate the definition of an abstract data type in the module. The type is exported together with the pertinent operations. Examples are *Files, Modules, Fonts, Texts, Viewers, MenuViewers, and TextFrames.* Modules whose names are in singular form

typically denote a resource that the module manages, be it a global variable or a device. The variable or the device is itself hidden (not exported) and becomes accessible through the module's exported procedures. Examples are all device drivers, *Input* for keyboard and mouse, *Kernel* for memory and disk, and *Display*. Exceptions are the command modules whose name is mostly chosen according to the activity they primarily represent, like *System*, and *Edit*

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.

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

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 *interactive tasks* 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.P*, followed by a textual parameter list. Consequently, any command can be called directly from within a text (so-called *tool*) simply by pointing at it with the mouse. However, the core of this chapter is a presentation of Oberon's text system as a case study in program modularization. The concerns of managing a text and displaying it are nicely separated. Both the text manager and the text display feature an abstract public interface as well as an internally hidden data structure. Finally in this chapter, Oberon's type-font management and the toolbox for editing are discussed.

Chapters 6 - 9 describe the *inner core*, 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.

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

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;
```

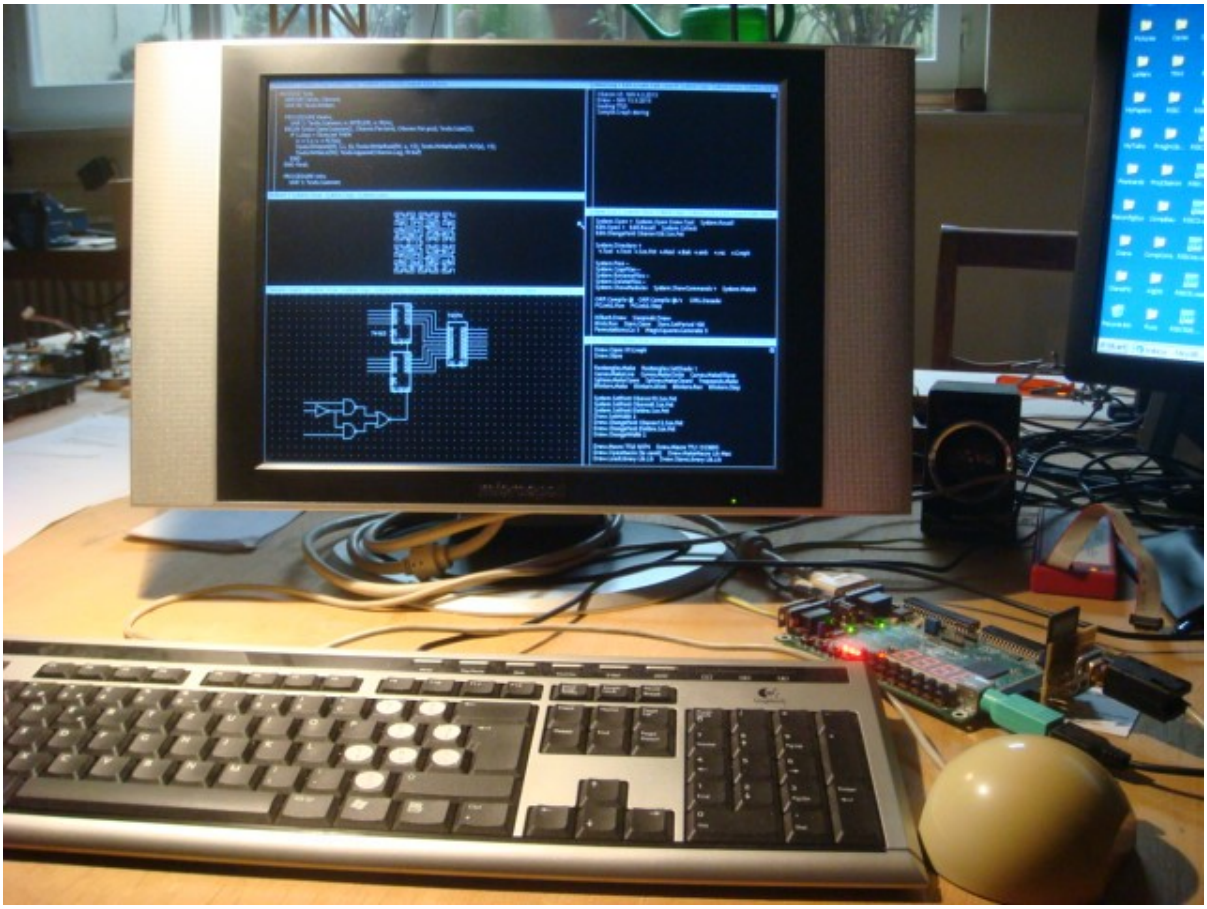


Figure 3.1 Typical Oberon display configuration with tool track on the right

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 object's compile-time interface and on the system integrity. It is fair to mention though that such a high degree of extensibility does not come for free. The price to pay is the obligation of explicit message *dispatching at runtime*. The following Chapters will capitalize on this property.

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

3.1.2. Background Tasks

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

```
Task = POINTER TO TaskDesc;
TaskDesc = RECORD state: 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 system's 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.

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 [Goldberg]. Perhaps the most conspicuous among several significant achievements of this endeavor is the idea of applying the desktop metaphor to the display screen. This metaphor comprises a desktop and 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*.

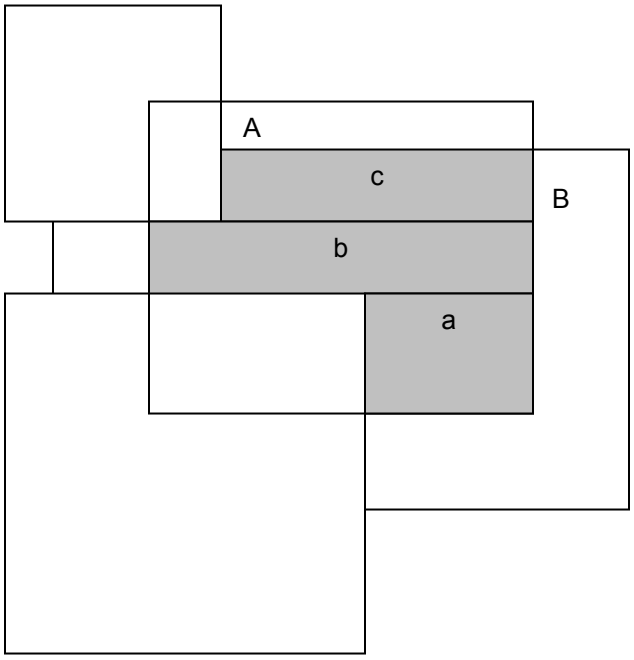


Figure 4.1 Desktop showing partially overlapping viewers

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 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 [Binding, Wille] 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 [Cohen]. 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.

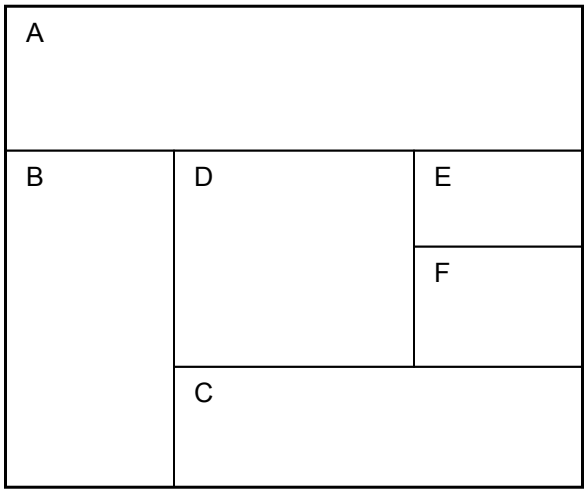


Figure 4.2 Viewer configuration resulting from unconstrained tiling

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

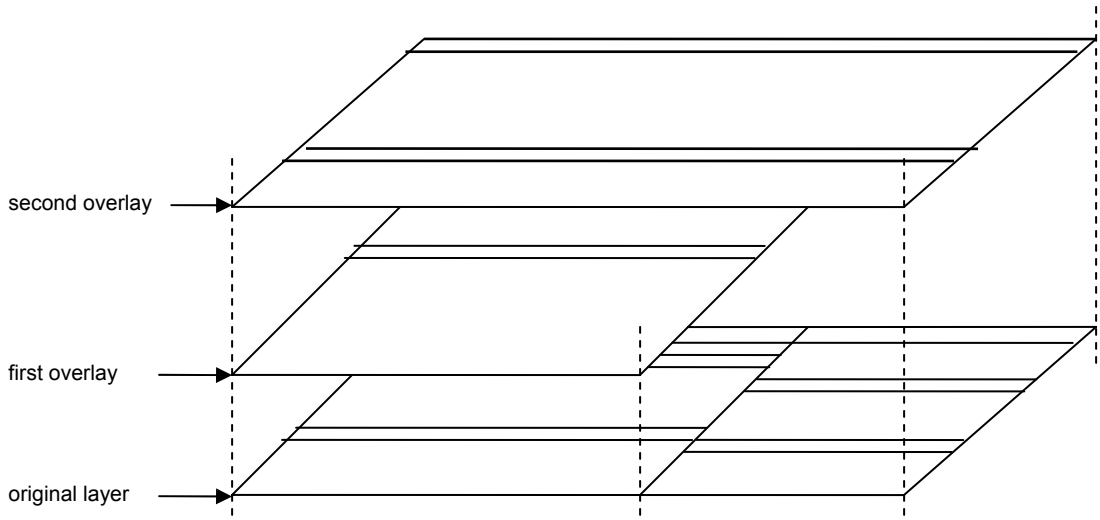


Figure 4.3 Overlay of tracks and sequences of tracks

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:

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

- 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).

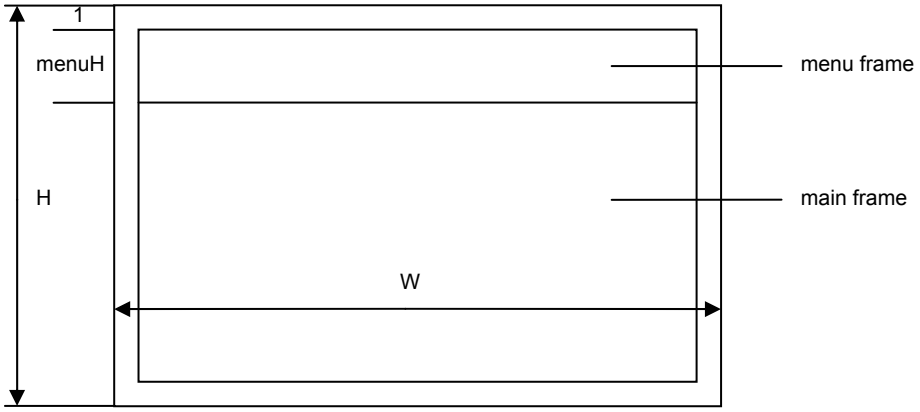


Figure 4.4 The compositional structure of a menu viewer

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.

Module	Type	Service
MenuViewer	Viewer	Message handling for menu viewers
Viewers	Viewer	Tiling viewer management
Display	Frame	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.

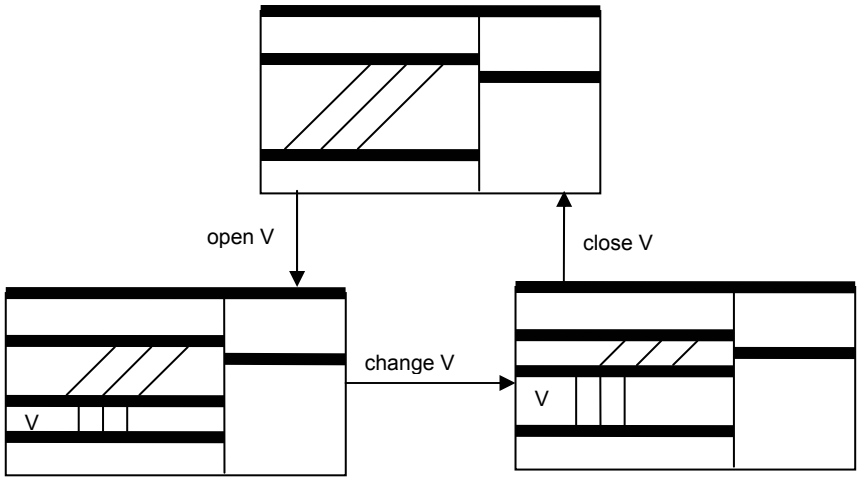


Figure 4.5 Basic operations on viewers

The last group of procedures provides miscellaneous services. Procedure *This* identifies the viewer displayed at *(X, Y).* Procedure *Nex*t 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 *Viewer*'s 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 module's 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.

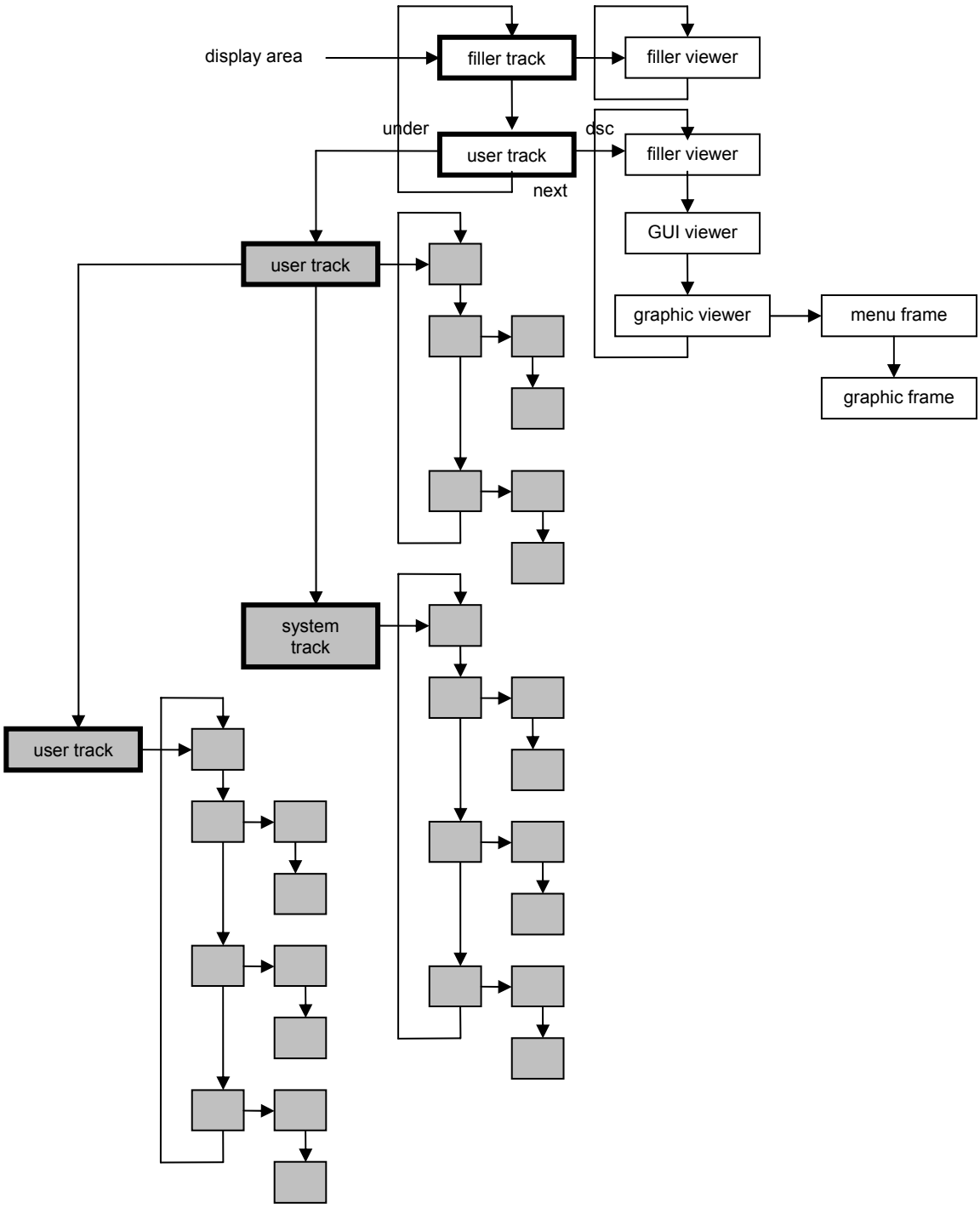


Figure 4.6 A snapshot of the internal data structure corresponding to Figure 4.3

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 # 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 }

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

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.

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

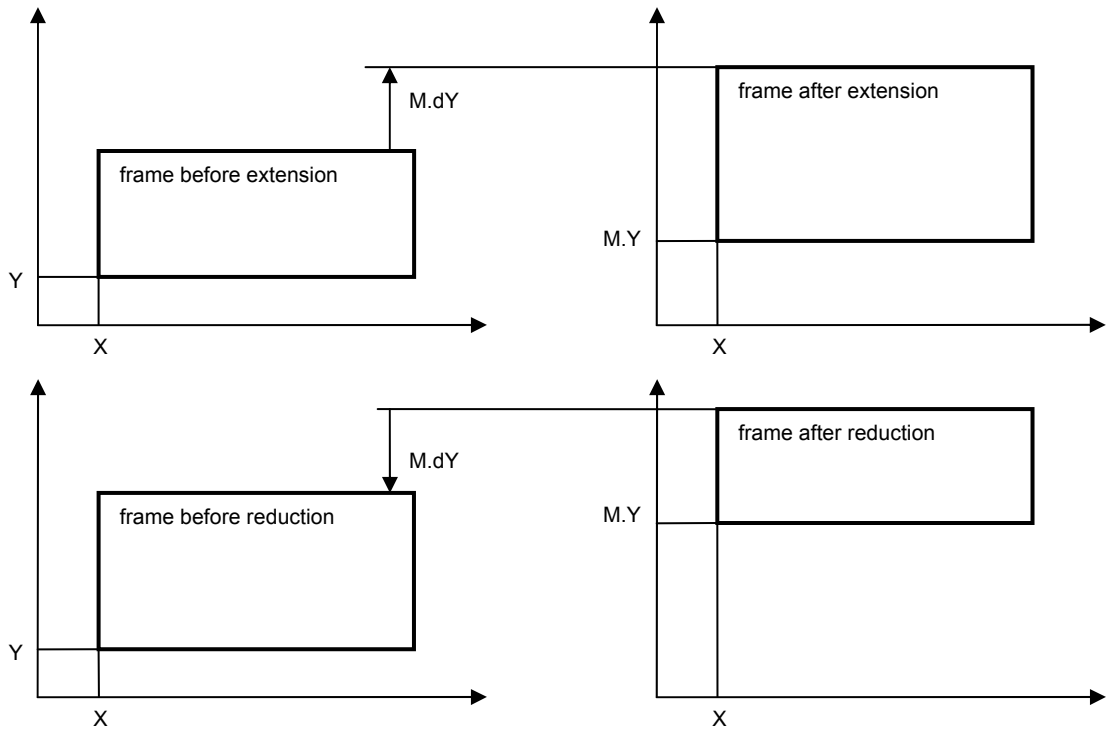


Figure 4.7 The modify frame operation

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:

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

Note that *invert* is equivalent with inverse video mode if s is *TRUE* for all pixels.

There are many different variants of raster operations. Some refer to a source block in the display area, others specify a constant pattern to be taken as source block. Some variants require replication of the source block within a given destination block (DX, DY, DW, DH) rather than simple placement.

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

```
DEFINITION Display;
  CONST  black = 0; white = 1; (*colors*)
    replace = 0; paint = 1; invert = 2; (*operation modes*)

  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 [Goldberg]. 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. Hence, the number of data bytes is ((w+7) DIV 8) * h. An example is shown in Figure 4.8.

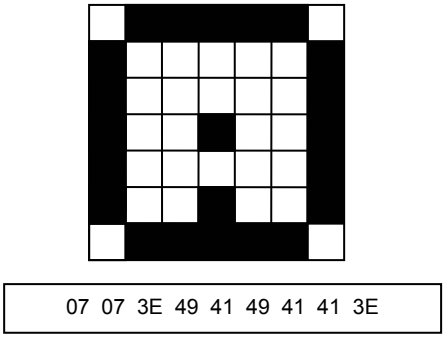


Figure 4.8 A pattern and its encoding as an array of bytes (in hex)

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.

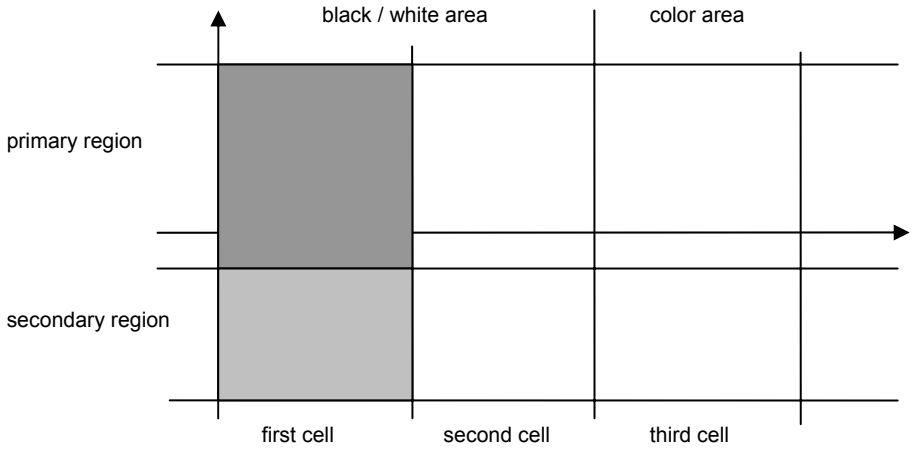


Figure 4.9 General, regular cell structure of display area

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 1/36'000 mm. 1/36'000 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.

References

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

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

[Wille] M. Wille, Overview: Entwurf und Realisierung eines Fenstersystems für Arbeitsplatzrechner, Diss. ETH Nr. 8771, 1988.

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

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

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*, or *s*. *len* gives the length of *s*,

nextCh typically exposes the character terminating the current token, and *line* counts the number of lines scanned.

The operations on scanners 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:

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

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

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

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

Our choice of an internal representation of text was determined by a catalogue of requirements and desired properties. The wish list looks like this:

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

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

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

2.) Every piece is represented by a descriptor *(f, pos, len),* where the components designate a file, a starting position, and a length respectively. The whole text is represented as a list of piece

descriptors (in short: piece list). The editing operations operate on the piece list rather than on the pieces themselves.

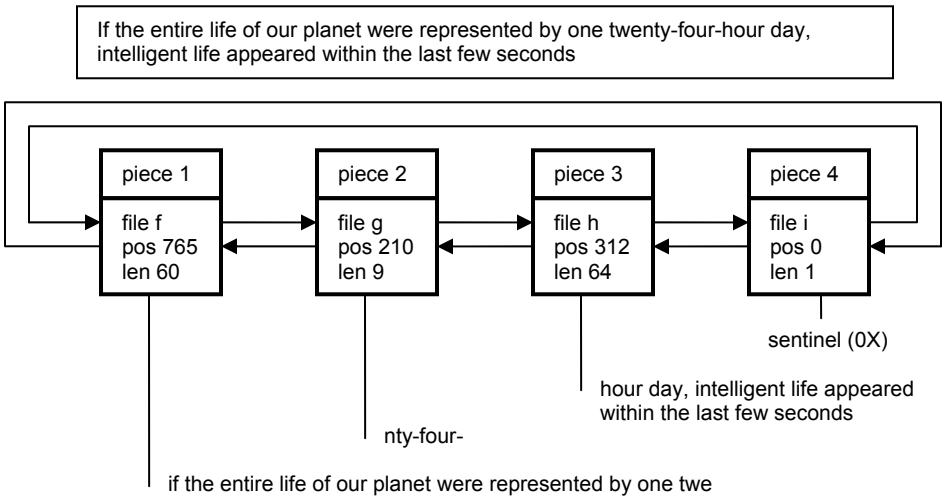


Figure 5.1 Piece chain representing a text

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.

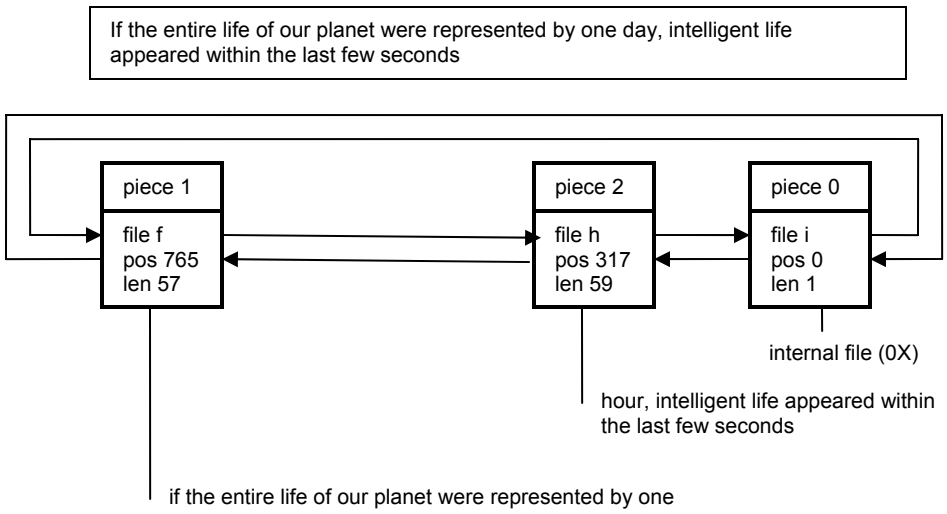


Figure 5.2 Piece chain after delete operation

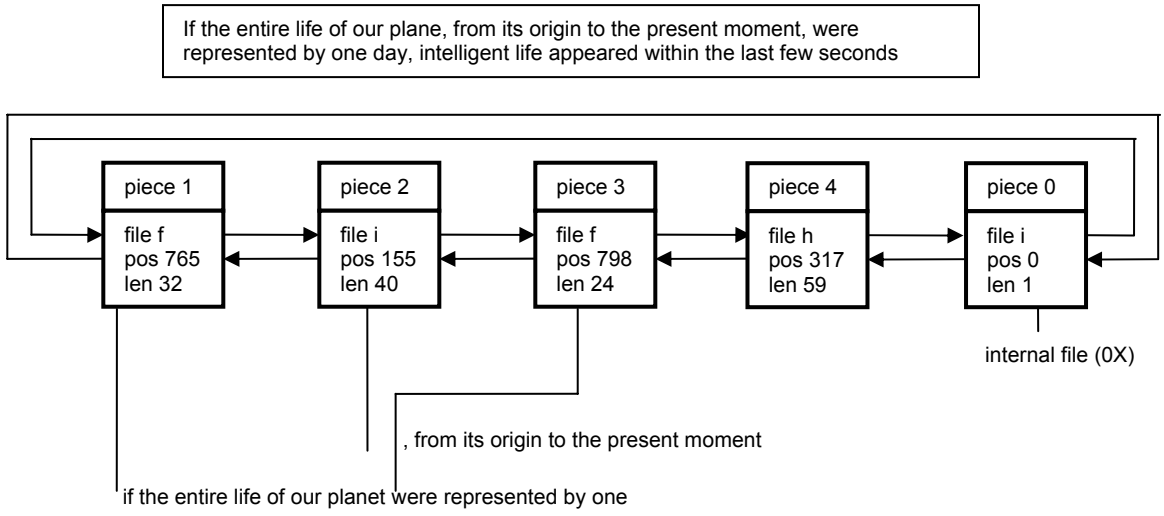


Figure 5.3 Piece chain after insert operation

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.

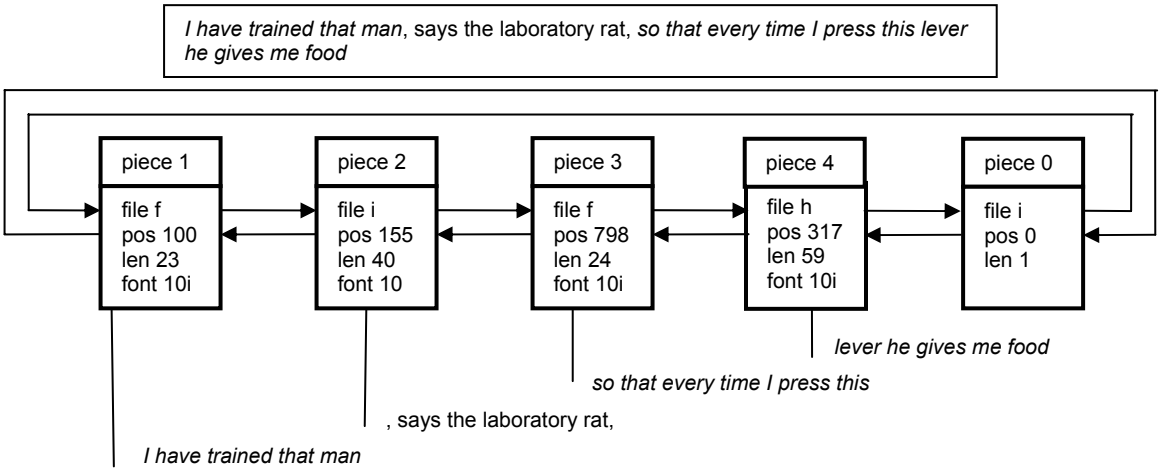


Figure 5.4 Generalized piece chain representing a text with looks

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

```
1)  PROCEDURE SplitPiece (p: Piece; off: INTEGER; VAR pr: Piece);
      VAR q: Piece;
    BEGIN
2)  IF off > 0 THEN NEW(q);
      q.fnt := p.fnt; q.col := p.col; q.voff := p.voff;
      q.len := p.len - off;
      q.f := p.f; q.off := p.off + off;
```

```
      p.len := off;
3)    q.next := p.next; p.next := q;
4)    q.prev := p; q.next.prev := q;
      pr := q
    ELSE pr := p
    END
  END SplitPiece;
```

Explanations:

1) return right part piece *pr* after split
2) generate new piece only if remaining length > 0
3) insert new piece in forward chain
4) insert new piece in backward chain

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

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

Explanations:

1) split piece to isolate point of insertion
2) adjust cache if necessary
3) merge pieces if possible
4) insert buffer
5) update text length
6) empty buffer
7) notify

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

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

Explanations:

1) read character from file and update looks in reader
2) if piece boundary reached
3) check if sentinel piece reached
4) move reader to next piece
5) position file rider

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

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

Writing a real number in decimal form is more complicated. It involves the computation of *m* and *d* from $x = n*2^e$ so that $x = m*10^d$ with $1.0 \leq m < 10.0$. First, *e* is obtained with the standard function UNPK(x, e), then *d* is computed (from the relationship $10^k = 2^{k*\log(10)}$) as $d = e/\log_2(10)$. In order to avoid a real division for obtaining *d*, we use the approximation $1.0 / \log_2(10) = 77$ DIV 256, and then compute x := x / Ten(e) or x := x * Ten(-e). Further details are to be taken from the listings of *WriteReal* and *WriteRealFix*.

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

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

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

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

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

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

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

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

Let us conclude this Section with two side-remarks and a summary.

Remarks:

For compatibility reasons, plain ASCII-files are accepted as text files as well. They are mapped to texts consisting of a single run with standard looks.

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

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

5.3. Text Frames

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

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

```
Frame = POINTER TO FrameDesc;

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

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

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

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

The auxiliary type *Location* summarizes information about a location in a text frame. Its definition is:

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

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

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

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

Explanations:

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

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

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

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

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

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

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

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

The module also supports mouse tracking inside text frames:

```
PROCEDURE TrackCaret (F: Frame; X, Y: INTEGER; VAR keysum: SET);
PROCEDURE TrackSelection (F: Frame; X, Y: INTEGER; VAR keysum: SET);
PROCEDURE TrackLine (F: Frame; X, Y: INTEGER; VAR org: INTEGER; VAR keysum: SET);
PROCEDURE TrackWord (F: Frame; X, Y: INTEGER; VAR pos: INTEGER; VAR keysum: SET);
```

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

```
Line = POINTER TO LineDesc;

LineDesc = RECORD
    len, wid: INTEGER;
    eot: BOOLEAN;
    next: Line
END;
```

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

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

```
Frame = POINTER TO FrameDesc;

FrameDesc = RECORD (Display.FrameDesc)
   text: Texts.Text;
   org, col, lsp: INTEGER;
   left, right, top, bot: INTEGER;
   markH, time: INTEGER;
   hasChar, hasSel, hasMark: BOOLEAN;
   carloc, selbeg, selend: Location;
→  trailer: Line
END;
```

Field *trailer* represents a sentinel element that closes the line list to a ring.

The line list contains useful summary information about the current contents of the text frame. It can be used beneficially by some related data types, for example by type *Location* that was introduced earlier:

```
Location = RECORD
   org, pos, dx, x, y: INTEGER;
→ lin: Line
END;
```

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

```
PROCEDURE Edit* (F: Frame; X, Y: INTEGER; Keys: SET);
  VAR M: CopyOverMsg;
    text: Texts.Text;
    buf: Texts.Buffer;
    v: Viewers.Viewer;
    loc0, loc1: Location;
    beg, end, time, pos: INTEGER;
    keysum: SET;
    fnt: Fonts.Font;
    col, voff: INTEGER;
BEGIN
  IF X < F.X + Min(F.left, barW) THEN  (*cursor is in scroll bar*)
    Oberon.DrawMouse(ScrollMarker, X, Y); keysum := Keys;
    IF Keys = {2} THEN   (*ML, scroll up*)
      TrackLine(F, X, Y, pos, keysum);
      IF (pos >= 0) & (keysum = {2}) THEN
        RemoveMarks(F); Oberon.RemoveMarks(F.X, F.Y, F.W, F.H);
        Show(F, pos)
      END
    ELSIF Keys = {1} THEN   (*MM*)  keysum := Keys;
      REPEAT Input.Mouse(Keys, X, Y); keysum := keysum + Keys;
        Oberon.DrawMouse(ScrollMarker, X, Y)
      UNTIL Keys = {};
      IF ~(keysum = {0, 1, 2}) THEN
        IF 0 IN keysum THEN pos := 0
        ELSIF 2 IN keysum THEN pos := F.text.len - 100
        ELSE pos := (F.Y + F.H - Y) * (F.text.len) DIV F.H
        END ;
        RemoveMarks(F); Oberon.RemoveMarks(F.X, F.Y, F.W, F.H);
        Show(F, pos)
      END
    ELSIF Keys = {0} THEN   (*MR, scroll down*)
      TrackLine(F, X, Y, pos, keysum);
      IF keysum = {0} THEN
        LocateLine(F, Y, loc0); LocateLine(F, F.Y, loc1);
        pos := F.org - loc1.org + loc0.org;
        IF pos < 0 THEN pos := 0 END ;
        RemoveMarks(F); Oberon.RemoveMarks(F.X, F.Y, F.W, F.H);
        Show(F, pos)
      END
    END
  ELSE  (*cursor is in text area*)
    Oberon.DrawMouseArrow(X, Y);
    IF 0 IN Keys THEN  (*MR: select*)
      TrackSelection(F, X, Y, keysum);
      IF F.hasSel THEN
        IF keysum = {0, 2} THEN (*MR, ML: delete text*)
          Oberon.GetSelection(text, beg, end, time);
```

```
            Texts.Delete(text, beg, end, TBuf);
            Oberon.PassFocus(Viewers.This(F.X, F.Y)); SetCaret(F, beg)
          ELSIF keysum = {0, 1} THEN  (*MR, MM: copy to caret*)
            Oberon.GetSelection(text, beg, end, time);
            M.text := text; M.beg := beg; M.end := end;
            Oberon.FocusViewer.handle(Oberon.FocusViewer, M)
          END
        END
      ELSIF 1 IN Keys THEN  (*MM: call*)
        TrackWord(F, X, Y, pos, keysum);
        IF (pos >= 0) & ~(0 IN keysum) THEN Call(F, pos, 2 IN keysum) END
      ELSIF 2 IN Keys THEN  (*ML: set caret*)
        Oberon.PassFocus(Viewers.This(F.X, F.Y));
        TrackCaret(F, X, Y, keysum);
        IF keysum = {2, 1} THEN (*ML, MM: copy from selection to caret*)
          Oberon.GetSelection(text, beg, end, time);
          IF time >= 0 THEN
            NEW(TBuf); Texts.OpenBuf(TBuf);
            Texts.Save(text, beg, end, TBuf); Texts.Insert(F.text, F.carloc.pos, TBuf);
            SetSelection(F, F.carloc.pos, F.carloc.pos + (end  - beg));
            SetCaret(F, F.carloc.pos + (end - beg))
          ELSIF TBuf # NIL THEN
            NEW(buf); Texts.OpenBuf(buf);
            Texts.Copy(TBuf, buf); Texts.Insert(F.text, F.carloc.pos, buf);
            SetCaret(F, F.carloc.pos + buf.len)
          END
        ELSIF keysum = {2, 0} THEN (*ML, MR: copy looks*)
          Oberon.GetSelection(text, beg, end, time);
          IF time >= 0 THEN
            Texts.Attributes(F.text, F.carloc.pos, fnt, col, voff);
            IF fnt # NIL THEN Texts.ChangeLooks(text, beg, end, {0,1,2}, fnt, col, voff) END
          END
        END
      END
    END
  END Edit;
```

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

1. In the scroll bar

primary key	secondary key	action
ML	-	scroll designated line to the top
MM	-	scroll proportional to mouse position
MM	ML	scroll to the end of the text
MM	MR	scroll to the beginning of the text

2. In the text area

primary key	secondary key	action
ML	-	set caret
ML	MM	copy selection to caret
ML	MT	copy looks
MM	-	call selected procedure
MR	-	select
MR	ML	delete selection
MR	MM	copy selection to caret

In the text area the keys are interpreted according to their generic semantics:

left key =	point key
middle key =	execute key
right key =	select key

Let us “zoom into” one of the editing operations, for example into *TrackCaret*.

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

Explanations:

1) guard guarantees non-empty line list
2) locates the character pointed at
3) drags caret to new location
4) - 5) tracks mouse and drags caret accordingly
6) set caret state

TrackCaret makes use of two auxiliary procedures *FlipCaret* and *LocateChar*. *FlipCaret* is used to turn off or on the pattern of the caret. *LocateChar* is an important operation that is used to locate the character at a given Cartesian position *(x, y)* within the frame.

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

Explanations:

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

```
   PROCEDURE LocateChar (F: Frame; x, y: INTEGER; VAR loc: Location);
     VAR R: Texts.Reader;
       patadr, pos, lim: INTEGER;
       ox, dx, u, v, w, h: INTEGER;
1)  BEGIN LocateLine(F, y, loc);
2)   lim := loc.org + loc.lin.len - 1;
3)   pos := loc.org; ox := F.left; dx := eolW;
4)   Texts.OpenReader(R, F.text, loc.org);
5)   WHILE pos # lim DO
6)     Texts.Read(R, nextCh);
7)     Fonts.GetPat(R.fnt.raster, nextCh, dx, u, v, w, h, patadr);
```

```
        IF ox + dx <= x THEN
          INC(pos); ox := ox + dx;
          IF pos = lim THEN dx := eolW END
        ELSE lim := pos
        END
      END ;
8)    loc.pos := pos; loc.dx := dx; loc.x := ox
  END LocateChar;
```

Explanations:

1) locate text line corresponding to at *y*
2) set limit to the last actual character on this line
3) start locating loop with first character on this line
4) setup reader and read first character of this line
5) - 7) scan through characters of this line until limit or *x* is reached
   6) get character width *dx* of current character
8) return location found

Note that the need to read characters from the text (again) in *LocateChar* has its roots in the so-called *proportional fonts* in which our rich texts are represented. We found that keeping character widths is an unnecessary optimization thanks to the buffering capabilities of the underlying file system. In the case of *fixed-pitch fonts* a simple division by the character width would be sufficient, of course.

Finally, procedure *LocateLine* uses the line list to locate the desired text line without reading text at all.

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

Explanations:

1) start with first line in the frame
2) - 3) traverse line chain until last line or y is reached
4) return found line

After text editing text, *rendering* is our next topic. Let us pursue the case in that a user pressed the point-key and then interclicked the middle key, corresponding to line 56) in procedure *Edit*. Remember that *notifier* is called at the end of every editing operation and in particular at the end of *Texts.Insert*. In case of standard text frames, the notifier simply *broadcasts* an update message into the display space:

```
PROCEDURE NotifyDisplay (T: Texts.Text; op, beg, end: INTEGER);
  VAR M: UpdateMsg;
BEGIN M.id := op; M.text := T; M.beg := beg; M.end := end; Viewers.Broadcast(M)
END NotifyDisplay;
```

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

```
  PROCEDURE Insert (F: Frame; beg, end: INTEGER);
    VAR R: Texts.Reader; L, L0, l: Line;
      org, len, curY, botY, Y0, Y1, Y2, dY, wid: INTEGER;
  BEGIN
```

```
      IF beg < F.org THEN F.org := F.org + (end - beg)
      ELSE
1)      org := F.org; L := F.trailer.next; curY := F.Y + F.H - F.top - asr;
        WHILE (L # F.trailer) & (org + L.len <= beg) DO
          org := org + L.len; L := L.next; curY := curY - lsp
2)      END;
3)      IF L # F.trailer THEN
          botY := F.Y + F.bot + dsr;
4)        Texts.OpenReader(R, F.text, org); Texts.Read(R, nextCh);
5)        len := beg - org; wid := Width(R, len);
6)        ReplConst (F.col, F, F.X + F.left + wid, curY - dsr, L.wid - wid, lsp, 0);
7)        DisplayLine(F, L, R, F.X + F.left + wid, curY, len);
8)        org := org + L.len; curY := curY - lsp;
          Y0 := curY; L0 := L.next;
          WHILE (org <= end) & (curY >= botY) DO
            NEW(l);
            Display.ReplConst(F.col, F.X + F.left, curY - dsr, F.W - F.left, lsp, 0);
            DisplayLine(F, l, R, F.X + F.left, curY, 0);
            L.next := l; L := l;
            org := org + L.len; curY := curY - lsp
9)        END;
10)       IF L0 # L.next THEN Y1 := curY;
11)         L.next := L0;
          WHILE (L.next # F.trailer) & (curY >= botY) DO
            L := L.next; curY := curY - lsp
12)       END;
          L.next := F.trailer;
          dY := Y0 - Y1;
          IF Y1 > curY + dY THEN
13)         Display.CopyBlock
              (F.X + F.left, curY + dY + lsp - dsr, F.W - F.left, Y1 - curY - dY,
               F.X + F.left, curY + lsp - dsr,
               0);
            Y2 := Y1 - dY
          ELSE Y2 := curY
          END;
14)       curY := Y1; L := L0;
          WHILE curY # Y2 DO
            Display.ReplConst(F.col, F.X + F.left, curY - dsr, F.W - F.left, lsp, 0);
            DisplayLine(F, L, R, F.X + F.left, curY, 0);
            L := L.next; curY := curY - lsp
15)       END
        END
      END
    END;
16)   UpdateMark(F)
  END Insert;
```

Some explanations:

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

Special care needs to be exercised in the implementation to avoid "flickering" and to minimize processing time. Concretely, the following measures are taken for this purpose:

1.) Avoid writing the same data again.
2.) Keep the number of newly rendered text lines at a minimum.
3.) Use raster operations (block moves) to adjust reusable displayed lines.

Of course, the rules governing the rendering and formatting process crucially influence the complexity of procedures like *Insert*. For text frames we have consciously chosen the simplest possible set of formatting rules. They can be summarized as:

1.) For a given text frame the distance between lines is constant.
2.) There are no implicit line breaks.

It is exactly this set of rules that makes it possible to display a text line in *one pass*. Two passes are inevitable if line distances have to adjust to font sizes or if lines must be broken implicitly.

Updating algorithms make use of the following one-pass rendering procedures *Width* and *DisplayLine*:

```
   PROCEDURE Width (VAR R: Texts.Reader; len: INTEGER): INTEGER;
     VAR patadr, pos, ox, dx, x, y, w, h: INTEGER;
1)  BEGIN pos := 0; ox := 0;
      WHILE pos < len DO
        Fonts.GetPat(R.fnt, nextCh, dx, x, y, w, h, pat);
        ox := ox + dx; INC(pos); Texts.Read(R, nextCh)
2)  END;
3)  RETURN ox
   END Width;
```

Explanations:

1) - 2) scan through *len* characters of this line
3) return accumulated width

Procedures *Width* and *LocateChar* are similar. Therefore the above comment about relying on the buffering capabilities of the underlying file system applies to procedure *Width* equally well.

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

Explanations:

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

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

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

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

Explanations

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

2) erase the old marker
3) draw the new marker

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

Putting into relation the different extensions of type *Display.Frame* that we came across in Chapters 4 and 5, we obtain the type hierarchy as shown in Figure 5.5.

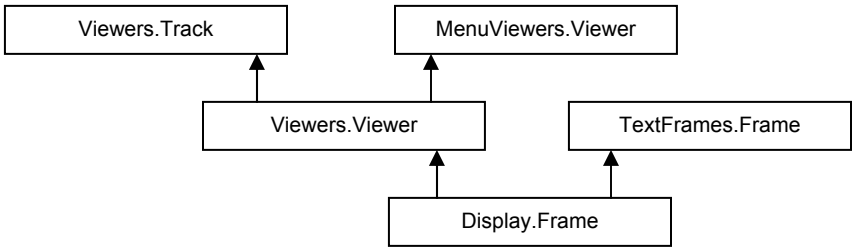


Figure 5.5 Extensions of tye *Display.Frame*

5.4. The Font Machinery

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

other kinds of icons may occur. In even more complex documents, mathematical or chemical formulae might flow within the text.

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

The imaging model of characters provides two levels of abstraction. On the first level, characters are black boxes specified by a set of metric data *x*, *y*, *w*, *h*, and *dx*. *(x, y)* is a vector from the current point of reference on the base line to the origin of the box. *w* and *h* are width and height of the box, and *dx* is the distance to the point of reference of the next character on the same base line. On the second level of abstraction, a character is defined by a digital pattern or *glyph* that is to be rendered into the box. Figure 5.6 visualizes this model of characters.

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

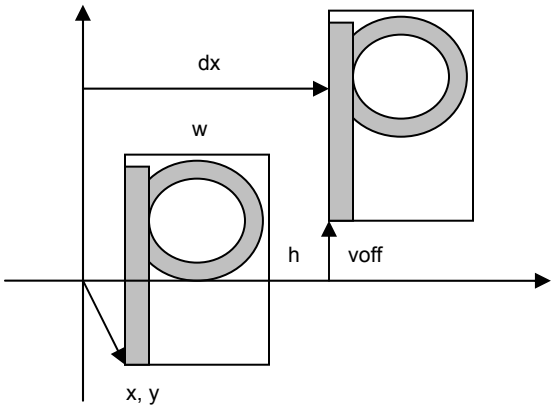


Figure 5.6 The geometric character model

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

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

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

```
MODULE Fonts;

  TYPE Font = POINTER TO FontDesc;
    FontDesc = RECORD
      name: ARRAY 32 OF CHAR;;
      height, minX, maxX, minY, maxY: INTEGER;
```

```
        next: Font
      END;

    VAR Default: Font;

    PROCEDURE GetPat(fnt: Font; ch: CHAR; VAR dx, x, y, w, h, patadr: INTEGER);
    PROCEDURE This (name: ARRAY OF CHAR): Font;
    PROCEDURE Free;
END Fonts.
```

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

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

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

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

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

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

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

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

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

5.5. The Edit toolbox

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

```
DEFINITION Edit;
   PROCEDURE Open; (*text viewer*)
   PROCEDURE Show; (*text*)
   PROCEDURE Locate; (*position*)
   PROCEDURE Search; (*pattern*)
   PROCEDURE Store; (*text*)

   PROCEDURE Recall; (*deleted text*)
   PROCEDURE CopyFont;
   PROCEDURE ChangeFont;
   PROCEDURE ChangeColor;
   PROCEDURE ChangeOffset;
END Edit.
```

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

References

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

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

6 The module loader

6.1. Linking and loading

When the execution of a command *M.P* is requested, module *M* containing procedure *P* must be loaded, unless it is already loaded because a command from the same module had been executed earlier or if the module had been imported by another module before. Modules are available in the form of so-called *object files*, generated by the compiler. The term *loading* refers to the transfer of the module code from the file into main memory, from where the processor fetches individual instructions. This transfer involves also a certain amount of transformation as required by the object file format on the one hand and the storage layout on the other. A system typically consists of many modules, and hence loading modules also involves linking them together, in particular linking them with already loaded modules. Before loading, references to another module's objects are relative to the base address of this module; the linking or binding process converts them into absolute addresses.

The linking process may require a significant amount of address computations. But they are simple enough and, if the data are organized in an appropriate way, can be executed very swiftly. Nevertheless, and surprisingly, in many operating systems linking needs more time than compilation. The remedy which system designers offer is a separation of linking from loading. A set of compiled modules is first linked; the result is a linked object file with absolute addresses. The loader then merely transfers the object file into main store.

We consider this an unfortunate solution. Instead of trying to cure an inadequacy with the aid of an additional processing stage and an additional tool, it is wiser to cure the malady at its core, namely to speed up the linking process itself. Indeed, there is no separate linker in the Oberon system. The linker and loader are integrated and fast enough to avoid any desire for pre-linking. Furthermore, the extensibility of a system crucially depends on the possibility to link additional modules to the ones already loaded by calls from any module. This is called *dynamic loading*. This is not possible with pre-linked object files. Newly loaded modules simply refer to the loaded ones, whereas pre-linked files lead to the presence of multiple copies of the same module code.

Evidently, the essence of linking is the conversion of relative addresses as generated by the compiler for all external references into absolute addresses as required during program execution. Before proceeding, we must consider an additional complication. Assume that a module *M1* is to be compiled which is a client of (that is, it imports) module *M0*. The interface of *M1* - in the form of a *symbol file* - does not specify the entry addresses of its exported procedures, but merely specifies a unique number (pno) for each one of them. The reason for this is that in this way the implementation of *M0* may be modified, causing a change of entry addresses, without affecting its interface specification. And this is a crucial property of the scheme of separate compilation of modules: changes of the implementation of *M0* must not necessitate the recompilation of clients (M1). The consequence is that the binding of entry addresses to procedure numbers must be performed by the linker. In order to make this possible, the object file must contain a list (table) of its entry addresses, one for each procedure number used as index to the table.

Similarly, the object file must contain a table of imported modules, containing their names. An external reference in the program code then appears in the form of a pair consisting of a module number (*mno*) - used as index to the import table (of modules) - and a procedure number (*pno*), used as index to the entry table of this module.

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

6.2. Module representation

The primary requirement is that a system must be represented in a form that allows to add new modules quickly. What is a sensible representation for this purpose? The simplest solution that comes to mind is a list of module blocks containing sections for the global data, for the program code, and perhaps meta data for the linking process. The list is rooted in a variable global to the loader module, here called *Modules*.

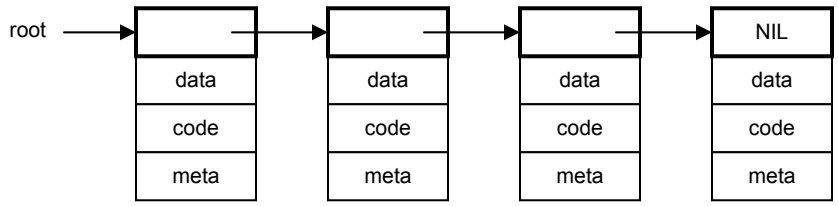


Fig. 6.1. System of 4 modules

The first part, containing the link to the next module, is called the *module descriptor*. On the Oberon System, it contains further links to the various sections of a module. The type *Module* is defined as follows:

```
TYPE Module = POINTER TO ModDesc;
   ModuleName = ARRAY 32 OF CHAR;
   ModDesc = RECORD
         name: ModuleName;
         next: Module;
         key, num, size, refcnt: INTEGER;
         data, code, imp, cmd, ent, ptr: INTEGER  (*addresses*)
   END ;
```

key is the module's key used for version consistency checking. The key changes if, and only if, the module's interface and thereby its symbol file changes. *num* is the module's number, which is the index of the module's entry in a global module table, referenced by the processor's MT register. The invariant relationship is

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

for all *mod* in the module list. *size* is the entire module block's size excluding the descriptor, and *refcnt* is the number of other modules importing this module. This number is used to check whether a module can be released by procedure *Modules.Free.*

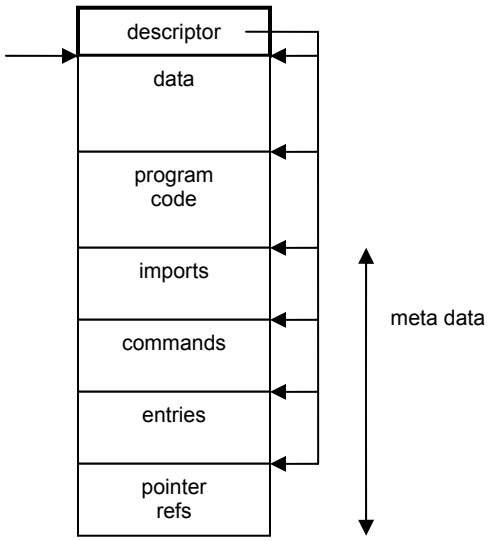


Figure 6.2. Module block headed by descriptor

The section with meta data follows the data and code areas and consists of several parts. *Imports* is an array of the modules imported by this module, each entry being the address of the respective module descriptor. *Commands* is a sequence of procedure identifiers followed by their offset in the code section. This section is used when activating a command. *Entries* is an array of offsets of all exported entities (including commands). This section is used by the loader itself for linking. *Pointer refs* is an array of offsets of global pointer variables in the data section. These are used by the garbage collector as the roots of graphs of heap objects in use.

6.3. The linking loader

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

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

```
mod := root;
WHILE (mod # NIL) & (name # mod.name) DO mod := mod.next END ;
IF mod = NIL THEN (*load*) F := ThisFile(name); Files.Set(R, F, 0);  ...
```

First, the header of the respective object file is read. It specifies the required size of the block which is allocated in the module area at the position indicated by the global variable *AllocPtr*. Then the list of imports of the module being imported is read, and these module are imported. Evidently procedure *Load* is used recursively. Because cyclic imports are excluded, recursion always terminates.

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

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

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

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

```
    offset := (dest - adr - 4) DIV 4;
    SYSTEM.PUT(adr, (offset MOD 1000000H) + 0F7000000H); (*compose*)
    adr := adr - disp*4
  END ;
```

After the module has been loaded successfully, its initialisation body is executed.

Apart from *Load,* module *Modules* also contains the procedures

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

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

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

6.4. The toolbox of the loader

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

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

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

6.5. The Oberon object file format

The name extension of object files is *.rsc*. Their syntax is the following:

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

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

7 The file system

7.1. Files

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

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

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

The file is not a static data structure like the array or the record, because the length may increase dynamically, i.e. during program execution. On the other hand, the sequence is less flexible than general dynamic structures, because it cannot change its form, but only its length, since elements can only be appended but not inserted. It might therefore be called a semi-dynamic structure.

The discipline of purely sequential access to a file is enforced by restricting access to calls of specific procedures, typically read and write procedures for scanning and generating a file. In the jargon of data processing, a file must be *opened* before reading or writing is possible. The opening implies the initialization of a reading and writing mechanism, and in particular the fixing of its initial position. Hence each (opened) file not only has a value and a length, but also a position attributed to it. If reading must occur from several positions (still sequentially) alternately, the file is "multiply opened"; it implies that the same file is represented by several variables, each denoting a different position.

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

Given a file *f*, a rider *r* is placed on a file by the call *Files.Set (r, f, pos)*, where pos indicates the position from which reading or writing is to start. Calls of *Files.Read (r, x)* and *Files.Write (r, x)*

implicitly increment the position beyond the element read or written, and the file is implicitly denoted via the explicit parameter *r*, which denotes a rider. The rider has two (visible) attributes, namely *r.eof* and *r.res*. The former is set to FALSE by *Files.Set*, and to TRUE when a read operation could not be performed, because the end of the file had been reached. r.res serves as a result variable in procedures *ReadBytes* and *WriteBytes* allowing one to check for correct termination.

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

```
DEFINITION Files;
    TYPE File = POINTER TO FileDesc;
            FileDesc = RECORD END ;
            Rider = RECORD eof: BOOLEAN; res: INTEGER END ;

    PROCEDURE Old(name: ARRAY OF CHAR): File;
    PROCEDURE New(name: ARRAY OF CHAR): File;
    PROCEDURE Register(f: File);
    PROCEDURE Close(f: File);
    PROCEDURE Purge(f: File);
    PROCEDURE Length(f: File): INTEGER;
    PROCEDURE Date(f: File): INTEGER);

    PROCEDURE Set(VAR r: Rider; f: File; pos: INTEGER);
    PROCEDURE ReadByte(VAR r: Rider; VAR x: BYTE);
    PROCEDURE ReadBytes(VAR r: Rider; VAR x: ARRAY OF BYTE; n: INTEGER);
    PROCEDURE Read(VAR r: Rider; VAR ch: CHAR);
    PROCEDURE ReadInt(VAR r: Rider; VAR n: INTEGER);
    PROCEDURE ReadSet(VAR r: Rider; VAR s: SET);
    PROCEDURE ReadReal(VAR r: Rider; VAR x: REAL);
    PROCEDURE ReadString(VAR r: Rider; VAR s: ARRAY OF CHAR);
    PROCEDURE ReadNum(VAR r: Rider; VAR n: INTEGER);

    PROCEDURE WriteByte(VAR r: Rider; x: BYTE);
    PROCEDURE WriteBytes(VAR r: Rider; x: ARRAY OF BYTE; n: INTEGER);
    PROCEDURE WriteInt(VAR r: Rider; n: INTEGER);
    PROCEDURE WriteSet(VAR r: Rider; s: SET);
    PROCEDURE WriteReal(VAR r: Rider; x: REAL);
    PROCEDURE WriteString(VAR r: Rider; x: ARRAY OF CHAR);
    PROCEDURE WriteNum(VAR r: Rider; n: INTEGER);
    PROCEDURE Pos(VAR r: Rider): INTEGER;
    PROCEDURE Base(VAR r: Rider): File;

    PROCEDURE Rename(old, new: ARRAY OF CHAR; VAR res: INTEGER);
    PROCEDURE Delete(name: ARRAY OF CHAR; VAR res: INTEGER);
END Files.
```

New(name) yields a new (empty) file without registering it in the directory. *Old(name)* retrieves the file with the specified name, or yields NIL, if it is not found in the directory. *Register(f)* inserts the name of f (specified in the call of *New*) in the directory. An already existing entry with this name is replaced. *Close(f)* must be called after writing is completed and the file is not to be

registered. *Close* actually stands for "close buffers", and is implied in the procedure *Register*. Procedure *Purge* will be explained at the end of section 7.2.

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

```
VAR f: Files.File; r: Files.Rider;

f := Files.Old(name);
IF f # NIL THEN
    Files.Set (r, f, 0); Files.Read (r, x);
    WHILE ~ r.eof DO ... x ...; Files.Read(r, x) END
END
```

The analogous template for a purely sequential writing is:

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

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

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

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

7.2 Implementation of files on a random-access store

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

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

The second most crucial design decision concerns the data structure in which sectors are organized; it determines how a succeeding sector is located. The simplest solution is to link sectors in a list. This is acceptable if access is to be restricted to purely sequential scans. Although this would be sufficient for most applications, it is unnecessarily restrictive for media

other than purely sequential ones (tapes). After all, it is sometimes practical to position a rider at an arbitrary point in the file rather than always at its beginning. This is made possible by the use of an indexed sector table, typically stored as a header in the file. The table is an array of the addresses of the file's data sectors. Unfortunately, the length of the table needed is unknown. Choosing a fixed length for all files is controversial, because it inevitably leads to either a limitation of file length (when chosen too small) that is unacceptable in some applications, or to a large waste of file space (when chosen too large). Experience shows that in practice most files are quite short, i.e. in the order of a few thousand bytes. The dilemma is avoided by a two-level table, i.e. by using a table of tables.

The scheme chosen in Oberon is slightly more complex in order to favor short files (< 64 K bytes): Each file header contains a table of 64 entries, each pointing to a 1K byte sector. Additionally, it contains a table of 12 entries, the so-called extensions, each pointing to an index sector containing 256 further sector pointers. The file length is thereby limited to 64 + 12*256 sectors, or 3'211'264 bytes (minus the length of the header). The chosen structure is illustrated in Fig. 7.1. sec[0] always points to the sector containing the file header.

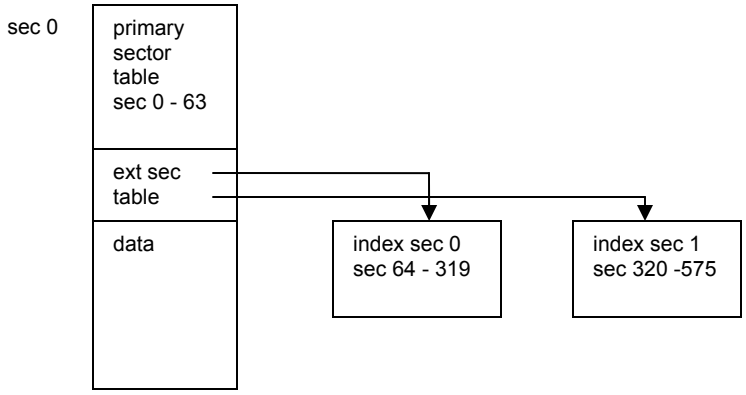


Figure 7.1 File header and extension sectors

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

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

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

```
Rider = RECORD
        eof: BOOLEAN; res, pos, adr: INTEGER;
        file: File
END
```

A rider is initialised by a call *Set(r, f, pos)*, which places the rider *r* on file *f* at position *pos*. From this it is clear that the rider record must contain fields denoting the attached file and the rider's position on it. We note that they are *not* exported. However, their values can be obtained by the

function procedures *Pos(r)* and *Base(r)*. This allows a (hidden) representation most appropriate for an efficient implementation of *Read* and *Write* without being unsafe.

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

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

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

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

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

```
MODULE MFiles;   (*NW 24.8.90 / 12.10.90 / 20.6.2013*)
  IMPORT SYSTEM, Kernel, FileDir;
  (*A file consists of a sequence of sectors. The first sector contains the header.
    Part of the header is the sector table, an array of addresses to the sectors.
    A file is referenced through riders each of which indicates a position.*)

  CONST
    HS  = FileDir.HeaderSize;
    SS  = FileDir.SectorSize;
    STS= FileDir.SecTabSize;
    XS  = FileDir.IndexSize;

  TYPE File*   = POINTER TO FileDesc;
    Index  = POINTER TO IndexRecord;
    IndexRecord = RECORD sec: FileDir.IndexSector END ;

    Rider* =
      RECORD eof*: BOOLEAN;
        res*, pos, adr: INTEGER;
        file: File
      END ;

    FileDesc =
      RECORD mark: INTEGER;
        name: FileDir.FileName;
        len, date: INTEGER;
        ext:  ARRAY FileDir.ExTabSize OF Index;
```

```
      sec: FileDir.SectorTable
    END ;

PROCEDURE Old*(name: ARRAY OF CHAR): File;
  VAR head: INTEGER;
    namebuf: FileDir.FileName;
BEGIN
  FileDir.Search(name, head); RETURN SYSTEM.VAL(File, head)
END Old;

PROCEDURE New*(name: ARRAY OF CHAR): File;
  VAR f: File; head: INTEGER;
BEGIN f := NIL; Kernel.AllocSector(0, head);
  IF head # 0 THEN
    f := SYSTEM.VAL(File, head); f.mark := FileDir.HeaderMark;
    f.len := HS; f.name := name;
    f.date := Kernel.Clock(); f.sec[0] := head
  END ;
  RETURN f
END New;

PROCEDURE Register*(f: File);
BEGIN
  IF (f # NIL) & (f.name[0] > 0X) THEN FileDir.Insert(f.name, f.sec[0]) END ;
END Register;

PROCEDURE Length*(f: File): INTEGER;
BEGIN RETURN f.len - HS
END Length;

PROCEDURE Date*(f: File): INTEGER;
BEGIN RETURN f.date
END Date;

PROCEDURE Set*(VAR r: Rider; f: File; pos: LONGINT);
  VAR m, n: INTEGER;
BEGIN  r.eof := FALSE; r.res := 0;
  IF f # NIL THEN
    IF pos < 0 THEN r.pos := HS
    ELSIF pos > f.len - HS THEN r.pos := f.len
    ELSE r.pos := pos + HS
    END ;
    r.file := f; m := r.pos DIV SS; n := r.pos MOD SS;
    IF m < STS THEN r.adr := f.sec[m] + n
    ELSE r.adr := f.ext[(m-STS) DIV XS].sec[(m-STS) MOD XS] + n
    END
  END
END Set;

PROCEDURE ReadByte*(VAR r: Rider; VAR x: BYTE);
  VAR m: INTEGER;
BEGIN
  IF r.pos < r.file.len THEN
    SYSTEM.GET(r.adr, x); INC(r.adr); INC(r.pos);
    IF r.adr MOD SS = 0 THEN
      m := r.pos DIV SS;
      IF m < STS THEN r.adr := r.file.sec[m]
      ELSE r.adr := r.file.ext[(m-STS) DIV XS].sec[(m-STS) MOD XS]
      END
    END
  ELSE x := 0; r.eof := TRUE
  END
END ReadByte;
```

```
  PROCEDURE WriteByte*(VAR r: Rider; x: BYTE);
    VAR k, m, n, ix: INTEGER;
  BEGIN
    IF r.pos < r.file.len THEN
      m := r.pos DIV SS; INC(r.pos);
      IF m < STS THEN r.adr := r.file.sec[m]
      ELSE r.adr := r.file.ext[(m-STS) DIV XS].sec[(m-STS) MOD XS]
      END
    ELSE
      IF r.adr MOD SS = 0 THEN
        m := r.pos DIV SS;
        IF m < STS THEN Kernel.AllocSector(0, r.adr); r.file.sec[m] := r.adr
        ELSE n := (m - STS) DIV XS; k := (m - STS) MOD XS;
          IF k = 0 THEN (*new index*)
            Kernel.AllocSector(0, ix); r.file.ext[n] := SYSTEM.VAL(Index, ix)
          END ;
          Kernel.AllocSector(0, r.adr); r.file.ext[n].sec[k] := r.adr
        END
      END ;
      INC(r.pos); r.file.len := r.pos
    END ;
    SYSTEM.PUT(r.adr, x); INC(r.adr)
  END WriteByte;

  PROCEDURE Pos*(VAR r: Rider): INTEGER;
  BEGIN RETURN r.pos - HS
  END Pos;

  PROCEDURE Base*(VAR r: Rider): File;
  BEGIN RETURN r.file
  END Base;
END MFiles.
```

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

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

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

The following procedures used for allocating, deallocating, and marking sectors in the sector reservation table are defined in module Kernel:

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

7.3 Implementation of files on a disk

First we recall that the organization of files as sets of individually allocated blocks (sectors) is inherently required by the allocation considerations of dynamically growing sequences. However, if the storage medium is a tape, a disk, or a flash-RAM, there exists an additional reason for the use of blocks. They constitute the subsequences to be individually buffered for transmission in order to overcome the timing constraints imposed by the medium. If an adequate space utilization is to be achieved, the blocks must not be too long. A typical size is 1, 2, or 4K bytes.

This necessity of buffering has a profound influence on the implementation of file access. The complication arises because the abstraction of the sequence of individual bytes needs to be maintained. The increase in complexity of file access is considerable, as can be seen by comparing the program listings of the two respective implementations.

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

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

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

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

1. The first buffer is created when the file is opened (New, Old).
2. Additional buffers may be allocated when a rider is placed (or repositioned) on the file.
3. At most four buffers are connected to the same file.
4. Purely sequential movements of riders do not cause allocation of buffers.
5. Separate buffers are generated when extensions of the file's sector table need be accessed (rider position > 64K). Each buffers the 256 sector addresses of the respective index sector.

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

```
File =      POINTER TO FileDesc;
Buffer =    POINTER TO BufferRecord;
Index =     POINTER TO IndexRecord;

FileDesc = RECORD next: File;
              aleng, bleng: INTEGER;          (*file length*)
              nofbufs: INTEGER;    (*no. of buffers allocated*)
              modH, registered: BOOLEAN;  (*header has been modified*)
              firstbuf: Buffer:          (*head of buffer chain*)
              sechint: DiskAdr;        (*sector hint*)
              name: FileDir.FileName;
              date: INTEGER;
```

```
        ext: ARRAY FileDir.ExTabSize OF Index;
        sec: ARRAY 64 OF DiskAdr
    END;

BufferRecord = RECORD apos, lim: INTEGER;(*lim = no. of bytes*)
        mod: BOOLEAN;       (*buffer has been modified*)
        next: Buffer;       (*buffer chain*)
        data: FileDir.DataSector
    END;

IndexRecord = RECORD adr: DiskAdr;
        mod: BOOLEAN;       (*index record has been modified*)
        sec: FileDir.IndexSector
    END;

Rider =  RECORD eof: BOOLEAN;       (*end of file reached*)
        res: INTEGER;               (*no. of unread bytes*)
        file: File;
        apos, bpos: INTEGER;        (*position*)
        buf: Buffer                 (*hint: likely buffer*)
    END ;
```

In order to increase efficiency of access, riders have been provided with a field containing the address of the element of the rider's position. From the conditions stated above for the allocation of buffers, it is evident that the value of this field can be a hint only. This implies that there can be no reliance on its information. Whenever it is used, its validity has to be checked. The check consists in a comparison of the riders' position *r.apos* with the hinted buffer's actual position *r.buf.apos*. If they differ, a buffer with the desired position must be searched and, if not present, allocated. The advantage of the hint lies in the fact that the hint is correct with a very high probability. The check is included in procedures *Read, ReadByte, Write, and WriteByte*.

Some fields of the record types require additional explanations:

1. The length is stored in a "preprocessed" form, namely by the two integers *aleng* and *bleng* such that aleng is a sector number and

```
length  = (aleng * SS) + bleng - HS
aleng   = (length + HS) DIV SS
bleng   = (length + HS) MOD SS
```

The same holds for the form of the position in riders (apos, bpos).

2. The field *nofbufs* indicates the number of buffers in the list headed by *firstbuf*:

```
1 <= nofbufs <= Maxbufs.
```

3. Whenever data are written into a buffer, the file becomes inconsistent, i.e. the data on the disk are outdated. The file is updated, i.e. the buffer is copied into the corresponding disk sector, whenever the buffer is reallocated, e.g. during sequential writing after the buffer is full and is "advanced". During sequential reading, a buffer is also advanced and reused, but needs not be copied onto disk, because it is still consistent. Whether a buffer is consistent or not is indicated by its state variable *mod* (modified). Similarly, the field *modH* in the file descriptor indicates whether or not the header had been modified.

4. The field *sechint* records the number of the last sector allocated to the file and serves as a hint to the kernel's allocation procedure, which allocates a next sector with an address larger than the hint. This is a measure to gain speed in sequential scans.

5. The buffer's position is specified by its field *apos*. Used as index in the file header's sector table, it yields the sector corresponding to the current buffer contents. The field *lim* specifies the number of bytes s stored in the buffer. Reading cannot proceed beyond this limiting index; writing beyond it implies an increase in the file's length. All buffers except the one for the last sector are filled and specify lim = SS.

6. The hidden rider field *buf* is merely a hint to speed up localization of the concerned buffer. A hint is likely, but not guaranteed to be correct. Its validity must be checked before use. The buffer hint is invalidated when a buffer is reallocated and/or a rider is repositioned.

The structure of riders remains practically the same as for files using main store. The hidden field *adr* is merely replaced by a pointer to the buffer covering the rider's position. A configuration of a file *f* with two riders is shown in Fig 7.2.

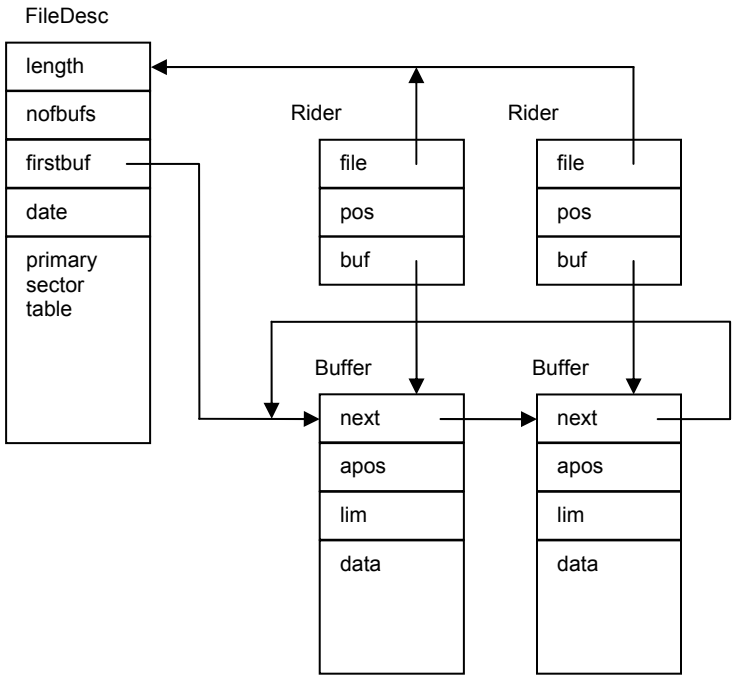


Figure 7.2 File f with two riders and two buffers

Some comments concerning module *Files* follow.

1. After the writing of a file has been completed, its name is usually registered in the directory. *Register* invokes procedure *Unbuffer*. It inspects the associated buffers and copies those onto disk which had been modified. During this process, new index sectors may have to be transferred as well. If a file is to remain anonymous and local to a module or command, i.e. is not to be registered, but merely to be read, the release of buffers must be specified by an explicit call to *Close* (meaning "close buffers"), which also invokes *Unbuffer*.

2. Procedure *Old* (and for reasons of consistency also *New*) deviates from the general Oberon programming rule that an object be allocated by the calling (instead of the called) module. This rule would suggest the statements

```
New(f); Files.Open(f, name)
```

instead of *f := Files.Old(name)*. The justification for the rule is that any extension of the type of *f* could be allocated, providing for more flexibility. And the reason for our deviation in the case of files is that, upon closer inspection, not a new file, but only a new descriptor is to be allocated. The distinction becomes evident when we consider that several statements *f := Files.Old(name)* with different *f* and identical name may occur, probably in different modules. In this case, it is necessary that the *same* descriptor is referenced by the delivered pointers in order to avoid file inconsistency. Each (opened) file must have exactly one descriptor. When a file is opened, the first action is therefore to inspect whether a descriptor of this file already exists. For this purpose, all descriptors are linked together in a list anchored by the global variable root and linked by the descriptor field *next*. This measure may seem to solve the problem of avoiding inconsistencies

smoothly. However, there exists a pitfall that is easily overlooked: All opened files would permanently remain accessible via *root*, and the garbage collector could never remove a file descriptor nor its associated buffers. This would be unacceptable. In order to hide this list from the garbage collector, it is represented by integers (addresses) instead of pointers.

3. Sector pointers are represented by sector numbers of type INTEGER. Actually, we use the numbers multiplied by 29. This implies that any single-bit error leads to a number which is not a multiple of 29, and hence can easily be detected. Thereby the crucial sector addresses are software parity checked and are safe (against single-bit errors) even on computers without hardware parity check. The check is performed by procedures *Kernel.GetSector* and *Kernel.PutSector*.

7.4 The file directory

A directory is a set of pairs, each pair consisting of a name (key) and an object (here: file). It serves to retrieve objects by their name. If efficiency matters, the directory is organized as an ordered set, ordered according to the keys. The most frequently used structures for ordered sets are trees and hash tables. The latter have disadvantages when the size of the set is unknown, particularly when its order of magnitude is unknown, and when deletions occur. The Oberon system therefore uses a tree structure for its file directory, more specifically a B-tree, which was developed especially for cases where not individual pairs, but only sets of pairs as a whole (placed on a disk sector) can be accessed.

For a thorough study of B-trees we refer the reader to the literature [1, 2]. Here it must suffice to specify the B-tree's principal characteristics:

1. In a B-tree of order N, each node (called *page*) contains m elements (pairs), where N <= m <= 2N, except the root, where 0 <= m <= 2N.
2. A page with m elements has either 0 descendants, in which case it is called a *leaf page*, or m + 1 descendants.
3. All leaf pages are on the same (bottom) level.

From 3. it follows that the B-tree is a balanced tree. Its height, and with it the longest path's length, has an upper bound of, roughly, 2 * log k, where k is the number of elements and the logarithm is taken to the base N and rounded up to the next larger integer. Its minimal height is log k taken to the base 2N.

On each page, space must be available for 2N elements and for 2N + 1 references to descendants. Hence, N is immediately determined by the size of a page and the size of elements. In the case of the Oberon system, names are limited to 32 characters (bytes), and the object is a reference to the associated file (4 bytes). Each descendant pointer takes 4 bytes, and the page size is given by the sector size (1024) minus the number of bytes needed to store m (2 bytes). Hence

N = ((1024 - 2 - 4) DIV (32 + 4 + 4)) DIV 2 = 12

A B-tree of height h and order 12 may contain the following minimal and maximal number of elements:

height	minimum	maximum
1	0	24
2	25	624
3	625	15624
4	15625	390624

It follows that the height of the B-tree will never be larger than 4, if the disk has a capacity of less than about 400 Mbyte, and assuming that each file occupies a single 1K sector. It is rarely larger than 3 in practice.

The definition of module *FileDir* shows the available directory operations. Apart from the procedures *Search, Insert, Delete*, and *Enumerate*, it contains some data definitions, and it should be considered as the non-public part of the file system's interface.

```
DEFINITION FileDir;
  IMPORT SYSTEM, Kernel;
  CONST
    FnLength = 32;      (*max length of file name*)
    SecTabSize = 64;    (*no. of entries in primary table*)
    ExTabSize = 12;
    SectorSize = 1024;
    IndexSize = SectorSize DIV 4;          (*no. of entries in index sector*)
    HeaderSize = 352;
    DirRootAdr = 29;
    DirPgSize = 24;     (*max no. of elements on page*)

  TYPE DiskAdr = INTEGER;
    FileName = ARRAY FnLength OF CHAR;
    SectorTable = ARRAY SecTabSize OF DiskAsr;
    ExtensionTable = ARRAY ExTabSize OF DiskAdr;
    EntryHandler = PROCEDURE (name: FileName; sec: DiskAdr; VAR continue: BOOLEAN);

    FileHeader = RECORD (*first page of each file on disk*)
        mark: INTEGER;
        name: FileName;
        aleng, bleng, date: INTEGER;
        ext: ExtensionTable;
        sec: SectorTable
      END ;

    IndexSector = RECORD (Kernel.Sector)
        x: ARRAY IndexSize OF LONGINT;
      END ;

    DataSector = ARRAY SectorSize OF BYTE;

    DirEntry = RECORD
        name: FileName;
        adr, p: DiskAdr
      END ;

    DirPage = RECORD
        mark: INTEGER;
        m: INTEGER;   (*no. of elements on page*)
        p0: DiskAdr;
        e: ARRAY DirPgSize OF DirEntry;
      END ;

  PROCEDURE Search(name: FileName; VAR fad: DiskAdr);
  PROCEDURE Insert(name: FileName; fad: DiskAdr);
  PROCEDURE Delete(name: FileName; VAR fad: DiskAdr);
  PROCEDURE Enumerate(prefix: ARRAY OF CHAR; proc: EntryHandler);

END FileDir.
```

Procedures *Search, Insert*, and *Delete* represent the typical operations performed on a directory. Efficiency of the first operation is of primary importance. But the B-tree structure also guarantees efficient insertion and deletion, although the code for these operations is complex. Procedure *Enumerate* is used to obtain excerpts of the directory. The programmer must guarantee that no directory changes are performed by the parametric procedure of Enumerate.

As in the presentation of module *Files*, we first discuss a version that uses main storage rather than a disk for the directory. This allows us to concentrate on the algorithms for handling the directory, leaving out the additional complications due to the necessity to read pages (sectors) into main store for selective updating and of restoring them onto disk. In particular, we point out

the definitions of the data types for B-tree nodes, called *DirPage*, and elements, called *DirEntry*. The component *E.p* of an entry *E* points to the page in which all elements (with index *k*) have keys *E.p.e[k].name* > *E.name*. The pointer *p.p0* points to a page in which all elements have keys *p.p0.e[k].name* < *p.e[0].name*. We can visualize these conditions by Fig. 7.3, where names have been replaced by integers as keys.

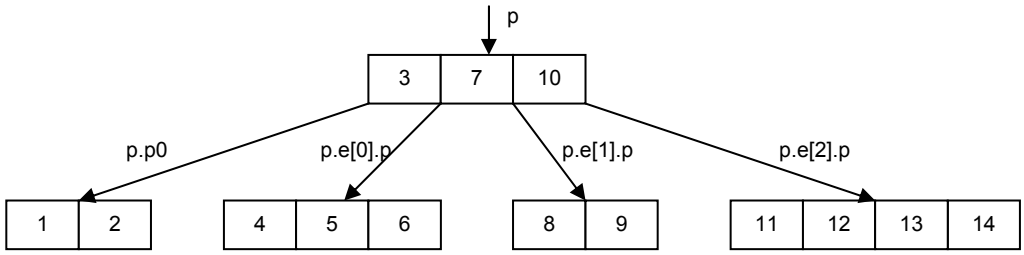


Figure 7.3 Example of a B-tree of order 2

Procedure *Search* starts by inspecting the root page. It performs a binary search among its elements, according to the following algorithm. Let e[0 ... m-1] be the ordered keys and x the search argument.

```
L := 0;  R := m;
WHILE L < R DO
    i := (L+R) DIV 2;
    IF x <= e[i] THEN R := i ELSE L := i  + 1 END
END;
IF (R < m) & (x = e[R]) THEN found END
```

The invariant is

```
e[L-1] < x <= e[R]
```

If the desired element is not found, the search continues on the appropriate descendant page, if there is one. Otherwise the element is not contained in the tree.

Procedures *insert* and *delete* use the same algorithm for searching an element within a page. However, they use recursion instead of iteration to proceed along the search path of pages. We recall that the depth of recursion is at most four. The reason for the use of recursion is that it facilitates the formulation of structural changes, which are performed during the "unwinding" of recursion, i.e. on the return path. First, the insertion point (respectively the position of the element to be deleted) is searched, and then the element is inserted (deleted).

Upon insertion, the number of elements on the insertion page may become larger than 2N, violating B-tree condition 1. This situation is called *page overflow*. The invariant must be reestablished immediately. It could be achieved by moving one element from either end of the array e onto a neighbouring page. However, we choose not to do this, and instead to split the overflowing page into two pages immediately. The process of a *page split* is visualized by Fig 7.4, in which we distinguish between three cases, namely R < N, R = N, and R > N, where R marks the insertion point. *a* denotes the overflowing, *b* the new page, and *u* the inserted element.

The 2N + 1 elements (2N from the full page *a*, plus the one element *u* to be inserted) are equally distributed onto pages *a* and *b*. One element *v* is pushed up in the tree. It must be inserted in the ancestor page of *a*. Since that page obtains an additional descendant, it must also obtain an additional element in order to maintain B-tree rule 2.

A page split may thus propagate, because the insertion of element v in the ancestor page may require a split once again. If the root page is full, it is split too, and the emerging element *v* is inserted in a new root page containing a single element. This is the only way in which the height of a B-tree can increase.

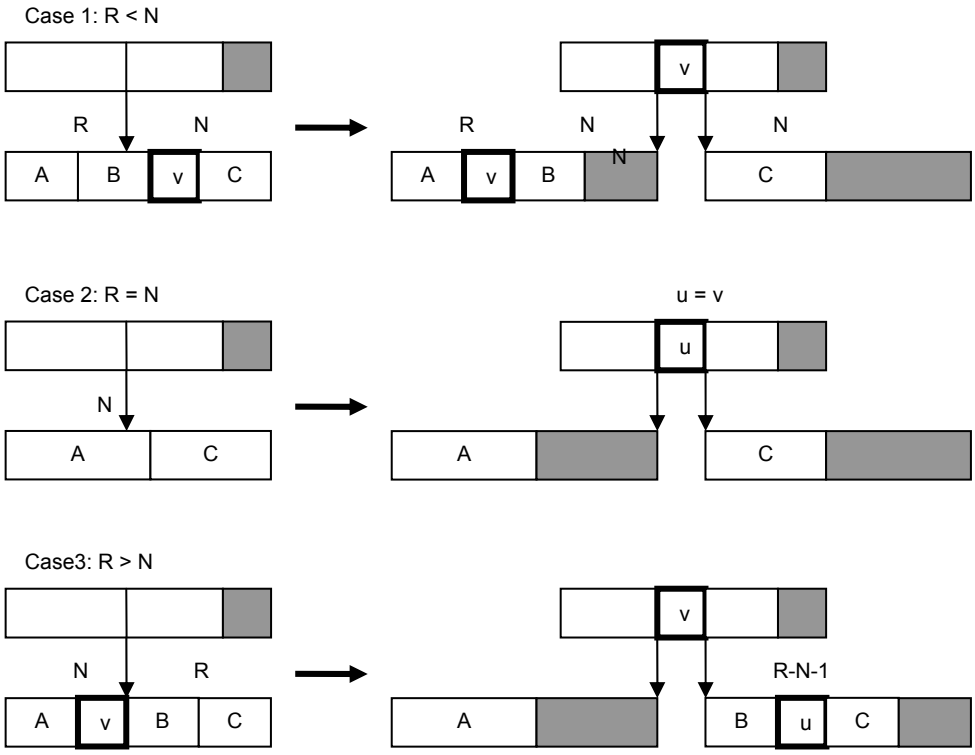


Figure 7.4 Page split when inserting element u

When an element is to be deleted, it cannot simply be removed, if it resides on an internal page. In this case, it is first replaced by another element, namely one of the two neighbouring elements on a leaf page, i.e. the next smaller (or next larger) element, which is always on a leaf page. In the presented solution, the replacing element is the largest on the left subtree (see procedure del). Hence, the actual deletion always occurs on a leaf page.

Upon deletion, the number of elements in a page may become less than N, violating invariant 1. This event is called *page underflow*. Since restructuring the tree is a relatively complicated operation, we first try to reestablish the invariant by borrowing an element from a neighbouring page. In fact, it is reasonable to borrow several elements, and thereby to decrease the likelihood of an underflow on the same page upon further deletions. The number of elements that could be taken from the neighbouring page b is b.m - N. Hence we will borrow

k = (b.m - N + 1) DIV 2

elements. The process of *page balancing* then distributes the elements of the underflowing and its neighbouring page equally to both pages (see procedure underflow).

However, if (and only if) the neighbouring page has no elements to spare, the two pages can and must be united. This action, called *page merge*, places the N-1 elements from the underflowing page, the N elements from the neighbouring page, plus one element from the ancestor page onto a single page of size 2N. One element must be taken from the ancestor page, because that page loses one descendant and invariant rule 2 must be maintained. The events of page balancing and merging are illustrated in Fig 7.5. *a* is the underflowing page, *b* its neighbouring page, and *c* their ancestor; *s* is the position in the ancestor page of (the pointer to the) underflowing page *a*. Two cases are distinguished, namely whether the underflowing page is the rightmost element (s = c.m) or not (see procedure *underflow*).

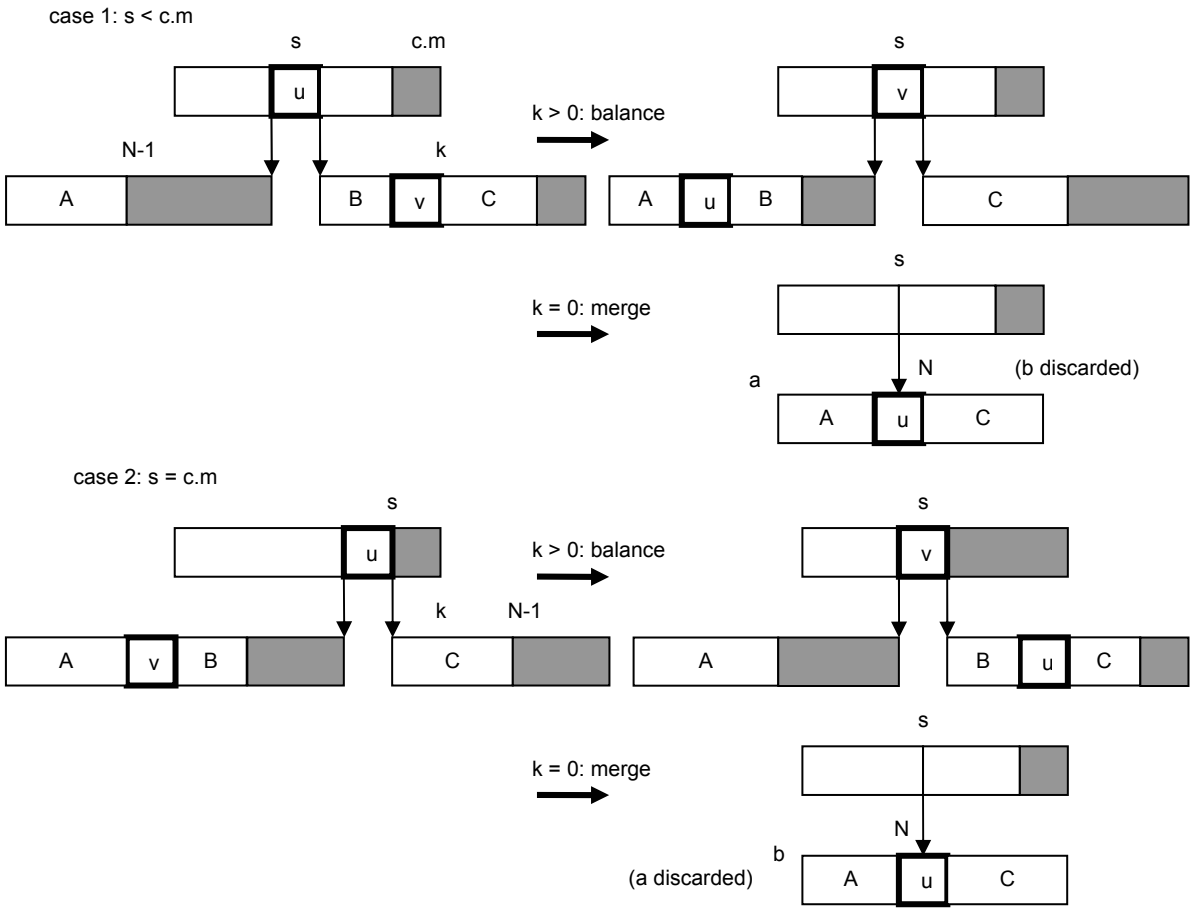


Figure 7.5 Page balancing and merging when deleting element

Similarly to the splitting process, merging may propagate, because the removal of an element from the ancestor page may again cause an underflow, and perhaps a merge. The root page underflows only if its last element is removed. This is the only way in which the B-tree's height can decrease.

```
MODULE BTree;
  IMPORT Texts, Oberon;

  CONST N = 3;

  TYPE Page = POINTER TO PageRec;

    Entry = RECORD
              key, data: INTEGER;
              p: Page
            END ;

    PageRec = RECORD
              m: INTEGER;  (*no. of entries on page*)
              p0: Page;
              e: ARRAY 2*N OF Entry
            END ;

  VAR root: Page; W: Texts.Writer;

PROCEDURE search(x: INTEGER; VAR p: Page; VAR k: INTEGER);
  VAR i, L, R: INTEGER; found: BOOLEAN; a: Page;
```

```
BEGIN a := root; found := FALSE;
  WHILE (a # NIL) & ~found DO
    L := 0; R := a.m;  (*binary search*)
    WHILE L < R DO
      i := (L+R) DIV 2;
      IF x <= a.e[i].key THEN R := i ELSE L := i+1 END
    END ;
    IF (R < a.m) & (a.e[R].key = x) THEN found := TRUE
    ELSIF R = 0 THEN a := a.p0 ELSE a := a.e[R-1].p
    END
  END ;
  p := a; k := R
END search;

PROCEDURE insert(x: INTEGER; a: Page; VAR h: BOOLEAN; VAR v: Entry);
  (*a # NIL. Search key x in B-tree with root a; if found, increment counter.
    Otherwise insert new item with key x.  If an entry is to be passed up,
    assign it to v. h := "tree has become higher"*)
  VAR i, L, R: INTEGER;
    b: Page; u: Entry;
BEGIN (*a # NIL & ~h*)
  L := 0; R := a.m;  (*binary search*)
  WHILE L < R DO
    i := (L+R) DIV 2;
    IF x <= a.e[i].key THEN R := i ELSE L := i+1 END
  END ;
  IF (R < a.m) & (a.e[R].key = x) THEN (*found*) INC(a.e[R].data)
  ELSE (*item not on this page*)
    IF R = 0 THEN b := a.p0 ELSE b := a.e[R-1].p END ;
    IF b = NIL THEN (*not in tree, insert*)
      u.p := NIL; h := TRUE; u.key := x
    ELSE insert(x, b, h, u)
    END ;
    IF h THEN (*insert u to the left of a.e[R]*)
      IF a.m < 2*N THEN
        h := FALSE; i := a.m;
        WHILE i > R DO DEC(i); a.e[i+1] := a.e[i] END ;
        a.e[R] := u; INC(a.m)
      ELSE NEW(b); (*overflow; split a into a,b and assign the middle entry to v*)
        IF R < N THEN (*insert in left page a*)
          i := N-1; v := a.e[i];
          WHILE i > R DO DEC(i); a.e[i+1] := a.e[i] END ;
          a.e[R] := u; i := 0;
          WHILE i < N DO b.e[i] := a.e[i+N]; INC(i) END
        ELSE (*insert in right page b*)
          DEC(R, N); i := 0;
          IF R = 0 THEN v := u
          ELSE v := a.e[N];
            WHILE i < R-1 DO b.e[i] := a.e[i+N+1]; INC(i) END ;
            b.e[i] := u; INC(i)
          END ;
          WHILE i < N DO b.e[i] := a.e[i+N]; INC(i) END
        END ;
        a.m := N; b.m := N; b.p0 := v.p; v.p := b
      END
    END
  END
END insert;

PROCEDURE underflow(c, a: Page; s: INTEGER; VAR h: BOOLEAN);
  (*a = underflowing page, c = ancestor page,
    s = index of deleted entry in c*)
  VAR b: Page;
```

```
      i, k: INTEGER;
BEGIN (*h & (a.m = N-1) & (c.e[s-1].p = a) *)
   IF s < c.m THEN (*b := page to the right of a*)
      b := c.e[s].p; k := (b.m-N+1) DIV 2; (*k = nof items available on page b*)
      a.e[N-1] := c.e[s]; a.e[N-1].p := b.p0;
      IF k > 0 THEN (*balance by moving k-1 items from b to a*) i := 0;
         WHILE i < k-1 DO a.e[i+N] := b.e[i]; INC(i) END ;
         c.e[s] := b.e[k-1]; b.p0 := c.e[s].p;
         c.e[s].p := b; DEC(b.m, k); i := 0;
         WHILE i < b.m DO b.e[i] := b.e[i+k]; INC(i) END ;
         a.m := N-1+k; h := FALSE
      ELSE (*merge pages a and b, discard b*)  i := 0;
         WHILE i < N DO a.e[i+N] := b.e[i]; INC(i) END ;
         i := s; DEC(c.m);
         WHILE i < c.m DO c.e[i] := c.e[i+1]; INC(i) END ;
         a.m := 2*N; h := c.m < N
      END
   ELSE (*b := page to the left of a*)  DEC(s);
      IF s = 0 THEN b := c.p0 ELSE b := c.e[s-1].p END ;
      k := (b.m-N+1) DIV 2; (*k = nof items available on page b*)
      IF k > 0 THEN i := N-1;
         WHILE i > 0 DO DEC(i); a.e[i+k] := a.e[i] END ;
         i := k-1; a.e[i] := c.e[s]; a.e[i].p := a.p0;
         (*move k-1 items from b to a, one to c*)  DEC(b.m, k);
         WHILE i > 0 DO DEC(i); a.e[i] := b.e[i+b.m+1] END ;
         c.e[s] := b.e[b.m]; a.p0 := c.e[s].p;
         c.e[s].p := a; a.m := N-1+k; h := FALSE
      ELSE (*merge pages a and b, discard a*)
         c.e[s].p := a.p0; b.e[N] := c.e[s]; i := 0;
         WHILE i < N-1 DO b.e[i+N+1] := a.e[i]; INC(i) END ;
         b.m := 2*N; DEC(c.m); h := c.m < N
      END
   END
END underflow;

PROCEDURE delete(x: INTEGER; a: Page; VAR h: BOOLEAN);
   (*search and delete key x in B-tree a; if a page underflow arises,
      balance with adjacent page or merge; h := "page a is undersize"*)
   VAR i, L, R: INTEGER; q: Page;

   PROCEDURE del(p: Page; VAR h: BOOLEAN);
      VAR k: INTEGER; q: Page;  (*global a, R*)
   BEGIN k := p.m-1; q := p.e[k].p;
      IF q # NIL THEN del(q, h);
         IF h THEN underflow(p, q, p.m, h) END
      ELSE p.e[k].p := a.e[R].p; a.e[R] := p.e[k];
         DEC(p.m); h := p.m < N
      END
   END del;

BEGIN (*a # NIL*)
   L := 0; R := a.m;  (*binary search*)
   WHILE L < R DO
      i := (L+R) DIV 2;
      IF x <= a.e[i].key THEN R := i ELSE L := i+1 END
   END ;
   IF R = 0 THEN q := a.p0 ELSE q := a.e[R-1].p END ;
   IF (R < a.m) & (a.e[R].key = x) THEN (*found*)
      IF q = NIL THEN (*a is leaf page*)
         DEC(a.m); h := a.m < N; i := R;
         WHILE i < a.m DO a.e[i] := a.e[i+1]; INC(i) END
      ELSE del(q, h);
         IF h THEN underflow(a, q, R, h) END
```

```
      END
    ELSE delete(x, q, h);
      IF h THEN underflow(a, q, R, h) END
    END
  END delete;

  PROCEDURE Search*(key: INTEGER; VAR data: INTEGER);
  BEGIN search(key, root, data)
  END Search;

  PROCEDURE Insert*(key: INTEGER; VAR data: INTEGER);
    VAR h: BOOLEAN; u: Entry; q: Page;
  BEGIN h := FALSE; u.data := data; insert(key, root, h, u);
    IF h THEN (*insert new base page*)
      q := root; NEW(root);
      root.m := 1; root.p0 := q; root.e[0] := u
    END
  END Insert;

  PROCEDURE Delete*(key: INTEGER);
    VAR h: BOOLEAN;
  BEGIN h := FALSE; delete(key, root, h);
    IF h THEN (*base page size underflow*)
      IF root.m = 0 THEN root := root.p0 END
    END
  END Delete;

  BEGIN NEW(root); root.m := 0
  END BTree.
```

The B-tree is also a highly appropriate structure for enumerating its elements, because during the traversal of the tree each page is visited exactly once, and hence needs to be read (from disk) exactly once too. The traversal is programmed by the procedure *Enumerate* and uses recursion. It calls the parametric procedure proc for each element of the tree. The type of proc specifies as parameters the name and the (address of) the enumerated element. The third parameter continue is a Boolean VAR-parameter. If the procedure sets it to FALSE, the process of enumeration will be aborted.

Enumerate is used for obtaining listings of the names of registered files. For this purpose, the actual procedure substituted for *proc* merely enters the given name in a text and ignores the address (sector number) of the file, unless it requires special file information such as the file's size or creation date.

The set of visited elements can be restricted by specifying a string which is to be a prefix to all enumerated names. The least name with the specified prefix is directly searched and is the name (key) of the first element enumerated. The process then proceeds up to the first element whose name does not have the given prefix. Thereby, the process of obtaining all elements whose key has a given prefix avoids traversal of the whole tree, resulting in a significant speedup. If the prefix is the empty string, the entire tree is traversed.

The principle behind procedure *Enumerate* is shown by the following sketch, where we abstract from the B-tree structure and omit consideration of prefixes:

```
PROCEDURE Enumerate(
        proc: PROCEDURE (name: FileName; adr: INTEGER; VAR continue: BOOLEAN));
    VAR continue: BOOLEAN; this: DirEntry;
BEGIN continue := TRUE; this := FirstElement;
    WHILE continue & (this # NIL) DO
        proc(this.name, this.adr, continue); this := NextEntry(this)
    END
END Enumerate
```

From this sketch we may conclude that during the process of traversal the tree structure must not change, because the function *NextEntry* quite evidently relies on the structural information stored in the elements of structure itself. Hence, the actions of the parametric procedure must not affect the tree structure. Enumeration must not be used, for example, to delete a given set of files. In order to prevent the misuse of the indispensible facility of element enumeration, the interface of *FileDir* is not available to users in general.

The handling of the directory stored on disk follows exactly the same algorithms. The accessed pages are fetched from the disk as a whole (each page fits onto a single disk sector) and stored in buffers of type *DirPage*, from where individual elements can be accessed. In principle, these buffers can be local to procedures *insert* and *delete*. A single buffer is allocated globally, namely the one used by procedure *Search*. The reason for this exception is not only that iterative searching requires one buffer only, but because procedure *Files.Old* and in turn *Search* may be called when the processor is in the supervisor mode and hence uses the system- (instead of the user-) stack, which is small and would not accommodate sector buffers.

Naturally, an updated page needs to be stored back onto disk. Omission of sector restoration is a programming error that is very hard to diagnose, because some parts of the program are executed very rarely, and hence the error may look sporadic and mistakenly be attributed to malfunctioning hardware.

Oberon's file directory represents a single, ordered set of name-file pairs. It is therefore also called a *fla*t directory. Its internal tree structure is not visible to the outside. In contrast, some file systems use a directory with a visible tree structure, notably UNIX. In a search, the name (key) guides the search path; the name itself displays structure, in fact, it is a sequence of names (usually separated by slashes or periods). The first name is then searched in the *root directory*, whose descendants are not files but *subdirectories*. The process is repeated, until the last name in the sequence has been used (and hopefully denotes a file).

Since the search path length in a tree increases with the logarithm of the number of elements, any subdivision of the tree inherently decreases performance since log(m + n) < log(m) + log (n) for any *m*, *n* > *1*. It is justified only if there exist sets of elements with common properties. If these property values are stored once, namely in the subdirectory referencing all elements with common property values, instead of in every element, not only a gain in storage economy results, but possibly also in accesses which depend on those properties. The common properties are typically an owner's name, a password, and access rights (read or write protection), properties that primarily have significance in a multi-user environment. Since Oberon was conceived explicitly as a single-user system, there is little need for such facilities, and hence a flat directory offers the best performance with a simple implementation.

Every directory operation starts with an access to the root page. An obvious measure for improving efficiency is to store the root page "permanently" in main store. We have chosen not to do this for four reasons:

1. If the hardware fails, or if the computer is switched off before the root page is copied to disk, the file directory will be inconsistent with severe consequences.

2. The root page has to be treated differently from other pages, making the program more complex.

3. Directory accesses do not dominate the computing process; hence, any improvement would hardly be noticeable in overall system performance. The payoff for the added complexity would be small.

4. Procedure Init is called upon system initialization in order to construct the sector reservation table. Therefore, this procedure (and the module) must be allowed to refer to the structure of a file's sector table(s), which is achieved by placing its definitions into the module *FileDir* (instead of *Files*). Unlike *Enumerate*, *Init* traverses the entire B-tree. The sector numbers of files delivered by *TraverseDir* are entered into a buffer. When full, the entries are sorted, whereafter each file's head sector is read and the sectors indicated in its sector table are marked as reserved. The

sorting speeds up the reading of the header sectors considerably. Nevertheless, the initialization of the sector reservation table clearly dominates the start-up time of the computer. For a file system with 10'000 files it takes in the order of 15s to record all files.

7.5. The toolbox of file utilities

We conclude this Chapter with a presentation of the commands which constitute the toolbox for file handling. These commands are contained in the tool module *System*, and they serve to copy, rename, and delete files, and to obtain excerpts of the file directory.

Procedures *CopyFiles, RenameFiles*, and *DeleteFiles* all follow the same pattern. The parameter text is scanned for file names, and for each operation a corresponding procedure is called. If the parameter text contains an arrow, it is interpreted as a pointer to the most recent text selection which indicates the file name. In the cases of *CopyFiles* and *RenameFiles* which require two names for a single action, the names are separated by "=>" indicating the direction of the copy or rename actions.

Procedure *Directory* serves to obtain excerpts of the file directory. It makes use of procedure *FileDir.Enumerate*. The parametric procedure *List* tests whether or not the delivered name matches the pattern specified by the parameter of the directory command. If it matches, the name is listed in the text of the viewer opened in the system track. Since the pattern may contain one or several asterisks (wild cards), the test consists of a sequence of searches of the pattern parts (separated by the asterisks) in the file name. In order to reduce the number of calls of *List, Enumerate* is called with the first part of the pattern as parameter prefix. Enumeration then starts with the least name having the specified prefix, and terminates as soon as all names with this prefix have been scanned.

If the specified pattern is followed by an option directive "!", then not only file names are listed, but also the listed files' creation date and length. This requires that not only the directory sectors on the disk are traversed, but that additionally for each listed file its header sector must be read. The two procedures use the global variables *pat* and *diroption*.

References

1. R. Bayer and E. M. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica, 1,* 3, (1972), 173-189.

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

8 Storage layout and management

8.1. Layout and run-time organization

A crucial property of the Oberon System is centralized resource management. Its advantage is that replication of management algorithms and a premature partitioning of resources are avoided. The disadvantage is that management algorithms are fixed once and forever and remain the same for all applications. The success of a centralized resource management therefore depends crucially on its flexibility and its efficient implementation. This chapter presents the scheme and the algorithms governing main storage in the Oberon System.

The storage layout of the Oberon System is determined by the structure of code and data typical in the use of modular, high-level programming languages, and in particular of the language Oberon. It suggests the subdivision of storage into three areas.

1. The *module space*. Every module specifies procedures (code) and global (static) variables. Its initialization part can be regarded as a procedure implicitly called after loading. Upon loading, space must be allocated for code and data. Typically, modules contain very few or no global variables, hence the size of the allocated space is primarily determined by the code. The combined code and data space is called a *block*. Blocks are allocated in the *module space* by the loader.

2. The *workspace (stack)*. Execution of every command invokes a sequence of procedures, each of which uses (possibly zero) parameters and local variables. Since procedure calls and completions follow a strict first-in last-out order, the stack is the uniquely suited strategy for storage allocation for local data. Deallocation upon completion of a procedure is achieved by merely resetting the pointer identifying the top of the stack. Since this operation is performed by a single instruction, it costs virtually no time. Because Oberon is a single-process system, *a single stack suffices.* Furthermore, after completion of a command, the stack is empty. This fact will be important in simplifying the scheme for reclamation of dynamically allocated space.

3. The *dynamic space (heap)*. Apart from global (static) variables, and local (stack-allocated) variables, a program may refer to anonymous variables referenced through pointers. Such variables are allocated truly dynamically through calls of an explicit operation (NEW). These variables are allocated in the so-called *heap*. Their deallocation is "automatic", when free storage is needed and they are no longer referenced from any of the loaded modules. This process is called *garbage collection*. Every record allocated in the heap contains a (hidden) pointer to the descriptor of its type called the *type tag*. It is used by the garbage collector.

Unfortunately, the number of distinct spaces is larger than two. If it were two, no arbitrary size limitation would be necessary; merely the sum of their sizes would be inherently limited by the size of the store. In the case of three spaces, arbitrarily determined size limits are unavoidable. Address-mapping hardware can alleviate (and delegate) this problem using a virtual address space which is so large that limits will hardly ever be reached.

Such a scheme is implemented by tables mapping virtual into physical addresses, requiring multiple memory accesses for every reference. Of course, the need for a double or a triple access for every memory reference is avoided by a translation cache in the (hardware) unit. Nevertheless, a decrease in performance is unavoidable for each cache miss. Furthermore, an additional subcycle is required for *every* access in order to look up the cached translation table. Without a virtual address scheme, each module block must consist of an integral number of physically adjacent pages. Holes generated by the release of modules must be reused. We employ the simple scheme of marking the released space as a hole, and of allocating a new block in the first hole encountered that is large enough (first-fit strategy). Considering the relative infrequency of module releases, efforts to improve the strategy are not worth the resulting added complexity.

It is remarkable that the code for module allocation and release without virtual addressing is only marginally more complicated than with it. The only remaining advantages of an MMU are a better

storage utilization, because no holes occur (a negligible advantage), and that inadvertent references to unloaded modules, e.g. via installed procedures, lead to an invalid address trap.

It is worth recalling that the concept of address mapping was introduced as a requirement for virtual memory implemented with disks as backing store, where pages could be moved into the background in order to obtain space for newly required pages, and could then be retrieved from disk on demand, i.e. when access was requested. This scheme is called *demand paging*. It is not used in the Oberon system, and one may fairly state that demand paging has lost its significance with the availability of large, primary stores.

Experience in the use of the RISC predecessor Ceres leads to the conclusion that whereas address translation through an MMU was an essential feature for multi-user operating systems, it constitutes a dispensible overkill for single-user workstations. The fact that modern semiconductor technology made it possible to integrate the entire translation and caching scheme into a single chip, or even into the processor itself, led to the hiding (and ignoring) of the scheme's considerable complexity. Its side effects on execution speed are essentially unpredictable. This makes systems with MMU virtually unusable for applications with tight real-time constraints. The RISC processor does indeed not feature an address mapping unit.

The RISC processor features 16 registers (of 32 bits). R0 - R11 are used for expression evaluation. R12 - R15 have fixed, system-wide usage:

R12	address of the module table MT (typically constant)
R13	base address for variables in the current module SB (static base)
R14	stack pointer SP
R15	return address LNK (fixed by RISC's BL instruction)

The used memory layout is shown in Figure 8.1.

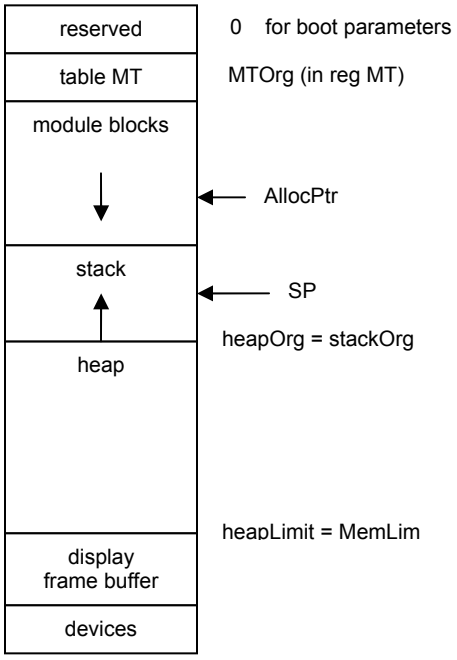


Figure 8.1 Storage layout

8.2. Management of dynamic storage

The term *dynamic storage* is used here for all variables that are allocated neither statically (global variables) nor on the stack (local variables), but through invocation of the intrinsic procedure NEW.

Such variables are anonymous and are referenced exclusively via pointers. The space in which they are allocated is called the *heap*.

The space allocated to such dynamic variables becomes free and reusable as soon as the last reference to it vanishes. This event is hard, and in multiprocess systems even impossible to detect. The usual remedy is to ignore it and instead to determine the accessibility of all allocated variables (records, objects) only at the time when more storage space is needed. This process is then called *garbage collection*.

The Oberon System does not provide an explicit deallocation procedure allowing the programmer to signal that a variable will no longer be referenced. The first reason for this omission is that usually a programmer would not know when to call for deallocation. And secondly, this "hint" could not be taken as trustworthy. An erroneous deallocation, i.e. one occurring when there still exist references to the object in question, could lead to a multiple allocation of the same space with disastrous consequences. Hence, it appears wise to fully rely on system management to determine which areas of the store are truly reusable.

Before discussing the scheme for storage reclamation, which is the primary subject of dynamic storage management, we turn our attention to the problem of allocation, i.e. the implementation of procedure NEW. The simplest solution is to maintain a list of free blocks and to pick the first one large enough. This strategy leads to a relatively large fragmentation of space and produces many small elements, particularly in the first part of the list. We therefore employ a somewhat more refined scheme and maintain four lists of available space. Three of them contain pieces of fixed size, namely 32, 64, and 128 bytes. The fourth list contains pieces whose size is any multiple of 256. We note that the choice of the values permits the merging of any two contiguous elements into an element of the next list. This scheme keeps fragmentation, i.e. the emergence of small pieces in large numbers, reasonably low with minimal effort. The body of procedure NEW consists of relatively few instructions, and typically only a small fraction of them needs to be executed.

The statement *NEW(p)* is compiled into an instruction sequence assigning the address of pointer variable *p* to a fixed register (R0) and the type tag to another register (R1). The type tag is a pointer to a type descriptor containing information required by the garbage collector. This includes the size of the space occupied and now to be allocated. The effect of *NEW* is the assignment of the address of the allocated block to *p*, and the assignment of the tag to a prefix of the block. (see Fig. 8.2)

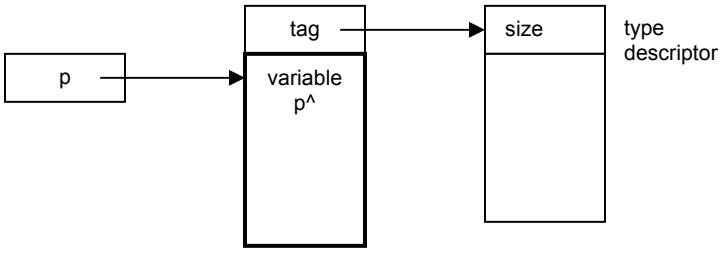


Figure 8.2 Allocation of dynamic variable p^ in the heap by procedure NEW(p)

In conclusion, we emphasize that this scheme makes the allocation of an object very efficient. Nevertheless, it is considerably more costly than that of a variable explicitly declared and therefore allocated either globally or on the stack.

We now turn to the problem of storage reclamation or *garbage collection*. There exist two essentially different schemes: the reference counting and the mark-scan schemes. In the former, every object carries a (hidden) reference count, indicating the number of existing references to it. The scheme works as follows:

1. NEW(p) initializes the reference count of p^ to 1.

2. An assignment q := p decrements the reference count of q^ by 1, performs the assignment, then increments the reference count of p^ by 1. When a reference count reaches zero, the element is linked into the free list.

There are two disadvantages inherent in this approach. The first is the non-negligible overhead in pointer assignments. The second is that circular data structures never become recognized as free, even if no external references point to their elements.

The Oberon system employs the second scheme which involves no hidden operations like the reference counting scheme, but relies on a process initiated when free storage has become scarce and more is needed. It consists of two phases. In the first phase, all referenced and therefore still accessible elements are marked. In the second phase, their unmarked complement is released. The first phase is called the *mark phase*, the second the *scan phase*. Its primary disadvantage is that the process may be started at moments unpredictable to the system's user. During the process, the computer then appears to be blocked. It follows that an interactive system using *mark-scan garbage collection* must guarantee that the process is sufficiently fast in order to be hardly noticeable. Modern processors make this possible, even with large main stores. Nevertheless, finding all accessible nodes in an entire computer system within, say, a second appears to be a formidable feat.

We recognize that the mark phase essentially is a tree traversal, or rather a forest traversal. The roots of the trees are all named pointer variables in existence. We shall postpone the question of how these roots are to be found, and first present a quick tutorial about tree traversal. In general, nodes of the traversed structure may contain many pointers (branches). We shall, however, first restrict our attention to a *binary tree*, because the essential problem and its solution can be explained better in this way.

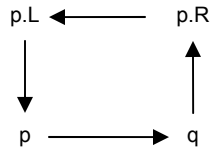
The essential problem alluded to is that of storage utilization by the traversal algorithm itself. Typically, information about the nodes already visited must be retained, be it explicitly, or implicitly as in the case of use of recursion. Such a strategy is plainly unacceptable, because the amount of storage needed is unpredictable and may become very large, and because garbage collection is typically initiated just when more storage is unavailable. The task may seem impossible, yet a solution lies in the idea of inverting pointers along the path traversed, thus keeping the return path open. It is embodied in the following procedure, whose task is to traverse the tree given by the parameter *root*, and to mark every node. Mark values are assumed to be initially 0. Let the data structure be defined by the types

```
Ptr =    POINTER TO Node;
Node =   RECORD m: INTEGER;  L, R: Ptr END;
```

and the algorithm by the procedure

```
PROCEDURE traverse(root: Ptr);
  VAR p, q, r; Ptr;
BEGIN p := root; q := root;
  REPEAT (* p # NIL *) INC(p.m); (*mark*)
    IF p.L # NIL THEN (*pointer rotation*)
      r := p.L; p.L :=  p.R; p.R := q; q := p; p := r
    ELSE
      p.L := p.R; p.R := q; q:= NIL
    END
  UNTIL p = q
END traverse
```

We note that only three local variables are required, independent of the size of the tree to be traversed. The third, *r*, is in fact merely an auxiliary variable to perform the rotation of values *p.L, p.R, q*, and *p* as shown in Fig. 8.3. A snapshot of a tree traversal is shown in Fig. 8.4.



.Figure 8.3 Rotation of four pointers

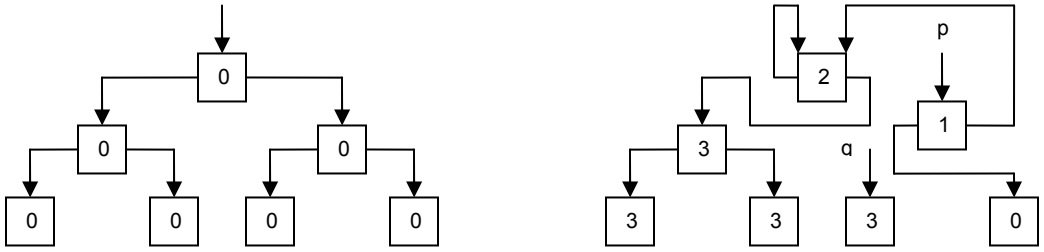


Figure 8.4 Tree traversal (original at left, snapshot at right)

The pair *p, q* of pointers marks the position of the process. The algorithm traverses the tree in a left to right, depth first fashion. When it returns to the root, all nodes have been marked.

How are these claims convincingly supported? The best way is by analyzing the algorithm at an arbitrary node. We start with the hypothesis *H* that, given the initial state *P*, the algorithm will reach state *Q*, (see Fig 8.5).

State *Q* differs from *P* by the node and its descendants *B* and *C* having been marked, and by an exchange of *p* and *q*. We now apply the algorithm to state *P*, assuming that *B* and *C* are not empty. The process is illustrated in Fig 8.5. *P0* stands for *P* in Fig. 8.4.

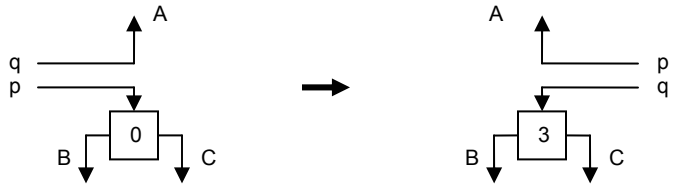


Figure 8.5 Transition from state P to Q

Transitions P0 → P1, P2 → P3, and P4 → P5 are the direct results of applying the pointer rotation as specified by the sequence of five assignments in the algorithm. Transitions P1 → P2 and P3 → P4 follow from the hypothesis *H* being applied to the states *P1* and *P3*: subtrees are marked and *p, q* interchanged. We note in passing that the node is visited three times. Progress is recorded by the mark value which is incremented from 0 to 3.

Fig. 8.6. demonstrates that, if *H* holds for steps P1 → P2 and P3 → P4, then it also holds for step P0 → P5, which visits the subtree *p*. Hence, it also holds for the step root → root, which traverses the entire tree.

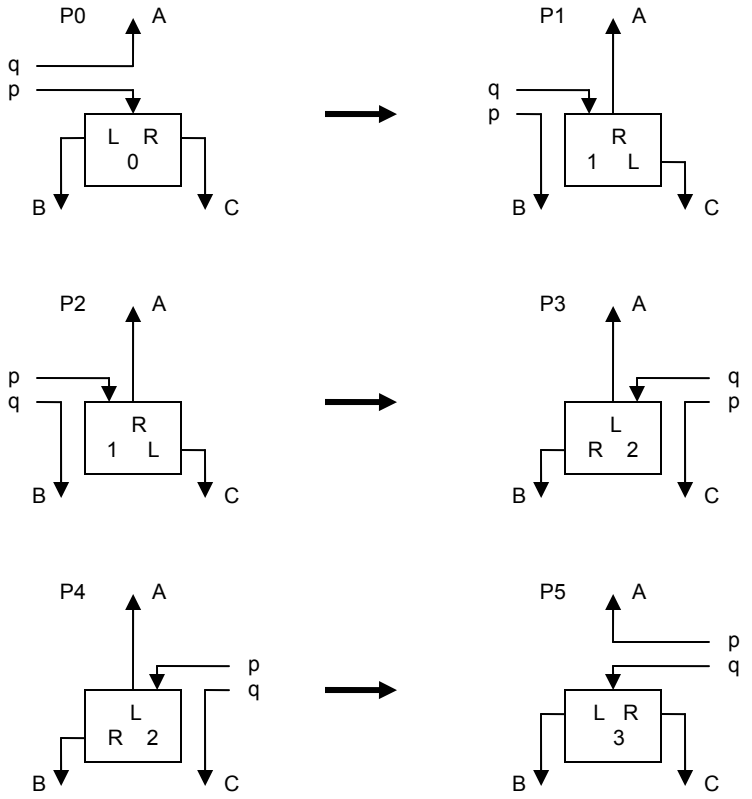


Figure 8.6 Transitions from P0 to P5, visiting nodes 3 times

This proof by recursion relies on the algorithm performing correct transitions also in the case of *p.L* being NIL, i.e. B being the empty tree. In this case, state *P1* is skipped; the first transition is P0 → P2 (see Figure 8.7).

If *p.L* is again NIL, i.e. also *C* is empty, the next transition is P2 → P4. This concludes the demonstration of the algorithm's correctness.

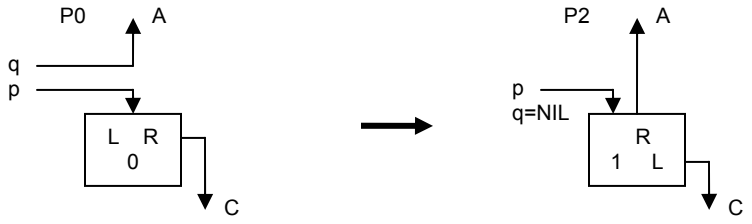


Figure 8.7 Direct transition from P0 to P2, if p.L = NIL

We now modify the algorithm of tree traversal to the case where the structure is not confined to a binary tree, but may be a tree of any degree, i.e. each node may have any number n of descendants. For practical purposes, however, we restrict n to be in the range $0 \leq n \leq N$, and therefore may represent all nodes by the type

```
Node = RECORD m, n: INTEGER;
         dsc: ARRAY N OF Node
       END
```

In principle, the binary tree traversal algorithm might be adopted almost without change, merely extending the rotation of pointers from *p.L, p.R, q, p* to *p.dsc[0], ... , p.dsc[n-1], q, p*. However, this

would be an unnecessarily inefficient solution. The following is a more effective variant. Moreover, it caters for the case of inhomogeneous graphs, where different nodes have different numbers of descendants. The key lies in associating with every node, in addition to the tag, a second private field *mk*. It serves two purposes. The first is as a mark, with *mk > 0* indicating that the node had been visited. The second is to store the address of the next descendant to be visited. The underlying data structure is shown in Figure 8.8. Type descriptors consist of the following fields:

size	in bytes, of the described record type,
base	a table of pointers to the descriptors of the base types (3 elements only)
offsets	of the descendant pointers in the described type (1 word each)

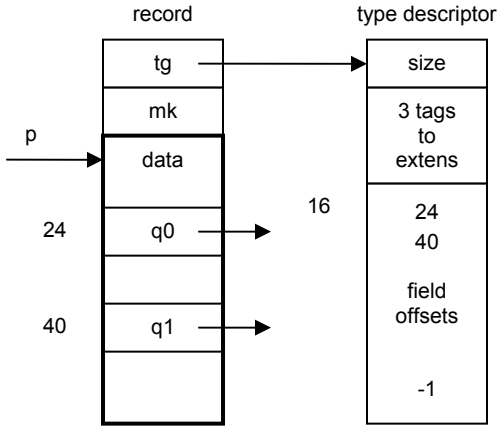


Figure 8.8 Record and its type descriptor

We note that the mark value, starting with zero (unmarked), is used as a counter of descendants already traversed, and hence as an index to the descendant field to be processed next. The algorithm can be applied not only to trees, but to arbitrary structures, including circular ones, if the continuation condition *p # 0* (actually *p >= heapOrg*) is extended to *(p >= heapOrg) & (offadr = 0)*. This causes a descendant that is already marked to be skipped. Here the array *M* stands for the entire memory.

```
PROCEDURE traverse(root: Ptr);
  VAR offadr, offset: INTEGER; p, q, r: Ptr;
BEGIN p := root; q := root;
  REPEAT (* p # NIL*) offadr := p.mk; (*mark*)
    IF offadr = 0 THEN tag := p.tg; offadr := tag + 16 ELSE INC(offadr, 4) END ;
    p. mk := offadr; offset := M[offadr];
    IF offset # -1 THEN  (*move down*)
      r := M[p+offset]; offadr := M[r-4];;
      IF offadr = 0 THEN M[p+offset] := q; q := p; p := r END
    ELSE (*move up*)
      offadr := M[q-4];offset := M[offadr]
      IF p # q THEN r := M[q+offset]; M[q+offset] := p; p := q; q := r END
    END
  UNTIL (p = q) & (offset = -1)
END traverse;
```

The mark is included in each record's hidden prefix. The prefix takes 2 words only; the first is used for the tag. The other is reserved for the garbage collector and used as mark and offset address. The end of the list of descendant pointers is marked by an entry with value -1. And finally, assignments involving *M* are expressed as

```
SYSTEM.GET(a, x)   for   x := M[a]
SYSTEM.PUT(a, x)   for   M[a] := x
```

The *scan phase* is performed by a relatively straight-forward algorithm. The heap, i.e. the storage area between *HeapOrg* and *HeapLimit* (the latter is a variable), is scanned element by element, starting at *HeapOrg*. Elements marked are unmarked, and unmarked elements are freed by linking them into the appropriate list of available space.

As the heap may always contain free elements, the scan phase must be able to recognize them in order to skip them or merge them with an adjacent free element. For this purpose, the free elements are also considered as prefixed. The prefix serves to determine the element's size and to recognize it as free due to a special (negative) mark value. The encountered mark values and the action to be taken are:

mk value	state	action
= 0	unmarked	collect, mark free
> 0	marked	unmark
< 0	free	skip or merge

8.3. The Kernel

The kernel lies at the bottom of the module hierarchy. It contains the procedures for dynamic storage allocation and retrieval as described before. The procedures are *New, Mark,* and *Scan*.

Kernel also contains the driver routines for the disk. They are used by modules *FileDir* and *Files.* The "disk" is actually an SD-card, a high-volume flash-RAM. It is accessed purely sequentially, byte-wise, by a standard, serial peripheral interface (SPI). Within Kernel a table called *SectorMap* is allocated keeping track of blocks (sectors) occupied by files. A single bit indicates, whether a sector is allocated or not. This table is accessed by the procedures *AllocSector, MarkSector,* and *FreeSector*. Reading and writing is done sector-wise by procedures *GetSector* and *PutSector*. Sector numbers are always a multiple of 29 for the purpose of redundancy checks.

Furthermore, the kernel contains a timer counting milliseconds and, perhaps, a real time clock, showing date and time. Clock data are packed into a single word as follows:

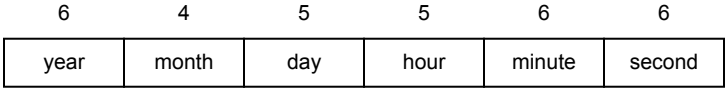


Figure 8.9 Encoding of date and time (year starting with 2000)

```
DEFINITION Kernel;  (*NW/PR  11.4.86 / 27.12.95 / 15.5.2013*)
   CONST SectorLength = 1024;
   TYPE Sector = ARRAY SectorLength OF BYTE;
   VAR allocated, NofSectors: INTEGER;
        heapOrg, heapLim: INTEGER;
        stackOrg, MemSize: INTEGER;
   PROCEDURE New(VAR ptr: INTEGER; tag: INTEGER);
   PROCEDURE Mark(pref: INTEGER);
   PROCEDURE Scan;
   PROCEDURE ResetDisk;
   PROCEDURE MarkSector(sec: INTEGER);
   PROCEDURE FreeSector(sec: INTEGER);
   PROCEDURE AllocSector(hint: INTEGER; VAR sec: INTEGER);
   PROCEDURE GetSector(src: INTEGER; VAR dest: Sector);
   PROCEDURE PutSector(dest: INTEGER; VAR src: Sector);
   PROCEDURE Time(): INTEGER; (*milliseconds*)
   PROCEDURE Clock(): INTEGER;
   PROCEDURE SetClock(dt: INTEGER);
   PROCEDURE Install(adr, procadr: INTEGER);
   PROCEDURE Init;
END Kernel.
```

8.4. The storage management's toolbox

The user can obtain information about the system's state and resources through its toolbox, a set of commands contained in the too module *System*. These commands are:

```
PROCEDURE Watch;
PROCEDURE Collect;  / n
PROCEDURE SetClock;   / year, month, day, hour, minute, second
```

Command *Watch* shows the amount of storage occupied in the heap, the number of disk sectors allocated on the disk, and the number of tasks installed. The command *Collect* allows to control the frequency of garbage collections. The number *n* indicates how many commands are executed before the next garbage collection.

9 Device drivers

9.1. Overview

Device drivers are collections of procedures that constitute the immediate interface between hardware and software. They refer to those parts of the computer hardware that are usually called *peripheral*. Computers typically contain a system bus which transmits data among its different parts. Processor and memory are considered as its internal parts; the remaining parts, such as disk, keyboard, display, etc, are considered as external or peripheral, notwithstanding the fact that they are often contained in the same cabinet or board.

Such peripheral devices are connected to the system bus via special registers (data buffers) and transceivers (switches, buffers in the sense of digital electronics). These registers and transceivers are addressed by the processor in the same way as memory locations - they are said to be *memory-mapped* - and they constitute the hardware interface between processor bus and device. References to them are typically confined to specific driver procedures which constitute the software interface.

Drivers are inherently hardware specific, and the justification of their existence is precisely that they encapsulate these specifics and present to their clients an appropriate abstraction of the device. Evidently, this abstraction must still reflect the essential characteristics of the device, but not the details (such as e.g. the addresses of its interface registers).

Our justification to present the drivers connecting the Oberon system with the RISC computer in detail is on the one hand the desire for completeness. But on the other hand it is also in recognition of the fact that their design represents an essential part of the engineering task in building a system. This part may look trivial from a conceptual point of view; it certainly is not so in practice.

In order to reduce the number of interface types, standards have been established. The RISC computer also uses such interface standards, and we will concentrate on them in the following presentations. The following devices are presented:

1. The *Keyboard* is considered as a serial device delivering one byte of input data per key stroke. It is connected by a serial line according to the PS/2 and ASCII (American Standard Code for Information Interchange) standards. The software is contained in module *Input* (Sect. 9.2), and the hardware is explained in Sect. 17.2.1.

2. The *Mouse* is a pointing device delivering coordinates in addition to key states. The software is also part of module *Input* (Sect. 9.2).

3. *Display*. The interface to the display is an area of memory that contains the displayed information, exactly one bit per pixel for a monochrome display. This area is called *frame buffer* or *bitmap* Here the size is of the display area is 768 lines and 1024 dots per line, representing a raster. The software is module *Display*, which primarily consists of operations to draw frequently occurring patterns. These operations are called *raster-ops*. They are explained in Section 4.5. The actual display requires a hardware interface called a *display controller*. The connection between the controller and the display follows the VGA-Standard (see Sect. 17.2.4).

4. *Disk*. Our RISC computer does not use a magnetic, rotating disk for storing non-volatile data. Instead, it uses an SD-card (flash-RAM). The driver is contained in module *Kerne*l (Section 8.3). The hardware is discussed in Sect. 17.2.2.

5. *Net.* In the original text, a network was presented consisting of a bus connecting many computers, based on the RS-485 standard. It was implemented by the serial communications controller Zilog 8530, operating at a frequency of 230 Kb/s. The name SCC has been retained as a generic interface, behind which the packet transport has now been re-implemented as a simple wireless network (Nordic nRF24L01 controller) in the regulation-free 2.4GHz industrial/scientific/medical (ISM) frequency band.

In all driver modules the implementation-dependent procedures SYSTEM.PUT, SYSTEM.GET, and SYSTEM.BIT are used to access the registers of the device interface. Their first parameter is the address of the register, the second an expression or variable.

9.2. Keyboard and mouse

The driver procedures for the keyboard and the mouse are located in module *Input. Available()* signals that a character has been typed on the keyboard, if its value is greater than 0. The character is read by calling *Read(ch)*. Module *Input* is restricting the data to the ASCII character set Latin-1, i.e. the values lie in the range 0X <= ch < 80X (7-bit values). *Mouse(k, x, y)* yields the current state of the mouse keys and the mouse's coordinates.

```
MODULE Input;
    PROCEDURE Available(): INTEGER;
    PROCEDURE Read(VAR ch: CHAR);
    PROCEDURE Mouse(VAR keys: SET; VAR x, y: INTEGER);
    PROCEDURE SetMouseLimits(w, h: INTEGER);
    PROCEDURE Init;
END Input.
```

The driver software accesses the keyboard via the Standard PS/2 interface represented by an 8-bit register for the received data *kbdCode*, and a single-bit flag indicating whether a byte had been received.

The keyboard codes received from the keyboard via a PS/2 line are *not* identical with the character values delivered to the *Read* procedure. A conversion is necessary. This is so, because modern keyboards treat all keys in the same way, including the ones for upper case, control, alternative, etc. Separate codes are sent to signal the pushing down and the release of a key, followed by another code identifying which key had been pressed or released. This requires, besides a translation table from codes to characters, a set of state variables. They are the global, Boolean variables *Recd, Up, Shift, Ctrl,* and *Ext.* Procedure *Peek* determines whether an actual character is present, or merely a code signalling a key shift. *Peek* controls the state.

Procedure *Mouse* fetches a word from the mouse interface register and decomposes it into its components (key state and coordinates). (*kb* is the bit indicating whether a code had been received from the keyboard).

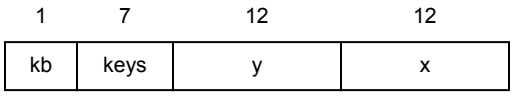


Fig. 9.1 Format of the mouse register

9.3. The SD-card (disk)

SD-card are high-volume memory devices based on flash-store technology. They are typically organized as individually accessible blocks of 1K bytes. The driver for the SD-card is contained in module *Kernel*, which also handles allocation and reservation of blocks, here in analogy to rotating disks still called *sectors*.

```
TYPE Sector = ARRAY SectorLength OF BYTE;

PROCEDURE GetSector(src: INTEGER; VAR dst: Sector);
PROCEDURE PutSector(dst: INTEGER; VAR src: Sector);
PROCEDURE AllocSector(hint: INTEGER; VAR sec: INTEGER);
PROCEDURE MarkSector(sec: INTEGER);
PROCEDURE FreeSector(sec: INTEGER);
```

Data transfer is sequebtial and handled by procedures *ReadSD* and *WriteSD* by issuing commands. These are for transmitting a block address, for receiving, and for sending a block of data. Synchronous transmission of sequences of words follows the SPI standard, which uses 3 lines, one for data input, one for dats output, and one for the clock (see also Section 17.2.2). The hardware interface contains a 32-bit register. The bit-rate is 8.3 MB/s.

9.4. Serial asynchronous interface (RS 232)

The RS-232 standard serves to transmit sequences of bytes over a data line asynchronously. This implies that there is no separate clock line (see also Section 17.2.3). The hardware interface contains a 10-bit register for the transmitter and one for the receiver. The data rate used here is 19200 bit/s. A byte is sent and received over the line by the following programs.

```
CONST data = -56; stat = -52;  (*device register addresses*)

PROCEDURE Send(x: BYTE);
BEGIN
  REPEAT UNTIL SYSTEM.BIT(stat, 1);
  SYSTEM.PUT(data, x)
END Send;

PROCEDURE Rec(VAR x: BYTE);
BEGIN
  REPEAT UNTIL SYSTEM.BIT(stat, 0);
  SYSTEM.GET(data, x)
END Rec;
```

These procedures are used in the driver module RS232 presented in Section 15.2. This module itself is not used in the Oberon core, but it was instrumental in building the System on a host computer and downloading it. It is characterized by a very simple interface.

9.5. Serial communications controller (SCC)

The interface of the driver for the network was taken over from the original design using a serial communocations controller Zilog 8530. The implementation changed totally. It was designed by Paul Reed for the wireless controller Nordic nRF24L01.

```
DEFINITION SCC;

  TYPE Header =
       RECORD valid: BOOLEAN; dadr, sadr, typ: BYTE;
         len: INTEGER; (*of data following header*)
       END ;

  PROCEDURE Start(filter: BOOLEAN);
  PROCEDURE Send(VAR head: Header; buf: ARRAY OF BYTE);
  PROCEDURE Available(): INTEGER;
  PROCEDURE ReceiveHead(VAR head: Header);
  PROCEDURE Receive(VAR x: BYTE);
  PROCEDURE Skip(m: INTEGER);
  PROCEDURE Stop;

END SCC.
```

10. The Network

10.1. Introduction

Workstations are typically, but not always, connected in a local environment by a network. There exist two basically different views of the architecture of such nets. The more demanding view is that all connected stations constitute a single, unified workspace (also called address-space), in which the individual processors operate. It implies the demand that the "thin" connections between processors are hidden from the users. At worst they might become apparent through slower data access rates between the machines. To hide the difference between access within a computer and access between computers is regarded primarily as a challenge to implementors.

The second, more conservative view, assumes that individual workstations are, although connected, essentially autonomous units which exchange data infrequently. Therefore, access of data on partner stations is initiated by explicit transfer commands. Commands handling external access are not part of the basic system, but rather are implemented in modules that might be regarded as applications.

In the Oberon System, we adhere to this second view, and in this chapter, we describe the module *Net*, which is an autonomous command module based on the network driver SCC. It can be activated on any station connected in a network, and all of them are treated as equals. Such a set of loosely coupled stations may well operate in networks with moderate transmission rates and therefore with low-cost hardware interfaces and twisted-pair wires.

An obvious choice for the unit of transferred data is the file. The central theme of this chapter is therefore file transfer over a network. Some additional facilities offered by a dedicated server station will be the subject of Chapter 11. The commands to be presented here are a few only: *SendFiles, ReceiveFiles*, and *SendMsg*.

As explained in Chapter 2, Oberon is a single-process system where every command monopolizes the processor until termination. When a command involves communication over a network, (at least) two processors are engaged in the action at the same time. The Oberon paradigm therefore appears to exclude such cooperation; but fortunately it does not, and the solution to the problem is quite simple.

Every command is initiated by a user operating on a workstation. For the moment we call it the *master* (of the command under consideration). The addressed station - obviously called the *server* - must be in a state where it recognizes the command in order to engage in its execution. Since the command - called a *request* - arrives in encoded form over the network, an Oberon task represented by a handler procedure must be inserted into the event polling loop of the system. Such a handler must have the general form

```
IF event present THEN handle event END
```

The guard, in this case, must imply that a request was received from the network. We emphasize that the event is sensed by the server only after the command currently under execution, if any, has terminated. However, data arrive at the receiver immediately after they are sent by the master. Hence, any sizeable delay is inherently inadmissible, and the Oberon metaphor once again appears to fail. It does not fail, however, because the unavoidable, genuine concurrency of sender and receiver action is handled within the driver module which places the data into a buffer. The driver is activated by an interrupt, and its receiver buffer effectively decouples the partners and removes the stringent timing constraints. All this remains completely hidden within the driver module.

10.2. The protocol

If more than a single agent participates in the execution of a command, a convention must be established and obeyed. It defines the set of requests, their encoding, and the sequence of data

exchanges that follow. Such a convention is called a *protocol*. Since in our metaphor, actions initiated by the master and the server strictly follow each other in alternation, the protocol can be defined using EBNF (extended Backus-Naur formalism), well-known from the syntax specification of languages. Items originating from the master will be written with normal font, those originating from the server appear in italics.

A simple form of the *ReceiveFile* request is defined as follows and will be refined subsequently:

ReceiveFile = SND filename (*ACK data* | *NAK*).

Here, the symbol SND represents the encoded request that the server send the file specified by the file name. ACK signals that the request is honoured and the requested data follow. The NAK symbol indicates that the requested file cannot be delivered. The transaction clearly consists of two parts, the request and the reply, one from each partner.

This simple-minded scheme fails because of the limitation of the size of each transmitted portion imposed by the network driver. We recall that module SCC restricts the data of each packet to 512 bytes. Evidently, files must be broken up and transmitted as a sequence of packets. The reason for this restriction is transmission reliability. The break-up allows the partner to confirm correct receipt of a packet by returning a short acknowledgement. Each acknowledgement also serves as request for the next packet. An exception is the last acknowledgement following the last data portion, which is characterized by its length being less than the admitted maximum. The revised protocol is defined as

ReceiveFile = SND filename (*DAT data* ACK {*DAT data* ACK} | *NAK*).

We now recall that each packet as defined in Section 9.3. is characterized by a type in its header. The symbols SND, DAT, ACK, and NAK indicate this packet type. The data portions of ACK and NAK packets are empty.

The revised protocol fails to cope with transmission errors. Correct transmission is checked by the driver through a cyclic redundancy check (CRC), and an erroneous packet is simple discarded. This implies that a receiver must impose a timing constraint. If an expected packet fails to arrive within a given time period (timeout), the request must be repeated. In our case, a request is implied by an acknowledgement. Hence, the acknowledgement must specify whether the next (normal case) or the previously requested (error case) packet must be sent. The solution is to attach a sequence number to each acknowledgement and to each data packet. These numbers are taken modulo 8, although in principle modulo 2 would suffice.

With the addition of a user identification and a password to every request, and of an alternate reply code NPR for "no permission", the protocol reaches its final form:

ReceiveFile = SND username password filename (*datastream* | *NAK* | *NPR*).
datastream = DAT_0 *data* ACK_1 {DAT_i *data* ACK_{i+1}}.

The protocol for file transmission from the master to the server is defined similarly:

SendFile = REC username password filename (*ACK0 datastream* | *NAK* | *NPR*).
datastream = DAT_0 data ACK_1 {DAT_i data ACK_{i+1}}.

The third request listed above, SendMsg, does not refer to any file, but merely transmits and displays a short message. It is included here for testing the link between two partners and perhaps for visibly acknowledging a rendered service by the message "done", or "thank you".

SendMsg = MSG message *ACK*.

10.3. Station addressing

Every packet must carry a destination address as well as the sender's address. Addresses are station numbers. It would certainly be inconvenient for a user to remember the station number of a desired partner. Instead, the use of symbolic names is preferred. We have become accustomed to use the partner's initials for this purpose.

The source address is inserted automatically into packet headers by the driver. It is obtained from a dip switch set when a computer is installed and connected. But where should the destination address come from? From the start we reject the solution of an address table in every workstation because of the potential inconsistencies. The concept of a centralized authority holding a name/address dictionary is equally unattractive, because of the updates required whenever a person uses a different computer. Also, we have started from the premise to keep all participants in the network equal.

The most attractive solution lies in a decentralized name service. It is based on the broadcast facility, i.e. the possibility to send a packet to all connected stations, bypassing their address filters with a special destination address (-1). The broadcast is used for emitting a name request containing the desired partner's symbolic name. A station receiving the request returns a reply to the requester, if that name matches its own symbolic name. The requester then obtains the desired partner's address from the source address field of the received reply. The corresponding simple protocol is:

NameRequest = NRQ partnername [*NRS*].

Here, the already mentioned timeout facility is indispensible. The following summarizes the protocol developed so far:

protocol = {request}.
request = ReceiveFile | SendFile | SendMsg | NameRequest.

The overhead incurred by name requests may be reduced by using a local address dictionary. In practice, a single entry is satisfactory. A name request is then needed whenever the partner changes.

10.4. The implementation

Module *Net* is an implementation of the facilities outlined above. The program starts with a number of auxiliary, local procedures. They are followed by procedure *Serve* which is to be installed as an Oberon task, and the commands *SendFiles, ReceiveFiles,* and *SendMsg*, each of which has its counterpart within procedure *Serve*. At the end are the commands for starting and stopping the server facility.

For a more detailed presentation we select procedure *ReceiveFiles*. It starts out by reading the first parameter which designates the partner station from the command line. Procedure *FindPartner* issues the name request, unless the partner's address has already been determined by a previous command. The global variable partner records a symbolic name (id) whose address is stored in the destination field of the global variable head0, which is used as header in every packet sent by procedure *SCC.SendPacket*. The variable partner may be regarded as a name cache with a single entry and with the purpose of reducing the number of issued name requests.

If the partner has been identified, the next parameter is read from the command line. It is the name of the file to be transmitted. If the parameter has the form *name0:name1*, the file stored on the server as *name0.name1* is fetched and stored locally as name1. Hence, name0 serves as a prefix of the file name on the server station.

Thereafter, the request parameters are concatenated in the local buffer variable *buf*. They are the user's name and password followed by the file name. (User name and password remain unused by the server presented here). The command package is dispatched by the call *Send(SND, k, buf)*, where *k* denotes the length of the command parameter string. Then the reply packet is awaited by calling *ReceiveHead*. If the received packet's type is DAT with sequence number 0, a new file is established. Procedure *ReadData* receives the data and stores them in the new file, obeying the protocol defined in Section 10.2. This process is repeated for each file specified in the list of file names in the command line.

Procedure *ReceiveHead(T)* receives packets and discards them until one arrives from the partner from which it is expected. The procedure represents an input filter in addition to the one provided by

the hardware. It discriminates on the basis of the packets' source address, whereas the hardware filter discriminates on the basis of the destination address. If no packet arrives within the allotted time T, a type code -1 is returned, signifying a timeout.

Procedure *ReceiveData* checks the sequence numbers of incoming data packets (type 0 - 7). If an incorrect number is detected, an ACK-packet with the previous sequence number is returned (type 16 - 23), requesting a retransmission. At most two retries are undertaken. This seems to suffice considering that also the server does not accept any other requests while being engaged in the transmission of a file.

The part corresponding to *ReceiveFiles* within procedure *Serve* is guarded by the condition *head1.typ = SND*. Variable *head1* is the recipient of headers whenever a packet is received by *ReceiveHead*. First, the request's parameters are scanned. *Id* and *pw* are ignored. Then the requested file is opened. If it exists, the transmission is handled by *ReceiveData*'s counterpart, procedure *SendData*. The time limit for receiving the next request is *T1*, whereas the limit of *ReceiveData* for receiving the next data packet is *T0. T1* is roughly *T0* multiplied by the maximum number of possible (re)transmissions. Before disengaging itself from a transaction, the sender of data waits until no further retransmission requests can be expected to arrive. The value *T0* (300) corresponds to 1s; the time for transmission of a packet of maximum length is about 16ms.

Procedure *SendFiles* is designed analogously; its counterpart in the server is guarded by the condition *head1.typ = REC*. The server accepts the request only if its state is unprotected (global variable *protected*). Otherwise the request is negatively acknowledged with an NPR packet. We draw attention to the fact that procedures *SendData* and *ReceiveData* are both used by command procedures as well as by the server.

11. A Dedicated file-distribution and mail-server

11.1. Concept and structure

In a system of loosely coupled workstations it is desirable to centralize certain services. A first example is a common file store. Even if every station is equipped with a disk for permanent data storage, a common file service is beneficial, e.g. for storing the most recent versions of system files, reference documents, reports, etc. A common repository avoids inconsistencies which are inevitable when local copies are created. We call this a *file distribution service*.

A centralized service is also desirable if it requires equipment whose cost and service would not warrant its acquisition for every workstation, particularly if the service is infrequently used. A prime example of this case is a *printing service*.

The third case is a communication facility in the form of *electronic mail*. The repository of messages must inherently be centralized. We imagine it to have the form of a set of mailboxes, one for each user in the system. A mailbox needs to be accessible at all times, i.e. also when its owner's workstation has been switched off.

A last example of a centralized service is a *time server*. It allows a station's real time clock to be synchronized with a central clock.

In passing we point out that every user has full control over his station, including the right to switch it on and off at any time. In contrast, the central server is continuously operational.

In this chapter, we present a set of server modules providing all above mentioned services. They rest on the basic Oberon System without module *Net* (see Chapter 10). In contrast to *Net*, module *NetServer*, which handles all network communication, contains no command procedures (apart from those for starting and stopping it). This is because it never acts as a master. The counterparts of its server routines reside in other modules, including (an extended version of) *Net*, on the individual workstations.

Routines for the file distribution service are the same as those contained in module *Net*, with the addition of permission checks based on the received user names and passwords. Routines for printing and mail service could in principle also be included in *NetServer* in the same way. But considerations of reliability and timing made this simple solution appear as unattractive. A weaker coupling in time of data transmission and data consumption is indeed highly desirable. Therefore, data received for printing or for dispatching into mailboxes are stored (by *NetServer*) into temporary files and thereafter "handed over" to the appropriate agent, i.e. the print server or the mail server.

This data-centered interface between servers - in contrast to procedural interfaces - has the advantage that the individual servers are independent in the sense that none imports any other. Therefore, their development could proceed autonomously. Their connection is instead a module which defines a data structure and associated operators for passing temporary files from one server to another. The data structure used for this purpose is the first-in-first-out queue. We call its elements tasks, because each one carries an objective and an object, the file to be processed. The module containing the FIFOs is called *Core*. The resulting structure of the involved modules is shown in Fig. 11.1.

Fig. 11.1. includes yet another server, *LineServer*, and shows the ease with which additional servers may be inserted in this scheme. They act as further sources and/or sinks for tasks, feeding or consuming the queues contained in *Core*. *LineServer* indeed produces and consumes tasks like *NetServer*. Instead of the RS-485 bus, it handles the RS-232 line which, connected to a modem, allows access to the server over telephone lines. We refrain from describing this module in further detail, because in many ways it is a mirror of *NetServer*.

A centralized, open server calls for certain protection measures against unauthorized use. We recall that requests always carry a user identification and a password as parameters. The server

checks their validity by examining a table of users. The respective routines and the table are contained in module *Core* (see Sect. 11.5).

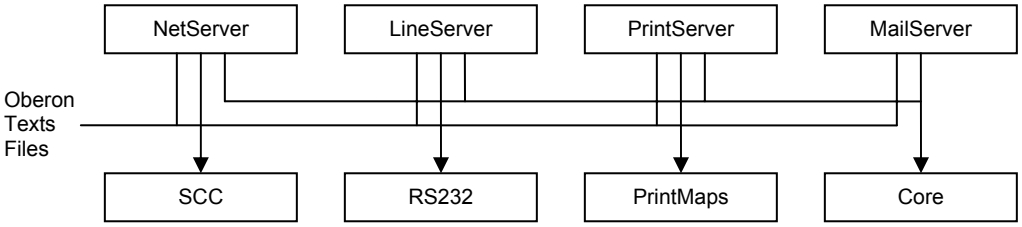


Figure 11.1 Module structure of server systems

11.2. Electronic Mail Service

The heart of an e-mail service is the set of mailboxes stored on the dedicated, central server. Each registered user owns a mailbox. The evidently necessary operations are the insertion of a message and its retrieval. In contrast to customary letter boxes, however, a retrieved message need not necessarily be removed from the box; its retrieval produces a copy. The box thereby automatically becomes a repository, and messages can be retrieved many times. This scheme calls for an additional command which removes a message from the box. Also, a command is needed for delivering a table of contents, in which presumably each message is represented by an indication of its sender and time of arrival.

The mail scheme suggested above results in the following commands:

Net.Mailbox ServerName. This command fetches a table of contents of the current user's mailbox from the specified server and displays it in a new viewer. The user's name and password must have been registered previously by the command *System.SetUser.*

Net.SendMail ServerName. The text in the marked viewer is sent to the specified server. In order to be accepted, the text must begin with at least one line beginning with "To:" and containing at least one recipient.

Net.ReceiveMail. This command is contained in the title bar (menu) of the viewer obtained when requesting the table of contents. Prior to issuing the command, the message to be read must have been specified by selecting a line in the table of contents in this viewer.

Net.DeleteMail. This command is also contained in the mailbox viewer's title bar. The message to be deleted must be selected before issuing the command.

The mail system presented here is primarily intended to serve as an exchange for short messages which are typically sent, received, read, and discarded. Mailboxes are not intended to serve as long term archives for a large and ever growing number of long pieces of text. This restrictiveness of purpose allows to choose a reasonably simple implementation and results in an efficient, practically instantaneous access to messages when the server is idle.

The Oberon mail server used at ETH also provides communication with external correspondents. It connects to an external mail server which is treated as a source and a sink for messages (almost) like other customers. Additionally, messages sent to that server need to be encoded into a standardized format, and those received need to be decoded accordingly. The parts of module *MailServer* for encoding and decoding are not described in this book. We merely divulge the fact that its design and implementation took a multiple of the time spent on the fast, local message exchange, to which we confine this presentation.

From the structures explained in Section 11.1. it follows that three agents are involved in the transfer of messages from the user into a mailbox. Therefore, additions to the server system distribute over three modules. New commands are added to module *Net* (see Section 10.4.); these procedures will be listed below. Their counterparts reside in module *NetServer* on the dedicated

computer. The third agent is module *MailServer*; both are listed below in this Section. The latter handles the insertion of arriving messages into mailboxes. The path which a message traverses for insertion and retrieval is shown in Fig. 11.2. Rectangles with bold edges mark storage.

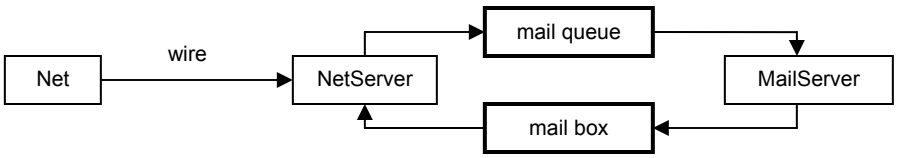


Figure 11.2 Path of messages to and from mailbox

Communication between the master station and the dedicated server runs over the network and therefore calls for an extension of its protocol (see Sect. 10.2.). The additions directly correspond to the four commands given above.

MailBox = MDIR username password (*datastream* | *NAK* | *NPR*).
SendMail = RML username password (*ACK datastream* | *NAK* | *NPR*).
ReceiveMail = SML username password msgno (*datastream* | *NAK* | *NPR*).
DeleteMail = DML username password msgno (*ACK* | *NAK* | *NPR*).

The message number is taken from the selected line in the mailbox viewer. The data transmitted are taken as (unformatted) texts. This is in contrast to file transfers, where they are taken as any sequence of bytes. The four command procedures listed below belong in module *Net*; they are listed together with the auxiliary procedures *SendText* and *ReceiveText* which closely correspond to *SendData* and *ReceiveData* (see Sect. 10.4).

We now turn our attention to the command procedures' counterparts in module *NetServer* listed in this Section. In order to explain these routines, a description of their interface with the mail server and a definition of the structure of mailboxes must precede. We begin with the simplest case, the counterpart of *SendMail*. It is the part of procedure *NetServer.Serve* which is guarded by the condition *typ = RML*, indicating a request to receive mail. As in all other services, the parameters username and password are read and the admissibility of the request is checked. The check is performed by procedure *Core.UserNo* which yields a negative number if service is to be refused. In the affirmative case, procedure *ReceiveData* obtains the message and stores it on a file, which is thereafter inserted into the mail queue as a task to be handled by the mail server at a later time. This may involve distribution of the message into several mailboxes.

Module *Core* is listed in Sect. 11.5. As mentioned before, it serves as link between the various server modules, defining the data types of the linking queues and also of mailboxes. Task queues are represented as FIFO-lists. The descriptor of type *Queue* contains a pointer to the first list element used for retrieval, and a pointer to the last element used for insertion (see Fig. 11.3). These pointers are not exported; instead, the next task is obtained by calling procedure *Core.GetTask*, and it is deleted by *Core.RemoveTask*. There e*xist two exported variables of type Queue: MailQueue consumed by MailServer, and* PrintQueue consumed by *PrintServer* (see Sect. 11.3.). (In fact, we use a third queue: *LineQueue* consumed by *LineServer*). Elements of queues are of type *TaskDesc* which specifies the file representing the data to be consumed. Additionally, it specifies the user number and identification of the task's originator. Three procedures are provided by module *Core* for handling task queues:

```
PROCEDURE InsertTask(VAR q: Queue; F: Files.File; VAR id: ARRAY OF CHAR; uno: INTEGER);

PROCEDURE GetTask(VAR q: Queue; VAR F: Files.File; VAR id: ARRAY OF CHAR; VAR uno:
INTEGER);

PROCEDURE RemoveTask(VAR q: Queue);
```

The server's counterparts of the remaining mail commands access mailboxes directly. The simplicity of the required actions - a result of a carefully chosen mailbox representation - and considerations of efficiency do not warrant a detour via task queue and mail server.

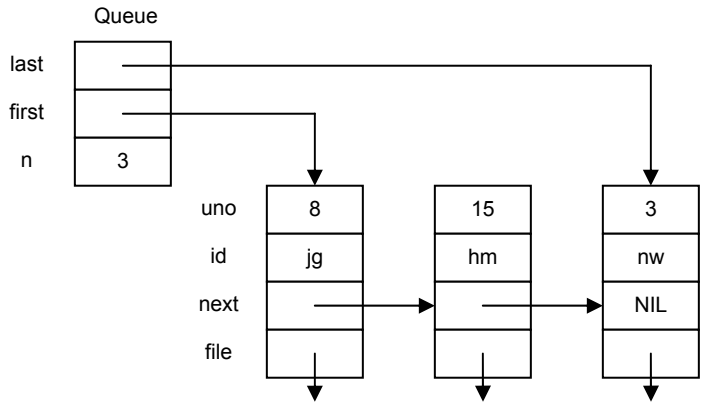


Figure 11.3 Structure of task queue

Every mailbox is represented as a file. This solution has the tremendous advantage that no special administration has to be introduced to handle a reserved partition of disk store for mail purposes. A mailbox file is partitioned into three parts: the block reservation part, the directory part, and the message part. Each part is quickly locatable, because the first two have a fixed length (32 and 31*32 = 992 bytes). The message part is regarded as a sequence of blocks (of 256 bytes), and each message occupies an integral number of adjacent blocks. Corresponding to each block, the block reservation part contains a single bit indicating whether or not the block is occupied by a message. Since the block reservation part is 32 bytes long, the message part contains at most 256 blocks, i.e. 64K bytes. The block length was chosen after an analysis of messages which revealed that the average message is less than 500 bytes long.

The directory part consists of an array of 31 elements of type *MailEntry*, a record with the following fields: *pos* and *len* indicate the index of the message's first block and the message's number of bytes; *time* and *date* indicate the message's time of insertion, and originator indicates the message's source. The entries are linked (field *next*) in chronological order of their arrival, and entry 0 serves as the list's header. It follows that a mailbox contains at most 30 messages. An example of a mailbox state is shown in Fig. 11.4.

```
MailEntry =   RECORD
                 pos, next: INTEGER;
                 len: LONGINT;
                 time, date: INTEGER;
                 originator: ARRAY 20 OF CHAR
              END ;
MResTab =     ARRAY 8 OF SET;
MailDir =     ARRAY 31 OF MailEntry;
```

We are now in a position to inspect the handler for requests for message retrieval. It is guarded by the condition *typ = SML*. After a validity check, the respective requestor's mailbox file is opened. The last mailbox opened is retained by the global variable MF which acts as a single entry cache. The associated user number is given by the global variable *mailuno*. Since typically several requests involving the same mailbox follow, this measure avoids the repeated reopening of the same file. Thereafter, a rider is directly positioned at the respective directory entry for reading the message's length and position in the message part. The rider is repositioned accordingly, and transmission of the message is handled by procedure *SendMail*.

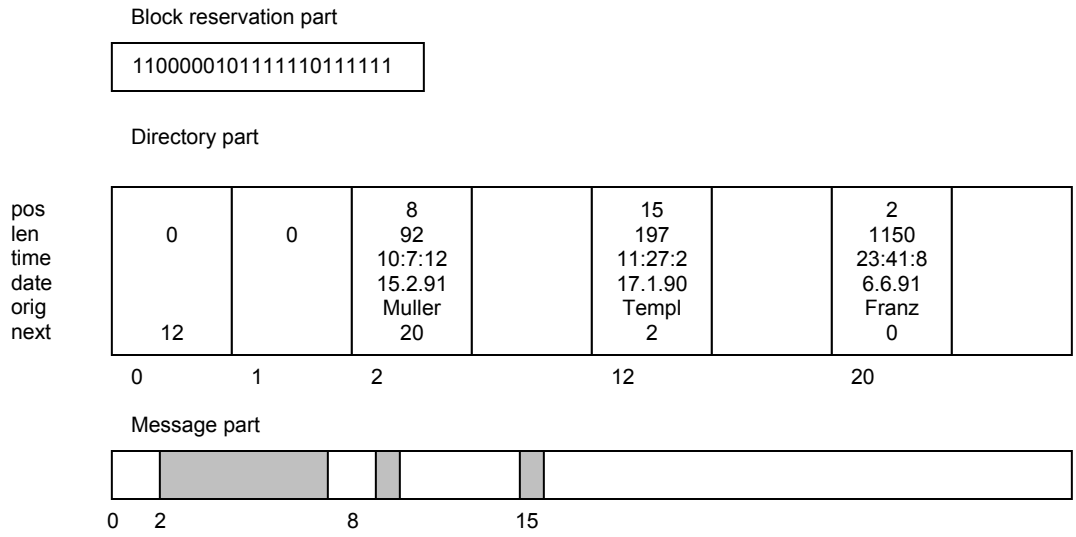


Figure 11.4 State of mailbox file

Requests for the mailbox directory are handled by the routine guarded by the condition *typ = MDIR*. The directory part must be read and converted into a text. This task is supported by various auxiliary procedures (Append) which concatenate supplied data in a buffer for latter transmission. We emphasize that this request does not require the reading of any other part of the file, and therefore is very swift.

The last of the four mail service requests (DML) deletes a specified message. Removal from the directory requires a relinking of the entries. Unused entries are marked by their *len* field having value 0. Also, the blocks occupied by the message become free. The block reservation part must be updated accordingly.

In passing we note that the use of files for representing mailboxes, in combination with the file distribution services residing on the same server station, allows anyone to access (and inspect) any mailbox. Although we do not claim that this system provides secure protection against snooping, a minimal effort for protection was undertaken by a simple encoding of messages in mailbox files. This encoding is not shown in the program listings contained in this book.

One operation remains to be explained in more detail: the processing of tasks inserted into the mail queue. It consists of the insertion of the message represented by the task's file into one or several mailboxes. It involves the interpretation of the message's header, i.e. lines containing addresses, and the construction of a new header containing the name of the originator and the date of insertion into the mailbox. These actions are performed by procedures in module *MailServer*. Its procedure *Serve* is installed as an Oberon Task, and it is guarded by the condition *Core.MailQueue.n > 0*, indicating that at least one message needs to be dispatched.

The originator's name is obtained from *Core.GetUserName(uno),* where uno is the user number obtained from the queue entry. The actual time is obtained from *Oberon.GetClock*. The form of the new header is shown by the following example:

From: Gutknecht
At: 12.08.91 09:34:15

The received message's header is then searched for recipients. Their names are listed in header lines starting with "To" (or "cc"). After a name has been read, the corresponding user number is obtained by calling *Core.UserNum*. Then the message is inserted into the designated mailbox by procedure *Dispatch*. The search for recipients continues, until a line is encountered that does not begin with "To" (or "cc"). A negative user number indicates that the given name is not registered. In

this case, the message is returned to the sender, i.e. inserted into the mailbox of the sender. An exception is the recipient "all" which indicates a broadcast to all registered users.

Procedure *Dispatch* first opens the mailbox file of the user specified by the recipient number rno. If a mailbox exists, its block reservation part (mrtab) and its directory part (mdir) are read. Otherwise a new, empty box is created. Then follows the search for a free slot in the directory and, if found, the search for a sufficient number of free, adjacent blocks in the message part. The number of required blocks is given by the message length. If either no free slot exists, or there is no large enough free space for the message part, the message is returned to the sender (identified by *sno*). If also this attempt fails, the message is redirected to the postmaster (with user number 0). The postmaster is expected to inspect his mailbox sufficiently often so that no overflow occurs. If the postmaster's mailbox also overflows, the message is lost.

Only if all conditions for a successful completion are satisfied, is insertion begun. It starts with the marking of blocks in the reservation table and with the insertion of the new directory information. Table and directory are then updated on the file. Thereafter, the message with the constructed new header is written into the message part.

Perhaps it may seem to the reader that the addition of a separate module *MailServer*, together with a new Oberon Task and the machinery of the mail queue is not warranted by the relative simplicity of the insertion operation, and that it could have been incorporated into module *NetServer* just as well as message extraction. The picture changes, however, if handling of external mail is to be added, and if access to mailboxes via other channels, such as the RS-232 line, is to be provided. The presented solution is based on a modular structure that facilitates such extensions without change of existing parts. External mail routines inevitably have to cope with message formats imposed by standards. Format transformations, encoding before sending to an external server and decoding before dispatching become necessary. Indeed, these operations have inflated module *MailServer* in a surprising degree. And lastly, the queuing machinery supports the easy insertion of additional message sources and provides a welcome decoupling and relaxation of timing constraints, particularly in the case of low-speed transmission media such as telephone lines.

11.4. Miscellaneous services

There exist a few additional services that are quite desirable under the presence of a central facility, and at the same time easy to include. They are briefly described in this section.

The set of commands of the file distribution service is augmented by *Net.DeleteFiles* and *Net.Directory*, allowing the remote deletion of files and inspection of the server's directory. The command procedures are listed below and must be regarded as part of module *Net* (Sect. 10.4). They communicate with their counterparts in module *NetServer* (Sect. 11.2.) according to the following protocol:

DeleteFile = DEL username password filename (*ACK* | *NAK* | *NPR*).
Directory = FDIR username password prefix (*datastream* | *NAK* | *NPR*).

The directory request carries a prefix; it uses procedure *FileDir.Enumerate* to obtain all file names starting with the specified prefix. Thereby the search can be limited to the relevant section of the directory.

Since requests to the server are always guarded by a password, a facility is necessary to set and change the password stored by the server. The respective command is *Net.SetPassword,* and its handler in the server is guarded by the condition *typ* = *NPW*. The corresponding protocol is

NewPassword = NPW username oldpassword
(ACK DAT newpassword (ACK | NAK) | NAK | NPR).

Finally, procedure Net.GetTime allows the workstation's real time clock to be adjusted to that of the central server. The protocol is

GetTime = TRQ TIM time date.

In concluding we summarize the entire protocol specification below. The combined server facility, comprising file distribution, electronic mail, printing, and time services is operating on a Ceres-1 computer (1 Mips) with a 2 MByte store, of which half is used by the printer's bitmap.

Summary of Protocol:

protocol	=	{request}.
request	=	ReceiveFile | SendFile | DeleteFile | Directory | MailBox | SendMail | ReceiveMail | DeleteMail | PrintStream | SendMsg | NameRequest | NewPassword | GetTime.
ReceiveFile	=	SND username password filename (datastream | NAK | NPR).
datastream	=	DAT0 data ACK1 {DATi data ACKi+1}.
SendFile	=	REC username password filename (ACK0 datastream | NAK | NPR).
datastream	=	DAT0 data ACK1 {DATi data ACKi+1}.
DeleteFile	=	DEL username password filename (ACK | NAK | NPR).
Directory	=	FDIR username password prefix (datastream | NAK | NPR).
MailBox	=	MDIR username password (datastream | NAK | NPR).
SendMail	=	RML username password (ACK datastream | NAK | NPR).
ReceiveMail	=	SML username password msgno (datastream | NAK | NPR).
DeleteMail	=	DML username password msgno (ACK | NAK | NPR).
PrintStream	=	PRT username password (ACK datastream | NAK | NPR).
SendMsg	=	MSG message ACK.
NameRequest	=	NRQ partnername [NRS].
NewPassword	=	NPW username oldpassword (ACK DAT newpassword (ACK | NAK) | NAK | NPR).
GetTime	=	TRQ TIM time date.

11.5. User Administration

It appears to be a universal law that centralization inevitably calls for an administration. The centralized mail and printing services make no exception. The typical duties of an administration are accounting and protection against misuse. It has to ensure that rendered services are counted and that no unauthorized user is taking advantage of the server. An additional duty is often the gathering of statistical data. In our case, accounting plays a very minor role, and the reason for the existence of the administration presented here is primarily protection.

We distinguish between two kinds of protection. The first is protection of the server's resources in general, the second is that of individual users' resources from being accessed by others. Whereas in the first case some validation of a user's identification might suffice, the second case requires the association of personal resources with user names. In any case, the central server must store data for each member of the set of registered users. Specifically, it must be able to check the admissibility of a user's request on the basis of stored information.

Evidently, a protection administration is similar in purpose and function to a lock. Quite regularly, locks are subjected to attempts of breaking them, and locksmiths are subjected to attempts of being outwitted. The race between techniques of breaking locks and that of better countermeasures is well known, and we do not even try to make a contribution to it. Our design is based on the premise that the Oberon Server operates in a harmonious environment. Nevertheless, a minimal amount of protection machinery was included. It raises the amount of effort required for breaking protection to a level which is not reached when curiosity alone is the motivation.

The data about users is held in a table in module *Core*. As was mentioned earlier, *Core* acts as connector between the various servers by means of task queues. Its second purpose is to provide the necessary access to user data via appropriate procedures.

In the simplest solution, each table entry would contain a user name only. For each request, the administration would merely test for the presence of the request's user name in the table. A significant step towards safe protection is the introduction of a password in addition to the user name. In order that a request be honoured, not only must the name be registered, but the delivered and the stored password must match. Evidently, abusive attempts would aim at recovering the

stored passwords. Our solution lies in storing an encoded password. The command *System.SetUser*, which asks for a user identification and a password, immediately encodes the password, and the original is stored nowhere. The encoding algorithm is such that it is difficult to construct a corresponding decoder.

The mail service requires a third attribute in addition to identification and encoded password: the user's name as it is used for addressing messages. Identification typically consists of the user's initials; for the name we suggest the full last name of the user and discourage cryptic abbreviations.

The printing service makes an accounting facility desirable. A fourth field in each user table entry serves as a count for the number of printed pages. As a result, there are four fields: *id, name, password,* and *count*. The table is not exported, but only accessible via procedures. *Core* is a good example of a resource hiding module. The program is listed below, and a few additional comments follow here.

Procedures *UserNo(id)* and *UserNum(name)* yield the table index of the identified user; it is called *user number* and is used as a short encoding for recipients and senders within the mail server. In other servers, the number is merely used to check a request's validity.

The user information must certainly survive any intermission of server operation, be it due to software, hardware, or power failure. This requires that a copy of the user information is held on backup store (disk). The simplest solution would be to use a file for this purpose. But this would indeed make protection too vulnerable: files can be accessed easily, and we have refrained from introducing a file protection facility. Instead, the backup of the user information is held on a few permanently reserved sectors on the server machine, which are inaccessible to the file system.

Apart from procedures and variables constituting the queuing mechanism for tasks, the procedures exported from module *Core* all belong to the administration, and they can be divided into two categories. The first category contains the procedures used by the three servers presented in this Chapter, and they are *UserNo, UserNum, IncPageCount, SetPassword, GetUserName* and *GetFileName*. The second category consists of the procedures *NofUsers* and *GetUser* for inspecting table entries, and *InsertUser, DeleteUser, ClearPassword, ClearCounts*, and *Init* for making changes to the table.

The client of the latter category is a module *Users* which is needed by the human administrator of the server facility.

The reader may at this point wonder why a more advanced concept of administration has not been chosen, which would allow the human administrator to operate the server remotely. A quick analysis of the consequences of this widely used approach reveals that a substantial amount of additions to our system would be required. The issue of security and protection would become inflated into dimensions that are hardly justified for our local system. The first consequence would be a differentiation among levels of protection. The administrator would become a so-called super-user with extra privileges, such as changing the user table. And so the game of trying to break the protection measures starts to become an interesting challenge.

We have resisted the temptation to introduce additional complexity. Instead, we assume that physical access to the server station is reserved to the administrator. Naturally, module Users and in particular the symbol file of *Core* do not belong to the public domain. In concluding, we may point out that the impossibility of activating users' programs on the server station significantly reduces the possibilities for inflicting damage from the exterior.

12 The compiler

12.1. Introduction

The compiler is the primary tool of the system builder. It therefore plays a prominent role in the Oberon System, although it is not part of the basic system. Instead, it constitutes a tool module - an application - with a single command: *Compile*. It translates program texts into machine code. Therefore, it is as a program inherently machine-dependent; it acts as the interface between source language and target computer.

In order to understand the process of compilation, the reader needs to be familiar with the source language *Oberon* defined in Appendix 1, and with the target computer *RISC,* defined in Appendix 2.

The language is defined as an infinite set of sequences of symbols taken from the language's vocabulary. It is described by a set of equations called syntax. Each equation defines a syntactic construct, or more precisely, the set of sequences of symbols belonging to that construct. It specifies how that construct is composed of other syntactic constructs. The meaning of programs is defined in terms of semantic rules governing each such construct.

Compilation of a program text proceeds by analyzing the text and thereby decomposing it recursively into its constructs according to the syntax. When a construct is identified, code is generated according to the semantic rule associated with the construct. The components of the identified construct supply parameters for the generated code.

It follows that we distinguish between two kinds of actions: analyzing steps and code generating steps. In a rough approximation we may say that the former are source language dependent and target computer independent, whereas the latter are source language independent and target computer dependent. Although reality is somewhat more complex, the module structure of this compiler clearly reflects this division. The main module of the compiler is ORP (for Oberon to RISC Parser) It is primarily dedicated to syntactic analysis, parsing. Upon recognition of a syntactic construct, an appropriate procedure is called the code generator module ORG (for Oberon to RISC Generator). Apart from parsing, ORP checks for type consistency of operands, and it computes the attributes of objects identified in declarations.

Whereas ORP mirrors the source language and is independent of a target computer, ORG reflects the target computer, but is independent of the source language.

Oberon program texts are regarded as sequences of symbols rather than sequences of characters. Symbols themselves, however, are sequences of characters. We refrain from explaining the reasons for this distinction, but mention that apart from special characters and pairs such as +, &, <=, also identifiers, numbers, and strings are classified as symbols. Furthermore, certain capital letter sequences are symbols, such as IF, END, etc. Each time the syntax analyzer (parser) proceeds to read the next symbol, it does this by calling procedure *Get*, which constitutes the so-called scanner residing in module ORS (for Oberon to RISC Scanner). It reads from the source text as many characters as are needed to recognize the next symbol.

In passing we note that the scanner alone reflects the definition of symbols in terms of characters, whereas the parser is based on the notion of symbols only. The scanner implements the abstraction of symbols. The recognition of symbols within a character sequence is called *lexical analysis*.

Ideally the recognition of any syntactic construct, say A, consisting of subconstructs, say B1, B2, ... , Bn, leads to the generation of code that depends only on (1) the semantic rules associated with A, and (2) on (attributes of) B1, B2, ... , Bn. If this condition is satisfied, the construct is said to be context-free, and if all constructs of a language are *context-free*, then also the language is context-free. Syntax and semantics of Oberon adhere to this rule, although with a significant exception. This

exception is embodied by the notion of declarations. The declaration of an identifier, say *x*, attaches permanent properties to *x*, such as the fact that *x* denotes a variable and that its type is *T*. These properties are "invisible" when parsing a statement containing *x*, because the declaration of *x* is not also part of the statement. The "meaning" of identifiers is thus inherently context-dependent.

Context-dependence due to declarations is the immediate reason for the use of a global data structure which represents the declared identifiers and their properties (attributes). Since this concept stems from early assemblers where identifiers (then called *symbols*) were registered in a linear table, the term *symbol table* tends to persist for this structure, although in this compiler it is considerably more complex than an array. Basically, it grows during the processing of declarations, and it is searched while expressions and statements are processed. Procedures for building and for searching are contained in module ORB.

A complication arises from the notion of exports and imports in Oberon. Its consequence is that the declaration of an identifier x may be in a module, say M, different from where x is referenced. If x is exported, the compiler includes x together with its attributes in the *symbol file* of the compiled module M. When compiling another module which imports M, that symbol file is read and its data are incorporated into the symbol table. Procedures for reading and writing symbol files are contained in module ORB, and no other module relies on information about the structure of symbol files.

The syntax is precisely and rigorously defined by a small set of syntactic equations. As a result, the parser is a reasonably perspicuous and short program. In spite of the high degree of regularity of the target computer, the process of code generation is more complicated, as shown by module ORG.

The resulting module structure of the compiler is shown in Fig. 12.1 in a slightly simplified manner. In reality OCS is imported by all other modules due to their need for procedure *OCS.Mark*. This, however, will be explained later.

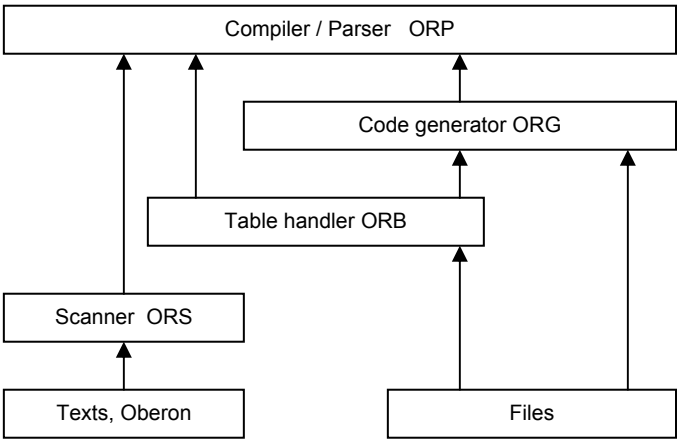


Figure 12.1 Compiler's module structure

12.2. Code patterns

Before it is possible to understand how code is generated, one needs to know *which* code is generated. In other words, we need to know the goal before we find the way leading to the goal. A fairly concise description of this goal is possible due to the structure of the language. As explained before, semantics are attached to each individual syntactic construct, independent of its context. Therefore, it suffices to list the expected code - instead of an abstract semantic rule - for each syntactic construct.

As a prerequisite to understanding the resulting instructions and in particular their parameters, we need to know where declared variables are stored, i.e. which are their addresses. This compiler

uses the straight-forward scheme of sequential allocation of consecutively declared variables. An address is a pair consisting of a base address (in a register) and an offset. Global variables are allocated in the module's data section and the respective base address register is SB (Static Base, see Chapter 6). Local variables are allocated in a procedure activation record on the stack; the respective base register is SP (Stack Pointer). Offsets are positive integers.

The amount of storage needed for a variable (called its *size*) is determined by the variable's type. The sizes of basic types are prescribed by the target computer's data representation. The following holds for the RISC processor:

Type	No. of bytes
BYTE, CHAR, BOOLEAN	1
INTEGER, REAL, SET, POINTER, PROCEDURE	4

The size of an array is the size of the element type multiplied by the number of elements. The size of a record is the sum of the sizes of its fields.

A complication arises due to so-called alignment. By alignment is meant the adjustment of an address to a multiple of the variable's size. Alignment is performed for variable addresses as well as for record field offsets. The motivation for alignment is the avoidance of double memory references for variables being "distributed" over two adjacent words. Proper alignment enhances processing speed quite significantly. Variable allocation using alignment is shown by the example in Fig. 12.2.

```
VAR b0: BYTE; int0: INTEGER; b1: BYTE; int1: INTEGER;
```

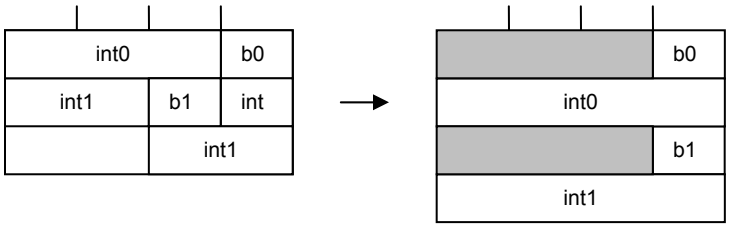


Figure 12.2. Alignment of variables

We note in passing that a reordering of the four variables lessens the number of unused bytes, as shown in Fig. 12.3.

```
VAR  int0, int1: INTEGER; b0, b1: BYTE;
```

Figure 12.3. Improved order of variables

Memory instructions compute the address as the sum of a register (base) and an offset constant. Local variables use the *stack pointer* SP (R14) as base, global variables the *static base* SB (R13) Every module has its own SB value, and therefore access to global (and imported) variables requires two instructions, one for fetching the base value, and one for loading or storing data. If the compiler can determine, whether the correct base value has already been loaded into the SB register, the former instruction is omitted.

The first 7 sample patterns contain global variables only, and their base SB is assumed to hold the appropriate value. Parameters of branch instructions denote jump distances from the instruction's own location (PC-relative).

Pattern 1: *Assignment of constants.* We begin with a simple example of assigning constants to variables. The variables used in this example are global; their base register is SB. Each assignment results in a single instruction. The constant is embedded within the instruction as a literal operand.

```
MODULE Pattern1;
    VAR ch: CHAR;              0
      k: INTEGER;              4
      x: REAL;                 8
      s: SET;                  12

BEGIN                                  module entry code
    ch := "0";                 40000030    MOV R0 R0    30H
                               B0D00000    STR  R0 SB     0
    k := 10;                   4000000A    MOV R0 R0    10
                               A0D00004    STR  R0 SB     4
    x := 1.0;                  60003F80    MOV' R0 R0   3F800000H
                               A0D00008    STR  R0 SB     8
    s := {0, 4, 8}             40000111    MOV R0 R0    111H
                               A0D0000C    STR  R0 SB     12
END Pattern1.                          module exit code
```

Pattern 2: *Simple expressions:* The result of an expression containing operators is always stored in a register before it is assigned to a variable or used in another operation.

Registers for intermediate results are allocated sequentially in ascending order R0, R1, ... , R11. Integer multiplication and division by powers of 2 are represented by shifts (LSL, ASR). Similarly, the modulus by a power of 2 is obtained by masking off leading bits. The operations of set union, difference, and intersection are represented by logical operations (OR, AND).

```
MODULE Pattern2;
    VAR i, j, k, n: INTEGER;   0, 4, 8, 12
      x, y: REAL;              16, 20
      s, t, u: SET;            24, 28, 32

BEGIN i := (i + 1) * (i - 1);  LDR R0 SB 0
                               ADD R0 R0 1
                               LDR R1 SB 0
                               SUB R1 R1 1
                               MUL R0 R0 R1
                               STR R0 SB 0
    k := k DIV 17;             LDR R0 SB 8
                               DIV R0 R0 17
                               STR R0 SB 8
    k := 8*n;                  LDR R0 SB 12
                               LSL R0 R0 3
                               STR R0 SB 8
    k := n DIV 2;              LDR R0 SB 12
                               ASR R0 R0 1
                               STR R0 SB 8
    k := n MOD 16;             LDR R0 SB 12
                               AND R0 R0 15
                               STR r0 SB 8
    x := -y / (x - 1.0);       LDR R0 SB 16
                               MOV' R1 R0 3F80H
                               FSB R0 R0 R1
                               LDR R1 SB 20
                               FDV R0 R1 R0
                               MOV R1 R0 0
                               FSB R0 R1 R0
                               STR R0 SB 16
    s := s + t * u             LDR R0 SB 28
                               LDR R1 SB 32
```

```
                                    AND R0 R0 R1
                                    LDR R1 SB 24
                                    OR R0 R1 R0
                                    STR R0 SB 24
END Pattern2.
```

Pattern3: *Indexed variables:* References to elements of arrays make use of the possibility to add an index value to an offset. The index must be present in a register and be multiplied by the size of the array elements. (For integers with size 4 this is done by a shift of 2 bits). Then this index is checked whether it lies within the bounds specified in the array's declaration. This is achieved by a comparison, actually a subtraction, and a subsequent branch instruction causing a trap, if the index is either negative or beyond the upper bound.

If the reference is to an element of a multi-dimensional array (matrix), its address computation involves several multiplications and additions. The address of an element $A[i_{k-1}, \ldots, i_1, i_0]$ of a k-dimensional array A with lengths $n_{k-1}, \ldots, n_1, n_0$ is

$$adr(A) + ((\ldots ((i_{k-1} * n_{k-2}) + i_{k-2}) * n_{k-3} + \ldots) * n_1 + i_1) * n_0 + i_0$$

Note that for index checks CMP is written instead of SUB to mark that the subtraction is merely a comparison, that the result remains unused and only the condition flag registers hold the result.

```
MODULE Pattern3;
    VAR i, j, k, n: INTEGER;                    0, 4, 8, 12
      a: ARRAY 10 OF INTEGER;                   16
      x: ARRAY 10, 10 OF INTEGER;               56
      y: ARRAY 10, 10, 10 OF INTEGER;           456
BEGIN
    k := a[i];                          LDR R0 SB  0
                                        CMP R1 R0  10
                                        BLHI  R12
                                        LSL R0 R0  2
                                        ADD R0 SB R0
                                        LDR R0 R0 16
                                        STR  R0 SB  8
    n := a[5];                          LDR R0 SB  36
                                        STR  R0 SB  12
    x[i, j] := 2;                       LDR R0 SB    0
                                        CMP R1 R0 10
                                        BLHI  R12
                                        MUL R0 R0  40
                                        ADD R0 SB R0
                                        LDR R1 SB  4
                                        CMP R2 R1  10
                                        BLHI  R12
                                        LSL R1 R1 2
                                        ADD R0 R0 R1
                                        MOV R1 R0  2
                                        STR  R1 R0   56
    y[i, j, k] := 3;                    LDR R0 SB  0
                                        CMP R1 R0 10
                                        BLHI  R12
                                        MUL R0 R0 400
                                        ADD R0 SB R0
                                        LDR R1 SB  4
                                        CMP R2 R1 10
                                        BLHI  R12
                                        MUL R1 R1 40
                                        ADD R0 R0 R1
                                        LDR R1 SB  8
                                        CMP R2 R1 10
```

```
                                    BLHI  R12
                                    LSL R1 R1 2
                                    ADD R0 R0 R1
                                    MOV R1 R0  3
                                    STR  R1 R0  456
    y[3, 4, 5] := 6                 MOV R0 R0   6
                                    STR  R0 SB 1836
END Pattern3.
```

Pattern 4: *Record fields and pointers:* Fields of records are accessed by computing the sum of the record's (base) address and the field's offset. If the record variable is statically declared, the sum is computed by the compiler.

```
MODULE Pattern4;
    TYPE Ptr = POINTER TO Node;
      Node = RECORD num: INTEGER;           0
        name: ARRAY 8 OF CHAR;              4
        next: Ptr                           12
      END ;
    VAR p, q: Ptr;                          12, 16
      r: Node;                              20

BEGIN
  r.num := 10;                  MOV R0 R0  10
                                STR  R0 SB  20
  p.num := 6                    LDR R0 SB  12  (p)
                                MOV R1 R0   6
                                STR  R1 R0   0
  p.name[7] := "0";             LDR R0 SB  12
                                MOV R1 R0  30H
                                STR  R1 R0  11   (4+7)
  p.next := q;                  LDR R0 SB   12
                                LDR R1 SB   16
                                STR  R1 R0   12
  p.next.next := NIL            LDR R0 SB   12  (p)
                                LDR R0 R0    12  (p.next)
                                MOV R1 R0  0   (NIL)
                                STR  R1 R0  12   (p.next.next)
END Pattern4.
```

Pattern 5: *Boolean expressions, If statements:* Conditional statements imply that parts of them are skipped. This is done by the use of branch instructions whose operand specifies the distance of the branch. The instructions refer to the condition-register as an implicit operand. Its value is determined by a preceding instruction, typically a compare or a bit-test instruction.

The Boolean operators & and OR are purposely not defined as total functions, but rather by the equations

```
p & q     =  if p then q else FALSE
p OR q    =  if p then TRUE else q
```

Consequently, Boolean operators must be translated into branches too. Evidently, branches stemming from if statements and branches stemming from Boolean operators should be merged, if possible. The resulting code therefore does not necessarily mirror the structure of the if statement directly, as can be seen from the code in *Pattern5*. We must conclude that code generation for Boolean expressions differs in some aspects from that for arithmetic expressions.

The example of *Pattern5* is also used to exhibit the code resulting from the standard procedures INC, DEC, INCL, and EXCL. These procedures provide an opportunity to use shorter code in those cases where a single two-operand instruction suffices, i.e. when one of the arguments is identical with the destination.

```
MODULE Pattern5;
   VAR n: INTEGER; s: SET;                  0, 4
BEGIN
   IF n = 0 THEN                            LDR R0 SB  0
                                            CMP R0 R0  0
                                            BNE  3
        INC(n)                              LDR R0 SB  0
                                            ADD R0 R0  1
                                            STR R0 SB  0
   END ;
   IF (n >= 0) & (n < 100) THEN             LDR SB R0 ...
                                            LDR R0 SB  0   (n)
                                            CMP R0 R0  0
                                            BLT  6
                                            LDR R0 SB  0
                                            CMP R0 R0  100
                                            BGE  3
        DEC(n)                              LDR R0 SB  0
                                            SUB R0 R0  1
                                            STR R0 R0  0
   END ;
   IF ODD(n) OR (n IN s) THEN               LDR SB R0 ...
                                            LDR R0 SB  0   (n)
                                            AND R0 R0  1
                                            BNE  5
                                            LDR R0 SB  4   (s)
                                            LDR R1 SB  0
                                            ADD R1 R1  1
                                            ROR R0 R0 R1
                                            BPL  2
        n := -1000                          MOV R0 R0 -1000
                                            STR R0 SB  0
   END ;
   IF n < 0 THEN                            LDR SB R0  ...
                                            LDR R0 SB  0
                                            CMP R0 R0  0
                                            BGE  3
        s := {}                             MOV R0 R0  0   {}
                                            STR R0 SB  4
                                            B  17
   ELSIF n < 10 THEN                        LDR SB R0 ...
                                            LDR R0 SB  0
                                            CMP R0 R0  10
                                            BGE  3
        s := {0}                            MOV R0 R0  1
                                            STR         R0 SB  4
                                            B  10
   ELSIF n < 100 THEN                       LDR SB R0 ...
                                            LDR R0 SB  0
                                            CMP R0 R0 100
                                            BGE  3
        s := {1}                            MOV R0 R0  2
                                            STR R0 SB  4
                                            B  3
   ELSE
        s := {2}                            MOV R0 R0 4
                                            LDR SB R0 ...
                                            STR R0 SB  4
   END
END Pattern5.
```

Pattern 6: *While and repeat statements.*

```
MODULE Pattern6;
  VAR i: INTEGER;
BEGIN i := 0;                    MOV R0 R0     0
                                 STR  R0 SB     0
  WHILE i < 10 DO                LDR SB R0     ...
                                 LDR R0 SB     0
                                 CMP R0 R0  10
                                 BGE  4
      i := i + 2                 LDR R0 SB  0
                                 ADD R0 R0  2
                                 STR R0 SB  0
  END ;                          B  -8
  REPEAT i := i - 1              LDR SB R0     ...
                                 LDR R0 SB     0
                                 SUB R0 R0     1
                                 STR  R0 SB     0
  UNTIL i = 0                    LDR R0 SB     0
                                 CMP R0 R0  0
                                 BNE  -7
END Pattern6.
```

Pattern 7: *For statements.*

```
MODULE Pattern7;
  VAR i, m, n: INTEGER;
BEGIN
  FOR i := 0 TO n-1 DO           MOV R0 R0   0
                                 LDR R1 SB   8
                                 SUB R1 R1 1
                                 CMP LNK R0 R1
                                 BGT   7
                                 STR  R0 SB   0
    m := 2*m                     LDR R0 SB   4
                                 LSL R0 R0  1
                                 STR  R0 SB  4
  END                            LDR R0 SB   0
                                 ADD R0 R0  1
                                 B   -11
END Pattern7.
```

Pattern 8: *Proper procedures:* Procedure bodies are surrounded by a prolog (entry code) and an epilog (exit code). They reposition the stack pointer SP (see Chapter 6), which holds the address of the procedure activation record on the stack. The immediate value of the first instruction indicates the space taken by variables local to the procedure, rounded up to the next multiple of 4.

Procedure calls use a branch and link (BL) instruction. Parameters are loaded into registers prior to the call and pushed on the stack after the call. Every parameter occupies a multiple of 4 bytes. In the case of value parameters the value is loaded, and in the case of VAR-parameters, the variable's address is loaded.

```
MODULE Pattern8;
  VAR i: INTEGER;

  PROCEDURE P(x: INTEGER; VAR y: INTEGER);
    VAR z: INTEGER;
  BEGIN                          SUB SP SP     16          adjust SP
                                 STR  LNK SP   0           push ret adr
                                 STR  R0 SP    4           push x
                                 STR  R1 SP    8           push @y
    z := x;                      LDR R0 SP     4           x
                                 STR  R0 SP    12          z
    y := z                       LDR R0 SP     12          z
```

```
                                LDR R1 SP      8        @y
                                STR  R0 R1     0        y
  END P;                        LDR LNK SP     0        pop ret adr
                                ADD SP SP     16
                                B R15

BEGIN P(5, i)                   MOV R0 R0      5
                                ADD R1 SB      0        @i
                                BL     -14               call
END Pattern8.
```

Pattern 9: *Function procedures.* They are handled in exactly the same manner as proper procedures, except that a result is returned in register R0. If the function is called in an expression at a place where intermediate results are held in registers, these values are put onto the stack before the call, and they are restored after return (not shown here).

```
MODULE Pattern9;
  VAR x: REAL;

  PROCEDURE F(x: REAL): REAL;
  BEGIN                         SUB SP SP 8
                                STR LNK SP 0          push ret adr
                                STR R0 SP  4          push x
    IF x >= 1.0 THEN            LDR R0 SP  4
                                MOV' R1 R0 3F80H
                                FSB R0 R0 R1
                                BLT        4
      x := F(F(x))              LDR R0 SP  4
                                BL -9
                                BL -10
                                STR R0 SP 4
    END ;
    RETURN x                    LDR R0 SP 4
  END F;                        LDR LNK SP 0          pop ret adr
                                ADD SP SP 8
                                B R15
END Pattern9.
```

Pattern 10: *Dynamic array parameters* are passed by loading a descriptor on the stack, regardless of whether they are value- or VAR- parameters. The descriptor consists of the actual variable's address and the array's length. (Only one-dimensional dynamic arrays are handled).

Elements of dynamic arrays are accessed like those of static arrays. However, even when the index is a constant, the check cannot be performed by the compiler.

```
MODULE Pattern10;
  VAR a: ARRAY 12 OF INTEGER;

  PROCEDURE P(x: ARRAY OF INTEGER);
      VAR i, n: INTEGER;
  BEGIN                         SUB SP SP  20
                                STR LNK SP  0
                                STR R0 SP 4           x
                                STR R1 SP 8  x.len
      n := x[i];                LDR R0 SP 12          i
                                LDR R1 SP  8          x.len
                                CMP R2 R0 R1
                                BLHI R12
                                LSL R0 R0  2
                                LDR R1 SP 4           x
```

```
                                ADD R0 R1 R0
                                LDR R0 R0      0
                                STR  R0 SP  16
      x[i+1] := n+5             LDR R0 SP  12          i
                                ADD R0 R0  1
                                LDR R1 SP  8           x.len
                                CMP R2 R0 R1
                                BLHI  R12
                                LSL R0 R0  2
                                LDR R1 SP  4           x
                                ADD R0 R1 R0
                                LDR R1 SP  16          n
                                ADD R1 R1 5
                                STR  R1 R0  0
  END P;                        LDR LNK SP      0
                                ADD SP SP     20
                                B  R15

BEGIN P(a);                     ADD R0 SB  0           a
                                MOV R1 R0 12           a.len
                                BL  -29
END Pattern10.
```

Pattern 11: *Sets.* This code pattern exhibits the construction of sets. If the specified elements are constants, the set value is computed by the compiler. Otherwise, sequences of move and shift instructions are used. Since shift instructions do not check whether the shift count is within sensible bounds, the results are unpredictable, if elements outside the range 0 .. 31 are involved.

```
MODULE Pattern11;
  VAR s: SET; m, n: INTEGER;
BEGIN
  s := {m};                     LDR R0 SB  4           m
                                MOV R1 R0  1
                                LSL R0 R1 R0
                                STR  R0 SB  0          s
  s := {0 .. n};                LDR R0 SB   8          n
                                MOV R1 R0  -2
                                LSL R0 R1 R0
                                XOR R0 R0  -1
                                STR  R0 SB   0
  s := {m .. 31};               LDR R0 SB    4         m
                                MOV R1 R0  31
                                MOV R2 R0   -2
                                LSL R1 R2 R1
                                MOV R2 R0     -1
                                LSL R0 R2 R0
                                XOR R0 R0 R1
                                STR  R0 SB  0          s
  s := {m .. n};                LDR R0 SB   4          m
                                LDR R1 SB   8          n
                                MOV R2 R0  -2
                                LSL R1 R2 R1
                                MOV R2 R0     -1
                                LSL R0 R2 R0
                                XOR R0 R0 R1
                                STR  R0 SB  0          s
  IF n IN {2, 3, 5, 7, 11, 13} THEN
                                MOV R0 R0  28ACH
                                LDR R1 SB    8
                                ADD R1 R1   1
                                ASR' R0 R0 R1
                                BPL   2
```

```
    m := 1                          MOV R0 R0   1
                                    STR  R0 SB   4          m
  END
END Pattern11.
```

Pattern 12: *Imported variables and procedures:* When a procedure is imported from another module, its address is unavailable to the compiler. Instead, the procedure is identified by a number obtained from the imported module's *symbol file*. In place of the offset, the branch instruction holds (1) the number of the imported module, (2) the number of the imported procedure, and (3) a link in the list of BL instructions calling an external procedure. This list is traversed by the linking loader, that computes the actual offset (fixup, see Chapter 6).

Imported variables are also referenced by a variable's number. In general, an access required two instructions. The first loads the static base register SB from a global table with the address of that module's data section. The module number of the imported variable serves as index. The second instruction loads the address of the variable, using the actual offset fixed up by the loader.

In the following example, modules Pattern12a and Pattern12b both export a procedure and a variable. They are referenced from the importing module Pattern12c.

```
MODULE Pattern12a;
  VAR k*: INTEGER;

  PROCEDURE P*;
  BEGIN k := 1
  END P;

END Pattern12a.


MODULE Pattern12b;
    VAR x*: REAL;

    PROCEDURE Q*;
    BEGIN x := 1
    END Q;

END Pattern12b.

MODULE Pattern12c;
    IMPORT Pattern12a, Pattern12b;

    VAR i: INTEGER; y: REAL;
BEGIN
    i := Pattern12a.k;          8D10xxxx    LDR SB 1 link     Pattern12a
                                80D00000    LDR R0 SB 0       Pattern12a.k
                                8D00xxxx    LDR SB 0 link     Pattern12c
                                A0D00000    STR R0 SB 0       Pattern12c.i
    y := Pattern12b.x;          8D20xxxx    LDR SB 2 link     Pattern12b
                                80D00000    LDR R0 SB 0       Pattern12b.x
                                8D00xxxx    LDR SB 0 link     Pattern12c
                                A0D00004    STR  R0 SB 4      Pattern12c.y
END Pattern12c.
```

Pattern 13: *Record extensions with pointers:* Fields of a record type R1, which is declared as an extension of a type R0, are simply appended to the fields of R0, i.e. their offsets are greater than those of the fields of R0. When a record is statically declared, its type is known by the compiler. If the record is referenced via a pointer, however, this is not the case. A pointer bound to a base type R0 may well refer to a record of an extension R1 of R0. Type tests (and type guards) allow to test for the actual type. This requires that a type can be identified at the time of program execution. Because the language defines name equivalence instead of structural equivalence of types, a type may be identified by a number. We use the address of a unique type descriptor for this purpose.

Therefore, type tests consist of a simple address comparison which is very fast. Type descriptors are stored in the module's area for data. Their address is called *type tag*. The tag of a (dynamically allocated) variable is stored as a prefix to its record (with offset -8).

A type descriptor contains - in addition to information stored for use by the garbage collector - a table of tags of all its base types. If, for instance, a type R2 is an extension of R1 which is an extension of R0, the descriptor of R2 contains the tags of R1 and R0 as shown in Fig. 12.4. The table has a fixed number of 3 entries.

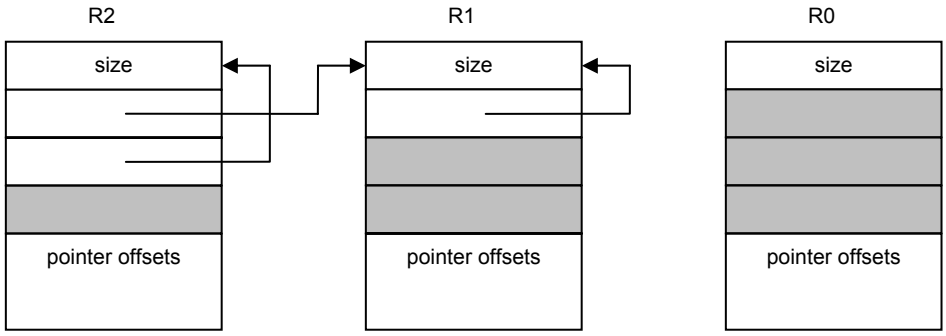


Figure 12.4 Type descriptors

A type test of the form *p IS T* then, consists of a comparison of the type tag of p^ at address p-8 with the tag held in the descriptor of T at the extension level of the type of p^. A type guard *p(T)* is synonymous to the statement

IF ~(p IS T) THEN abort END

The following example features 3 record types with associated pointer types, and hence also 3 type descriptors. Each descriptor is 5 words long. Their addresses, and therefore their tags, are 0, 20, and 40 respectively.

```
 0 00000020 FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF
20 00000020 00014006 FFFFFFFF FFFFFFFF FFFFFFFF
40 00000020 00014005 00028001 FFFFFFFF FFFFFFFF
```

```
MODULE Pattern13;
  TYPE
    P0 = POINTER TO R0;
    P1 = POINTER TO R1;
    P2 = POINTER TO R2;
    R0 = RECORD x: INTEGER END ;
    R1 = RECORD (R0) y: INTEGER END ;
    R2 = RECORD (R1) z: INTEGER END ;
  VAR
    p0: P0;                60
    p1: P1;                64
    p2: P2;                68
BEGIN
  p0.x := 0;               LDR R0 SB 60
                           MOV R1 R0 0        p0.x
                           STR  R1 R0 0       no type check
  p1.y := 1;               LDR R0 SB 64
                           MOV R1 R0 1
                           STR  R1 R0 4       p1.y
  p0(P1).y := 3;           LDR R0 SB 60       p0
                           LDR R1 R0 -8       tag(p0)
                           LDR R1 R1 4
                           ADD R2 SB 20       TD P1
                           CMP R3 R2 R1
```

```
                        BLNE  R12
                        MOV R1 R0 3
                        STR  R1 R0 4        p0.z
  p0(P2).z := 5;        LDR R0 SB 60        p0
                        LDR R1 R0 -8        tag(p0)
                        LDR R1 R1 8
                        ADD R2 SB 40        TD P2
                        CMP R3 R2 R1
                        BLNE  R12
                        MOV R1 R0 5
                        STR  R1 R0 8        p0.z
  IF p1 IS P2 THEN      LDR R0 SB 64        p1
                        LDR R1 R0 -8        tag(p1)
                        LDR R1 R1 8
                        ADD R2 SB 40        TD P2
                        CMP R3 R2 R1
                        BNE 2
    p0 := p2            LDR R0 SB     68
                        STR R0 SB     60
  END
END Pattern13.
```

Pattern 14: *Record extensions as VAR parameters:* Records occurring as VAR-parameters may also require a type test at program execution time. This is because VAR-parameters effectively constitute hidden pointers. Type tests and type guards on VAR-parameters are handled in the same way as for variables referenced via pointers, with a slight difference, however. Statically declared record variables may be used as actual parameters, and they are not prefixed by a type tag. Therefore, the tag has to be supplied together with the variable's address when the procedure is called, i.e. when the actual parameter is established. Record structured VAR-parameters therefore consist of address and type tag. This is similar to dynamic array descriptors consisting of address and length.

```
 0 00000020 FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF
20 00000020 00014006 FFFFFFFF FFFFFFFF FFFFFFFF

MODULE Pattern14;
    TYPE
      R0 = RECORD a, b, c: INTEGER END ;
      R1 = RECORD (R0) d, e: INTEGER END ;
    VAR
      r0: R0;               40
      r1: R1;               52

    PROCEDURE P(VAR r: R0);
    BEGIN                   ...

      r.a := 1;             LDR R1 SP 4         r
                            STR R0 R1 0         r.a
      r(R1).d := 2          LDR R0 SP 8         tag(r)
                            LDR R0 R0 4
                            ADD R1 SB 20        R1
                            CMP R2 R1 R0
                            BLNE  R12
                            MOV R0 R0 2
                            LDR R1 SP 4         r
                            STR R0 R1 12        r.d
    END P;                  ...

BEGIN                       ...
    P(r0);                  ADD R0 SB 40        r0
                            ADD R1 SB 0         tag(R0)
                            BL  P
```

```
    P(r1)                ADD R0 SB 52        r1
                         ADD R1 SB 20        tag(R1)
                         BL  P
END Pattern14.           ...
```

Pattern 15: *Array assignments and strings.*

```
MODULE Pattern15;
  VAR s0, s1: ARRAY 32 OF CHAR;

  PROCEDURE P(x: ARRAY OF CHAR);
  END P;

BEGIN s0 := "ABCDEF";    ADD R0 SB   0       @s0
                         ADD R1 SB   64      @"ABCDEF"
                         LDR R2 R1   0
                         ADD R1 R1   4
                         STR R2 R0   0
                         ADD R0 R0   4
                         ASR R2 R2   24      test for 0X
                         BNE     -6

  s0 := s1;              ADD R0 SB   0       @s0
                         ADD R1 SB   32      @s1
                         MOV R2 R0   8       len
                         LDR R3 R1   0
                         ADD R1 R1   4
                         STR R3 R0   0
                         ADD R0 R0   4
                         SUB R2 R2   1
                         BNE     -6

  P(s1);                 ADD R0 SB   32      @s1
                         MOV R1 R0   32      len
                         BL  -38             P

  P("012345");           ADD R0 SB   72      @"012345"
                         MOV R1 R0   7       len  (incl 0X)
                         BL  -42             P

  P("%")                 ADD R0 SB   80      @"%"
                         MOV R1 R0   2       len
                         BL  -46             P
END Pattern15.
```

Pattern 16: *Predeclared procedures.*

```
MODULE Pattern16;
  VAR m, n: INTEGER;
      x: REAL; u: SET;
      a, b: ARRAY 10 OF INTEGER;
      s, t: ARRAY 16 OF CHAR;
BEGIN
  INC(m);                ADD R0 SB   0       @m
                         LDR R1 R0   0
                         ADD R1 R1   1
                         STR R1 R0   0
  DEC(n, 10);            ADD R0 SB   4       @n
                         LDR R1 R0   0
                         SUB R1 R1   10
                         STR R1 R0   0
  INCL(u, 3);            ADD R0 SB   12      @u
                         LDR R1 R0   0
                         OR R1 R1    8       {3}
                         STR R1 R0   0
```

```
   EXCL(u, 7);              ADD R0 SB  12        @u
                            LDR R1 R0   0
                            AND R1 R1  -129      -{7}
                            STR R1 R0   0
   ASSERT(m < n);           LDR R0 SB  0
                            LDR R1 SB  4
                            CMP R0 R0 R1
                            BLGE  R12
   UNPK(x, n);              LDR R0 SB  8         x
                            ASR R1 R0   23
                            SUB R1 R1  127
                            STR R1 SB   4        n
                            LSL R1 R1  23
                            SUB R0 R0 R1
                            STR R0 SB   8        x
   PACK(x, n);              LDR R0 SB   8        x
                            LDR R1 SB   4        n
                            LSL R1 R1  23
                            ADD R0 R0 R1
                            STR R0 SB   8        x
   s := "0123456789";       ADD R0 SB  96        @s
                            ADD R1 SB  128       adr of string
                            LDB R2 R1   0        loop
                            ADD R1 R1  4
                            STB R2 R0   0
                            ADD R0 R0   4
                            ASR R2 R2  24
                            BNE       -6
   IF s < t THEN            ADD R0 SB  96        @s
                            ADD R1 SB  112       @t
                            LDB R2 R0   0        loop
                            ADD R0 R0   1
                            LDB R3 R1   0
                            ADD R1 R1   1
                            CMP R4 R2 R3
                            BNE   2
                            CMP R4 R2  0
                            BNE   -8
                            BGE   3

      m := 1                MOV R0 R0  1
                            STR R0 SB   0        m

   END
END Pattern16.
```

Pattern 17: *Predeclared functions.*

```
MODULE Pattern17;
   VAR m, n: INTEGER;
       x, y: REAL;
       b: BOOLEAN; ch: CHAR;
BEGIN
   n := ABS(m);             LDR R0 SB 0          m
                            CMP R0 R0 0
                            BGE 2
                            MOV R1 R0 0
                            SUB R0 R1 R0
                            STR R0 SB 4          n
   y := ABS(x);             LDR R0 SB 8          x
                            LSL R0 R0 1
```

```
                        ROR R0 R0 1
                        STR R0 SB 12        y
b := ODD(n);            LDR R0 SB 4         n
                        AND R0 R0 1
                        BEQ  2
                        MOV R0 R0 1
                        B  1
                        MOV R0 R0 0
                        STB R0 SB 16        b
n := ORD(ch);           LDB R0 SB 17        ch
                        STR R0 SB 4         n
n := FLOOR(x);          LDR R0 SB 8         x
                        MOV' R1 R0 4B00H
                        FAD" R0 R0 R1       floor
                        STR R0 SB 4         n
y := FLT(m);            LDR R0 SB 0         m
                        MOV' R1 R0 4B00H
                        FAD' R0 R0 R1       float
                        STR R0 SB 12        y
n := LSL(m, 3);         LDR R0 SB 0         m
                        LSL R0 R0 3
                        STR R0 SB 4         n
n := ASR(m, 8);         LDR R0 SB 0
                        ASR R0 R0 8
                        STR R0 SB 4
m := ROR(m, n);         LDR R0 SB 0
                        LDR R1 SB 4
                        ROR R0 R0 R1
                        STR R0 SB 0

END Pattern17.
```

12.3. Internal data structures and module interfaces

12.3.1. Data structures

In Section 12.1 it was explained that declarations inherently constitute context-dependence of the translation process. Although parsing still proceeds on the basis of a context-free syntax and relies on contextual information only in a few isolated instances, information provided by declarations affects the generated code significantly. During the processing of declarations, their information is transferred into the "symbol table", a data structure of considerable complexity, from where it is retrieved for the generation of code.

This dynamic data structure is defined in module ORB in terms of two record types called *Object* and *Struct*. These types pervade all other modules with the exception of the scanner. They are therefore explained before further details of the compiler are discussed (see module ORB below).

For each declared identifier an instance of type *Object* is generated. The record holds the identifier and the properties associated with the identifier given in its declaration. Since Oberon is a *statically typed* language, every object has a type. It is represented in the record by its *typ* field, which is a pointer to a record of type *Struct*. Since many objects may be of the same type, it is appropriate to record the type's attributes only once and to refer to them via a pointer. The properties of type *Struct* will be discussed below.

The kind of object which a table entry represents is indicated by the field *class*. Its values are denoted by declared integer constants: *Var* indicates that the entry describes a variable, *Con* a constant, *Fld* a record field, *Par* a VAR-parameter, and *Proc* a procedure. Different kinds of entries carry different attributes. A variable or a parameter carries an address, a constant has a value, a record field has an offset, and a procedure has an entry address, a list of parameters, and a result type. For each class the introduction of an extended record type would seem advisable. This was not done, however, for three reasons. First, the compiler was first formulated in (a subset of)

Modula-2 which does not feature type extension. Second, not making use of type extensions would make it simpler to translate the compiler into other languages for porting the language to other computers. And third, all extensions were known at the time the compiler was planned. Hence extensibility provided no argument for the introduction of a considerable variety of types. The simplest solution lies in using the multi-purpose fields *val* and *dsc* for class-specific attributes. For example, *val* holds an address for variables, parameters, and procedures, an offset for record fields, and a value for constants.

The definition of a type yields a record of type *Struct*, regardless of whether it occurs within a type declaration, in which case also a record of type *Object* (class = *Typ*) is generated, or in a variable declaration, in which case the type remains anonymous. All types are characterized by a form and a size. A type is either a basic type or a constructed type. In the latter case it refers to one or more other types. Constructed types are arrays, records, pointers, and procedural types. The attribute *form* refers to this classification. Its value is an integer

Just as different object classes are characterized by different attributes, different forms have different attributes. Again, the introduction of extensions of type *Struct* was avoided. Instead, some of the fields of type *Struct* remain unused in some cases, such as for basic types, and others are used for form-specific attributes. For example, the attribute *base* refers to the element type in the case of an array, to the result type in the case of a procedural type, to the type to which a pointer is bound, or to the base type of a (extended) record type. The attribute *dsc* refers to the parameter list in the case of a procedural type, or to the list of fields in the case of a record type.

As an example, consider the following declarations. The corresponding data structure is shown in Fig. 12.5. For details, the reader is referred to the program listing of module ORB and the respective explanations.

```
CONST  N = 100;
TYPE   Ptr = POINTER TO Rec;
       Rec = RECORD n: INTEGER; p, q: Ptr END ;
VAR    k: INTEGER;
       a: ARRAY N OF INTEGER;
PROCEDURE P(x: INTEGER): INTEGER;
```

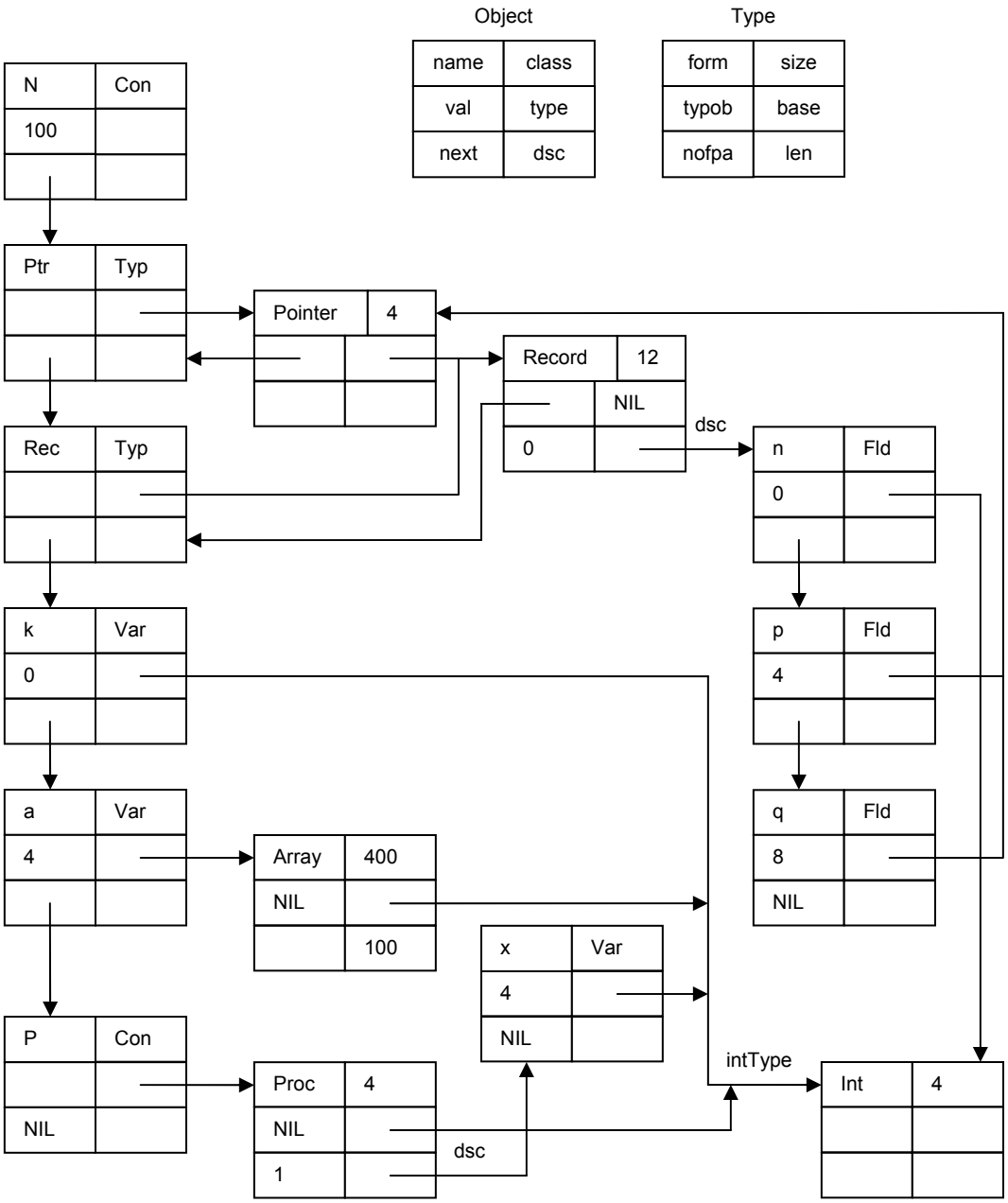


Figure 12.5. Representation of declarations

Only entries representing constructed types are generated during compilation. An entry for each basic type is established by the compiler's initialization. It consists of an *Object* holding the standard type's identifier and a *Struct* indicating its form, denoted by one of the values *Byte, Bool, Char, Int, Real,* or *Set*. The object records of the basic types are anchored in global pointer variables in module ORB (which actually should be regarded as constants).

Not only are entries created upon initialization for basic types, but also for all standard procedures. Therefore, every compilation starts out with a symbol table reflecting all standard, pervasive identifiers and the objects they stand for.

We now return to the subject of *Objects*. Whereas objects of basic classes (*Const, Var, Par, Fld, Typ, SProc, SFunc* and *Mod*) directly reflect declared identifiers and constitute the context in which statements and expressions are compiled, compilations of expressions typically generate

anonymous entities of additional, non-basic modes. Such entities reflect selectors, factors, terms, etc., i.e. constituents of expressions and statements. As such, they are of a transitory nature and hence are not represented by records allocated on the heap. Instead, they are represented by record variables local to the processing procedures and are therefore allocated on the stack. Their type is called *Item* and is a slight variation of the type Object. Items are not referenced via pointers.

Let us assume, for instance, that a term *x*y* is parsed. This implies that the operator and both factors have been parsed already. The factors *x* and *y* are represented by two variables of type *Item* of *Var* mode. The resulting term is again described by an item, and since the product is transitory, i.e. has significance only within the expression of which the term is a constituent, it is to be held in a temporary location, in a register. In order to express that an item is located in a register, a new, non-basic mode *Reg* is introduced.

Effectively, all non-basic modes reflect the target computer's architecture, in particular its addressing modes. The more addressing modes a computer offers, the more item modes are needed to represent them. The additional item modes required by the RISC processor are. They are declared in module ORG:

Reg	direct register mode
RegI	indirect register mode
Cond	condition code mode

The use of the types *Object, Item,* and *Struct* for the various modes and forms, and the meaning of their attributes are explained in the following tables:

	Objects:		Items:		
	class	val	a	b	r
0	Undef				
1	Const	val	val		
2	Var	adr	adr	base	
3	Par	adr	adr	off	
4	Fld	off	off		
5	Typ	TDadr	TDadr	modno	
6	SProc	num			
7	SFunc	num			
8	Mod				
10	Reg				regno
11	RegI		off		regno
12	Cond		Tjmp	Fjmp	condition code

Structures:

	form	nofpar	len	dsc	base
7	Pointer				base type
10	ProcTyp	nofpar		param	result type
12	Array		nofel		element typ
13	Record	ext lev	desc adr	fields	extension type

Items have an attribute called *lev* which is part of the address of the item. Positive values denote the level of nesting of the procedure in which the item is declared; lev = 0 implies a global object. Negative values indicate that the object is imported from the module with number *-lev*.
The three types *Object, Item,* and *Struct* are defined in module ORB, which also contains procedures for accessing the symbol table.

12.3.2. Module interfaces

Before embarking on a presentation of the compiler's main module, the parser, an overview of its remaining modules is given in the form of their interfaces. The reader is invited to refer to them when studying the parser.

The interface of the scanner module ORS is simple. It defines the numeric values of all symbols. But its chief constituent is procedure *Get*. Each call yields the next symbol from the source text, identified by an integer. Global variables represent attributes of the read symbol in certain cases. If a number was read, *ival* or *rval* hold its numeric value. If an identifier or a string was read, *str* holds the ASCII values of the characters read.

Procedure *Mark* serves to generate a diagnostic output indicating a brief diagnostic and the scanner's current position in the source text. This procedure is located in the scanner, because only the scanner has access to its current position. *Mark* is called from all other modules.

```
DEFINITION ORS;  (*Scanner*)
   IMPORT Texts, Oberon;

   TYPE Ident = ARRAY 32 OF CHAR;
   VAR ival, slen: INTEGER;
         rval: REAL;
         id: Ident;
         str: ARRAY 256 OF CHAR;
         errcnt: BOOLEAN;

   PROCEDURE Mark (msg: ARRAY OF CHAR);
   PROCEDURE Get (VAR sym: INTEGER);
   PROCEDURE Init (source: Texts.Text; pos: INTEGER);
END ORS.
```

Module ORB defines the basic data structures representing declared objects and their types. It also contains procedures for accessing these structures. *NewObj* serves to insert a new identifier, and it returns a pointer to the allocated object. *ThisObj* returns the pointer to the object whose name equals the global scanner variable *ORS.id. Thisimport* and *thisfield* deliver imported objects and record fields with names equal to *ORS.id.*

Procedure *Import* serves to read the specified symbol file and to enter its identifier in the symbol table (class = *Mod*). Finally, *Export* generates the symbol file of the compiled module, containing descriptions of all objects and structures marked for export.

```
DEFINITION ORB;  (*Base table handler*)
   TYPE
      Object = POINTER TO ObjDesc;
      Type = POINTER TO TypeDesc;
      ObjDesc = RECORD
            class, lev, exnp: INTEGER;
            expo, rdo: BOOLEAN;
            next, dsc: Object;
            type: Type;
            name: ORS.Ident;
            val: INTEGER
      END ;
      TypeDesc = RECORD
            form, ref, mno: INTEGER;  (*ref is used for import/export only*)
            nofpar: INTEGER;  (*for records: extension level*)
            len: INTEGER;  (*for records: address of descriptor*)
            dsc, typobj: Object;
            base: Type;
            size: INTEGER
      END ;

   VAR topScope: Object;
      byteType, boolType, charType, intType, realType, setType,
      nilType, noType, strType: Type;
```

```
   PROCEDURE Init;
   PROCEDURE Close;
   PROCEDURE NewObj (VAR obj: Object; id: ORS.Ident; class: INTEGER);
   PROCEDURE thisObj (): Object;
   PROCEDURE thisimport (mod: Object): Object;
   PROCEDURE thisfield (rec: Type): Object;
   PROCEDURE OpenScope;
   PROCEDURE CloseScope;
   PROCEDURE Import (VAR modid, modid1: ORS.Ident);
   PROCEDURE Export (VAR modid: ORS.Ident;
            VAR newSF: BOOLEAN; VAR key: INTEGER);
END ORB.
```

Module ORG contains the procedures for code generation. The names of these procedures indicate the respective constructs for which code is to be produced. Note that an individual code generator procedure is provided for every standard, predefined procedure. This is necessary, because they generate in-line code.

```
DEFINITION ORG;
   CONST WordSize* = 4;
   TYPE Item* = RECORD
        mode*: INTEGER;
        type*: ORB.Type;
        a*, b*, r: INTEGER;
        rdo*: BOOLEAN  (*read only*)
      END ;
   VAR pc: INTEGER;

   PROCEDURE MakeConstItem*(VAR x: Item; typ: ORB.Type; val: INTEGER);
   PROCEDURE MakeRealItem*(VAR x: Item; val: REAL);
   PROCEDURE MakeStringItem*(VAR x: Item; len: INTEGER);
   PROCEDURE MakeItem*(VAR x: Item; y: ORB.Object; curlev: INTEGER);
   PROCEDURE Field*(VAR x: Item; y: ORB.Object);   (* x := x.y *)
   PROCEDURE Index*(VAR x, y: Item);   (* x := x[y] *)
   PROCEDURE DeRef*(VAR x: Item);
   PROCEDURE BuildTD*(T: ORB.Type; VAR dc: INTEGER);
   PROCEDURE TypeTest*(VAR x: Item; T: ORB.Type; varpar, isguard: BOOLEAN);

   PROCEDURE Not*(VAR x: Item);   (* x := ~x,  Boolean operators *)
   PROCEDURE And1*(VAR x: Item);   (* x := x & *)
   PROCEDURE And2*(VAR x, y: Item);
   PROCEDURE Or1*(VAR x: Item);   (* x := x OR *)
   PROCEDURE Or2*(VAR x, y: Item);

   PROCEDURE Neg*(VAR x: Item);   (* x := -x, arithmetic operators *)
   PROCEDURE AddOp*(op: LONGINT; VAR x, y: Item);   (* x := x +- y *)
   PROCEDURE MulOp*(VAR x, y: Item);   (* x := x * y *)
   PROCEDURE DivOp*(op: INTEGER; VAR x, y: Item);   (* x := x op y *)
   PROCEDURE RealOp*(op: INTEGER; VAR x, y: Item);   (* x := x op y *)

   PROCEDURE Singleton*(VAR x: Item);  (* x := {x}, set operators *)
   PROCEDURE Set*(VAR x, y: Item);   (* x := {x .. y} *)
   PROCEDURE In*(VAR x, y: Item);  (* x := x IN y *)
   PROCEDURE SetOp*(op: INTEGER; VAR x, y: Item);   (* x := x op y *)

   PROCEDURE IntRelation*(op: INTEGER; VAR x, y: Item);   (* x := x < y *)
   PROCEDURE SetRelation*(op: INTEGER; VAR x, y: Item);   (* x := x < y *)
   PROCEDURE RealRelation*(op: INTEGER; VAR x, y: Item);   (* x := x < y *)
   PROCEDURE StringRelation*(op: INTEGER; VAR x, y: Item);   (* x := x < y *)

   PROCEDURE StrToChar*(VAR x: Item);   (*assinments*)
   PROCEDURE Store*(VAR x, y: Item); (* x := y *)
   PROCEDURE StoreStruct*(VAR x, y: Item); (* x := y *)
   PROCEDURE CopyString*(VAR x, y: Item);  (*from x to y*)
```

```
    PROCEDURE VarParam*(VAR x: Item; ftype: ORB.Type);  (*parameters*)
    PROCEDURE ValueParam*(VAR x: Item);
    PROCEDURE OpenArrayParam*(VAR x: Item);
    PROCEDURE StringParam*(VAR x: Item);

    PROCEDURE For0*(VAR x, y: Item);   (*For Statements*)
    PROCEDURE For1*(VAR x, y, z, w: Item; VAR L: LONGINT);
    PROCEDURE For2*(VAR x, y, w: Item);

    (* Branches, procedure calls, procedure prolog and epilog *)
    PROCEDURE Here*(): LONGINT;
    PROCEDURE FJump*(VAR L: LONGINT);
    PROCEDURE CFJump*(VAR x: Item);
    PROCEDURE BJump*(L: LONGINT);
    PROCEDURE CBJump*(VAR x: Item; L: LONGINT);
    PROCEDURE Fixup*(VAR x: Item);
    PROCEDURE PrepCall*(VAR x: Item; VAR r: LONGINT);
    PROCEDURE Call*(VAR x: Item; r: LONGINT);
    PROCEDURE Enter*(parblksize, locblksize: LONGINT; int: BOOLEAN);
    PROCEDURE Return*(form: INTEGER; VAR x: Item; size: LONGINT; int: BOOLEAN);

    (* In-line code procedures*)
    PROCEDURE Increment*(upordown: LONGINT; VAR x, y: Item);
    PROCEDURE Include*(inorex: LONGINT; VAR x, y: Item);
    PROCEDURE Assert*(VAR x: Item);
    PROCEDURE New*(VAR x: Item);
    PROCEDURE Pack*(VAR x, y: Item);
    PROCEDURE Unpk*(VAR x, y: Item);
    PROCEDURE Led*(VAR x: Item);
    PROCEDURE Get*(VAR x, y: Item);
    PROCEDURE Put*(VAR x, y: Item);
    PROCEDURE Copy*(VAR x, y, z: Item);
    PROCEDURE LDPSR*(VAR x: Item);
    PROCEDURE LDREG*(VAR x, y: Item);

    (*In-line code functions*)
    PROCEDURE Abs*(VAR x: Item);
    PROCEDURE Odd*(VAR x: Item);
    PROCEDURE Floor*(VAR x: Item);
    PROCEDURE Float*(VAR x: Item);
    PROCEDURE Ord*(VAR x: Item);
    PROCEDURE Len*(VAR x: Item);
    PROCEDURE Shift*(fct: LONGINT; VAR x, y: Item);
    PROCEDURE ADC*(VAR x, y: Item);
    PROCEDURE SBC*(VAR x, y: Item);
    PROCEDURE UML*(VAR x, y: Item);
    PROCEDURE Bit*(VAR x, y: Item);
    PROCEDURE Register*(VAR x: Item);
    PROCEDURE H*(VAR x: Item);
    PROCEDURE Adr*(VAR x: Item);
    PROCEDURE Condition*(VAR x: Item);

    PROCEDURE Open*(v: INTEGER);
    PROCEDURE SetDataSize*(dc: LONGINT);
    PROCEDURE Header*;
    PROCEDURE Close*(VAR modid: ORS.Ident; key, nofent: LONGINT);
END ORG.
```

12. 4. The Parser

The main module ORP constitutes the parser. Its single command *Compile* - at the end of the program listing - identifies the source text according to the Oberon command conventions. It then calls procedure *Module* with the identified source text as parameter. The command forms are:

ORP.Compile @	The most recent selection identifies the beginning of the source text.
ORP.Compile ^	The most recent selection identifies the name of the source file.
ORP.Compile f0 f1 ... ~	f0, f1, ... are the names of source files.

File names and the characters @ and ^ may be followed by an option specification /s. Option s enables the compiler to overwrite an existing symbol file, thereby invalidating clients.

The parser is designed according to the proven method of top-down, recursive descent parsing with a look-ahead of a single symbol. The last symbol read is represented by the global variable *sym*. Syntactic entities are mirrored by procedures of the same name. Their goal is to recognize the specified construct in the source text. The start symbol and corresponding procedure is *Module*. The principal parser procedures are shown in Fig. 12.6., which also exhibits their calling hierarchy. Loops in the diagram indicate recursion in the syntactic definition.

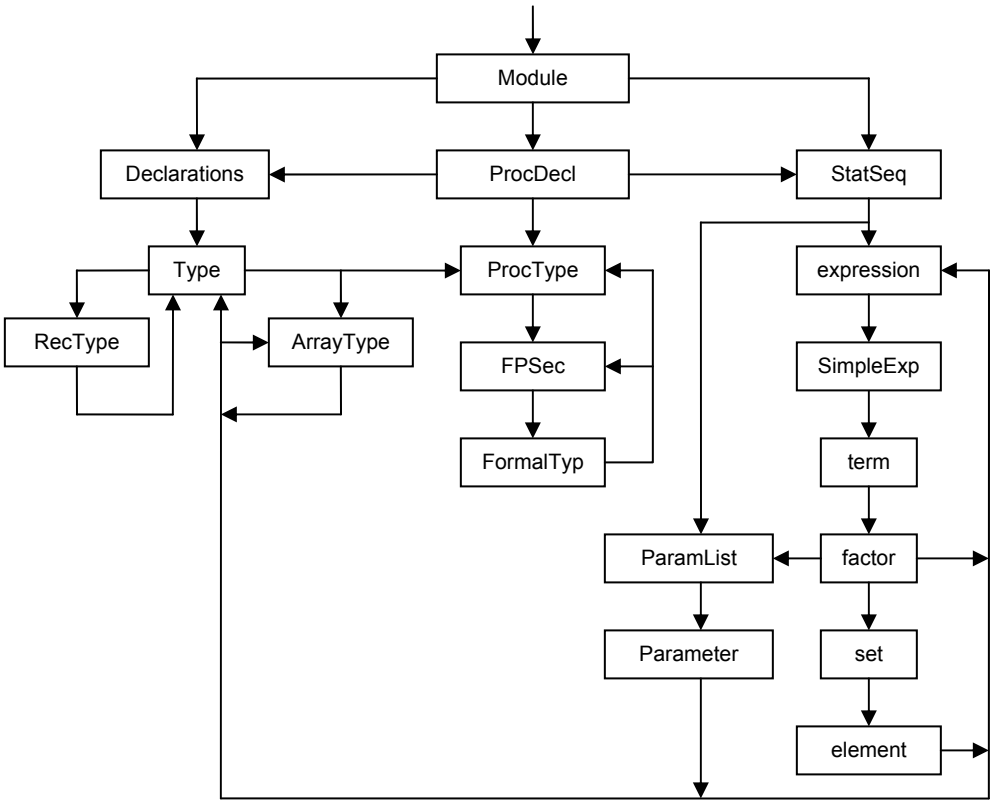


Figure 12.6 Parser procedure hierarchy

The rule of parsing strictly based on a single-symbol look-ahead and without reference to context is violated in three places. The prominent violation occurs in statements. If the first symbol of a statement is an identifier, the decision of whether an assignment or a procedure call is to be recognized is based on contextual information, namely the class of the identified object. The second violation occurs in *qualident*; if the identifier x preceding a period denotes a module, it is recognized together with the subsequent identifier as a qualified identifier. Otherwise x supposedly denotes a record variable. The third violation is made in procedure *selector*; if an identifier is followed by a left parenthesis, the decision of whether a procedure call or a type guard is to be recognized is again made on the basis of contextual information, namely the mode of the identified object.

A fairly large part of the program is devoted to the discovery of errors. Not only should they be properly diagnosed. A much more difficult requirement is that the parsing process should continue on the basis of a good guess about the structure that the text should most likely have. The parsing process must continue with some assumption and possibly after skipping a short piece of the

source text. Hence, this aspect of the parser is mostly based on heuristics. Incorrect assumptions about the nature of a syntactic error lead to secondary error diagnostics. There is no way to avoid them. A reasonably good result is obtained by the fact that procedure *ORS.Mark* inhibits an error report, if it lies less than 10 characters ahead of the last one. Also, the language Oberon is designed with the property that most large constructs begin with a unique symbol, such as IF, WHILE, CASE, RECORD, etc. These symbols facilitate the recovery of the parsing process in the erroneous text. More problematic are open constructs which neither begin nor end with key symbols, such as types, factors, and expressions. Relying on heuristics, the source text is skipped up to the first occurrence of a symbol which may begin a construct that follows the one being parsed. The employed scheme may not be the best possible, but it yields quite acceptable results and keeps the amount of program devoted to the handling of erroneous texts within justifiable bounds.

Besides the parsing of text, the Parser also performs the checking for type consistency of objects. This is based on type information held in the global table, gained during the processing of declarations, which is also handled by the routines which parse. Thereby an unjustifiably large number of very short procedures is avoided. However, the strict target-computer independence of the parser is violated slightly: Information about variable allocation strategy including alignment, and about the sizes of basic types is used in the parser module. Whereas the former violation is harmless, because the allocation strategy is hardly controversial, the latter case constitutes a genuine target-dependence embodied in a number of explicitly declared constants. Mostly these constants are contained in the respective type definitions, represented by records of type *Type* initialized by ORB. The following procedures allocate objects and generate elements of the symbol table:

Declarations	Object(Con), Object(Typ), Object(Var)
ProcedureDeclaration	Object(xProc)
FormalType	Object(Var), Object(Par)
ORB.Import	Object(Mod)
RecordType	Object(Fld), Type(Record)
ArrayType	Type(Array)
ProcedureType	Type(ProcTyp)
Type	Type(Pointer)
FormalType	Type(Array)

An inherently nasty subject is the treatment of forward references in a single-pass compiler. In Oberon, there are two such cases:

1. Forward declarations of procedures. They have been eliminated from the revision of the Oberon language in the year 2007 as they should be avoided if ever possible. If it is impossible, a remedy is to declare a variable of the given procedure type, and assign the procedure to be forwarded to this variable. The nastiness of procedure forward declarations originates in the necessity to specify parameters and result type in the forward declaration. These must be repeated in the actual procedure declaration, and one expects that a compiler verifies the equality (or equivalence) of the two declarations. This is a heavy burden for a case that very rarely occurs.

2. Forward declarations of pointer types also constitutes a nasty exception, but its exclusion would be difficult to justify. If in a pointer declaration the base type (to which the pointer is bound) is not found in the symbol table, a forward reference is therefore automatically assumed. An entry for the pointer type is generated anyway (see procedure *Type*) and an element is inserted in the list of pointer base types to be fixed up. This list is headed by the global variable *pbsList.* When later in the text a declaration of a record type is encountered with the same identifier, the forward entry is recognized and the proper link is established (see procedure *Declarations*).

The compiler must check for undefined forward references when the current declaration scope is closed. This check is performed at the end of procedure *Declarations.*

The with statement had been eliminated from the language in its revision of 2007. Here it reappears in the form of a case statement, whose cases are not labelled by integers, but rather by types. What formerly was written as

```
IF x IS T1 THEN
    WITH x: T1 DO ... x ... END
ELSIF x IS T2 THEN
    WITH x: T2 DO ... x ... END
ELSIF ...
END
```

is now written more simply and more efficiently as

```
CASE x OF
 T1: ... x ... |
 T2: ... x ... |
...
END
```

where *T1* and *T2* are extensions of the type *T0* of the case variable *x*. Compilation of this form of case statement merges the regional type guard of the former with statements with the type test of the former if statements. This case statement represents the only case where a symbol table entry - the type of *x* - is modified during compilation of statements. When the end of the with statement is reached, the change must be reverted.

12.5. The scanner

The scanner module ORS embodies the lexicographic definitions of the language, i.e. the definition of abstract symbols in terms of characters. The scanner's substance is procedure *Get*, which scans the source text and, for each call, identifies the next symbol and yields the corresponding integer code. It is most important that this process be as efficient as possible. Procedure *Get* recognizes letters indicating the presence of an identifier (or reserved word), and digits signalling the presence of a number. Also, the scanner recognizes comments and skips them. The global variable *ch* stands for the last character read.

A sequence of letters and digits may either denote an identifier or a key word. In order to determine which is the case, a search is made in a table containing all key words for each would-be identifier. This table is sorted alphabetically and according to the length of reserved words. It is initialized when the compiler is loaded.

The presence of a digit signals a number. Procedure *Number* first scans the subsequent digits (and letters) and stores them in a buffer. This is necessary, because hexadecimal numbers are denoted by the postfix letter H (rather than a prefix character). The postfix letter X specifies that the digits denote a character.

There exists one case in the language Oberon, where a look-ahead of a single character does not suffice to identify the next symbol. When a sequence of digits is followed by a period, this period may either be the decimal point of a real number, or it may be the first element of a range symbol (..). Fortunately, the problem can be solved locally as follows: If, after reading digits and a period, a second period is present, the number symbol is returned, and the look-ahead variable *ch* is assigned the special value 7FX. A subsequent call of *Get* then delivers the range symbol. Otherwise the period after the digit sequence belongs to the (real) number.

12.6. Searching the symbol table, and handling symbol files

12.6.1. The structure of the symbol table

The symbol table constitutes the context in which statements and expressions are parsed. Each procedure establishes a scope of visibility of local identifiers. The records registering identifiers belonging to a scope are linked as a linear list. They are of type *Object*. Each object has a type.

Types are represented by records of type *Type*. These two types pervade the entire compiler, and they are defined as follows:

```
TYPE Object = POINTER TO ObjDesc;
  Type = POINTER TO TypeDesc;

  ObjDesc = RECORD
      class, lev, exno: INTEGER;
      expo, rdo: BOOLEAN;   (*exported / read-only*)
      next, dsc: Object;
      type: Type;
      name: ORS.Ident;
      val:  INTEGER
    END ;

  TypeDesc = RECORD
      form, ref, mno: INTEGER;  (*ref is only used for import/export*)
      nofpar: INTEGER;  (*for procedures; extension level for records*)
      len:  INTEGER;  (*for arrays, len < 0 => open array; for records: adr of descriptor*)
      dsc, typobj: Object;
      base: Type;  (*for arrays, records, pointers*)
      size:  INTEGER;  (*in bytes; always multiple of 4, except for Byte, Bool and Char*)
    END ;
```

Procedures for generating and searching the lists are contained in module ORB. If a new identifier is to be added, procedure *NewObj* first searches the list, and if the identifier is already present, a double definition is diagnosed. Otherwise the new element is appended, thereby preserving the order given by the source text.

Procedures, and therefore also scopes, may be nested. Each scope is represented by the list of its declared identifiers, and the list of the currently visible scopes are again connected as a list. Procedure *OpenScope* appends an element and procedure *CloseScope* removes it. The list of scopes is anchored in the global variable *topScope* and linked by the field *dsc.* It is treated like a stack. It consists of elements of type *Object*, each one being the header (*class* = *Head*) of the list of declared entities. As an example, the procedure for searching an object (with name *ORS.id*) is shown here:

```
PROCEDURE thisObj*(): Object;
  VAR s, x: Object;
BEGIN s := topScope;
  REPEAT x := s.next;
    WHILE (x # NIL) & (x.name # ORS.id) DO x := x.next END ;
    s := s.dsc
  UNTIL (x # NIL) OR (s = NIL);
  RETURN x
END thisObj;
```

A snapshot of a symbol table for an example with nested scopes is shown in Fig. 12.6. It is taken when the following declarations are parsed and when the statement *S* is reached.

```
VAR x: INTEGER;

PROCEDURE P(u: INTEGER);
BEGIN ... END P;

PROCEDURE Q(v: INTEGER);
  PROCEDURE R(w: INTEGER);
  BEGIN S END R;
BEGIN ... END Q;
```

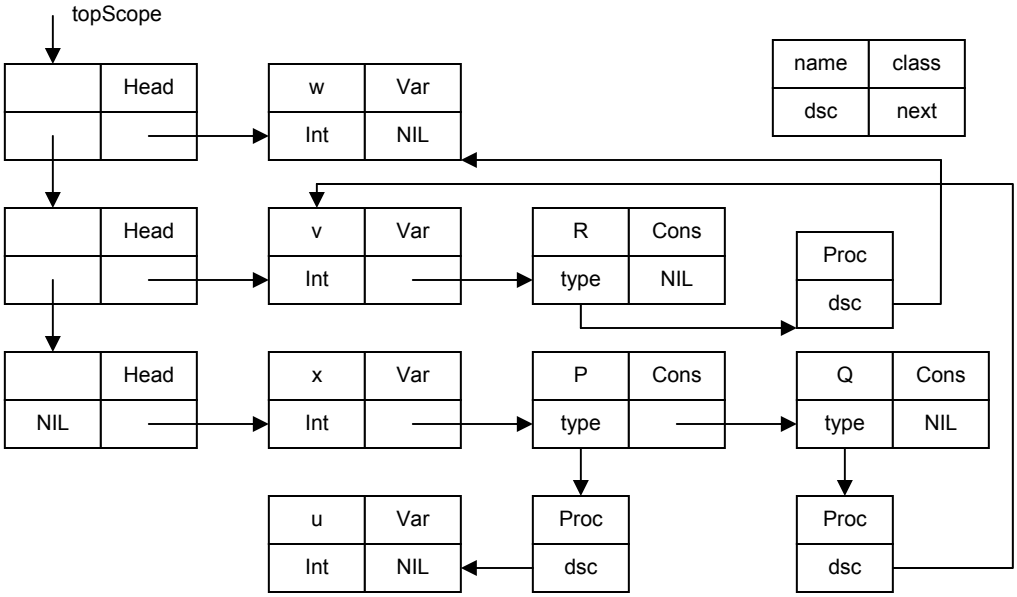


Fig. 12.7 Snapshot of a symbol table

A search of an identifier proceeds first through the scope list, and for each header its list of object records is scanned. This mirrors the scope rule of the language and guarantees that if several entities carry the same identifier, the most local one is selected. The linear list of objects represents the simplest implementation by far. A tree structure would in many cases be more efficient for searching, and would therefore seem more recommendable. Experiments have shown, however, that the gain in speed is marginal. The reason is that the lists are typically quite short. The superiority of a tree structure becomes manifest only when a large number of global objects is declared. We emphasize that when a tree structure is used for each scope, the linear lists must still be present, because the order of declarations is sometimes relevant in interpretation, e.g. in parameter lists.

Not only procedures, but also record types establish their own local scope. The list of record fields is anchored in the type record's field *dsc*, and it is searched by procedure *thisField*. If a record type *R1* is an extension of *R0*, then *R1*'s field list contains only the fields of the extension proper. The base type *R0* is referenced by the *BaseTyp* field of *R1*. Hence, a search for a field may have to proceed through the field lists of an entire sequence of record base types.

12.6.2. Symbol files

The major part of module ORB is devoted to input and output of symbol files. A symbol file is a linearized form of an excerpt of the symbol table containing descriptions of all exported (marked) objects. All exports are declared in the global scope. Procedure *Export* traverses the list of global objects and outputs them to the symbol file.

The structure of a symbol file is defined by the syntax specified below. The following terminal symbols are class and form specifiers or reference numbers for basic types with fixed values:

Classes: Con = 1, Var = 2, Par = 3, Fld = 4; Typ = 5

Forms: Byte = 1, Bool = 2, Char = 3, Int = 4, LInt = 5, Set = 6,
Pointer = 7, NoTyp = 9, ProcTyp = 10, Array = 12, Record = 13

Syntax:

```
SymFile  = null key name versionkey {object}.
object   =  (CON name type (value | exno) | TYP name type [{fix} 0] | VAR name type expno).
type     =  ref (PTR type | ARR type len | REC type {field} 0 | PRO type {param} 0].
```

```
field  =    FLD name type offset.
param  =    (VAR | PAR) type.
```

A procedure type description is contains a parameter list. Similarly, a record type description with form specifier *Record* contains the list of field descriptions. Note that a procedure is considered as a constant of a procedure type.

Objects have types, and types are referenced by pointers. These cannot be written on a file. The straight-forward solution would be to use the type identifiers as they appear in the program to denote types. However, this would be rather crude and inefficient, and second, there are anonymous types, for which artificial identifiers would have to be generated.

An elegant solution lies in consecutively numbering types. Whenever a type is encountered the first time, it is assigned a unique *reference number.* For this purpose, records (in the compiler) of type *Type* contain the field *ref*. Following the number, a description of the type is then written to the symbol file. When the type is encountered again during the traversal of the data structure, only the reference number is issued, with negative sign. The global variable *ORB.Ref* functions as the running reference number.

When reading a symbol file, a positive reference number is followed by the type's description. A pointer to the type read is assigned to the global table *typtab* with the reference number as index. When a negative reference number is read (it is *not* followed by a type description), then the type is identified by *typtab[-ref]* (see procedure *InType*). In the following example, types are identified by their reference number (e.g. R #14), and later referenced by this number (^14).

```
MODULE A;
  CONST Ten* = 10; Dollar* = "$";
  TYPE R* = RECORD u*: INTEGER; v*: SET END ;
    S* = RECORD w*: ARRAY 4 OF R END ;
    P* = POINTER TO R;
    A* = ARRAY 8 OF INTEGER;
    B* = ARRAY 4, 5 OF REAL;
    C* = ARRAY 10 OF S;
    D* = ARRAY OF CHAR;
  VAR x*: INTEGER;
  PROCEDURE Q0*;
  BEGIN END Q0;
  PROCEDURE Q1*(x, y: INTEGER): INTEGER;
  BEGIN RETURN x+y END Q1;
END A.

class = CON Ten [^4]   10
class = CON Dollar [^3]   36
class = TYP R [#14  form = REC [^9]  exno = 1  extlev = 0  size = 8 { v [^6]   4 u [^4]   0}]()
class = TYP S [#15  form = REC [^9]  exno = 2  extlev = 0  size = 32 { w [#0  form = ARR [^14]  len = 4  size = 32]   0}]()
class = TYP P [#16  form = PTR [^14]]()
class = TYP A [#17  form = ARR [^4]  len = 8  size = 32]()
class = TYP B [#18  form = ARR [#0  form = ARR [^5]  len = 5  size = 20]  len = 4  size = 80]()
class = TYP C [#19  form = ARR [^15]  len = 10  size = 320]()
class = TYP D [#20  form = ARR [^3]  len = -1  size = 8]()
class = VAR x [^4]    3
class = CON Q0 [#0  form = PRO [^9]()]    4
class = CON Q1 [#0  form = PRO [^4]( class = VAR [^4] class = VAR [^4])]    5
```

After a symbol file has been generated, it is compared with the file from a previous compilation of the same module, if one exists. Only if the two files differ and if the compiler's s-option is enabled, is the old file replaced by the new version. The comparison is made by comparing byte after byte without consideration of the file's structure. This somewhat crude approach was chosen because of its simplicity and yielded good results in practice.

A symbol file must not contain addresses (of variables or procedures). If they did, most changes in the program would result in a change of the symbol file. This must be avoided, because changes in

the implementation (rather than the interface) of a module are supposed to remain invisible to the clients. Only changes in the interface are allowed to effect changes in the symbol file, requiring recompilation of all clients. Therefore, addresses are replaced by *export numbers*. The variable *exno* (global in *ORP*) serves as running number (see *ORP.Declarations* and *ORP.ProcedureDecl*). The translation from export number to address is performed by the loader. Every code file contains a list (table) of addresses (of variables and entry points for procedures). The export number serves as index in this table to obtain the requested address. Export numbers are generated by the parser.

Objects exported from some module *M1* may refer in their declaration to some other module *M0* imported by *M1*. It would be unacceptable, if an import of *M1* would then also require the import of *M0*, i.e. imply the automatic reading of *M01*'s symbol file. It would trigger a chain reaction of imports that must be avoided. Fortunately, such a chain reaction can be avoided by making symbol files *self-contained,* i.e. by including in every symbol file the description of entities that stem from other modules. Such entities are always types.

The inclusion of types imported from other modules seems simple enough to handle: type descriptions must include a reference to the module from which the type was imported. This reference is the name and key of the respective module. However, there exists one additional complication that cannot be ignored. Consider a module *M1* importing a variable *x* from a module *M0*. Let the type *T* of *x* be defined in module *M0*. Also, assume *M1* to contain a variable y of type *M0.T*. Evidently, *x* and *y* are of the same type, and the compiler compiling *M2* must recognize this fact. Hence, when importing *M0* during compilation of *M1*, the imported element *T* must not only be registered in the symbol table, but it must also be recognized as being identical to the *T* already imported from *M2* directly. It is rather fortunate that the language definition specifies equivalence of types on the basis of names rather than structure, because it allows type tests at execution time to be implemented by a simple address comparison.

The measures to be taken to satisfy the new requirements are as follows:

1. Every type element in a symbol file is given a module number. Before a type description is emitted to the file.
2. If a type to be exported has a name and stems from another, imported module, then also the name and key of the module are attached, from which the type stems (see end of procedure *ORB.OutType* and end of *ORB.InType*).

An additional detail is worth being mentioned here: Hidden pointers. We recall that individual fields of exported record types may be hidden. If marked (by an asterisk) they are exported and therefore visible in importing modules. If not marked, they are not exported and remain invisible, and evidently seem to be omissible in symbol files. However, this is a fallacy. They need to be included in symbol files, although without name, because of meta information to be provided for garbage collection. This is elucidated as follows:

Assume that a module M1 declares a global pointer variables of a type imported from module M0.

```
MODULE M0;
   TYPE Ptr = POINTER TO Rec0;
      Rec0* = RECORD p*, q: Ptr ... END ;
END M0.

MODULE M1;
   VAR p: M0.Ptr;
      r: RECORD f: M0.Ptr; ... END ;
END M1.
```

Here *p* and *r.f* are roots of data structures that must be visited by the garbage collector. If they are not, they will not be marked, and therefore collected with disastrous and entirely unpredictable consequences. The crux is that not only exported pointers (p.p) must be listed, but also hidden ones (p.q), although they are not accessible in module M1.

We chose to include hidden pointers in symbol files without their names, but with their type being of the form *ORB.NilTyp*. This must be considered in procedure ORG.FindPtrs, where the condition *typ.form = ORB.Pointer* must be extended to *(typ.form = ORB.Pointer) OR (typ.form = ORB.NilTyp).*

But the story does not end here. Assume that in the example above module M1 declares a type *Rec1* as a n extension of *M0.Rec0*. This requires the generation of a type descriptor. And this descriptor must include not only field *p*, but also the hidden field *q*. This is achieved by also extending the condition *typ.form = ORB.Pointer* in *ORG.FindPtrFlds* to *(typ.form = ORB.Pointer) OR (typ.form = ORB.NilTyp).*

12. 7. The code generator

The routines for generating instructions are contained in a single module: ORG. They are fairly numerous, and therefore the interface of ORG is quite large. It is a procedural interface. This implies that there is no "intermediate code" or "intermediate data structure" between parser and code generator. This is one reason for the compactness of the code generator. The other is the regularity and simplicity of the processor architecture. In order to understand the following material, the reader is supposed to be familiar with this architecture (Appendix 2) and the generated code patterns for individual language constructs (Section 12.2).

A distinguishing feature of this compiler is that parsing proceeds top-down according to the principle of recursive descent in the parsing tree. This implies that for every syntactic construct a specific procedure is called. It carries the same name as the construct. It also implies that properties of the parsed construct can be represented by parameters of the parsing procedures. Consider, for example, the construct of simple expression:

```
SimpleExpression  =  term {"+" term}.
```

The corresponding parsing procedure is

```
PROCEDURE SimpleExpression(VAR x: Item);
   VAR y: Item;
BEGIN term(x);
   WHILE sym = plus DO ORS.Get(sym); term(y); ORG.AddOp(x, y) END
END SimpleExpression
```

The generating procedure *AddOp* receives two parameters representing the operands, and returns the result through the first parameter. This scheme carries the invaluable advantage of using operands efficiently allocated on the stack rather than dynamically allocated on the heap and subject to automatic storage retrieval (garbage collection). Here the processed operands quietly disappear from the stack upon exit from the parser procedure.

The parameters representing syntactic constructs are of type *Item* defined in ORG. This data type is rather similar to the type *Object* (in ORB). After all, it serves the same purpose; but it represents internal items rather than declared objects.

```
TYPE Item = RECORD
      mode: INTEGER;
      type: ORB.Type;
      a, b, r: INTEGER;
      rdo: BOOLEAN  (*read only*)
END
```

The attribute *class* of *Object* is renamed *mode* in *Item*. In fact, in some sense different classes evoke different (corresponding) *addressing modes* as featured by the processor architecture. According to the architecture, additional modes may have to be introduced. Thanks to the simplicity of RISC, only three are needed:

```
Reg = 10;   The item x is located in  register x.r
RegI = 11;   The item x is addressed indirectly through register x.r plus offset x.a
Cond = 12;   The item is represented by the condition bit registers
```

Instructions are emitted sequentially and emitted by the four procedures *Put0, Put1, Put2, Put3*. They directly correspond to the instruction formats of the RISC processor (see Chapter 11). The instructions are stored in the array *code* and the compiler variable *pc* serves as running index.

```
PROCEDURE Put0(op, a, b, c: INTEGER);          format F0
PROCEDURE Put1(op, a, b, im: INTEGER);         format F1
PROCEDURE Put2(op, a, b, off: INTEGER);        format F2
PROCEDURE Put3(op, cond, off: INTEGER);        format F3
```

12.7.1. Expressions

Expressions consist of operands and operators. They are evaluated and have a value. First, a number of make-procedures transform objects into items (see Section 12.3.2). The principal one is *MakeItem*. Typical objects are variables (class, mode = *Var*). Global variables are addressed with base register SB (x.r = 13), local variables with the stack pointer SP (x.r = 14). VAR-parameters are addressed indirectly; the address is on the stack (class, mode = *Par, Ind*). *x.a* is the offset from the stack pointer.

Before an operator can be applied to operands, these must first be transferred (loaded) into registers. This is because the RISC performs operations only on registers. The loading is achieved by procedure *load* (and *loadAdr*) in ORG. The resulting mode is *Reg*. In allocating registers, a strict stack principle is used, starting with R0, up to R11. This is certainly not an optimal strategy and provides ample room for improvement (usually called optimization). The compiler variable RH indicates the next free register (top of register stack).

Base address SB is, as the name suggests, static. But this holds only within a module. It implies that on every transfer to a procedure in another module, the static base must be adjusted. The simplest way is to load SB before every external call, and to restore it to its old value after return from the procedure. We chose a different strategy: loading on demand (see below: global variables).

If a variable is indexed, has a field selector, is dereferenced, or has a type guard, this is detected in the parser by procedure *selector*. It calls generators *Index, Field, DeRef,* or *TypeTest* accordingly (see Section 12.3.2. and patterns 1 - 4 in Section 12.2). These procedures cause item modes to change as follows:

mode transition of x	instructions emitted	construct
1. Index(x, y) (y is loaded into y.r)		
Var --> RegI	ADD y.r, SP, y.r	array variable
Par --> RegI	LDR RH, SP, x.a ADD y.r, RH, y.r	array parameter
RegI --> RegI	ADD x.r, x.r, y.r	indexed array
2. Field(x, y) (y.mode = Fld, y.a = field offset)		
Var --> Var	none	field designator, add offset to x.a
RegI --> RegI	none	add field offset to x.a
Par --> Par	none	add field offset to x.b
3. DeRef(x)		
Var --> RegI	LDR RH, SP, x.a	dereferenced x^
Par --> RegI	LDR RH, SP, x.a LDR RH, RH, x.b	dereferenced parameter x^
RegI --> RegI	LDR x.r, x.r, x.a	

A fairly large number of procedures then deal with individual operators. Specifically, they are *Not, And1, And2, Or1, Or2* for Boolean operators, *Neg, AddOp, MulOp, DivOp* for operations on integers, *RealOp* for operations on real numbers, and *Singleton, Set, In, and SetOp* for operations

on Sets. And finally, following the same pattern, are the procedures for relations (comparisons) *IntRelation, SetRelation, RealRelation, StringRelation.* (see Appendix for listing of ORG). We note in particular that if all operands are constants, their evaluation is performed by the compiler and not delegated to run-time. This is an important efficiency factor.

12.7.2. Relations

RISC does not feature any compare instruction. Instead, subtraction is used, because an implicit comparison wth 0 is performed along with any arithmetic (or load) instruction. Instead of *x* < *y* we use *x-y* < 0. This is possible, because in addition to the computed difference deposited in a register, also the result of the comparison is deposited in the condition flags *N* (difference negative) and *Z* (difference zero). Relations therefore yield a result item *x* with mode *Cond*. *x.r (= relmap[sym])* identifies the relation. Branch instructions (jumps) are executed conditionally depending on these flags. The value *x.r* is then used when generating branch instructions. For example, the relation *x* < *y* is translated simply into

```
LDR  R0, SP, x
LDR  R1, SP, y
CMP  R0, R0, R1
```

and the resulting item mode is *x.mode = Cond, x.r := "less"*. (The mnemonic CMP is synonymous with SUB). More about relations and Boolean expressions will be explained in Section 12.7.6.

12.7.3. Set operations

The type SET represents sets of small integers in the range from 0 to 31. Bit *i* being 1 signals that *i* is an element of the set. This is a convenient representation, because the logical instructions directly mirror the set operations: AND implements set intersection, OR set union, and XOR the symmetric set difference. This representation also allows a simple and efficient implementation of membership tests. The instructions for the expression *n IN s* is generated by procedure *In*. Assuming the value *n* in register R0, and the set *s* in R1, we obtain

```
ADD  R0, R0, 1
ROR  R1, R1, R0     rotate s by i+1 position, the relevant bit moving to the sign bit
```

The resulting item mode is Cond with x.r = "minus".

Of some interest are the procedures for generating sets, i.e. for processing {*m*}, {*m .. n*}, and {*m, n*}, where *m, n* are integer expressions.

We start with {*m*}. It is generated by procedure *Singleton* using a shift instruction. Assuming *m* in R0, the resulting code is

```
MOV  R1, 0, 1
LSL  R0, R1, R0   shift 1 by m bit positions to the left
```

Somewhat more sophisticated is the generation of {*m .. n*} by procedure *Set*. Assuming *m* in R0, and *n* is R1, the resulting code is

```
MOV  R2, 0, -2
LSL  R1, R2, R1   shift -2 by n bit positions to the left
MOV  R2, 0, -1
LSL  R0, R2, R0   shift -1 by m bit positions to the left
XOR  R0, R0, R1
```

The set {*m, n*} is generated as the union of {*m*} and {*n*}. If any of the element values is a constant, several possibilities of code improvement are possible. For details, the reader is referred to the source code of ORG.

12.7.4. Assignments

Statements have an effect, but no result like expressions. Statements are executed, not evaluated. Assignments alter the value of variables through store instructions. The computation of the address of the affected variable follows the same scheme as for loading. The value to be assigned must be in a register.

Assignments of arrays (and records) are an exceptional case in so far as they are performed not by a single store instruction, but by a repetition. Consider y := x, where x, and y are both arrays of *n* integers. Assuming that the address of *y* is in register R0, that of *x* in R1, and the value *n* in R2. Then the resulting code is

```
L       LDR  R3, R1, 0   source
        ADD  R1, R1, 4
        STR  R3, R0, 0   destination
        ADD  R0, R0, 4
        SUB  R2, R2, 1   counter
        BNE  L
```

12.7.5. Conditional and repetitive statements

These statements are implemented using branch instructions (jumps) as shown in Section 12.2, Patterns 5 - 7. In all repetitive statements, backward jumps occur. Here, at the point of return the value of the global variable ORG.pc is saved in a local (!) variable of the involved parsing procedure. It is retrieved when the backward jump is emitted. We note that branch instructions use a displacement rather than an absolute destination address. It is the difference between the branch instruction and the destination of the jump.

A difficulty, however, arises in the case of forward jumps, a difficulty inherent in all single-pass compilers: When the branch is issued, its destination is still unknown. It follows that the branch displacement must be later inserted when it becomes known, when the destination is reached. This is called a *fixup*. Here the method of fixup lists is used. The place of the instruction with still unknown destination is held in a variable *L* local to the respective parsing procedure. If several branches have the same destination, L is the heading of a list of the instructions to be fixed up, with its links placed in the instructions themselves in the place of the eventual jump displacement. This shown for the if statement by an excerpt of *ORP.StatSequence* with local variable L0:

```
ELSIF sym = ORS.if THEN
  ORS.Get(sym); expression(x); ORG.CFJump(x);
  StatSequence; L0 := 0;
  WHILE sym = ORS.elsif DO
    ORS.Get(sym); ORG.FJump(L0); ORG.Fixup(x); expression(x);
    ORG.CFJump(x); Check(ORS.then, "no THEN"); StatSequence
  END ;
  IF sym = ORS.else THEN ORS.Get(sym); ORG.FJump(L0); ORG.Fixup(x); StatSequence
  ELSE ORG.Fixup(x)
  END ;
  ORG.FixLink(L0);
```

where in module ORG:

```
PROCEDURE CFJump(VAR x: Item);  (*conditional forward jump*)
BEGIN
  IF x.mode # Cond THEN loadCond(x) END ;
  Put3(BC, negated(x.r), x.a); FixLink(x.b); x.a := pc-1
END CFJump;

PROCEDURE FJump(VAR L: LONGINT);  (*unconditional forward jump*)
BEGIN Put3(BC, 7, L); L := pc-1
END FJump;

PROCEDURE fix(at, with: LONGINT);
BEGIN code[at] := code[at] DIV C24 * C24 + (with MOD C24)
END fix;
```

```
PROCEDURE FixLink(L: LONGINT);
   VAR L1: LONGINT;
BEGIN invalSB;
   WHILE L # 0 DO L1 := code[L] MOD 40000H; fix(L, pc-L-1); L := L1 END
END FixLink;

PROCEDURE Fixup(VAR x: Item);
BEGIN FixLink(x.a)
END Fixup;
```

In while-, repeat-, and for statements essentially the same technique is used with the support of the identical procedures in ORG.

12.7.6. Boolean expressions

In the case of arithmetic expressions, our compilation scheme results in a conversion from infix to postfix notation (x+y => xy+). This is not applicable for Boolean expressions, because the operators & and OR are defined as follows:

x & y --> if x then y else FALSE
x OR y -- > if x then TRUE else y

This entails that depending on the value of *x*, *y* must not be evaluated. As a consequence, jumps may have to be taken across the code for *y*. Therefore, the same technique of conditional evaluation must be used as for conditional statements. In the case of an expression *x & y (x OR y),* procedure ORG.And1 resp. ORG.Or1 must be called just after parsing *x* (see ORP.term resp. ORP.SimpleExpression). Only after parsing also *y* can the generators ORG.And2 resp. ORG(Or2) be called, providing the necessary fixups of forward jumps.

```
PROCEDURE And1(VAR x: Item);   (* x := x & *)
BEGIN
   IF x.mode # Cond THEN loadCond(x) END ;
   Put3(BC, negated(x.r), x.a); x.a := pc-1; FixLink(x.b); x.b := 0
END And1;

PROCEDURE And2(VAR x, y: Item);
BEGIN
   IF y.mode # Cond THEN loadCond(y) END ;
   x.a := merged(y.a, x.a); x.b := y.b; x.r := y.r
END And2;
```

A negative consequence of this scheme having condition flags in the processor is that when an item with mode *Cond* has to be transferred into mode *Reg*, as in a Boolean assignment, an unpleasantly complex instruction sequence must be generated. Fortunately, this case occurs quite rarely.

12.7.7. Procedures

Before embarking on an explanation of procedure calls, entries and exits, we need to know how recursion is handled and how storage for local variables is allocated. Procedure calls cause a sequence of frames to be allocated in a stack fashion. These frames are the storage space for local variables. Each frame is headed by a single word containing the return address of the call. This address is deposited in R15 by the call instructions (BL, branch and link). The compiler "knows" the size of the frame to be allocated, and thus merely decrements the stack pointer SP (R14) by this amount. Upon return, SP is incremented by the same amount, and PC is restored by a branch instruction. In the following example, a procedure P is called, calling itself Q, and Q calling P again (recursion). The stack then contains 3 frames (see Figure 12.7).

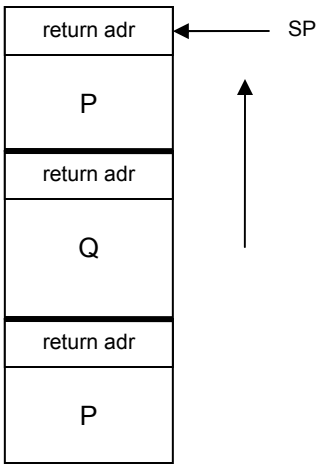


Figure 12.7 Stack frames

Scheme and layout determine the code sequences for call, entry and exit of procedures. Here is an example of a procedure *P* with 2 parameters:

```
Call:     LDR  R0, param0
          LDR  R1, param1
          BL   P

Prolog:   SUB  SP, SP, size      decrement SP
          STR  LNK, SP, 0        push return adr
          STR  R0, SP, 4         push parameter 0
          STR  R1, SP, 8         push parameter1 ....

Epilog:   LDR  LNK, SP, 0        pop return adr
          ADD  SP, SP, size      increment SP
          BR  LNK
```

When the call instruction is executed, parameters reside in registers, starting with R0. For function procedures, the result is passed in register R0. This scheme is very efficient; storing the parameters occurs only in a single place, namely at procedure entry, and not before each call. However, it has severe consequences for the entire register allocation strategy. Throughout the compiler, registers *must* be allocated in strict stack fashion. Furthermore, parameter allocation *must* start with R0. This is a distinct drawback for function calls. If registers are occupied by other values loaded prior to the call, they must be cleared, i.e. the parameters must be saved and reloaded after return. This is rather cumbersome (see procedures *ORG.SaveRegisters* and *ORG.RestoreRegisters*).

```
F(x)              no register saving
x + F(x)
F(F(x))
(x+1) + F(x)      register saving necessary
```

12.7.8. Type extension

Static typing is an important principle in programming languages. It implies that every constant, variable or function is of a certain data type, and that this type can be derived by reading the program text without executing it. It is the key principle to introduce important redundancy in languages in such a form that a compiler can detect inconsistencies. It is therefore the key element for reducing the number of errors in programs.

However, it also acts as a restriction. It is, for example, impossible to construct data structures (arrays, trees) with different types of elements. In order to relax the rule of strictly static typing, the notion of *type extension* was introduced in Oberon. It makes it possible to construct

inhomogeneous data structures without abandoning type safety. The price is that the checking of type consistency must in certain instances be deferred to run-time. Such checks are called *type tests*. The challenge is to defer to run-time as few checks as possible and as many as needed.

The solution in Oberon is to introduce families of types, and compatibility among their members. Their members are thus related, and a family forms a hierarchy. The principle idea is the following: Any record type T0 can be extended into a new type T1 by additional record fields (attributes). T1 is then called an *extension* of T0, which in turn is said to be T1's *base type*. T1 is then type compatible with T0, but not vice-versa. This property ensures that in many cases static type checking is still possible. Furthermore, it turns out that run-time tests can be made very efficient, thus minimizing the overhead for maintaining type safety.

For example, given the declarations

```
TYPE R0 = RECORD u, v: INTEGER END ;
     R1 = RECORD (R0) w: INTEGER END
```

we say that R1 is an *extension* of R0. R0 has the fields u and v, R1 has u, v, and w. The concept becomes useful in combination with pointers. Let

```
TYPE P0 = POINTER TO R0;
     P1 = POINTER TO R1;
VAR p0: P0;  p1: P1;
```

Now it is possible to assign p1 to p0 (because a P1 is always also a P0), but not p0 to p1, because a P0 need not be a P1. This has the simple consequence that a variable of type P0 may well point to an extension of R0. Therefore, data structures can be declared with a base type, say P0, as common element type, but in fact they can individually differ, they can be any extension of the base type.

Obviously, it must be possible to determine the actual, current type of an element even if the base type is statically fixed. This is possible through a *type test*, syntactically a Boolean factor:

```
p0 IS P1                          (short for p0^ IS R1)
```

Furthermore, we introduce the *type guard*. In the present example, the designator p0.w is illegal, because there is no field *w* in a record of type R0, even if the current value of p0^ is a R1. As this case occurs frequently, we introduce the short notation *p0(P1).w*, implying a test *p0 IS P1* and an abort if the test is not met.

It is important to mention that this technique also applies to formal variable parameters of record type, as they also represent a pointer to the actual parameter. Its type may be any extension of the type specified for the formal parameter in the procedure heading.

How are type test and type guard efficiently implemented? Our first observation is that they must consist of a single comparison only, similar to index checks. This in turn implies that types must be identified by a single word. The solution lies in using the unique address of the *type descriptor* of the (record) type. Which data must this descriptor hold? Essentially, type descriptors (TD) must identify the base types of a given type. Consider the following hierarchy:

```
TYPE T =  RECORD … END ;
     T0 =   RECORD (T) … END ;          extension level 1
     T1 =   RECORD (T) … END ;          extension level 1
     T00 =  RECORD (T0) … END ;         extension level 2
     T01 =  RECORD (T0) … END ;         extension level 2
     T10 =  RECORD (T1) … END ;         extension level 2
     T11 =  RECORD (T1) … END ;         extension level 2
```

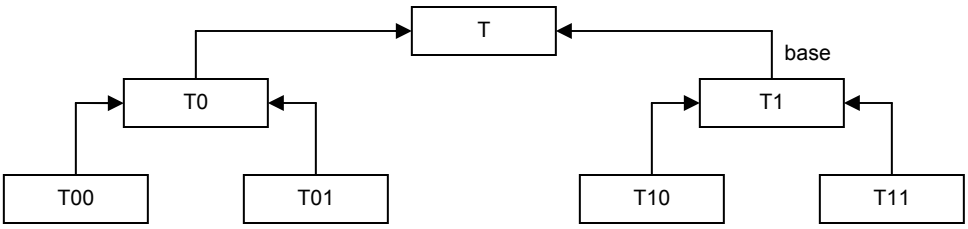


Figure 12.8. A type hierarchy

In the symbol table, the field *base* refers to the ancestor of a given record type. Thus base of the type representing T11 points to T1, etc. Run-time checks, however, must be fast, and hence cannot proceed through chains of pointers. Instead, each TD contains an array with references to the ancestor TDs (including itself). For the example above, the TDs are as follows:

```
TD(T)  =   [T]
TD(T0) =   [T, T0]
TD(T1) =   [T, T1]
TD(T00) =  [T, T0, T00]
TD(T01) =  [T, T0, T01]
TD(T10) =  [T, T1, T10]
TD(T11) =  [T, T1, T11]
```

Evidently, the first element can be omitted, as it always refers to the common base of the type hierarchy. The last element always points to the TD's owner. TDs are allocated in the data area, the area for variables.

References to TDs are called *type tags*. They are required in two cases. The first is for records referenced by pointers. Such dynamically allocated records carry an additional, hidden field holding their type tag. (A second additional word is reserved for use by the garbage collector. The offset of the tag field is therefore -8). The second case is that of record-typed VAR-parameters. In this case the type tag is explicitly passed along with the address of the actual parameter. Such parameters therefore require two words/registers.

A type test then consists of a test for equality of two type tags. In *p IS T* the first tag is that of the n'th entry of the TD of *p*^, where n is the extension level of *T*. The second tag is that of type *T*. This is shown in Pattern13 in Section 12.2 (see also Fig. 12.4). The test then is as follows:

```
p^.tag^[n] = adr(T),  where n is the extension level of T
```

When declaring a record type, it is not known how many extensions, nor how many levels will be built on this type. Therefore TD's should actually be infinite arrays. We decided to restrict them to 3 levels only. The first entry, which is never used for checking, is replaced by the size of the record.

12.7.9. Import and export, global variables

Addresses of imported objects are not available to the compiler. Their computation must be left to the module loader (see Chapter 6). Similar to handling addresses of forward jumps, the compiler puts the necessary information in place of the actual address into the instruction itself. In the case of procedure calls, this is quite feasible, because the BL instruction features an offset field of 24 bits. The information consists of the module number and the export number of the imported object. In addition, there is a link to the previous instruction referring to an imported procedure. The origin of the list of procedure call fixups is rooted in the compiler variable *fixorgP*, and of the 24 bits in each BL instruction 4 bits are used for the module number, 8 bits for the object's export number, and 12 for the link. The loader need only scan this list to fix up the addresses (jump offsets).

Matters are more complex in the case of data. Object records in the symbol table have a field *lev*. It indicates the nesting level of variables local to procedures. It is also used for the module number in the case of variables of imported modules. Note that when importing, objects designating modules are inserted in the symbol table, and the list of their own objects are attached in the field *dsc*. In this

latter case, the module numbers have an inverted sign (are negative). Such imported objects are static, i.e. have a fixed address. In principle their absolute address could be computed (fixed) by the module loader. However, this is not practicable, because RISC instructions have an address offset of 16 bits only. It is therefore necessary in the general case to use a base address in conjunction with the offset. We use a single register for holding the *static base* (SB, R13). This register need be reloaded for every access to an imported variable. However, the compiler keeps track of external accesses; if a variable is to be accessed from the same module as the previous case, then reloading is avoided (see procedure *GetSB* and global compiler variable *curSB*).

This base address is fetched from a table global to the entire system. This module table contains one entry for every module loaded, namely the address of the module's data section. The address of the table is permanently in register MT (= R12). An access to an imported variable therefore always requires two instructions:

```
LDR  SB, MT, modno*4    base address of data section
LDR  R0, SB, offset     offset computed by the loader from object's export number
```

Considering the fact that references to external variables are (or should be) rare, this circumstance is of no great concern. (Note also that such accesses are read-only). More severe is the fact that we also treat global variables contained in the same module by the same technique. Their level number is 0. One might use a specific base register for the base of the current module. Its content would then have to be reloaded upon every procedure call and after every return. This is common technique, but we have here chosen to reload only when necessary, i.e. only when an access is at hand. This strategy rewards the programmer who sensibly uses global variables rarely.

12.7.10. Traps

This compiler provides an extensive system of safeguard by providing run-time checks (aborts) in several cases:

trap number	trap cause
1	array index out of range
2	type guard failure
3	array or string copy overflow
4	access via NIL pointer
5	illegal procedure call
6	integer division by zero
7	assertion violated

These checks are implemented very efficiently in order not to downgrade a program's performance. Involved is typically a single compare instruction, plus a conditional branch (BLR MT). It is assumed that entry 0 of the module table contain not a base address (module numbers start with 1), but a branch instruction to an appropriate trap routine. The trap number is encoded in bits 4:7 of the branch instruction.

The predefined procedure *Assert* generates a conditional trap with trap number 7. For example, the statement *Assert(m = n)* generates

```
LDR    R0, m
LDR    R1, n
CMP    R0, R0, R1
BLR    1, 7CH        branch and link if unequal through R12, trap number 7
```

Procedure *New*, representing the operator *NEW*, has been implemented with the aid of the trap mechanism. (This is in order to omit in ORG any reference to module *Kernel*, which contains the allocation procedure *New*). The generated code for the statement NEW(p) is

```
ADD    R0, SP, p     address of p
ADD    R1,SB,  tag   type tag
BLR    7, 0CH        branch and link unconditionally through R12 (MT), trap number 0
```

13 A graphics editor

13.1. History and goal

The origin of graphics systems as they are in use at this time was intimately tied to the advent of the high-resolution bit-mapped display and of the mouse as pointing device. The author's first contact with such equipment dates back to 1976. The Alto computer at the Xerox Palo Alto Research Center is justly termed the first workstation featuring those characteristics. The designer of its first graphics package was Ch. Thacker who perceived the usefulness of the high-resolution screen for drawing and processing schematics of electronic circuits. This system was cleverly tailored to the needs encountered in this activity, and it was remarkable in its compactness and effectiveness due to the lack of unnecessary facilities. Indeed, its acronym was SIL, for Simple ILlustrator.

After careful study of the used techniques, the author designed a variant, programmed in Modula-2 (instead of BCPL) for the PDP-11 Computer, thereby ordering and exhibiting the involved data structures more explicitly. In intervals of about two years, that system was revised and grew gradually into the present Draw system. The general goal remained a simple line drawing system: emphasis was placed on a clear structure and increase of flexibility through generalization of existing rather than indiscriminate addition of new features.

In the history of this evolution, three major transitions can be observed. The first was the move from a single "window", the screen, to multiple windows including windows showing different excerpts of the same graphic. This step was performed on the Lilith computer which resembled the Alto in many ways. The second major transition was the application of the object-oriented style of programming, which allowed the addition of new element types to the basic system, making it extensible. The third step concerned the proper integration of the Draw system with Oberon's text system. The last two steps were performed using Oberon and the Ceres computer.

We refrain from exhibiting this evolution and merely present the outcome, although the history might be an interesting reflection of the evolution of programming techniques in general, containing many useful lessons. We stress the fact, however, that the present system rests on a long history of development, during which many features and techniques were introduced and later discarded or revised. The size of the system's description is a poor measure of the effort that went into its construction; deletion of program text sometimes marks bigger progress than addition.

The goal of the original SIL program was to support the design of electronic circuit diagrams. Primarily, SIL was a line drawing system. This implies that the drawings remain uninterpreted. However, in a properly integrated system, the addition of modules containing operators that interpret the drawings is a reasonably straight-forward proposition. In fact, the Oberon system is ideally suited for such steps, particularly due to its command facility.

At first, we shall ignore features specially tailored to circuit design. The primary one is a macro facility to be discussed in a later chapter.

The basic system consists of the modules *Draw, GraphicFrames,* and *Graphics*. These modules contain the facilities to generate and handle horizontal and vertical lines, text captions, and macros. Additional modules serve to introduce other elements, such as rectangles and circles, and the system is extensible, i.e. further modules may be introduced to handle further types of elements.

13.2. A brief guide to Oberon's line drawing system

In order to provide the necessary background for the subsequent description of the Draw system's implementation, a brief overview is provided in the style of a user's guide. It summarizes the facilities offered by the system and gives an impression of its versatility.

The system called *Draw* serves to prepare line drawings. They contain lines, text captions, and other items, and are displayed in graphic viewers (more precisely: in menu viewers' graphic frames). A

graphic viewer shows an excerpt of the drawing plane, and several viewers may show different parts of a drawing. The most frequently used commands are built-in as mouse clicks and combinations of clicks. Additional commands are selectable from texts, either in viewer menus (title bars) or in the text called *Draw.Tool*. Fig. 13.1. shows the display with two graphic viewers at the left and the draw tool text at the right. The mouse buttons have the following principal functions whenever the cursor lies in a graphic frame:

left:	draw / set caret
middle:	move / copy
right:	select

A mouse command is identified (1) by the key k0 pressed initially, (2) by the initial position P0 of the cursor, (3) by the set of pressed keys k1 until the last one is released, and (4) the cursor position P1 at the time of release.

13.2.1. Basic commands

The command *Draw.Open* opens a new viewer and displays the graph with the name given as parameter. We suggest that file names use the extension *Graph*.

Drawing a line. In order to draw a horizontal or vertical line from P0 to P1, the left key is pressed with the cursor at P0 and, while the key is held, the mouse and cursor is moved to P1. Then the key is released. If P0 and P1 differ in both their x and y coordinates, the end point is adjusted so that the line is either horizontal or vertical.

Writing a caption. First the cursor is positioned where the caption is to appear. Then the left key is clicked, causing a crosshair to appear. It is called the *caret*. Then the text is typed. Only single lines of texts are accepted. The DEL key may be used to retract characters (backspace).

Selecting. Most commands require the specification of operands, and many implicitly assume the previously selected elements - the *selection* - to be their operands. A single element is selected by pointing at it with the cursor and then clicking the right mouse button. This also causes previously selected elements to be deselected. If the left key is also clicked, their selection is retained. This action is called an *interclick*. To select several elements at once, the cursor is moved from *P0* to *P1* while the right key is held. Then all elements lying within the rectangle with diagonally opposite corners at *P0* and *P1* are selected. Selected lines are displayed as dotted lines, selected captions (and macros) by inverse video mode. A macro is selected by pointing at its lower left corner. The corner is called *sensitive area*.

Moving. To move (displace) a set of elements, the elements are first selected and then the cursor is moved from P0 to P1 while the middle key is held. The vector from *P0* to *P1* specifies the movement and is called the *displacement vector*. *P0* and *P1* may lie in different viewers displaying the same graph. Small displacements may be achieved by using the keyboard's cursor keys.

Copying. Similarly, the selected elements may be copied (duplicated). In addition to pressing the middle key while indicating the displacement vector, the left key is interclicked. The copy command may also be used to copy elements from one graph into another graph by moving the cursor from one viewer into another viewer displaying the destination graph. A text caption may be copied from a text frame into a graphic frame and vice-versa. There exist two ways to accomplish this: 1. First the caret is placed at the destination position, then the text is selected and the middle key is interclicked. 2. First the text is selected, then the caret is placed at the destination position and the middle key is interclicked.

Shifting the plane. You may shift the entire drawing plane behind the viewer by specifying a displacement vector pressing the middle button (like in a move command) and interclicking the right button.

The following table shows a summary of the mouse actions:

left	draw line
left (no motion)	set caret

left + middle	copy selected caption to caret
left + right	set secondary caret
middle	move selection
middle + left	copy selection
middle + right	shift drawing plane
right	select area
right (no motion)	select object
right + middle	copy caption to caret
right + left	select without deselection

13.2.2. Menu commands

The following commands are displayed in the menu (title bar) of every graphic viewer. They are activated by being pointed at and by clicking the middle button.

Draw.Delete	The selected elements are deleted.
Draw.Store	The drawing is written as file with the name shown in the title bar. The original file is renamed by appending ".Bak".
Draw.Restore	The entire frame is redrawn
Draw.Ticks	The frame displays a pattern of dots (ticks) to facilitate positioning.

The two viewers in Fig. 13.1. display different parts of the same graphic. The second view was obtained from the generic *System.Copy* command and a subsequent shift of the drawing plane.

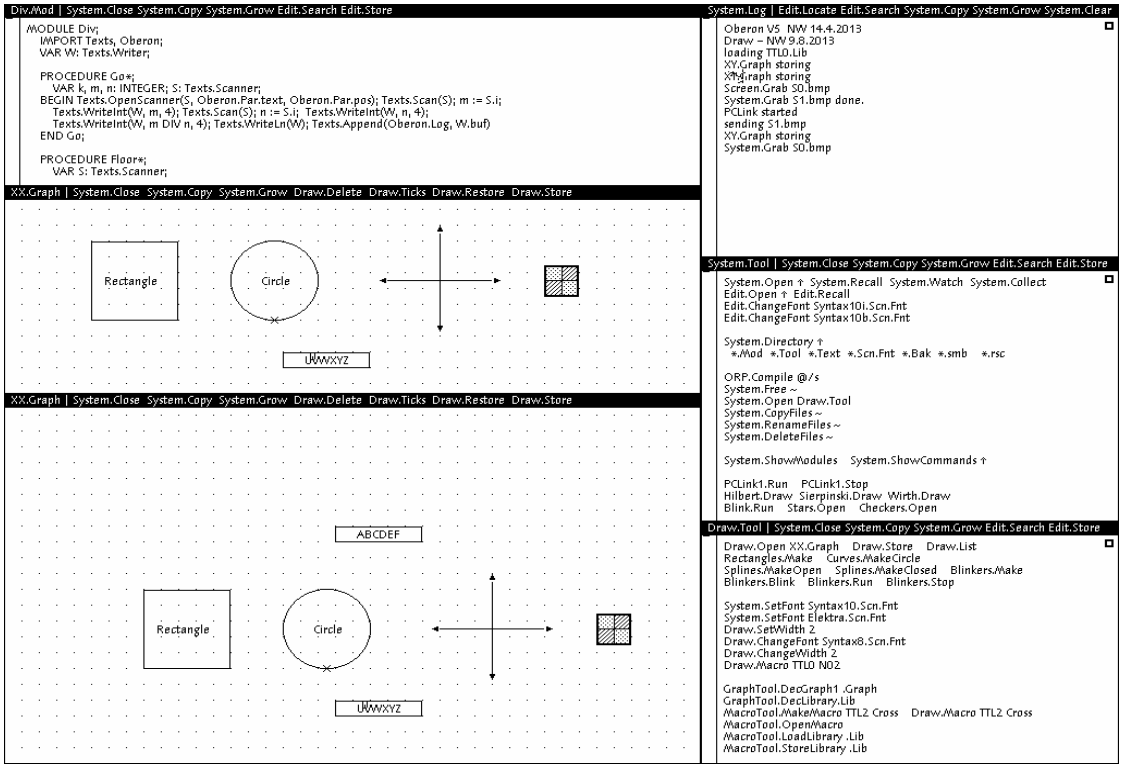


Figure 13.1 Display with graphics duplicated viewers

13.2.3. Further commands

The following commands are listed in the text *Draw.Tool*, but may appear in any text.

Draw.Store *name*	The drawing in the marked viewer is stored as a file with the specified name.

The subsequent commands change attributes of drawing elements, such as line width, text font, and color, and they apply to the most recent selection.

Draw.SetWidth *w*	default = 1, 0 < *w* < 7.
Draw.ChangeFont *fontname*	
Draw.ChangeColor *c*	
Draw.ChangeWidth *w*	(0 < *w* < 7)

The *ChangeColor* command either take a color number in the range 1 .. 15 or a string as parameter. It serves to copy the color from the selected character.

13.2.4. Macros

A macro is a (small) drawing that can be identified as a whole and be used as an element within a (larger) drawing. Macros are typically stored in collections called *libraries*, from where they can be selected and copied individually.

Draw.Macro *lib mac* The macro *mac* is selected from the library named *lib* and inserted in the drawing at the caret's position.

An example for the use of macros is drawing electronic circuit diagrams. The basic library file containing frequently used TTL components is called *TTL0.Lib*, and a drawing showing its elements is called *TTL0.Graph* (see Figure 13.2).

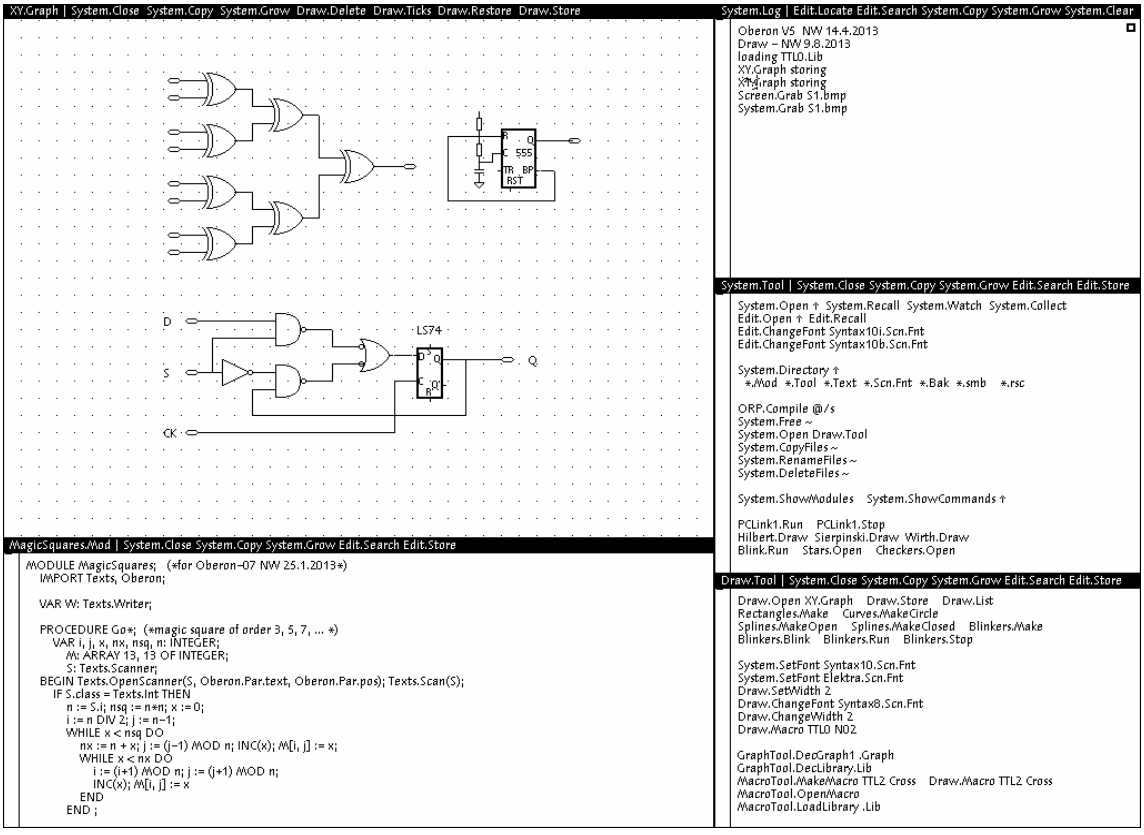


Figure 13.2 Viewer with circuit macros of TTL0 library

13.2.5. Rectangles

Rectangles can be created as individual elements. They are frequently used for framing sets of elements. Rectangles consist of four lines which are selectable as a unit. The attribute commands *Draw.SetWidth, System.SetColor, Draw.ChangeWidth,* and *Draw.ChangeColor* also apply to

rectangles. Rectangles are selected by pointing at their lower left corner and are created by the following steps:

1. The caret is placed where the lower left corner of the new rectangle is to lie.
2. A secondary caret is placed where the opposite corner is to lie (ML + MR).
3. The command *Rectangles.Make* is activated.

13.2.6. Oblique lines, circles, and ellipses

Further graphic elements are (oblique) lines, circles, and ellipses. The sensitive area of circles and ellipses is at their lowest point. They are created by the following steps:

Lines:
1. The caret is placed where the starting point is to lie.
2. A secondary caret is placed at the position of the end.
3. The command *Curves.MakeLine* is activated.

Circles:
1. The caret is placed where the circle's center is to lie.
2. A secondary caret is placed, its distance from the center specifying the radius.
3.The command *Curves.MakeCircle* is activated.

Ellipses:
1. The caret is placed where the center is to lie.
2. A second caret is placed. Its horizontal distance from the first caret specifies one axis.
3. A third caret is placed. Its vertical distance from the first caret specifies the other axis.
4. The command *Curves.MakeEllipse* is activated.

13.2.7. Spline curves

Spline curves are created by the following steps:

1. The caret is placed where the starting point is to lie.
2. Secondary carets are placed at the spline's fixed points (at most 20).
3. The command *Splines.MakeOpen* or *Splines.MakeClosed* is activated.

13.2.8. Constructing new macros

A new macro is constructed and inserted in the library lib under the name mac as follows:

1. All elements which belong to the new macro are selected.
2. The caret is placed at the lower left corner of the area to be spanned by the macro.
3. A secondary caret is placed at the upper right corner of the area to be spanned.
4. The command *MacroTool.MakeMacro lib mac* is activated.

An existing macro can be decomposed (opened) into its parts as follows:

1. The macro is selected.
2. The caret is placed at the position where the decomposition is to appear.
3. The command *MacroTool.OpenMacro* is activated.

The command *MacroTool.StoreLibrary lib file* stores the library *lib* on the specified file. Only the macros presently loaded are considered as belonging to the library. If one wishes to add some macros to an existing library file, all of its elements must first be loaded. This is best done by opening a graph containing all macros of the desired library file.

13.3. The core and its structure

Like a text, a graphic consists of elements, subsequently to be called *objects*. Unlike a text, which is a sequence of elements, a graphic is an unordered set of objects. In a text, the position of an element need not be explicitly indicated (stored); it is recomputed from the position of its predecessor each time it is needed, for example for displaying or selecting an element. In a graphic, each object must carry its position explicitly, as it is independent of any other object in the set. This is an essential difference, requiring a different treatment and much more storage space for an equal number of objects.

Although this is an important consideration in the choice of a representation of a data structure, the primary determinants are the kind of objects to be included, and the set of operations to be applied to them. Here SIL set a sensible starting point. To begin with, there exist only two kinds of objects, namely straight, horizontal and vertical lines, and short texts for labelling lines, called *captions*. It is surprising how many useful task can be fulfilled with only these two types of objects.

The typical operations to be performed on objects are creating, drawing, moving, copying, and erasing. Those performed on a graphic are inserting, searching, and deleting an object. For the operations on objects, data indicating an object's position (and possibly color), its length and width in the case of lines, and the character string in the case of captions suffice. For the operations on the graphic, some data structure representing the set of objects must be chosen. Without question, a dynamic structure is most appropriate, and it requires the addition of some linking fields to the record representing an object. Without further deliberation, and with the idea that graphics to be handled with this system contain hundreds rather than tens of thousands of objects, we chose the simplest solution, the linear list. A proper modularization in connection with information hiding will make it possible to alter this choice without affecting client modules.

Although in general the nature of a user interface should not influence the representation chosen for the abstract data structure, we need to take note of the manner in which parameters of certain operations are denoted. It is, for example, customary in interactive graphics systems to select the objects to which an operation is to apply *before* invoking that operation. Their *selection* is reflected in their visual appearance in some way, and gives the user an opportunity to verify the selection (and to change it, if necessary) before applying the operation (such as deletion). For an object to be selectable means that it must record a state (selected/unselected). We note that it is important that this state is reflected by visual appearance.

As a consequence, the property *selected* is added to every object record. We now specify the data types representing lines and captions as follows and note that both types must be extensions of the same base type in order to be members of one and the same data structure.

```
TYPE Object =     POINTER TO ObjectDesc;
  ObjectDesc =    RECORD
                    x, y, w, h, col: INTEGER;
                    selected: BOOLEAN;
                    next: Object
                  END ;

  Line =          POINTER TO LineDesc;
  LineDesc =      RECORD (Object) END ;

  Caption =       POINTER TO CaptionDesc
  CaptionDesc =   RECORD (Object)
                    pos, len: INTEGER
                  END
```

Selection of a single element is typically achieved by pointing at the object with mouse and cursor. Selection of a set of objects is achieved by specifying a rectangular area, implying selection of all objects lying within it. In both cases, the search for selected elements proceeds through the linked list and relies on the position and size stored in each object's descriptor. As a consequence, the rule was adopted that every object not only specify a position through its coordinates *x, y*, but also the rectangular area within which it lies (width *w*, height *h*). It is thus easy to determine whether a given point identifies an object, as well as whether an object lies fully within a rectangular area.

In principle, each caption descriptor carries the sequence of characters (string) representing the caption. The simplest realization would be an array structured field, limiting the length of captions to some fixed, predetermined value. First, this is highly undesirable (although used in early versions of the system). And second, texts carry attributes (color, font). It is therefore best to use a global "scratch text", and to record a caption by the position and length of the string in this immutable text.

A procedure *drawGraphic* to draw all objects of a graphic now assumes the following form:

```
PROCEDURE drawObj(obj: Object);
BEGIN
    IF obj IS Line THEN drawLine(obj(Line))
    ELSIF obj IS Caption THEN drawCaption(obj(Caption))
    ELSE (*other object types, if any*)
    END
END drawObj;

PROCEDRE drawGraphic(first: Object);
    VAR obj: Object;
BEGIN obj := first;
    WHILE obj # NIL DO drawObj(obj); obj := obj.next END
END drawGraphic
```

The two procedures typically are placed in different modules, one containing operations on objects, the other those on graphics. Here the former is the service module, the latter the former's client. Procedures for, e.g, copying elements, or determining whether an object is selectable, follow the same pattern as *drawGraphic*.

This solution has the unpleasant property that all object types are anchored in the base module. If any new types are to be added, the base module has to be modified (and all clients are to be - at least - recompiled). The object-oriented paradigm eliminates this difficulty by inverting the roles of the two modules. It rests on binding the operations pertaining to an object type to each object individually in the form of procedure-typed record fields as shown in the following sample declaration:

```
ObjectDesc =  RECORD
                  x, y, w, h, col: INTEGER; selected: BOOLEAN;
                  draw: PROCEDURE (obj: Object);
                  write:  PROCEDURE (obj: Object; VAR R: Files.Rider);
                  next: Object
              END
```

The procedure *drawGraphic* is now formulated as follows:

```
PROCEDURE drawGraphic(first: Object);
    VAR obj : Object;
BEGIN obj := first;
    WHILE obj 9 NIL DO obj.draw(obj);  obj := obj.next END
END drawGraphic;
```

The individual procedures - in object-oriented terminology called *methods* - are assigned to the record's fields upon its creation. They need no further discrimination of types, as this role is assumed by the assignment of the procedures upon their installation. We note here that the procedure fields are never changed; they assume the role of *constants* rather than variables associated with each object.

This example exhibits in a nutshell the essence of object-oriented programming, *extensibility* as its purpose and the *procedure-typed* record field as the technique.

The given solution, as it stands, has the drawback that each object (instance, variable) contains several procedures (of which three are listed), and therefore leads to a storage requirement that should be avoided. Furthermore, it defines once and for all the number of operations applicable to objects, and also their parameters and result types. A different approach with the same underlying principle removes these drawbacks. It employs a single installed procedure which itself discriminates among the operations according to different types of parameters. The parameters of the preceding solution are merged into a single record called a *message*. The unified procedure is called a *handler*, and messages are typically extensions of a single base type *Msg*.

```
TYPE Msg =        RECORD END;
    DrawMsg =     RECORD (Msg) END;
    WriteMsg =    RECORD (Msg) R: Files.Rider END ;

    ObjectDesc =  RECORD
                      x, y, w, h, col: INTEGER; selected: BOOLEAN;
```

```
                    handle: PROCEDURE (obj: Object; VAR M: Msg);
                    next: Object
                END ;

PROCEDURE  Handler (obj: Object; VAR M: Msg);
    (*this procedure is assigned to the handle field of every line object*)
BEGIN
    IF M IS DrawMsg THEN drawLine(obj(Line))
    ELSIF M IS WriteMsg THEN writeLine(obj(Line), M(WriteMsg).R)
    ELSE ...
    END
END ;

PROCEDURE drawGraphic(first: Objec; VAR M: Msg);
    VAR obj: Object;
BEGIN obj := first;
    WHILE obj 9 NIL DO obj.handle(obj, M);  obj := obj.next END
END drawGraphics
```

In the present system, a combination of the two schemes presented so far is used. It eliminates the need for individual method fields in each object record as well as the cascaded IF statement for discriminating among the message types. Yet it allows further addition of new methods for later extensions without the need to change the object's declaration. The technique used is to include a single field (called *do*) in each record (analogous to the handler). This field is a pointer to a method record containing the procedures declared for the base type. At least one of them uses a message parameter, i.e. a parameter of record structure that is extensible.

```
TYPE Method =       POINTER TO MethodDesc;
     Msg =          RECORD END;
     Context =      RECORD END;

     Object =       POINTER TO ObjectDesc;
     ObjectDesc =   RECORD
                        x, y, w, h, col: INTEGER; selected: BOOLEAN;
                        do: Method; next: Object
                    END;

MethodDesc =        RECORD
                        new: Modules.Command;
                        copy: PROCEDURE (obj, to: Object);
                        draw, handle: PROCEDURE (obj: Object; VAR M: Msg);
                        selectable: PROCEDURE (obj: Object; x, y: INTEGER): BOOLEAN;
                        read: PROCEDURE (obj: Object; VAR R: Files.Rider; VAR C: Context);
                        write: PROCEDURE (obj: Object; cno: INTEGER;
                            VAR R: Files.Rider; VAR C: Context);
                    END
```

A single method instance is generated when a new object type is created, typically in the initialization sequence of the concerned module. When a new object is created, a pointer to this record is assigned to the do field of the new object descriptor. A call then has the form *obj.do.write(obj, R).* This example exhibits the versatility of Oberon's type extension and procedure variable features very well, and it does so without hiding the data structures involved in a dispensible, built-in run-time mechanism.

The foregoing deliberations suggest the system's modular structure shown in Figure 13.3.:

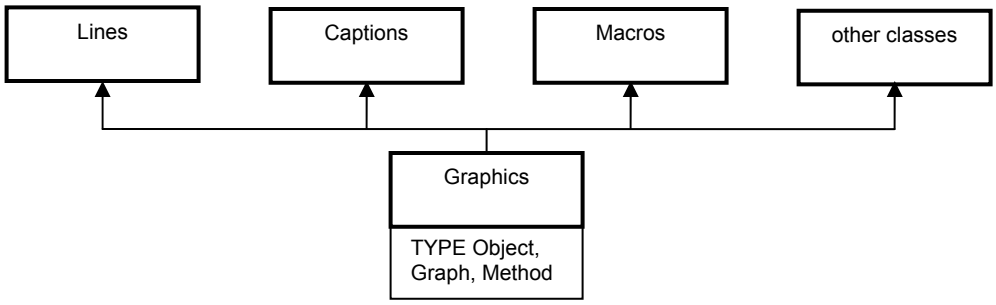


Figure 13.3 Clients of module *Graphics*

The modules in the top row implement the individual object types' methods, and additionally provide commands, in particular *Make* for creating new objects. The base module specifies the base types and procedures operating on graphics as a whole.

Our system, however, deviates from this scheme somewhat for several reasons:

1. Implementation of the few methods requires relatively short programs for the basic objects. Although a sensible modularization is desirable, we wish to avoid an atomization, and therefore merge parts that would result in tiny modules with the base module.

2. The elements of a graphic refer to fonts used in captions and to libraries used in macros. The writing and reading procedures therefore carry a context consisting of fonts and libraries as an additional parameter. Routines for mapping a font (library) to a number according to a given context on output, and a number to a font (library) on input are contained in module *Graphics*.

3. In the design of the Oberon System, a hierarchy of four modules has proven to be most appropriate:

   0. Module with base type handling the abstract data structure.
   1. Module containing procedures for the representation of objects in frames (display handling).
   2. Module containing the primary command interpreter and connecting frames with a viewer.
   3. A command module scanning command lines and invoking the appropriate interpreters.

The module hierarchy of the Graphics System is here shown together with its analogy, with the Text System:

Function	Graphics	Text
3. Command Scanner	Draw	Edit
2. Viewer Handler	MenuViewers	MenuViewers
1. Frame Handler	GraphicFrames	TextFrames
0. Base	Graphics	Texts

As a result, module *Graphics* does not only contain the base type *Object*, but also its extensions *Line* and *Caption* (and *Macro*). Their methods are also defined in *Graphics*, with the exception of drawing methods, which are defined in *GraphicFrames*, because they refer to frames.

So far, we have discussed operations on individual objects and the structure resulting from the desire to be able to add new object types without affecting the base module. We now turn our attention briefly to operations on graphics as a whole. They can be grouped into two kinds, namely operations involving a graphic as a set, and those applying to the selection, i.e. to a subset only.

The former kind consists of procedures Add, which inserts a new object, *Draw*, which traverses the set of objects and invokes their drawing methods, *ThisObj*, which searches for an object at a given position, *SelectObj*, which marks an object to be selected, *SelectArea*, which identifies all objects lying within a given rectangular area and marks them, *Selectable*, a Boolean function, and *Enumerate*, which applies the parametric procedure handle to all objects of a graphic. Furthermore, the procedures *Load, Store, Print,* and *WriteFile* belong to this kind.

The set of operations applying to selected objects only consist of the following procedures: *Deselect, DrawSel* (drawing the selection according to a specified mode), *Change* (changing certain attributes of selected objects like width, font, color), *Move, Copy, CopyOver* (copying from one graphic into another), and finally *Delete*. Also, there exists the important procedure *Open* which creates a new graphic, either loading a graphic stored as a file, or generating an empty graphic.

The declaration of types and procedures that have emerged so far are summarized in the following excerpt of the module's interface definition.

```
DEFINITION Graphics;  (*excerpt without macros*)
  IMPORT Files, Fonts, Texts, Modules, Display;

  CONST NameLen = 32;

  TYPE Graph = POINTER TO GraphDesc;
    Object = POINTER TO ObjectDesc;
    Method = POINTER TO MethodDesc;

    ObjectDesc = RECORD
      x, y, w, h: INTEGER;
      col: BYTE;
      selected, marked: BOOLEAN;
      do: Method
    END ;

    Msg = RECORD END ;
    WidMsg = RECORD (Msg) w: INTEGER END ;
    ColorMsg = RECORD (Msg) col: INTEGER END ;
    FontMsg = RECORD (Msg) fnt: Fonts.Font END ;
    Name = ARRAY NameLen OF CHAR;

    GraphDesc = RECORD sel: Object;
      time: INTEGER
     END ;

    Context = RECORD END ;

    MethodDesc = RECORD
      module, allocator: Name;
      new: Modules.Command;
      copy: PROCEDURE (obj, to: Object);
      draw, change: PROCEDURE (obj: Object; VAR msg: Msg);
      selectable: PROCEDURE (obj: Object; x, y: INTEGER): BOOLEAN;
      read: PROCEDURE (obj: Object; VAR R: Files.Rider; VAR C: Context);
      write: PROCEDURE (obj: Object; cno: INTEGER; VAR R: Files.Rider; VAR C: Context);
    END ;

    Line = POINTER TO LineDesc;
    LineDesc = RECORD (ObjectDesc) END ;

    Caption = POINTER TO CaptionDesc;
    CaptionDesc = RECORD (ObjectDesc) pos, len: INTEGER END ;

  VAR width, res: INTEGER;
    T: Texts.Text;
    LineMethod, CapMethod, MacMethod: Method;

  PROCEDURE New(obj: Object);
  PROCEDURE Add (G: Graph; obj: Object);
  PROCEDURE Draw (G: Graph; VAR M: Msg);
  PROCEDURE ThisObj (G: Graph; x, y: INTEGER): Object;
  PROCEDURE SelectObj (G: Graph; obj: Object);
  PROCEDURE SelectArea (G: Graph; x0, y0, x1, y1: INTEGER);


  PROCEDURE Deselect (G: Graph);
  PROCEDURE DrawSel (G: Graph; VAR M: Msg);
```

```
    PROCEDURE Change (G: Graph; VAR M: Msg);
    PROCEDURE Move (G: Graph; dx, dy: INTEGER);
    PROCEDURE Copy (Gs, Gd: Graph; dx, dy: INTEGER);
    PROCEDURE Delete (G: Graph);

    PROCEDURE FontNo (VAR W: Files.Rider; VAR C: Context; fnt: Fonts.Font):  INTEGER;
    PROCEDURE WriteObj (VAR W: Files.Rider; cno:  INTEGER; obj: Object);
    PROCEDURE Store (G: Graph; VAR W: Files.Rider);
    PROCEDURE WriteFile (G: Graph; name: ARRAY OF CHAR);
    PROCEDURE Font (VAR R: Files.Rider; VAR C: Context): Fonts.Font;
    PROCEDURE Load (G: Graph; VAR R: Files.Rider);
    PROCEDURE Open (G: Graph; name: ARRAY OF CHAR);
END Graphics.
```

13.4. Displaying graphics

The base module *Graphics* defines the representation of a set of objects in terms of a data structure. The particulars are hidden and allow the change of structural representation by an exchange of this module without affecting its clients. The problems of displaying a graphic on a screen or a printed page are not handled by this module; they are delegated to the client module *GraphicFrames*, which defines a frame type for graphics which is an extension of *Display.Frame*, just like *TextFrames.Frame* is an extension of *Display.Frame*. In contrast to text frames, however, a graphic instead of a text is associate with it.

```
FrameDesc = RECORD (Display.FrameDesc)
        graph: Graphics.Graph;
        Xg, Yg, X1, Y1, x, y, col: INTEGER;
        marked, ticked: BOOLEAN;
        mark: LocDesc
    END
```

Every frame specifies its coordinates *X, Y* within the display area, its size by the attributes *W* (width) and *H* (height), and its background color *col*. Just as a frame represents a (rectangular) section of the entire screen, it also shows an excerpt of the drawing plane of the graphic. The coordinate origin need coincide with neither the frame origin nor the display origin. The frame's position relative to the graphic plane's origin is recorded in the frame descriptor by the coordinates *Xg, Yg*.

The additional, redundant attributes *x, y, X1, Y1* are given by the following invariants, and they are recorded in order to avoid their frequent recomputation.

```
X1  =  X + W,          Y1  =  Y + H
x   =  X + Xg,         y   =  Y1 + Yg
```

X and *Y* (and hence also *X1* and *Y1*) are changed when a viewer is modified, i.e. when the frame is moved or resized. *Xg* and *Yg* are changed when the graph's origin is moved within a frame. The meaning of the various values is illustrated in Figure 13.4.

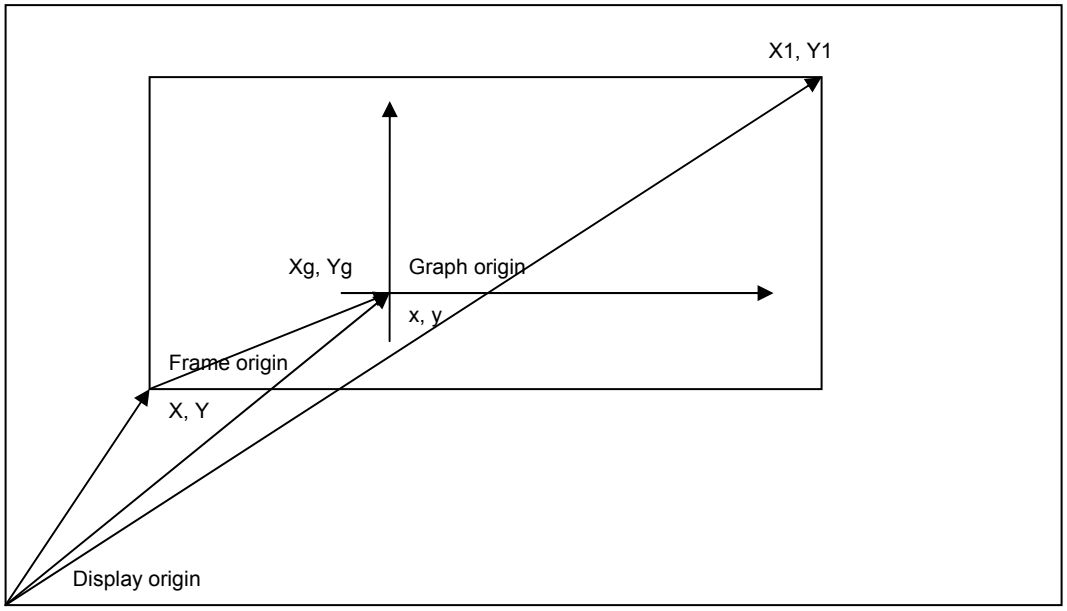


Figure 13.4 Frame and graph coordinates

As a consequence, the display coordinates *u, v* of an object *z* of a graph displayed in a frame *f* are computed as

u = z.x + f.x, v = z.y + f.y

In order to determine whether an object *z* lies within a frame *f*, the following expression must hold:

(f.X <= u) & (u + z.w <= f.X1) & (f.Y <= v) & (v + z.h <= f.Y1)

The record field *marked* indicates whether or not the frame contains a caret. Its display position is recorded in the field called *mark*. A frame may contain several (secondary) carets; they form a list of location descriptors.

When an object is displayed (drawn), its state must be taken into consideration in order to provide visible user feedback. The manner in which selection is indicated, however, may vary among different object types. This can easily be realized, because every object (type) is associated with an individual drawing procedure. The following visualizations of selection have been chosen:

Selected lines are shown in a grey tone (raster pattern).
Selected captions are shown with "inverse video".

Change of state is a relatively frequent operation, and if possible a complete repainting of the involved objects should be avoided for reasons of efficiency. Therefore, procedures for drawing an object are given a mode parameter, in addition to the obvious object and frame parameters. The parameters are combined into the message record of type *DrawMsg*.

```
DrawMsg =    RECORD (Graphics.Msg)
                f: Frame;
                mode, x, y, col: INTEGER
             END
```

The meaning of the mode parameter's four possible values are the following:

mode = 0: draw object according to its state,
mode = 1: draw reflecting a transition from normal to selected state,
mode = 2: draw reflecting a transition from selected to normal state,
mode = 3: erase.

In the case of captions, for instance, the transitions are indicated by simply inverting the rectangular area covered by the caption. No rewriting of the captions' character patterns is required.

A mode parameter is also necessary for reflecting object deletion. First, the selected objects are drawn with *mode* indicating erasure. Only afterwards are they removed from the graphic's linked list.

Furthermore, the message parameter of the drawing procedure contains two offsets *x* and *y*. They are added to the object's coordinates, and their significance will become apparent in connection with macros. The same holds for the color parameter.

The drawing procedures are fairly straight-forward and use the four basic raster operations of module *Display*. The only complication arises from the need to clip the drawing at the frame boundaries. In the case of captions, a character is drawn only if it fits into the frame in its entirety. The raster operations do not test (again) whether the indicated position is valid.

At this point we recall that copies of a viewer (and its frames) can be generated by the *System.Copy* command. Such copies display the same graphic, but possibly different excerpts of them. When a graphic is changed by an insertion, deletion, or any other operation, at a place that is visible in several frames, all affected views must reflect the change. A direct call to a drawing procedure indicating a frame and the change does therefore not suffice. Here again, the object-oriented style solves the problem neatly: In place of a direct call a message is broadcast to all frames, the message specifying the nature of the required updates.

The broadcast is performed by the general procedure *Viewers.Broadcast(M)*. It invokes the handlers of all viewers with the parameter *M*. The viewer handlers either interpret the message or propagate it to the handlers of their subframes. Procedure *obj.handle* is called with a control message as parameter when pointing at the object and clicking the middle mouse button. This allows control to be passed to the handler of an individual object.

The definition of module *GraphicFrames* is summarized by the following interface:

```
DEFINITION GraphicFrames;
    IMPORT Display, Graphics;

    TYPE Frame = POINTER TO FrameDesc;
      Location = POINTER TO LocDesc;

      LocDesc = RECORD
        x, y: INTEGER;
        next: Location
      END ;

      FrameDesc = RECORD (Display.FrameDesc)
        graph: Graphics.Graph;
        Xg, Yg, X1, Y1, x, y, col: INTEGER;
        marked, ticked: BOOLEAN;
        mark: LocDesc
      END ;

      (*mode = 0: draw according to selected, 1: normal -> selected, 2: selected -> normal, 3: erase*)

      DrawMsg = RECORD (Graphics.Msg)
        f: Frame;
        x, y, col, mode: INTEGER
      END ;

    PROCEDURE Restore (F: Frame);
    PROCEDURE Focus (): Frame;
    PROCEDURE Selected (): Frame;
    PROCEDURE This(x, y: INTEGER): Frame;
    PROCEDURE Draw (F: Frame);
    PROCEDURE Erase (F: Frame);
    PROCEDURE DrawObj (F: Frame; obj: Graphics.Object);
    PROCEDURE EraseObj (F: Frame; obj: Graphics.Object);
    PROCEDURE Change (F: Frame; VAR msg: Graphics.Msg);
    PROCEDURE Defocus (F: Frame);
    PROCEDURE Deselect (F: Frame);
    PROCEDURE Macro (VAR Lname, Mname: ARRAY OF CHAR);
```

```
    PROCEDURE Open (G: Frame; graph: Graphics.Graph);
END GraphicFrames.
```

Focus and *Selected* identify the graphic frame containing the caret, or containing the latest selection. *Draw, Erase,* and *Handle* apply to the selection of the specified frame's graphic. And *Open* generates a frame displaying the specified graphic.

13.5. The user interface

Although the display is the prime constituent of the interface between the computer and its user, we chose the title of this chapter for a presentation primarily focussed on the computer's input, i.e. on its actions instigated by the user's handling of keyboard and mouse, the editing operations. The design of the user interface plays a decisive role in a system's acceptance by users. There is no fixed set of rules which determine the optimal choice of an interface. Many issues are a matter of subjective judgement, and all too often convention is being mixed up with convenience. Nevertheless, a few criteria have emerged as fairly generally accepted.

We base our discussion on the premise that input is provided by a keyboard and a mouse, and that keyboard input is essentially to be reserved for textual input. The critical issue is that a mouse - apart from providing a cursor position - allows to signal actions by the state of its keys. Typically, there are far more actions than there are keys. Some mice feature a single key only, a situation that we deem highly unfortunate. There are, however, several ways to "enrich" key states:

1. Position. Key states are interpreted depending on the current position of the mouse represented by the cursor. Typically, interpretation occurs by the handler installed in the viewer covering the cursor position, and different handlers are associated with different viewer types. The handler chosen for interpretation may even be associated with an individual (graphic) object and depend on that object's type.

2. Multiple clicks. Interpretation may depend on the number of repeated clicks (of the same key), and/or on the duration of clicks.

3. Interclicks. Interpretation may depend on the combination of keys depressed until the last one is released. This method is obviously inapplicable for single-key mice.

Apart from position dependence, we have quite successfully used interclicks. A ground rule to be observed is that frequent actions should be triggered by single-key clicks, and only variants of them should be signalled by interclicks. The essential art is to avoid overloading this method.

Less frequent operations may as well be triggered by textual commands, i.e. by pointing at the command word and clicking the middle button. Even for this kind of activation, Oberon offers two variations:

1. The command is listed in a menu (title bar). This solution is favoured when the respective viewer is itself a parameter to the command, and it is recommended when the command is reasonably frequent, because the necessary mouse movement is relatively short.

2. The command lies elsewhere, typically in a viewer containing a tool text.

Lastly, we note that any package such as *Draw* is integrated within an entire system together with other packages. Hence it is important that the rules governing the user interfaces of the various packages do not differ unnecessarily, but that they display common ground rules and a common design "philosophy". *Draw*'s conventions were, as far as possible and sensible, adapted to those of the text system. The right key serves for selection, the left for setting the caret, and the middle key for activating general commands, in this case moving and copying the entire graphic. Inherently, drawing involves certain commands that cannot be dealt with in the same way as for texts. A character is created by typing on the keyboard; a line is created by dragging the mouse while holding the left key. Interclicks left-middle and right-middle are treated in the same way as in the text system (copying a caption from the selection to the caret), and this is not surprising, because text and graphics are properly integrated, i.e. captions can be copied from texts into graphics and vice-versa.

Using different conventions depending on whether the command was activated by pointing at the caption within a text frame or within a graphics frame would be confusing indeed.

13.6. Macros

For many applications it is indispensible that certain sets of objects may be named and used as objects themselves. Such a named subgraph is called a *macro*. A macro thus closely mirrors the sequence of statements in a program text that is given a name and can be referenced from within other statements: the procedure. The notion of a graphic object becomes recursive, too. The facility of recursive objects is so fundamental that it was incorporated in the base module *Graphics* as the third class of objects.

Its representation is straight-forward: in addition to the attributes common to all objects, a field is provided storing the head of the list of elements which constitute the macro. In the present system, a special node is introduced representing the head of the element list. It is of type *MacHeadDesc* and carries also the name of the macro and the width and height of the rectangle covering all elements. These values serve to speed up the selection process, avoiding their recomputation by scanning the entire element list.

The recursive nature of macros manifests itself in recursive calls of display procedures. In order to draw a macro, drawing procedures of the macro's element types are called (which may be macros again). The coordinates of the macro are added to the coordinates of each element, which function as offsets. The color value of the macro, also a field of the parameter of type *DrawMsg*, overrides the colors of the elements. This implies that macros always appear monochrome.

An application of the macro facility is the design of schematics of electronic circuits. Circuit components correspond to macros. Most components are represented by a rectangular frame and by labelled connectors (pins). Some of the most elementary components, such as gates, diodes, transistors, resistors, and capacitors are represented by standardized symbols. Such symbols, which may be regarded as forming an alphabet of electronic circuit diagrams, are appropriately provided in the form of a special font, i.e. a collection of raster patterns. Three such macros are shown in Figure 13.5, together with the components from which they are assembled. The definitions of the data types involved are:

```
Macro =          POINTER TO MacroDesc;
MacroDesc =      RECORD (ObjectDesc) mac: MacHead END ;

MacHead =        POINTER TO MacHeadDesc;
MacHeadDesc =    RECORD name: Name;
                      w, h: INTEGER; lib: Library
                 END ;

Library =        POINTER TO LibraryDesc;
LibraryDesc =    RECORD name: Name END
```

Procedure *DrawMac(mh, M)* displays the macro with head *mh* according to the draw message parameter *M* which specifies a frame, a position within the frame, a display mode, and an overriding color.

In the great majority of applications, macros are not created by their user, but are rather provided from another source, in the case of electronic circuits typically by the manufacturer of the components represented by the macros. As a consequence, macros are taken from a collection (inappropriately) called a *library*. In our system, a macro is picked from such a collection by the command *Draw.Macro* with a library name and a macro name as parameters. It inserts the specified macro at the place of the caret by calling *GraphicFrames.Macro*, which in turn calls *Graphics.Add.*

At last, we mention that selection of a macro is visualized by covering with a dot pattern the entire rectangular area occupied by the macro. This emphasizes the fact that the macro constitutes an object as a whole.

The design of new macros is a relatively rare activity. Macros are used rather like characters of a font; the design of new macros and fonts is left to the specialist. Nevertheless, it was decided to incorporate the ingredients necessary for macro design in the basic system. They consist of a few procedures only which are used by a tool module called *MacroTool* (see Section 16.3).

MakeMac integrates all elements lying within a specified rectangular area into a new macro. *OpenMac* reverses this process by disintegrating the macro into its parts. *InsertMac* inserts a specified macro into a library. *NewLib* creates a new, empty library, and *StoreLib* generates a library file containing all macros currently loaded into the specified library. The details of these operations may be examined in the program listings provided later in this Chapter. Summarizing, the following procedures are exported from module *Graphics* related to handling macros:

```
PROCEDURE GetLib(name: ARRAY OF CHAR; replace: BOOLEAN; VAR Lib: Library);
PROCEDURE ThisMac(L: Library; Mname: ARRAY OF CHAR): MacHead;
PROCEDURE DrawMac(mh: MacHead; VAR M: Msg);
```

and the following are added for creating new macros and libraries:

```
PROCEDURE NewLib(Lname: ARRAY OF CHAR): Library;
PROCEDURE StoreLib(L: Library; Fname: ARRAY OF CHAR);
PROCEDURE RemoveLibraries;
PROCEDURE OpenMac(mh: MacHead; G: Graph; x, y: INTEGER);
PROCEDURE MakeMac(G: Graph; x, y, w, h: INTEGER; Mname: ARRAY OF CHAR): MacHead;
PROCEDURE InsertMac(mh: MacHead; L: Library; VAR new: BOOLEAN);
```

13. 7. Object classes

Although surprisingly many applications can be covered satisfactorily with the few types of objects and the few facilities described so far, it is nevertheless expected that a modern graphics system allow the addition of further types of objects. The emphasis lies here on the word addition instead of change. New facilities are to be providable by the inclusion of new modules without requiring any kind of adjustment, not even recompilation of the existing modules. In practice, their source code would quite likely not be available. It is the triumph of the object-oriented programming technique that this is elegantly possible. The means are the extensible record type and the procedure variable, features of the programming language, and the possibility to load modules on demand from statements within a program, a facility provided by the operating environment.

We call, informally, any extension of the type *Object* a *class*. Hence, the types *Line, Caption,* and *Macro* constitute classes. Additional classes can be defined in other modules importing the type *Object*. In every such case, a set of methods must be declared and assigned to a variable of type *MethodDesc*. They form a so-called *method suite*. Every such module must also contain a procedure, typically a command, to generate a new instance of the new class. This command, likely to be called *Make*, assigns the method suite to the *do* field of the new object.

This successful decoupling of additions from the system's base suffices, almost. Only one further link is unavoidable: When a new graphic, containing objects of a class not defined in the system's core, is loaded from a file, then that class must be identified, the corresponding module with its handlers must be loaded - this is called *dynamic loading* - and the object must be generated (allocated). Because the object in question does not already exist at the time when reading the object's attribute values, the generating procedure cannot possibly be installed in the very same object, i.e. it cannot be a member of the method suite. We have chosen the following solution to this problem:

1. Every new class is implemented in the form of a module, and every class is identified by the module name. Every such module contains a command whose effect is to allocate an object of the class, to assign the message suite to it, and to assign the object to the global variable *Graphics.new*.
2. When a graphics file is read, the class of each object is identified and a call to the respective module's allocation procedure delivers the desired object. The call consists of two parts: a call to *Modules.ThisMod*, which may cause the loading of the respctive class module *M*, and a call of

Modules.ThisCommand. Then the data of the base type *Object* are read, and lastly the data of the extension are read by a call to the class method *read*.

The following may serve as a template for any module defining a new object class *X*. Two examples are given in Section 13.9, namely *Rectangles* and *Curves*.

```
MODULE Xs;
  IMPORT Files, Oberon, Graphics, GraphicFrames;

  TYPE X* = POINTER TO XDesc;
    XDesc = RECORD (Graphics.ObjectDesc) (*additional data fields*) END ;

  VAR method: Graphics.Method;

  PROCEDURE New*;
    VAR x: X;
  BEGIN NEW(x); x.do := method; Graphics.new := x
  END New;

  PROCEDURE* Copy(obj, to: Graphics.Object);
  BEGIN to(X)^ := obj(X)^
  END Copy;

  PROCEDURE* Draw(obj: Graphics.Object; VAR msg: Graphics.Msg);
  BEGIN ...
  END Draw;

  PROCEDURE* Selectable(obj: Graphics.Object; x, y: INTEGER): BOOLEAN;
  BEGIN ...
  END Selectable;

  PROCEDURE* Change(obj: Graphics.Object; VAR msg: Graphics.Msg);
  BEGIN
    IF msg IS Graphics.ColorMsg THEN obj.col := msg(Graphics.ColorMsg).col
    ELSIF msg IS ... THEN ...
    END
  END Handle;

  PROCEDURE* Read(obj: Graphics.Object; VAR W: Files.Rider; VAR C: Context);
  BEGIN (*read X-specific data*)
  END Write;

  PROCEDURE* Write(obj: Graphics.Object; cno: SHORTINT;
    VAR W: Files.Rider; VAR C: Context);
  BEGIN Graphics.WriteObj(W, cno, obj); (*write X-specific data*)
  END Write;

  PROCEDURE Make*;   (*command*)
    VAR x: X;  F: GraphicFrames.Frame;
  BEGIN F := GraphicFrames.Focus();
    IF F # NIL THEN
      GraphicFrames.Deselect(F);
      NEW(x); x.x := F.mark.x - F.x; x.y := F.mark.y - F.y; x.w := ... ; x.h := ... ;
      x.col := Oberon.CurCol; x.do := method;
      GraphicFrames.Defocus(F); Graphics.Add(F.graph, x); GraphicFrames.DrawObj(F, x)
    END
  END Make;

BEGIN NEW(method); method.module := "Xs"; method.allocator := "New";
  method.copy := Copy; method.draw := Draw; method.selectable := Selectable;
  method.handle := Handle; method.read := Read; method.write := Write; method.print := Print
END Xs.
```

We wish to point out that also the macro and library facilities are capable of integrating objects of new classes, i.e. of types not occurring in the declarations of macro and library facilities. The complete interface definition of module *Graphics* is obtained from its excerpt given in Sect. 13.3, augmented by the declarations of types and procedures in Sect. 13.6. and 13.7.

13.8. The implementation

13.8.1. Module Draw

Module *Draw* is a typical command module whose exported procedures are listed in a tool text. Its task is to scan the text containing the command for parameters, to check their validity, and to activate the corresponding procedures, which primarily are contained in modules *Graphics* and *GraphicFrames.* The most prominent among them is the *Open* command. It generates a new viewer containing two frames, namely a text frame serving as menu, and a graphic frame.

We emphasize at this point that graphic frames may be opened and manipulated also by other modules apart from *Draw*. In particular, document editors that integrate texts and graphics - and perhaps also other entities - would refer to *Graphics* and *GraphicFrames* directly, but not make use of *Draw* which, as a tool module, should not have client modules.

```
DEFINITION Draw;
    PROCEDURE Open;
    PROCEDURE Delete;
    PROCEDURE SetWidth;
    PROCEDURE ChangeColor;
    PROCEDURE Store;
    PROCEDURE Macro;

    PROCEDURE OpenMacro;
    PROCEDURE MakeMacro;
    PROCEDURE LoadLibrary;
END Draw.
```

13.8.2. Module *GraphicFrames*

Module *GraphicFrames* contains all routines concerned with displaying, visualizing graphic frames and their contents, i.e. graphics. It also contains the routines for creating new objects of the base classes, i.e. lines, captions, and macros. And most importantly, it specifies the appropriate frame handler which interprets input actions and thereby defines the user interface. The handler discriminates among the following message types:

1. Update messages. According to the *id* field of the message record, either a specific object or the entire selection of a graphic are drawn according to a mode. The case *id = 0* signifies a restoration of the entire frame including all objects of the graphic.

2. Selection, focus, and position queries. They serve for the identification of the graphic frame containing the latest selection, containing the caret (mark) or the indicated position. In order to identify the latest selection, the time is recorded in the graph descriptor whenever a new selection is made or when new objects are inserted.

3. Input messages. They originate from the central loop of module *Oberon* and indicate either a mouse action (track message) or a keyboard event (consume message).

4. Control messages from *Oberon*. They indicate that all marks (selection, caret, star) are to be removed (neutralize), or that the focus has to be relinquished (defocus).

5. Selection and copy messages from *Oberon*. They constitute the interface between the graphics and the text system, and make possible identification and copying of captions between graphic and text frames.

6. Modify messages from *MenuViewers*. They indicate that a frame has to be adjusted in size and position because a neighbouring viewer has been reshaped, or because its own viewer has been repositioned

7. Display messages. They originate from procedure *InsertChar* and handle the displaying of single characters when a caption is composed (see below).

The frame handler receiving a consume message interprets the request through procedure *InsertChar*, and receiving a track message through procedure *Edit*. If no mouse key is depressed, the cursor is simply drawn, and thereby the mouse is tracked. Instead of the regular arrow, a crosshair is used as cursor pattern. Thereby immediate visual feedback is provided to indicate that now mouse actions are interpreted by the graphics handler (instead of, e.g., a text handler). Such feedback is helpful when graphic frames appear not only in a menuviewer, but as subframes of a more highly structured document frame.

Procedure *Edit* first tracks the mouse while recording further key activities (interclicks) until all keys are released. The subsequent action is determined by the perceived key clicks. The actions are (the second key denotes the interclick):

keys = left	set caret, if mouse was not moved, otherwise draw new line,
keys = left, middle	copy text selection to caret position
keys = left, right	set secondary caret (mark)
keys = middle	move selection
keys = middle, left	copy selection
keys = middle, right	shift origin of graph
keys = right	select (either object, or objects in area)
keys = right, middle	copy selected text to caret position

When copying or moving a set of selected objects, it must be distinguished between the cases where the source and the destination graphics are the same or are distinct. In the former case, source and destination positions may lie in the same or in different frames.

Procedure *InsertChar* handles the creation of new captions. The actual character string is appended to the global text *T*, and the new object records its position within *T* and its length.

A complication arises because the input process consists of as many user actions as there are characters, and because other actions may possibly intervene between the typing. It is therefore unavoidable to record an insertion state, which is embodied by the global variable *newcap*. When a character is typed, and *newcap* = *NIL*, then a new caption is created consisting of the single typed character. Subsequent typing results in appending characters to the string (and *newcap*). The variable is reset to NIL, when the caret is repositioned. The BS character is interpreted as a backspace by procedure *DeleteChar*.

Since the caption being generated may be visible simultaneously in several frames, its display must be handled by a message. For this reason, the special message *DispMsg* is introduced, and as a result, the process of character insertion turns out to be a rather complex action. To avoid even further complexity, the restriction is adopted that all characters of a caption must use the same attributes (font, color).

The definition of the interface of *GraphicFrames* is listed in Section 13.3.

13.8.3. Module *Graphics*

The preceding presentations of the interface definitions have explained the framework of the graphics system and set the goals for their implementation. We recall that the core module *Graphics* handles the data structures representing sets of objects without reliance on the specifications of individual objects. Even the structural aspects of the object sets are not fixed by the interface. Several solutions, and hence several implementations are imaginable.

Here we present the simplest solution for representing an abstract, unordered set: the linear, linked list. It is embodied in the object record's additional, hidden field *next*. Consequently, a graphic is represented by the head of the list. The type *GraphDesc* contains the hidden field *first* (see listing of *Graphics*). In addition, the descriptor contains the exported field *sel* denoting a selected element, and the field *time* indicating the time of its selection. The latter is used to determine the most recent selection in various viewers.

Additional data structures become necessary through the presence of macros and classes. Macros are represented by the list of their elements, like graphics. Their header is of type *MacHeadDesc* in analogy to *GraphDesc*. In addition to a macro's name, width, and height, it contains the field *first*, pointing to the list's first element, and the field *lib*, referring to the library from which the macro stems.

A library descriptor is similarly structured: In addition to its name, the field *first* points to the list of elements (macros) of the library, which are themselves linked through the field *next*. Fig. 13.6. shows the data structure containing two libraries. It is anchored in the global variable *firstLib*.

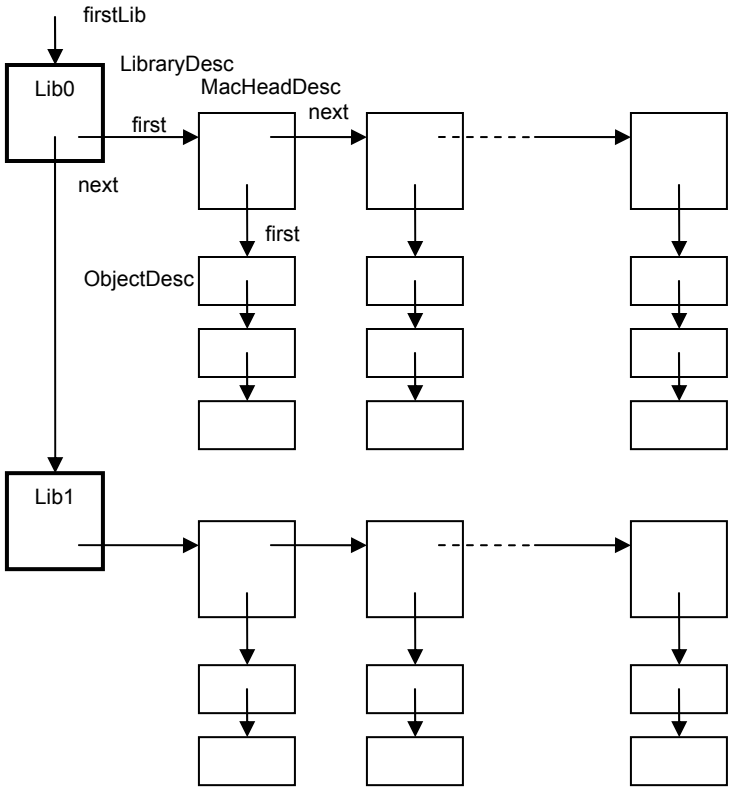


Fig. 13.6 Data structure for two libraries, each with three macros

Libraries are permanently stored as files. It is evidently unacceptable that file access be required upon every reference to a macro, e.g. each time a macro is redrawn. Therefore a library is loaded into primary store, when one of its elements is referenced for the first time. Procedure *ThisMac* searches the data structure representing the specified library and locates the header of the requested macro.

We emphasize that the structures employed for macro and library representation remain hidden from clients, just like the structure of graphics remains hidden within module *Graphics*. Thus, none of the linkage fields of records (*first, next, sel*) are exported from the base module. This measure retains the possibility to change the structural design decisions without affecting the client modules. But partly it is also responsible for the necessity to include macros in the base module.

A large fraction of module *Graphics* is taken up by procedures for reading and writing files representing graphics and libraries. They convert their internal data structure into a sequential form and vice-versa. This would be a rather trivial task, were it not for the presence of pointers referring to macros and classes. These pointers must be converted into descriptions that are position-independent, such as names. The same problem is posed by fonts (which are also represented by pointers).

Evidently, the replacement of every pointer by an explicit name would be an uneconomical solution with respect to storage space as well as speed of reading and writing. Therefore, pointers to fonts

and libraries - themselves represented as files - are replaced by indices to font and library dictionaries. These dictionaries establish a context and are constructed while a file is read. They are used only during this process and hence are local to procedure *Load* (or *Open*). For classes, a dictionary listing the respective allocation procedures is constructed in order to avoid repeated calls to determine the pertinent allocator.

When a graphics file is generated by procedure *Store*, local dictionaries are constructed of fonts, libraries, and classes of objects that have been written onto the file. Upon encountering a caption, a macro, or any element whose font, library, or class is not contained in the respective dictionary, a pair consisting of index and name is emitted to the file, thereby assigning a number to each name. These pairs are interspersed within the sequence of object descriptions.

When the graphic file is read, these pairs trigger insertion of the font, library, or class in the respective dictionary, whereby the name is converted into a pointer to the entity, which is obtained by a loading process embodied by procedures *Fonts.This, GetLib,* and *GetClass*. Both the *Load* and *Store* procedures traverse the file only once. The files are self-contained in the sense that all external quantities are represented by their names. The format of a graphics file is defined in Extended BNF syntax as follows:

```
file =    tag stretch.
stretch = {item} 255.
item =    0 0 fontno fontname | 0 1 libno libname | 0 2 classno classname allocname |
          1 data |  2 data fontno string |  3 data libno macname |  classno data extension.
data =    x  y  w  h  color.
```

All class numbers are at least 4; the values 1, 2, and 3 are assigned to lines, captions, and macros. *x, y, w, h* are two-byte integer attributes of the base type *Object*. The attribute *color* takes a single byte. The first byte of an item being 0 signifies that the item is an identification of a new font, library, or class. If the second byte is 0, a new font is announced, if 1 a new library, and if 2 a new class of elements.

The same procedures are used for loading and storing a library file. In fact, *Load* and *Store* read and write a file stretch representing a sequence of elements which is terminated by a special value (255). In a library file each macro corresponds to a stretch, and the terminator is followed by values specifying the macro's overall width, height, and its name. The structure of library files is defined by the following syntax:

```
libfile =  libtag {macro}.
macro =    stretch w h name.
```

The first byte of each element is a class number within the context of the file and identifies the class to which the element belongs. An object of the given class is allocated by calling the class' allocation procedure, which is obtained from the class dictionary in the given context. The class number is used as dictionary index. The presence of the required allocation procedure in the dictionary is guaranteed by the fact that a corresponding index/name pair had preceded the element in the file.

The encounter of such a pair triggers the loading of the module specifying the class and its methods. The name of the pair consists of two parts: the first specifies the module in which the class is defined, and it is taken as the parameter of the call to the loader (see procedure *GetClass*). The second part is the name of the relevant allocation procedure which returns a fresh object to variable *Graphics.new*. Thereafter, the data defined in the base type *Object* are read.

Data belonging to an extension follow those of the base type, and they are read by the extension's *read* method. This part must always be headed by a byte specifying the number of bytes which follow. This information is used in the case where a requested module is not present; it indicates the number of bytes to be skipped in order to continue reading further elements.

A last noteworthy detail concerns the *Move* operation which appears as surprisingly complicated, particularly in comparison with the related copy operation. The reason is our deviation from the principle that a graphics editor must refrain from an interpretation of drawings. Responsible for this

deviation was the circumstance that the editor was at first primarily used for the preparation of circuit diagrams. They suggested the view that adjoining, perpendicular lines be connected. Consequently, the horizontal or vertical displacement of a line was to preserve connections. Procedure *Move* must therefore identify all connected lines, and subsequently extend or shorten them.

The definition of the interface of *Graphics* is listed in Section 13.3.

13.9. Rectangles and curves

13.9.1. Rectangles

In this section, we present two extensions of the basic graphics system which introduce new classes of objects. The first implements rectangles which are typically used for framing a set of objects. They are, for example, used in the representation of electronic components (macros, see Fig. 13.2). Their implementation follows the scheme presented at the end of chapter 13.7 and is reasonably straightforward, considering that each rectangle merely consists of four lines. Additionally, a background raster may be specified.

One of the design decisions occurring for every new class concerns the way to display the selection. In this case we chose, in contrast to the cases of captions and macros, not inverse video, but a small square dot in the lower right corner of the rectangle. The data type *Rectangle* contains one additional field: *lw* indicates the line width.

In spite of the simplicity of the notion of rectangles, their drawing method is more complex than might be expected. The reason is that drawing methods are responsible for appropriate clipping at frame boundaries. In this case, some of the component lines may have to be shortened, and some may disappear altogether.

Procedure *Handle* provides an example of a receiver of a control message. It is activated as soon as the middle mouse button is pressed, in contrast to other actions, which are initiated after the release of all buttons. Therefore, this message allows for the implementation of actions under control of individual handlers interpreting further mouse movements. In this example, the action serves to change the size of the rectangle, namely by moving its lower left corner.

```
DEFINITION Rectangles;
    TYPE Rectangle = POINTER TO RectDesc;

        RectDesc = RECORD (Graphics.ObjectDesc)
            lw: INTEGER
          END ;

    VAR method: Graphics.Method;

    PROCEDURE New;
    PROCEDURE Make;
END Rectangles.
```

13.9.2. Oblique lines and circles

The second extension to be presented is module *Curves*. It introduces two new kinds of objects: lines which are not necessarily horizontal or vertical, and circles. All are considered to be variants of the same type *Curve*, the variant being specified by the field *kind* of the object record. Selection is indicated by a small rectangle at the end of a line and at the lowest point of a circle.

In order to avoid computations involving floating-point numbers and to increase efficiency, Bresenham algorithms are used. The algorithm for a line defined by $bx - ay = 0$ (for $b \leq a$) is given by the following statements:

```
x := 0; y := 0; h := (b – a) DIV 2;
WHILE x <= a DO Dot(x, y);
   IF h <= 0 THEN INC(h, b) ELSE INC(h, b-a); INC(y) END ;
```

```
    INC(x)
  END
```

The Bresenham algorithm for a circle given by the equation $x^2 + y^2 = r^2$ is:

```
  x := r; y := 0; h := 1-r;
  WHILE y <= x DO Dot(x, y);
    IF h < 0 THEN INC(h, 2*y + 3) ELSE INC(h, 2*(y-x)+5); DEC(x) END ;
    INC(y)
  END

  DEFINITION Curves;
    TYPE Curve = POINTER TO CurveDesc;

        CurveDesc = RECORD (Graphics.ObjectDesc)
              kind, lw: INTEGER
            END ;

    (*kind: 0 = up-line, 1 = down-line, 2 = circle*)

    VAR method: Graphics.Method;
    PROCEDURE MakeLine;
    PROCEDURE MakeCircle*;
  END Curves.
```

14 Building and maintenance tools

14.1. The Startup Process

An aspect usually given little attention in system descriptions is the process of how a system is started. Its choice, however, is itself an interesting and far from trivial design consideration and will be described here in some detail. Moreover, it directly determines the steps in which a system is developed from scratch, mirroring the steps in which it builds itself up from a bare store to an operating body.

The startup process typically proceeds in several stages, each of them bringing further facilities into play, raising the system to a higher level towards completion. The term for this strategy is *boot strapping* or, in modern computer jargon, *booting*.

Stage 0 is initiated when power is switched on or when the reset button is pressed and released. To be precise, power-on issues a reset signal to all parts of the computer and holds it for a certain time. Pushing the reset button therefore appears like a power-on without power having been switched off. Release of the reset signal triggers the built-in FPGA hardware to load a short *configuration bit-stream* from a ROM residing on the Spartan board, called the *platform flash*, into a BRAM within the FPGA. This program is called *boot loader*. Being stored in a ROM, it is always present. The BRAM is address-mapped onto an upper part of the address space, and the RISC processor starts execution at this address.

In Stage 1 the boot loader loads the *inner core*, which consists of modules *Kernel, FileDir, Files,* and *Modules.* The loader first inspects the link register. If its value is 0, a cold start is indicated. (If the value of the link register is not 0, this signals an abort caused by pressing button 3 on the board. Then loading is skipped and control is immediately returned to the Oberon command loop). The disk (SD-card, SPI) is initialized.

The boot loader terminates with a branch to location 0, which transfers control to the just loaded module *Modules*, the regular loader.

Stage 2 starts with the initialization body of module *Modules* which calls the bodies of *Kernel, FileDir* and *Files*, establishing a working file system. Then it calls itself, requesting to load the central module *Oberon*. This implicitly causes the loading of its own imports, namely *Input, Display, Viewers, Fonts,* and *Texts,* establishing a working viewer and text system.

This loading of the *outer core* must be interpreted as the continuation of the loading of the inner core. To allow proper continuation, the boot loader has deposited the following data in fixed locations:

0	A branch instruction to the initializing body of module *Modules*
12	The limit of available memory
16	The address of the end of the module space loaded
20	The current root of the links of loaded modules
24	The current limit of the module area

In Stage 3, *Oberon* calls the loader to load the tool module *System*, and with it its imports *MenuViewers* and *TextFrames*. The initialization of *System* causes the opening of the viewers for the system tool and the system log. Control then returns to *Oberon* and its central loop for polling input events. Normal operation begins. The booting process is summarized in Figure 14.1.

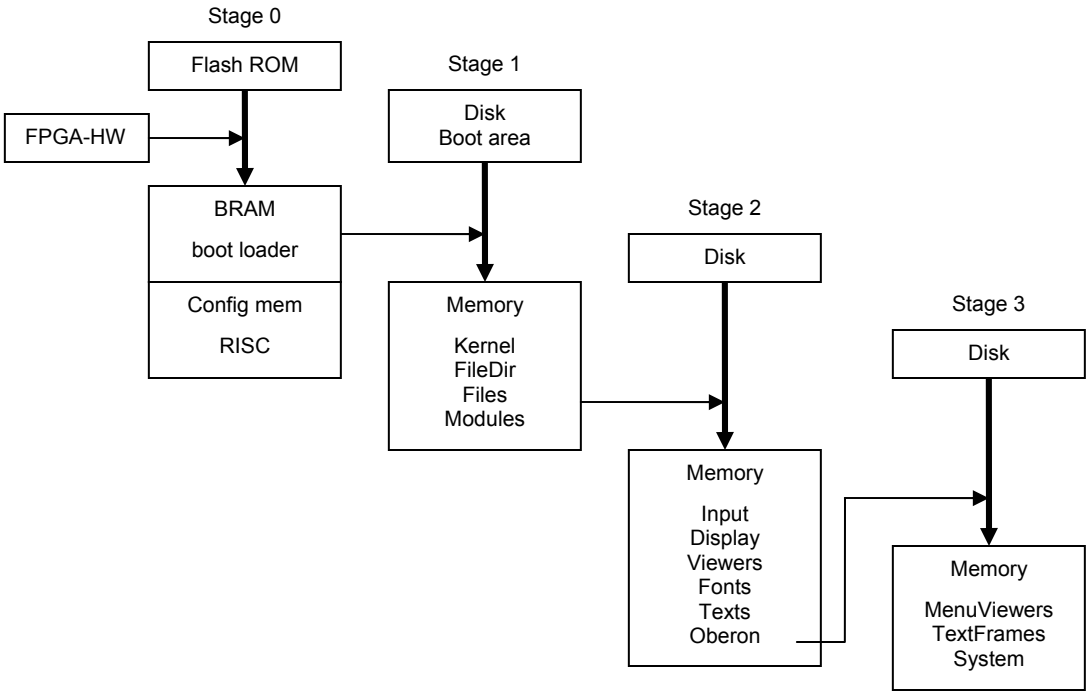


Figure 14.1 The four stages of the booting process

This describes the normal case of startup. But, how did the boot loader ever get into the platform-flash, and how did the inner core ever get into the boot area of the disk, and how did the files of the outer core get into the file store? In fact, how did the file store get initialized? This is described in the following section on building tools.

Precisely to solve this problem, the boot loader has been provided with a second source of the boot data. Instead of from the disk, it may be fetched over a data link, in this case the RS-232 data line. This choice is set by switch 0.

0 load from the "boot track" of the disk (sectors 2 - 63)
1 load from the RS-232 line (or a network, if available)

In case 1, the data stream originates at a host computer, on which presumably the boot file had been generated or even the entire system had been built.

In order to keep the boot loader as simple as possible - remember that it is placed in a small flash memory on every workstation and therefore cannot be changed without a special effort - the format of the byte stream representing the inner core must be simple. We have chosen the following structure, which had never to be changed during the entire development effort of the Oberon System because of both its simplicity and generality:

```
BootFile    = {block}.
block       = size address {byte}.   (size and address are words)
```

The address of the last block, distinguished by *size* = 0, is interpreted as the address of the starting point of Stage 2.

In this step, a module called *Oberon0* is used as the top module, rather than *Modules*. This module communicates with the host computer via the RS-232 line and in addition features various inspection tools. In particular it contains a command copying the just loaded inner core into the disk (see also Section 14.2).

Still, how did the hardware configuration data and the boot loader get into the Flash ROM? This step requires the help of proprietary tools of the FPGA manufacturer. Regrettably, their incantation ceremony typically is rather complex.

After all necessary Verilog modules have been synthesized, the result is the configuration file *RISCTop.bit.* The necessary source files are

RISCTop.v, RISC.v, Multiplier.v, Divider.v, FPAdder.v. FP.Multiplier.v, FP.Divider.v, dbram32.v
RS232R.v, RS232T.v, SPI.v, XGS.v, PS2.v, RISC.ucf

Thereafter, the boot loader is compiled and, together with the result of the configuration of the RISC hardware, loaded into the configuration memory of the FPGA. This Stage 0 is partly done with proprietory software (dependant on the specific FPGA) and is described in a separate installation guide.

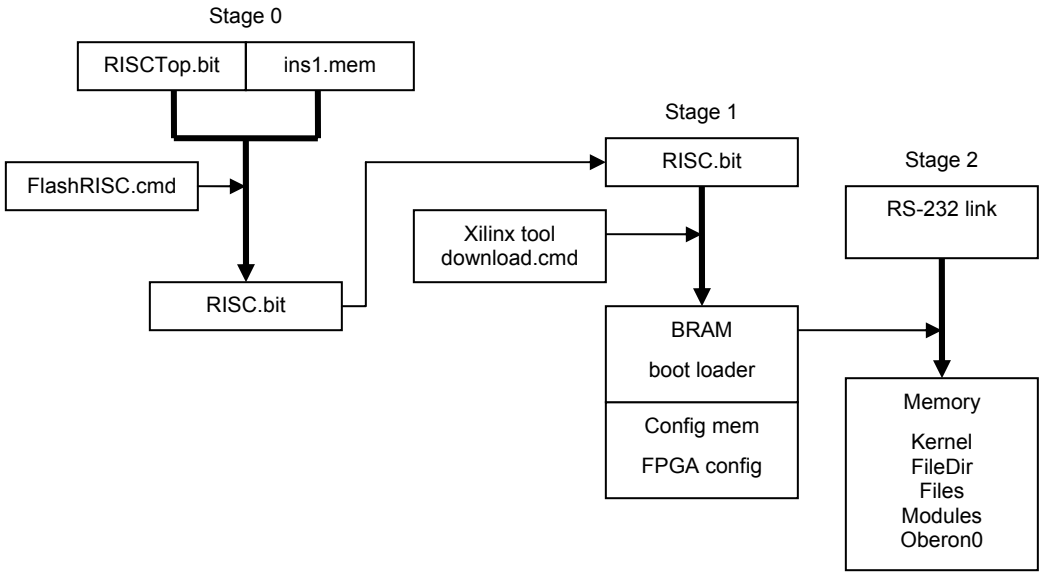


Figure 14.2 Booting from host computer

A simple boot loader reading from the RS-232 line and using the stream format described above is shown here:

```
MODULE* BootLoad;
  IMPORT SYSTEM;
  CONST MT = 12; SP = 14; MemLim = 0E7F00H;
    swi = -60; led = -60; data = -56; ctrl = -52; (*device addresses*)

  PROCEDURE RecInt(VAR x: INTEGER);
    VAR z, y, i: INTEGER;
  BEGIN z := 0;  i := 4;
    REPEAT i := i-1;
      REPEAT UNTIL SYSTEM.BIT(ctrl, 0);
      SYSTEM.GET(data, y); z := ROR(z+y, 8)
    UNTIL i = 0;
    x := z
  END RecInt;

  PROCEDURE Load;
    VAR len, adr, dat: INTEGER;
  BEGIN RecInt(len);
    WHILE len > 0 DO
      RecInt(adr);
      REPEAT RecInt(dat); SYSTEM.PUT(adr, dat); adr := adr + 4; len := len - 4 UNTIL len = 0;
      RecInt(len)
```

```
    END ;
    SYSTEM.GET(4, adr); SYSTEM.LDREG(13, adr); SYSTEM.LDREG(12, 20H)
  END Load;

BEGIN SYSTEM.LDREG(SP, MemLim); SYSTEM.LDREG(MT, 20H); SYSTEM.PUT(led, 128);
END BootLoad.
```

Another detail that must not be ignored is the handling of traps. They are implemented as a single BRL instruction, jumping conditionally to the address stored in register MT, that is, to entry 0 of the module table (which is not a module address). This address is deposited by the initialization of module *System*, which contains the trap handler. However, traps may also occur during the startup process. So, a temporary trap handler must also be installed at the very start, that is, when initializing *Kernel*.

Finally, it is worth mentioning that small Oberon programs can also be loaded and executed without the Oberon core. In fact, the boot loader is just one such example. Programs of this kind must be marked by an asterisk immediately after the symbol MODULE. This causes the compiler to generate a different starting sequence Such programs are loaded, like the boot loader in Stage 0, by the Xilinx downloader. They must not import other modules.

14.2. Building Tools

Let us summarize the prerequisites for startup:

0. The FPGA configuration and bootloader must reside in the ROM (platform flash)
1. The boot file must reside on the boot area of the disk.
2. The modules of the outer core must reside in the file system.
3. The default font and *System.Tool* must be present in the file system.

These conditions are usually met. But they are not satisfied, if either a new, bare machine is present, or if the disk store is defective. In these cases, the prerequisites must be established with the aid of suitable tools. The tools needed for the case of the bare machine or the incomplete file store are called *building tools*, those required in the case of defects are called *maintenance tools*.

Building tools allow to establish the preconditions for the boot process on a bare machine. Establishing condition 0 requires a tool for downloading the hardware configuration of the FPGA resulting from circuit synthesis, and it requires a compiler for generating the boot loader. Condition 0 is established in Stage 0.

Establishing condition 1 requires a tool for composing the boot file, and one to load it into the boot area. The former is the compiler, presumably running on a host computer. The resulting files are linked by a linker (ORL) generating a "binary" file. This file is then downloaded from the host computer to the RISC running its boot loader. Here we use an extended inner core, where the main module is not *Modules*, but *Oberon0*. The reason is that Oberon0 allows to perform the subsequent stages by accepting commands over a communication channel (here the RS-232 line). Hence, for the following stages, the tool on the RISC is Oberon0, communicating with the host computer's module ORC.

Establishing condition 2 implies the building of a file directory and the loading of files. The pair *Oberon0* and ORC contains commands for initializing a file system, for loading files over the line connection, and for moving the inner core to the disk's boot area. In addition, *Oberon0* contains further commands for file system, memory and disk inspection. Note that loading (and starting) *Oberon0* automatically starts the entire Oberon system.

There remains the important question of how *Oberon0* is loaded onto a bare machine. It is done by the boot loader with switch 1 being up. The boot file contains the inner core with the top module being *Oberon0* rather than *Modules*. The procedure is the following:

1. Select the alternative boot source by setting switch0 = 1.
2. Reset and send the boot file from the host The boot file is transferred and *Oberon0* is started.

3. Read all files from the host, (which supposedly holds all files needed for the outer core).
4. Invoke the command which loads *Oberon*. This loads the outer core, sets up the display, and starts the central loop.

A more modern solution would be to select the network as alternative boot file source. We rejected this option in order to keep net access routines outside the ROM, in order to keep the startup of a computer independent of the presence of a network and foreign sources, and also in consideration of the fact that there exist machines which operate in a stand-alone mode. As it turns out, the need for the alternative boot file source arises very rarely.

The boot linker ORL, presumably running on a host computer, where the FPGA-tools are available, is almost identical to the module loader, with the exception that object code is not deposited in newly allocated blocks, but is output in the form a file. The name of the top module of the inner core is supplied as parameter.

ORL.Link Modules	generates the regular boot file
ORL.Link Oberon0	generates the build-up boot file

Oberon0 imports two modules taking care of communication with ORL on the host computer. They are the basic module RS232, and module *PCLink1* for file transfer. The latter constitutes a task, accepting commands over the line from ORL. Their interfaces are shown below:

```
DEFINITION RS232;
    PROCEDURE Send(x: BYTE);
    PROCEDURE Rec(VAR x: BYTE);
    PROCEDURE SendInt(x: INTEGER);
    PROCEDURE SendHex(x: INTEGER);
    PROCEDURE SendReal(x: REAL);
    PROCEDURE SendStr(x: ARRAY OF CHAR);
    PROCEDURE RecInt(VAR x: INTEGER);
    PROCEDURE RecReal(VAR x: REAL);
    PROCEDURE RecStr(VAR x: ARRAY OF CHAR);
    PROCEDURE Line;
    PROCEDURE End;
END RS232.

DEFINITION PCLink1;
    PROCEDURE Run*;
    PROCEDURE Stop*;
END PCLink1.
```

The command interpreter is a simple loop, accepting commands specified by an integer followed by parameters which are either integers or names. User-friendliness was not attributed any importance at this point, and it would indeed be merely luxury. We refrain from elaborating on further details and concentrate on providing a list of commands provided by *Oberon0*. This should give the reader an impression of the capabilities and limitations of this tool module for system initiation and for error searching. (*name* stands for a string, and *a, secno, m, n* stand for integers).

	parameters	action
0	s	send and mirror s
1	a, n	show (in hex) M[a], M[a+4], ... , M[a + n*4]
2	w	fill display with words w
3	secno	show disk sector
4	filename	read file
6	-	start PC-link
7	-	show allocation, nof sectors, switches, and timer
10	-	list modules
11	modname	list commands
12	prefix	list files (enumerate directory)
13	filename	delete file

20	modname	load module
21	modname	unload module
22	name	call command
50	adr, list of values	write memory
51	adr, n	clear memory (n words)
52	secno, list of values	write sector
53	secno, n	clear sector (n words)
100	-	load boot track
101	-	clear file directory

Oberon0 imports modules *Kernel, FileDir, Files, Modules, RS232, PCLink1.* This is the inner core plus facilities for communication.

14.3. Maintenance Tools

An important prerequisite for Stage 2 (and the following stages) in the boot process has not been mentioned above. Recall that the initialization of module *FileDir* constructs the disk sector reservation table in the *Kernel* from information contained on the disk. Obviously, its prerequisite is an intact, consistent file directory. A single unreadable, corrupted file directory or file header sector lets this process fail, and booting becomes impossible. To cope with this (fortunately rare) situation, a maintenance tool has been designed: module *DiskCheck*.

DiskCheck is organized similarly to *Oberon0* as a simple command interpreter, but it imports only *Kernel* and *RS232*. Hence, booting involves only Stages 1 and 2 without any access to the disk. Operating *DiskCheck* requires care and knowledge of the structure of the file system (Chapter 7). The available commands are the following:

	parameters	action
0	s	send and mirror integer (test)
1	a, n	show (in hex) M[a], M[a+4], ... , M[a + n*4]
2	secno	show disk sector
3	secno	show head sector
4	secno	show directory sector
5	-	traverse directory
6	secno	clear header sector
7	-	clear directory (root page)

The essential command is the file directory traversal (5). It lists all faulty directory sectors, showing their numbers. It also lists faulty header sectors. No changes are made to the file system.

If a faulty header is encountered, it can subsequently be cleared (6). Thereby the file is lost. It is not removed from the directory, though. But its length will be zero.

Program *DiskCheck* must be extremely robust. No data read can be assumed to be correct, no index can be assumed to lie within its declared bounds, no sector number can be assumed to be valid, and no directory or header page may be assumed to have the expected format. Guards and error diagnostics take a prominent place.

Whereas a faulty sector in a file in the worst case leads to the loss of that file, a fault in a sector carrying a directory page is quite disastrous. Not only because the files referenced from that page, but also those referenced from descendant pages become inaccessible. A fault in the root page even causes the loss of all files. The catastrophe is of such proportions, that measures should be taken even if the case is very unlikely. After all, it may happen, and it indeed has occurred.

The only way to recover files that are no longer accessible from the directory is by scanning the entire disk. In order to make a search at all possible, every file header carries a mark field that is given a fixed, constant value. It is very unlikely, but not entirely impossible, that data sectors which happen to have the same value at the location corresponding to that of the mark, may be mistaken to be headers.

The tool performing such a scan is called *Scavenger*. It is, like *DiskCheck,* a simple command interpreter with the following available commands:

	parameters	action
0	s	send and mirror integer (test)
1	n	Scan the first n sectors and collect headers
2	-	Display names of collected files
3	-	Build new directory
4	-	Transfer new directory to the disk
5	-	Clear display

During the scan, a new directory is gradually built up in primary store. Sectors marked as headers are recorded by their name and creation date. The scavenger is the reason for recording the file name in the header, although it remains unused there by the Oberon System. Recovery of the date is essential, because several files with the same name may be found. If one is found with a newer creation date, the older entry is overwritten.

Command *W* transfers the new directory to the disk. For this purpose, it is necessary to have free sectors available. These have been collected during the scan: both old directory sectors (identified by a directory mark similar to the header mark) and overwritten headers are used as free locations.

The scavenger has proven its worth on more than one occasion. Its main drawback is that it may rediscover files that had been deleted. The deletion operation by definition affects only the directory, but not the file. Therefore, the header carrying the name remains unchanged and is discovered by the scan. All in all, however, it is a small deficiency.

Reference

1. N. Wirth. Designing a System from Scratch. *Structured Programming, 1*, (1989), 10-18.

15 Tool and service modules

In this chapter, a few modules are presented that do not belong to Oberon's system core. However, they belong to the system in the sense of being basic, and of assistance in some way, either to construct application programs, to communicate with external computers, or to analyze existing programs.

15.1. Basic mathematical functions

Module *Math* contains the basic standard functions that had been postulated already in 1960 by Algol 60. They are

sqrt(x)	the square root
exp(x)	the exponential function
ln(x)	the natural logarithm
sin(x)	the sine function
cos(x)	the cosine function

They are presented here only briefly without discussing their approximation methods. However, we point out how advantage can be taken from knowledge about the internal representation of floating-point numbers.

15.1.1. Conversion between integers and floating-point numbers

The Oberon System adopts the standard format postulated by IEEE. Here we restrict it to the 32-bit variant. A floating-point number *x* consists of 3 parts

s	the sign	1 bit
e	the exponent	8 bits
m	the mantissa	23 bits

Its value is defined as $x = (-1)^s \times 2^{e+127} \times (1.m)$. A number is in normalized form, if its mantissa satisfies $1.0 \leq m < 2.0$. It is assumed that numbers are always normalized, and therefore the leading 1-bit is omitted. The exception is the singular value 0, which cannot be normalized. It must therefore be treated as a special case.

It follows that integers and floating-point numbers are represented quite differently, and that conversion operations are necessary to transfer a number from one format to the other. This is the reason why the Oberon language keeps the two types INTEGER and REAL separate. Conversion must be explicitly specified by using the two predefined functions

n := FLOOR(x)	REAL→ INTEGER
x := FLT(n)	INTEGER → REAL

Note: FLOOR(x) rounds toward -inf. For example FLOOR(1.5) = 1, FLOOR(-1.5) = -2.

The RISC processor does not feature specific instructions implementing these functions. Instead, the compiler generates inline code using the FAD instruction with special options suppressing normalization. This option is specified by the *u* and *v* modifier bits of the instruction.

The FLOOR function is realized by adding 0 with an exponent of 127 + 24 and suppressing the insertion of a leading 1-bit (u = 1). This causes the mantissa of the argument to be shifted right until its exponent is equal to 151. The RISC instructions are:

```
MOV'    R1 R0  4B00H            R1 := 4B000000H
FAD'    R0 R0 R1
```

The FLT function is implemented also by adding 0 with an exponent of 151 and forced insertion of a leading 1-bit (v = 1).

```
MOV'    R1 R0  4B00H
FAD"    R0 R0 R1
```

There are two predefined procedures for packing and unpacking a floating-point number:

PACK(x, e) x := x × 2^e (for x > 0)

UNPK(x, e) assign to *x* and *e*, such that x×2^e = x0, where *x0* is the original value of *x*, *x* becomes normalized, that is 1.0 ≤ x < 2.0

Assuming R0 = x and R1 = e, the instruction sequence for PACK(x, e) is

```
LSL     R1 R1    23
ADD     R0 R0 R1
STR     R0 x
```

Again assuming x = R0, the instruction sequence for UNPK(x, e) is

```
ASR     R1 R0    23
SUB     R1 R1   127
STR     R1 e
LSL     R1 R1    23
SUB     R0 R0 R1
STR     R0 x
```

15.1.2. The square root function

We rely on the definition x = 2^e×m Using the intrinsic UNPK procedure, the components *m* and *e* are obtained from *x*. Then the square root is computed according to the formulas

sqrt(x) = $2^{(e \text{ DIV } 2)}$ × sqrt(m) if e is even,

sqrt(x) = $2^{(e \text{ DIV } 2 - 1)}$ × sqrt(2) × sqrt(m) if e is odd.

The advantage is that the argument of the square root now lies in the narrow interval [1.0, 2.0], and therefore is easier and faster to approximate by a continued fraction.

```
PROCEDURE sqrt(x: REAL): REAL;
  CONST c1 = 0.70710680;   (* 1/sqrt(2) *)
    c2 = 0.590162067;
    c3 = 1.4142135;  (*sqrt(2)*)
  VAR s: REAL; e: INTEGER;
BEGIN ASSERT(x >= 0.0);
  IF x > 0.0 THEN
    UNPK(x, e);
    s := c2*(x+c1);
    s := s + (x/s);
    s := 0.25*s + x/s;
    s := 0.5 * (s + x/s);
    IF ODD(e) THEN s := c3*s END ;
    PACK(s, e DIV 2)
  ELSE s := 0.0
  END ;
  RETURN s
END sqrt;
```

15.1.3. The exponential function

Since our floating-point format is based on an exponent of 2, we first use the formula

exp(x) = e^x = 2^y with y = x × $\log_2(e)$ = x / ln(2) $\log_2(e)$ = 1.4426951

and first compute *y*. We decompose *y* into its integral part n = FLOOR(y) and its fractional part y0 = y - n. Since 2^n × 2^{y0} = 2^{n+y0}, the result is the sum of the exponent *n* and the mantissa 2^{y0}. Again, the advantage of the decomposition is that the argument *y0* of the polynomial approximation lies in the narrow interval [1.0, 2.0].

```
PROCEDURE exp(x: REAL): REAL;
   CONST c1 = 1.4426951;  (*1/ln(2) *)
      p0 = 1.513864173E3;
      p1 = 2.020170000E1;
      p2 = 2.309432127E-2;
      q0 = 4.368088670E3;
      q1 = 2.331782320E2;
   VAR n: INTEGER; p, y, yy: REAL;
BEGIN y := c1*x;  (*1/ln(2)*)
   n := FLOOR(y + 0.5); y := y - FLT(n);
   yy := y*y;
   p := ((p2*yy + p1)*yy + p0)*y;
   p := p/((yy + q1)*yy + q0 - p) + 0.5;
   PACK(p, n+1); RETURN p
END exp;
```

15.1.4. The logarithm

Again we take advantage of the presence of an exponent in the floating-point representation and use the equations

$$\ln (a \times b) = \ln a + \ln b$$

$$\ln (2^e \times m) = \log_2(2^e \times m) \times \ln(2) = e \times \ln(2) + \ln m$$

```
PROCEDURE ln(x: REAL): REAL;
   CONST c1 = 0.70710680;   (* 1/sqrt(2) *)
         c2 =  0.69314720;  (* ln(2) *)
         p0 = -9.01746917E1;
         p1 =  9.34639006E1;
         p2 = -1.83278704E1;
         q0 = -4.50873458E1;
         q1 =  6.176106560E1;
         q2 = -2.07334879E1;
   VAR e: INTEGER; y: REAL;
BEGIN ASSERT(x > 0.0); UNPK(x, e);
   IF x < c1 THEN x := x*2.0; e := e-1 END ;
   x := (x - 1.0)/(x + 1.0);
   y := c2 * FLT(e) + x * ((p2*x + p1)*x + p0) / (((x + q2)*x + q1)*x + q0);
   RETURN y
END ln;
```

15.1.5. The sine function

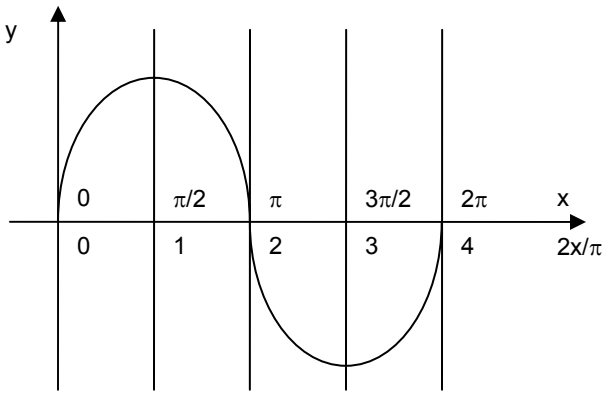


Figure 15.1 Sine function y = sin(x)

First, the argument *x* is transposed into the interval [0, π/4] by computing

n := FLOOR(y+0.5); y := (y - n)

and then distinguish between two approximating polynomials depending on whether x < π/4.

```
PROCEDURE sin(x: REAL): REAL;
  CONST c1 =  6.3661977E-1;   (*2/pi*)
    p0 =  7.8539816E-1;
    p1 = -8.0745512E-2;
    p2 =  2.4903946E-3;
    p3 = -3.6576204E-5;
    p4 =  3.1336162E-7;
    p5 = -1.7571493E-9;
    p6 =  6.8771004E-12;
    q0 =  9.9999999E-1;
    q1 = -3.0842514E-1;
    q2 =  1.5854344E-2;
    q3 = -3.2599189E-4;
    q4 =  3.5908591E-6;
    q5 = -2.4609457E-8;
    q6 =  1.1363813E-10;
  VAR n: INTEGER; y, yy, f: REAL;
BEGIN y := c1*x;
  IF y >= 0.0 THEN n := FLOOR(y + 0.5) ELSE n := FLOOR(y - 0.5) END ;
  y := (y - FLT(n)) * 2.0; yy := y*y;
  IF ODD(n) THEN f := (((((q6*yy + q5)*yy + q4)*yy + q3)*yy + q2)*yy + q1)*yy + q0
  ELSE f := ((((((p6*yy + p5)*yy + p4)*yy + p3)*yy + p2)*yy + p1)*yy + p0)*y
  END ;
  IF ODD(n DIV 2) THEN f := -f END ;
  RETURN f
END sin;
```

15.2. A data link

Module *PCLink* serves to transfer data (files) to and from another system. Data are transmitted as a sequence of blocks. Each block is a sequence of bytes. The number of data bytes lies between 0 and 255. They are preceded by a single byte indicating the length. Blocks are 255 bytes long, except the last block, whose length is less than 255.

Here, the transmission channel is an RS-232 line. The interface consists of two registers, one for a data byte (address = -56), and one for the status (address = -52). Bit 0 of this status register indicates, whether a byte had been received. Bit 1 of the status register indicates, whether the byte in the data register had been sent. (Note: the default transmission rate of the RISC is 9600 bit/s).

This module represents a server running as an Oberon task which must be activated by the command *Run*. A server running on the partner system must be the master issuing requests. The command sequence is a REC byte, a SND byte, or a REQ byte (for testing the connection). REC and SND must be followed by a file name, and the sequence of blocks.

Every block is acknowledged by the receiver sending an ACK byte, for which the sender waits before sending the next block. There is no synchronization within blocks. Because writing bytes onto a file may involve operations of unpredictable duration, the received bytes are not written to the file immediately. They are buffered and only output after the entire block had been received.

```
MODULE PCLink;  (*NW 8.2.2013  for Oberon on RISC*)
  IMPORT SYSTEM, Files, Texts, Oberon;
  CONST data = -56; stat = -52;
    BlkLen = 255;
    REQ = 20H; REC = 21H; SND = 22H; ACK = 10H; NAK = 11H;

  VAR T: Oberon.Task; W: Texts.Writer;
  PROCEDURE Rec(VAR x: BYTE);
  BEGIN
```

```
   REPEAT UNTIL SYSTEM.BIT(stat, 0);
   SYSTEM.GET(data, x)
END Rec;

PROCEDURE RecName(VAR s: ARRAY OF CHAR);
   VAR i: INTEGER; x: BYTE;
BEGIN i := 0; Rec(x);
   WHILE x > 0 DO s[i] := CHR(x); INC(i); Rec(x) END ;
   s[i] := 0X
END RecName;

PROCEDURE Send(x: BYTE);
BEGIN
   REPEAT UNTIL SYSTEM.BIT(stat, 1);
   SYSTEM.PUT(data, x)
END Send;

PROCEDURE Task;
   VAR len, n, i: INTEGER;
      x, ack, len1, code: BYTE;
      name: ARRAY 32 OF CHAR;
      F: Files.File; R: Files.Rider;
      buf: ARRAY 256 OF BYTE;
BEGIN
   IF  SYSTEM.BIT(stat, 0) THEN (*byte available*)
      Rec(code);
         IF code = SND THEN  (*send file*)
            RecName(name); F := Files.Old(name);
            IF F # NIL THEN
               Send(ACK); len := Files.Length(F); Files.Set(R, F, 0);
               REPEAT
                  IF len >= BlkLen THEN len1 := BlkLen ELSE len1 := len END ;
                  Send(len1); n := len1; len := len - len1;
                  WHILE n > 0 DO Files.ReadByte(R, x); Send(x); DEC(n) END ;
                  IF ack # ACK THEN  len := 0 END
               UNTIL len1 < BlkLen
            ELSE Send(11H)
            END
         ELSIF code = REC THEN (*receive file*)
            RecName(name); F := Files.New(name);
            IF F # NIL THEN
               Files.Set(R, F, 0); Send(ACK);
               REPEAT Rec(x); len := x; i := 0;
                  WHILE i < len DO Rec(x); buf[i] := x; INC(i) END ;
                  i := 0;
                  WHILE i < len DO Files.WriteByte(R, buf[i]); INC(i) END ;
                  Send(ACK)
               UNTIL len < 255;
               Files.Register(F); Send(ACK)
            ELSE Send(NAK)
            END
         ELSIF code = REQ THEN Send(ACK) (*for testing*)
         END
   END
END Task;

PROCEDURE Run*;
BEGIN Oberon.Install(T); Texts.WriteString(W, "PCLink started");
   Texts.WriteLn(W); Texts.Append(Oberon.Log, W.buf)
END Run;

PROCEDURE Stop*;
```

```
    BEGIN Oberon.Remove(T); Texts.WriteString(W, "PCLink stopped");
      Texts.WriteLn(W); Texts.Append(Oberon.Log, W.buf)
    END Stop;

BEGIN Texts.OpenWriter(W); T := Oberon.NewTask(Task, 0)
END PCLink.
```

15.3. A generator of graphic macros

The module *MacroTool* serves to create macros for the graphic system (Ch. 13). It provides the commands *OpenMacro, MakeMacro, LoadLibrary* and *StoreLibrary.*

OpenMacro decomposes the selected macro into its elements and places them at the position of the caret. This command is typically the first if an existing macro is to be modified.

MakeMacro L M collects all selected objects in the frame designated by the star pointer and unites them into macro *M*. This macro is displayed at the caret position and inserted into library *L*. If no such library exists, a new one is created.

LoadLibrary L loads the library *L* (under file name *L.Lib).* Note that a library must have been stored, before it can be loaded.

StoreLibrary stores library *L* (with filename *L.Lib).*

The required modules are *Texts, Oberon, Graphics, GraphicFrames.*

16 Implementation of the RISC processor

16.1. Introduction

The design of the processor to be described here in detail was guided by two intentions. The first was to present an architecture that is distinct in its regularity, minimal in the number of features, yet complete and realistic. It should be ideal to present and explain the main principles of processors. In particular, it should connect the subjects of architectural and compiler design, of hardware and software, which are so closely interconnected.

Clearly “real”, commercial processors are far more complex than the one presented here. We concentrate on the fundamental concepts rather than on their elaboration. We strive for a fair degree of completeness of facilities, but refrain from their “optimization”. In fact, the dominant part of the vast size and complexity of modern processors and software is due to speed-up called optimization. It is the main culprit in obfuscating the basic principles, making them hard, if not impossible to study. In this light, the choice of a RISC (Reduced Instruction Set Computer) is obvious.

The use of an FPGA provides a substantial amount of freedom for design. Yet, the hardware designer must be much more aware of availability of resources and of limitations than the software developer. Also, timing is a concern that usually does not occur in software, but pops up unavoidably in circuit design. Nowadays circuits are no longer described in terms of elaborate diagrams, but rather as a formal text. This lets circuit and program design appear quite similar. The circuit description language – we here use *Verilog* – appears almost the same as a programming language. But one must be aware that differences still exist, the main one being that in software we create mostly sequential processes, whereas in hardware everything “runs” concurrently. However, the presence of a language – a textual definition – is an enormous advantage over graphical schemata. Even more so are systems (tools) that compile such texts into circuits, taking over the arduous task of placing components and connecting them (routing). This holds in particular for FPGAs, where components and wires connecting them are limited, and routing is a very difficult and time-consuming matter.

The development of this RISC progressed through several stages. The first was the design of the architecture itself, (more or less) independent of subsequent implementation considerations. Then followed a first implementation called RISC-0. For this a *Harvard Architecture* was chosen, implying that two distinct memories are used for program and for data. For both chip-internal *block RAMs* were used. The Harvard architecture allows for a neat separation of the arithmetic from the control unit.

But these blocks of RAM are relatively small on the used Spartan-3 development board (1 - 4K words). This board, however, provides also an FPGA-external static RAM with a capacity of 1 MByte. In a second effort, the BRAM for data was replaced by this SRAM. Both instructions and data are placed into the SRAM, resulting in a *von Neumann* architecture.

The RISC hardware is characterized by three interfaces. The first is the programmer's interface, the architecture, that is, those aspects that are relevant to the programmer, in particular, the instruction set. It is described in Appendix A2. The second is the hardware interface between the processor core and its environment, described here. The third is that which connects the environment with physical devices such as memory, keyboard and display. This is described in Chapter 17.

```
module RISC5(
   input clk, rst, stallX,
   input [31:0] inbus, codebus,
   output [19:0] adr,                  // memory and device addresses
   output rd, wr, ben,                 // read, write,  byte enable control signals for memory
   output [31:0] outbus);
```

The main parts of the hardware interface are three busses, the data input and output busses, the code bus, and the address bus. Signals *rd* and *wr* indicate, whether a read or a write operation is to be performed. *ben* indicates a byte (rather than word) access. The entire processor operates synchronously on the clock *clk* (25 MHz on Spartan-3), *rst* is the reset signal (from a push button on the development board), and *stall* is the input to stall the processor.

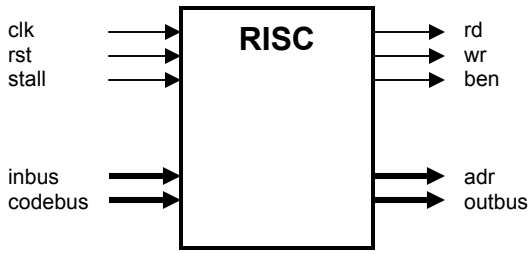


Figure 16.1 The processor's interface

First we concentrate on the implementation of the processor core, its realization in the form of circuits. They are divided into two parts, the arithmetic/logic unit processing data, and the control unit determining the flow of instructions.

16.2. The arithmetic and logic unit

The ALU features a bank of 16 registers with 32 bit *words*. Arithmetic and logical operations, represented by instructions, always operate on these registers. Data can be transferred between memory and registers by separate load and store instructions. This is an important characteristic of RISC architectures, developed between 1975 and 1985. It contrasts with the earlier CISC architectures (Complex instruction set): Memory is largely decoupled from the processor. A second important characteristic is that *most instructions take a single clock cycle* (25 MHz) for their execution. The exceptions are access to memory, multiplication and division.. More about this will be presented later. This single-cycle rule makes such processors *predictable* in performance. The number of cycles and the time required for executing any instruction sequence is precisely defined. Predictability is essential in all real-time applications.

The data processing unit consisting of ALU and registers is shown in Figure 16.2. Evidently, data cycle from registers through the ALU, where an operation is performed, and the result is deposited back into a register. The ALU embodies the circuits for arithmetic operations, logical operations, and shifts. The operations available are listed below. They are described in more detail in Appendix A2. The operand *n* is either a register or a part of the instruction itself.

0	MOV a, n	R.a := n	
1	LSL a, b, n	R.a := R.b ← n	(shift left by n bits)
2	ASR a, b, n	R.a := R.b → n	(shift right by n bits with sign extension)
3	ROR a, b, n	R.a := R.b rot n	(rotate right by n bits)
4	AND a, b, n	R.a := R.b & n	logical operations
5	ANN a, b, n	R.a := R.b & ~n	
6	IOR a, b, n	R.a := R.b or n	inclusive or
7	XOR a, b, n	R.a := R.b xor n	exclusive or
8	ADD a, b, n	R.a := R.b + n	integer arithmetic
9	SUB a, b, n	R.a := R.b – n	
10	MUL a, b, n	R.a := R.b x n	
11	DIV a, b, n	R.a := R.b div n	

```
12    FAD  a, b, c    R.a := R.b + R.c          floating-point arithmetic
13    FSB  a, b, c    R.a := R.b – R.c
14    FML  a, b, c    R.a := R.b x R.c
15    FDV  a, b, c    R.a := R.b / R.c
```

The following excerpt describes the essence of the ALU circuits. It is written in the HDL Verilog and refers to the following wires and registers.

```
wire [31:0] IR;
wire p, q, u, v, w;          // instruction fields IR[31], IR[30], IR[29], IR[28], IR[16]
wire [3:0] op, ira, irb, irc; // instruction fields IR[19:16], IR[27:24], IR[23:20], IR[3:0]
wire [15:0] imm;             // instruction field IR[15:0]

wire [31:0] A, B, C0, C1, regmux;
wire [31:0]  s3, t3, quotient, fsum, fprod, fquot;
wire [32:0] aluRes;
wire [63:0] product;

reg [31:0] R [0:15];         // array of 16 registers
reg  N, Z, C, OV;            // condition flags
```

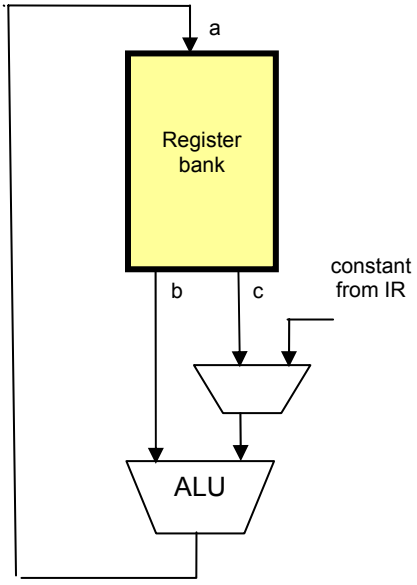


Fig. 16.2. Processor core with ALU and registers

B and C0 are the outputs from the register bank, and A is its input. The register numbers *ira* for port A, *irb* for port B, and *irc* for port C0 are taken from 4-bit fields of the instruction register IR. C1 is the multiplexer selecting among the register output C0 and the immediate field *imm*. *s3* and *t3* are outputs of the shift units (Sect. 16.2.1). *product* is the output of the multiplier (16.2.2), *quotient* and *remainder* those of the divider (16.2.3), *fsum* that of the floating-point adder (16.2.4), *fprod* that of the floating-point multiplier (16.2.5), and *fquot* the output of the floating-point divider (16.2.6).

```
assign A = R[ira];
assign B = R[irb];
assign C0 = R[irc];
assign C1 = q ? {{16{v}}, imm} : C0;
```

The following represents the main instruction decoding and selection of results. The opcodes refer to specific values of fields *p* and *op* of IR. Note that *if x then y else z* is denoted in Verilog by *x ? y : z.*

```
assign aluRes =
```

```
MOV ?     (q ? (~u ? {{16{v}}, imm} : {imm, 16'b0}) :
                (~u ? C0 : (~irc[0] ? H : {N, Z, C, OV, 20'b0, 8'b01010000}))) :
LSL ?           t3 :                    // output of left shift unit
(ASR|ROR) ?  s3 :                       // output of right shift unit
AND ?           B & C1 :
ANN ?           B & ~C1 :
IOR  ?          B | C1 :
XOR ?           B ^ C1 :
ADD ?           B + C1 + (u & C) :
SUB ?           B - C1 - (u & C) :
MUL ?           product [31:0] :        // output of multiplier
DIV ?           quotient :
(FAD|FSB) ?  fsum :
FML ?           fprod :
FDV ?           fquot : 0;
```

The input to the register bank, *regmux,* is selected from either *alures, inbus* (for LDR instructions), or the program address *nxpc* (for branch and link instructions). The signal *regwr* determines, whether data are to be stored (written) into the register bank. Details must be gathered from the respective program listing RISC.v.

```
always @ (posedge clk) begin
   R[ira] <= regwr ? regmux : A;
   N <= regwr ? regmux[31] :  N;
   Z <= regwr ? (regmux == 0) : Z;
    C <= (ADD|SUB) ? aluRes[32] : C;
   OV <= (ADD|SUB) ? aluRes[32] ^ aluRed[31] : OV ;
end
```

Whenever a register is written, the condition flags are also affected. They are N (*aluRes* negative), *Z* (*aluRes* zero), C (carry), and OV (overflow). The latter apply only to addition and subtraction.

16.2.1 Shifters

Shifters are multi-way multiplexers. For a 32-bit word, the simplest solution would be 32 32-way multiplexers. But this is hardly economical. On the FPGA used here, 4-way muxes are basic cells. It is therefore beneficial, to compose a shifter out of 4-way muxes. Now the obvious solution is to use 3 levels of muxes through which data flow. The first level shifts by amounts of 0, 1, 2, or 3, the second by amounts of 0, 4, 8, 12, and the third by 0 or 16. This scheme is programmed as follows for left shifts (instruction LSL) with B as input, *sc0* = C1[1:0] and *sc1* = C1[3:2] as shift counts, and *t3* as output:

```
assign t1 =   (sc0 == 3) ? {B[28:0], 3'b0} :
              (sc0 == 2) ? {B[29:0], 2'b0} :
              (sc0 == 1) ? {B[30:0], 1'b0} : B;
assign t2 =   (sc1 == 3) ? {t1[19:0], 12'b0} :
              (sc1 == 2) ? {t1[23:0], 8'b0} :
              (sc1 == 1) ? {t1[27:0], 4'b0} : t1;
assign t3 =   C1[4] ? {t2[15:0], 16'b0} : t2;
```

The solution for right shifts is analogous. An additional level of multiplexing is required, shifting in either the sign bit (ASR with sign propagation) or bits from the low end of the word (ROR), making a barrel shifter. This selection is controlled by the instruction bit w = IR[16].

16.2.2. Multiplication

Multiplication is an inherently more complex operation than addition and subtraction. After all, multiplication can be composed (of a sequence) of additions. There are many methods to implement multiplication, all – of course – based on the same concept of a series of additions. They show the fundamental problem of trade-off between time and space (circuitry). Some solutions operate with a minimum of circuitry, namely a single adder used for all 32 additions executed sequentially (in time). They obviously sacrifice speed. The other extreme is multiplication

in a single cycle, using 32 adders in series (in space). This solution is fast, but the amount of required circuitry is high..

Before we present the sequential solution, let us briefly recapitulate the basics of a multiplication *p := x × y*. Here *p* is the product, *x* the multiplier, and *y* the multiplicand. Let *x* and *y* be *unsigned* integers. Consider *x* in binary form.

$$x = x_{31}\times 2^{31} + x_{30}\times 2^{30} + \ldots + x_1\times 2^1 + x_0\times 2^0$$

Evidently, the product is the sum of 32 terms of the form $x_k\times 2^k\times y$, i.e. of *y* left shifted by *k* positions multiplied by x_k. Since x_k is either 0 or 1, the product is either 0 or *y* (shifted). Multiplication is thus performed by an adder and a selector. The selector is controlled by x_k, a bit of the multiplier. Instead of selecting this bit among $x_0 \ldots x_{31}$, we right shift x by one bit in each step. Then the selection is always according to x_0. The *add-shift step* then is

```
IF ODD(x) THEN p := p + y END ;
y := 2*y; x := x DIV 2
```

whereby multiplication by 2 is done by a left shift, and division by 2 by a right shift: As an example, consider the multiplication of two 4-bit integers x = 5 and y = 3, requiring 4 steps:

	p	x	y	
	0000'0000	0101	0000'0011	
add y to p	0000'0011	010**1**	0000'0011	
shift	0000'0011	0010	0000'0110	
add 0 to p	0000'0011	001**0**	0000'0110	
shift	0000'0011	0001	0000'1100	
add y to p	0000'1111	000**1**	0000'1100	
shift	0000'1111	0000	0001'1000	
add 0 to p	0000'1111	000**0**	0001'1000	
shift	0000'1111	0000	0011'0000	p = 15

The shifting of *x* to the right also suggests that instead of shifting *y* to the left in each step, we keep *y* in the same position and shift the partial sum *p* to the right. We notice that the size of *x* decreases by 1 in each step, whereas the size of *p* increases by 1. This allows to pack *p* and *x* into a single double register <B, A> with a shifting border line. At the end, it contains the product p = x × y.

	p	x	
	0000	0101	
add y to p	00**11**	010**1**	
shift	00011	010	
add 0 to p	00011	01**0**	
shift	000011	01	
add y to p	00**11**11	0**1**	
shift	0001111	0	
add 0 to p	0001111	**0**	
shift	00001111		p = 15

p = {B[31:0], A{31:[32-k]}, x = A[31-k:0] k = 0 … 31

The multiplier is controlled by a rudimentary state machine S, actually a simple 5-bit counter running from 0 to 31. The multiplier is shown schematically in Figure 16.3.

The multiplier interprets its operands as signed (*u* = 0) or unsigned (*u* = 1) integers. The difference between unsigned and signed representation is that in the former case the first term has a negative weight ($-x_{31}\times 2^{31}$). Therefore, implementation of signed multiplication requires very little change: Term 31 is subtracted instead of added (see complete program listing below).

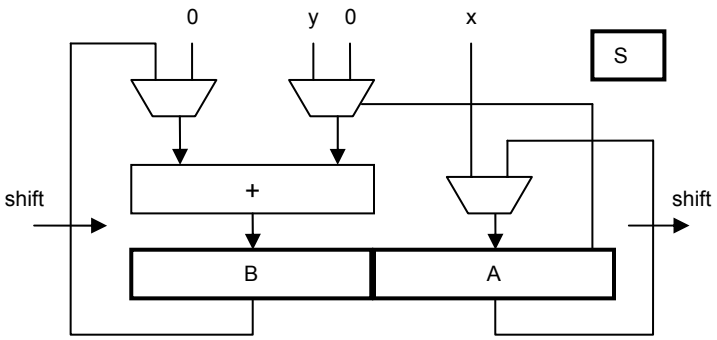


Figure 16.3. Schematic of multiplier

During execution of the 32 add-shift steps the processor must be stalled. The process proceeds and the counter S advances as long as the input MUL is active (high). MUL indicates that the current operation is a multiplication, and the signal is stable until the processor advances to the next instruction. This happens when step 31 is reached (Figure 16.4).

```
stall  =  MUL & ~(S == 31);
S  <=  MUL ? S+1 : 0;
```

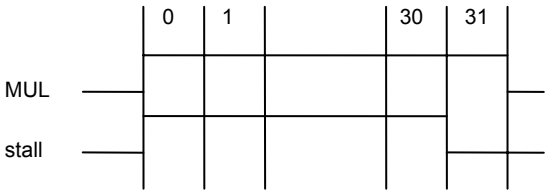


Figure 16.4. Generating *stall*

The details of the simple multiplier are listed below:

```
module Multiplier(
  input CLK, MUL, u,
  output stall,
  input [31:0] x, y,
  output [63:0] z);

reg [4:0] S;    // state
reg [31:0] B2, A2;  // high and low parts of partial product
wire [32:0] B0, B00, B01;
wire [31:0] B1, A0, A1;

assign stall = MUL & ~(S == 31);
assign B00 = (S == 0) ? 0 : {B2[31] & u, B2};
assign B01 = A0[0] ? {y[31] & u, y} : 0;
assign B0 = ((S == 31) & u) ? B00 - B01 : B00 + B01;
assign B1 = B0[32:1];
assign A0 = (S == 0) ? x : A2;
assign A1 = {B0[0], A0[31:1]};
assign z = {B1, A1};

always @ (posedge(CLK)) begin
  B2 <= B1; A2 <= A1;
  S <= MUL ? S+1 : 0;
end
endmodule
```

Implementing multiplication in hardware made the operation about 30 times faster than its solution by software. A significant factor! As multiplication is a relatively rare operation – at least in comparison with addition and subtraction – early RISC designs (MIPS, SPARC, ARM) refrained from its full implementation in hardware. Instead, an instruction called *multiply step* was provided, performing a single add-shift step in one clock cycle. A multiplication was then programmed by a sequence of 32 step instructions, typically provided as a subroutine. This measure of economy was abandoned, when hardware became faster and cheaper.

The FPGA used on the Spartan-3 board features a welcome facility for speeding up multiplication, namely fast 18 x 18 bit multiplier units. These are made available as basic cells of the FPGA, and they multiply in a single clock cycle. Considering an operand $x = x_1 \times 2^{16} + x_0$, the product is obtained as the sum of only 4 terms:

$$p = x \times y = x_1 \times y_1 \times 2^{32} + (x_0 \times y_1 + x_1 \times y_0) \times 2^{16} + x_0 \times y_0$$

Thereby multiplication of two 32-bit integers can be performed in 2 cycles only, one for multiplications, one for addition. Four multipliers are needed. For details, the reader is referred to the program listing (module *Multiplier1*).

16.2.3. Division

Division is similar to multiplication in structure, but slightly more complicated. We present its implementation by a sequence of 32 *shift-subtract steps*, the complement of add-shift. We here discuss division of *unsigned* integers only.

$$q = x \text{ DIV } y \qquad r = x \text{ MOD } y$$

q is the *quotient*, *r* the *remainder*. These are defined by the invariants

$$x = q \times y + r \qquad \text{with} \qquad 0 \le r < y$$

Both *q* and *r* are held in registers. Initially we set r to x, the dividend, and then subtract multiples of y (the divisor) from it, each time checking that the result is not negative. This *shift-subtract step* is

```
r := 2*r; q := 2*q;
IF r – y ≥ 0 THEN r := r – y END
```

As an example, consider the division of the 8-bit integer *x* = 14 by the 4-bit integer *y* = 4, where multiplication and division by 2 are done by shifts:

	r	q	y	
	0000'1110	0000	0001'1000	
shift	0000'1110	0000	0001'1000	r < y
sub 0 from r	0000'1110	0000	0000'1100	
shift	0000'1110	0000	0000'1100	r >= y
sub y from r	0000'0010	000**1**	0000'1100	
shift	0000'0010	0010	0000'0110	r < y
sub 0 from r	0000'0010	0010	0000'0110	
shift	0000'0010	0100	0000'0011	r < y
sub y from r	0000'0010	0100	0000'0011	q = 4, r = 2

As with multiplication this arrangement may be simplified by putting *r* and *q* into a double-length shift register, and by shifting *r* to the left instead of *y* to the right. This results in

	r	q	
	0000'1110		
shift	0001'110	0	r < Y
sub 0 from r	0001'110	0	
shift	0011'10	00	r >= Y
sub y from r	0000'10	0**1**	
shift	0001'0	010	r < Y
sub 0 from r	0001'0	010	

shift	0010	0100	r < Y
sub 0 from r	0010	0100	q = 4, r = 2

This scheme is represented by the circuit shown in Figure 16.5.

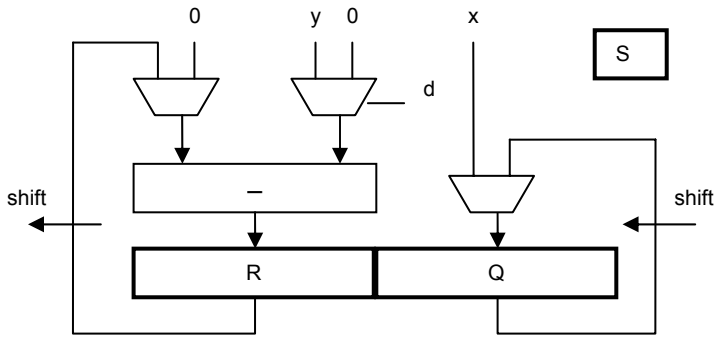


Figure 16.5. Schematic of divider

Stall generation is the same as for the multiplier. A division takes 32 clock cycles. Further details are shown in the subsequent program listing.

```
module Divider(
  input clk, DIV,
  output stall,
  input [31:0] x, y,
  output [31:0] quot, rem);

reg [4:0] S;  // state
reg [31:0] r3, q2;
wire [31:0] r0, r1, r2, q0, q1, d;

assign stall = DIV & ~(S == 31);
assign r0 = (S == 0) ? 0 : r3;
assign d = r1 - y;
assign r1 = {r0[30:0], q0[31]};
assign r2 = d[31] ? r1 : d;
assign q0 = (S == 0) ? x : q2;
assign q1 = {q0[30:0], ~d[31]};
assign rem = r2;
assign quot = q1;

always @ (posedge(clk)) begin
  r3 <= r2; q2 <= q1;
  S <= DIV ? S+1 : 0;
end
endmodule
```

16.3. Floating-point arithmetic

The RISC uses the IEEE Standard for representing REAL (floating-point) numbers with 32 bits. The word is divided into 3 fields: *s* for the sign, *e* for the exponent, and *m* for the mantissa. The value is

$$x = (-1)^s \times 2^{e-127} \times 1.m \quad \text{with} \quad 1.0 \le m < 2.0 \quad \text{(normalized form)}$$

Numbers are represented in sign-magnitude form. This implies that for sign inversion only the sign bit must be inverted, and exponent and mantissa remain unchanged.

Zero is a special case represented by 32 0-bits, and therefore has to be treated separately. Furthermore, e = 255 denotes "not a number". It is generated in the case of arithmetic overflow.

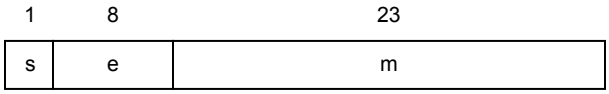


Figure 16.6 IEEE standard floating-point representation of REAL numbers

16.3.1. Floating-point addition

If two numbers are to be added, they must have the same exponent. This implies that the summand with the smaller exponent must be denormalized. *m* is shifted to the right and *e* is incremented accordingly. That is, if *d* is the difference of the two exponents, *m* is multiplied by 2^d, and *e* is incremented by d. After the addition, the sum must be rounded and post-normalized. *m* is shifted to the left and *e* is decremented accordingly. The shift amount is determined by the position of the leftmost one-bit. This results in the scheme shown in Figure 16.7, and the module's interface is

```
module FPAdder(
  input clk, run, u, v,
  input [31:0] x, y,
  output stall,
  output [31:0] z);
```

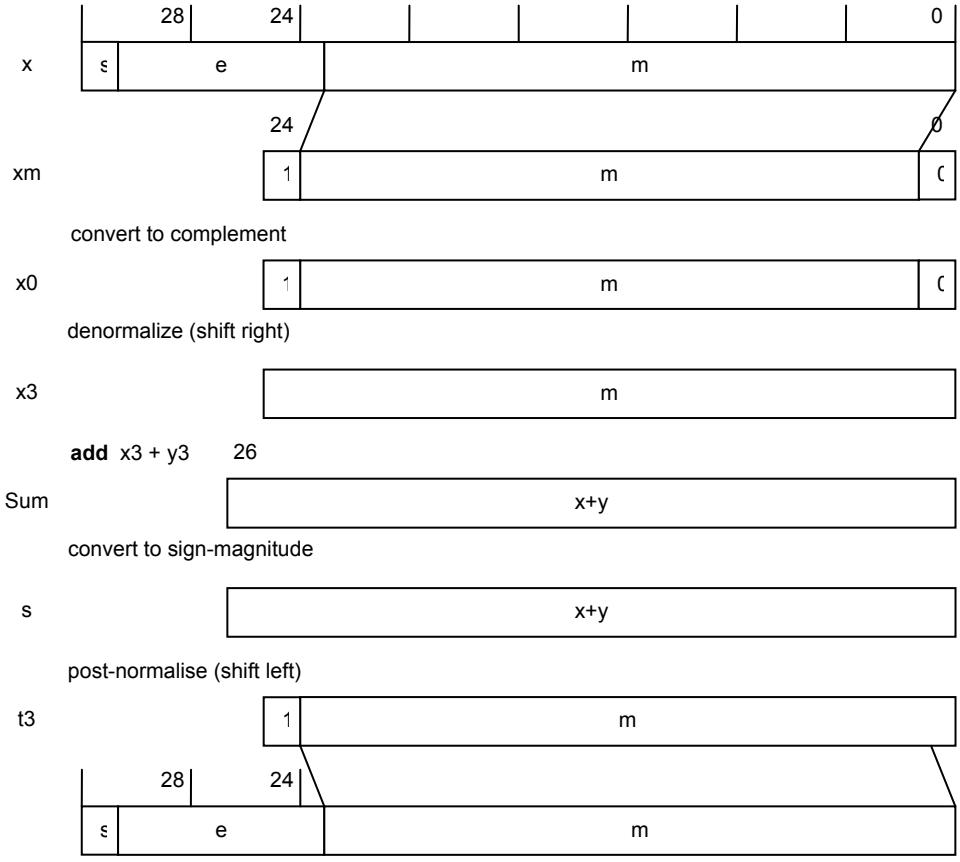


Figure 16.7 Steps of floating-point addition

It is important to achieve proper rounding. This is done by extending the mantissa of both operands by a guard bit, initialized to 0. A one is added (effectively 0.5) and at the end the guard bit is discarded.

The two predefined conversion functions FLT and FLOOR are conveniently implemented as additions. A denormalized 0 is added to the argument, effecting the proper shift. In the case of FLT (modifier bit u = 1), denormalization is omitted (no 1-bit inserted), and in the case of FLOOR (modifier bit v = 1), post-normalization is suppressed.

16.3.2. Floating-point multiplication

A product is given by the equation

$$p = x \times y = (2^{xe} \times xm) \times (2^{ye} \times ym) = 2^{xe+ye} \times (xm * ym)$$

$$p = (xs, xe, xm) \times (ys, ye, ym) = (xs \text{ xor } ys, xe + ye, xm \times ym)$$

That is, exponents are added, mantissas multiplied. Denormalization is not needed. Post-normalization is a right shift of at most one bit, because if $1.0 \leq xm, ym < 2.0$, the result satisfies $1.0 \leq xm*ym < 4.0$. The sign of the product is the exclusive or of the signs of the arguments. The multiplier module's interface is

```
module FPMultiplier(
  input clk, run,
  input [31:0] x, y,
  output stall,
  output [31:0] z);
```

16.3.3. Floating-point division

A quotient is given by the equation

$$q = x / y = (2^{xe} \times xm) / (2^{ye} \times ym) = 2^{xe-ye} \times (xm / ym)$$

$$q = (sx, ex, mx) / (sy, ey, my) = (sx \text{ xor } sy, ex - ey, mx / my)$$

That is, exponents are subtracted, mantissas divided. Denormalization is not needed. Post-normalization requires a left shift by at most a single bit, because if $1.0 \leq xm, ym < 2.0$, the result satisfies $0.5 \leq xm/ym < 2.0$. The sign of the product is the exclusive or of the signs of the arguments. The divider module's interfaces is

```
module FPDivider(
   input clk, run,
   input [31:0] x, y,
   output stall,
   output [31:0] z);
```

16.4. The Control Unit

The control unit determines the sequence of executed instructions. It contains two registers, the program counter PC holding the address of the current instruction, and the current instruction register IR holding the instruction currently being interpreted. Instructions are obtained from memory through the *codebus* (see interface), from where the decoding signals emanate. Mostly, the arithmetic unit and the control unit operate concurrently (in parallel). While the arithmetic unit performs the operation held in register IR and data signals flow through the ALU, the control unit fetches in the same clock cycle the next instruction from memory in the location with the address held in PC. Next address and next instruction are latched in the registers at the end of a cycle. This scheme constitutes a one-element pipeline of instructions.

The principal task of the control unit is to generate the address of the next instruction. There are essentially only four cases:

0. Zero on reset.

1. The next instructions address is PC+1 (all instructions except branches)
2. The branch target PC+1 + offset. (Branch instructions).
3. It is taken from a data register. (This is used for returning from procedures).

This is reflected by the following program text, and shown in Figure 16.8.

```
reg [17:0] PC;
reg [31:0] IRBuf;
wire [31:0] IR;
wire [31:0] pmout;
wire [17:0] pcmux, nxpc;
wire cond;

IR = codebus;
nxpc = PC + 1;
pcmux = (~rst) ? 0 :
        (stall) ? PC :  // stall
        (BR & cond & u) ? off + nxpc :
        (BR & cond & ~u) ? C0[19:2]  :
        nxpc;

always @ (posedge clk)  PC <= pcmux; end
```

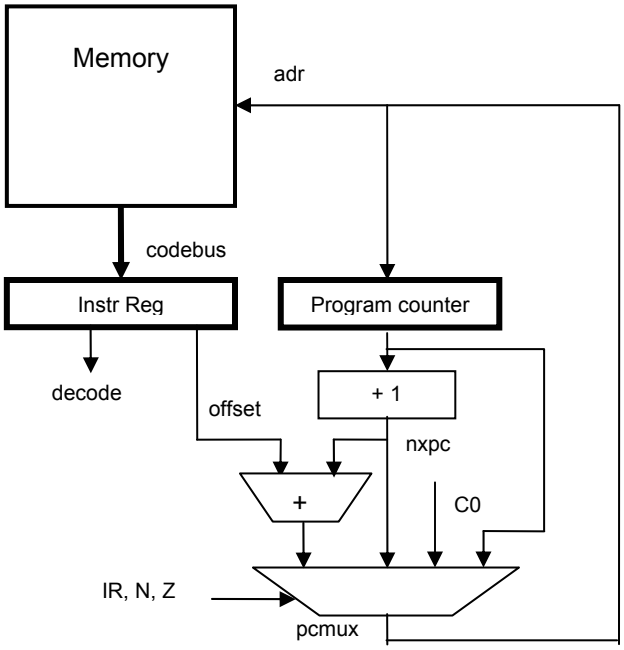


Fig. 16.8. The control unit

Branches are the only conditional instructions. Whether a branch is taken or not, is determined by the combination of the condition flags selected by the condition code field of the branch instruction. IR[27] is the condition sense inversion bit.

```
reg N, Z, C, OV;  // condition flags
wire S;
assign S = N ^ OV;
assign cond = IR[27] ^
    ((cc == 0) & N |          // MI, PL
    (cc == 1) & Z |           // EQ, NE
    (cc == 2) & C |           // CS, CC
    (cc == 3) & OV |          // VS, VC
    (cc == 4) & (C|Z) |       // LS, HI
    (cc == 5) & S |           // LT, GE
```

```
(cc == 6) & (S|Z) |        // LE, GT
(cc == 7));                // T, F
```

There is, unfortunately, a complication obfuscating the simple scheme presented so far. It stems from the necessity to initialize the processor. Only registers and memory blocks (BRAM) can be initialized and loaded by the available FPGA-tools. How, then, is a program (in our case the boot loader) moved into memory, the chip-external SRAM? The following scheme has been chosen:

The initial program is loaded into a BRAM (1K x 32). This block is memory-mapped into high-end addresses in the range of the data stack. On startup, the flag *PMsel* is set and IR is loaded from *pmout* (from the BRAM) at *StartAdr*. At the end of the program (boot loader), a branch instruction with destination 0 jumps to the beginning of the program that had just been loaded into SRAM by the boot loader. This is, presumably, but not necessarily, the operating system. The following changes and additions are required:

```
localparam StartAdr = 18'b111111100000000000; // 0FE000H

reg PMsel;  // memory select for instruction fetch
reg [31:0] IRBuf;

dbram32 PM (  // BRAM
   .clka (clk),
   .rdb (pmout),  // output port
   .ab (pcmux[10:0]));  // address

assign IR = PMsel ? pmout : IRBuf;

always @ (posedge clk) begin
    PMsel <= ~rst | (pcmux[17:11] == 7'b1111111);
    IRBuf <= stall ? IRBuf : codebus;
    ...
end ;
```

17 The processor's environment

The RISC processor is embedded in an environment (module RISCTop.v) connecting it with elements that are FPGA-chip external, but whose are provided on the Spartan development board (Figure 17.1). The environment consists of an address decoder, a data multiplexer, and interfaces to the memory and peripheral devices..

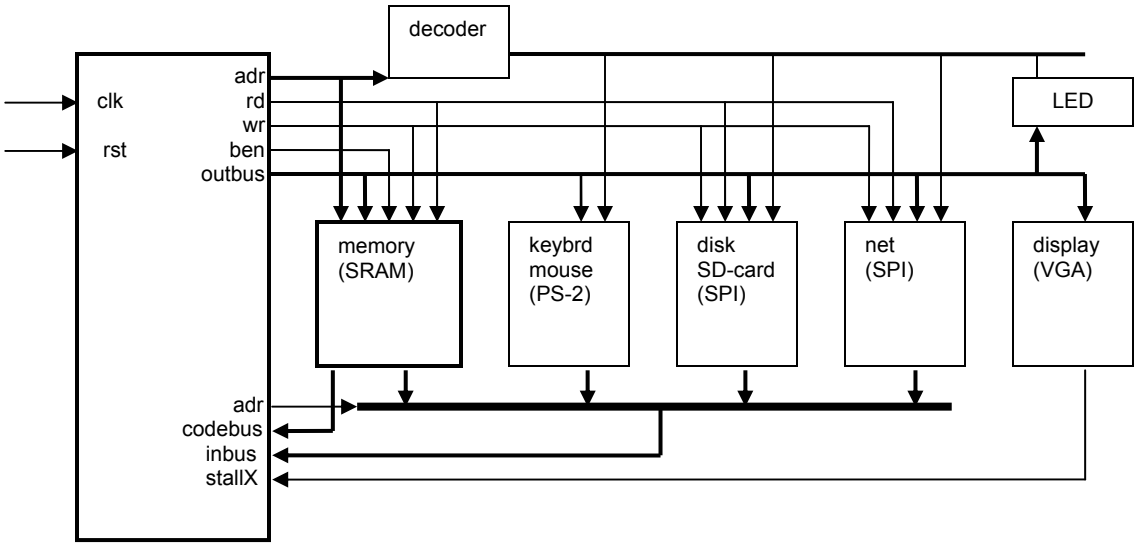


Figure 17.1 The RISC configuration

The decoder for output and the multiplexer for input determine the various addresses of devices:

adr		input	output
0	0FFFFFFC0H	millisecond counter	reserved
4	0FFFFFFC4H	switches	LEDs
8	0FFFFFFC8H	RS-232 data	RS-232 data
12	0FFFFFFCCH	RS-232 status	RS-232 control
16	0FFFFFFD0H	SPI data (SD-card, net)	SPI data (SD-card, nat)
20	0FFFFFFD4H	SPI status	SPI control
24	0FFFFFFD8H	PS/2 keyboard	
28	0FFFFFFDCH	mouse	

The circuitry connecting with the SRAM is part of this module, whereas the drivers for the other devices are described in separate modules. Note: The signals to and from devices must be listed in the heading of the top module, which is not imported by any other module. Their pin numbers are specified in a configuration file (.ucf). For details, the reader is referred to the program listing, as several items are rather dependent on the given Spartan-3 board.

17.1. The SRAM memory

The design of the circuitry around a static RAM is quite straight forward. The only controls are a read (SRoe) and a write enable signal (SRwe). Since the SRAM multiplexes data lines for input and output, a tri-state driver (SRbuf) must be used on the FPGA. This is shown schematically in Figure 17.2.

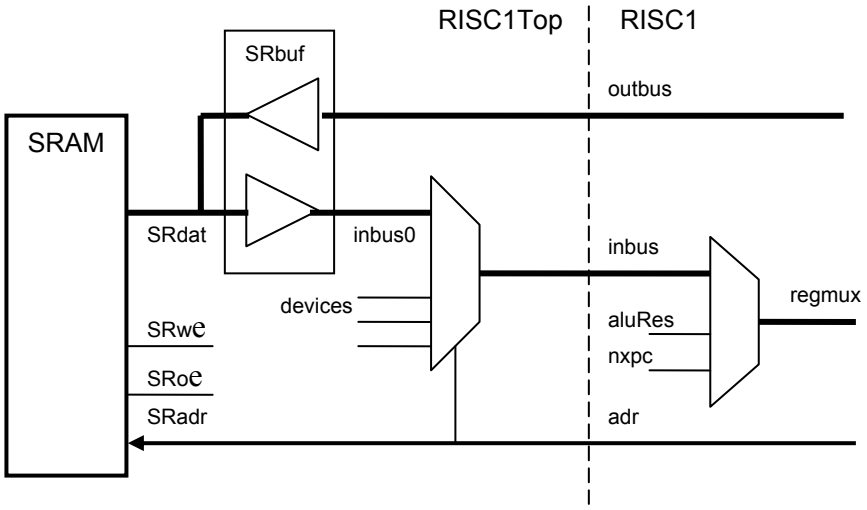


Figure 17.2. Connections between processor and SRAM

However, there is a complication: the feature of byte-wise access. After all, the present RAM is 32 bits wide (actually there are two 512K x 16-bit chips in parallel). Evidently, some multiplexing is unavoidable. The task is significantly eased by the chip's feature of four separate write enables, one for each byte of a word. The selection of the byte affected is determined by address bits 0 an 1 (which are ignored in the case of word-access). This scheme is shown in Figure 17.3. The codebus bypasses the multiplexers.

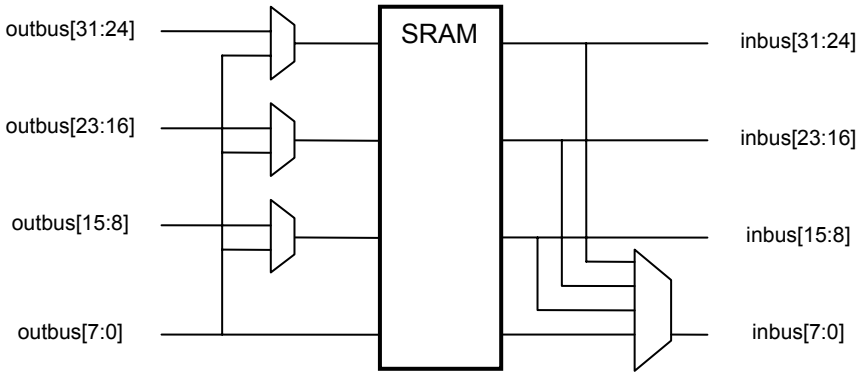


Figure 17.3 Multiplexers for SRAM byte access

17.2. Peripheral interfaces

Each of the interfaces to external media is implemented as a separate module and can therefore easily be exchanged. Modules are connected with the processor by the input and the output bus, and by enable signals *wr* and *rd*.

17.2.1. The PS/2 interface for the keyboard

PS/2 is mostly used for input devices. It uses 2 wires (apart from ground)., one for data, one for the clock. It uses a *synchronous* transmission, and the clock is driven by the device. Here it is used for the keyboard and the mouse (see Sect. 17.2.5). Transmission occurs in packets of 8 bits. An optional third wire serves for output. It is not used in this application. The interface is very simple and consists of an 8-bit buffer register. The following describes the interface for the keyboard.

A bit is shifted into the data register whenever the clock shows a falling edge, i.e. the clock signal *Q0* is low and the clock delayed by one cycle *Q1* is high.

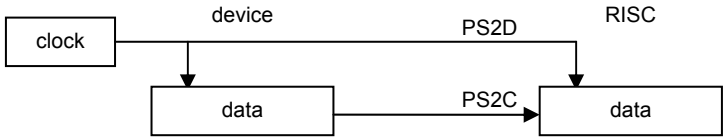


Figure 17.4 The PS/2 configuration

In the driver for the keyboard a 16-byte fifo buffer is inserted, forming a queue. This is necessary in order to avoid loss of characters when the processor is tied up in computation.

```
module PS2(
   input clk, rst,
   input done,   // "byte has been read"
   output rdy,   // "byte is available"
   output shift, // shift in, transmitter
   output [7:0] data,
   input PS2C,   // serial input
   input PS2D);   // clock

reg Q0, Q1;  // synchronizer and falling edge detector
reg [10:0] shreg;
reg [3:0] inptr, outptr;
reg [7:0] fifo [15:0];  // 16 byte buffer
wire endbit;

assign endbit = ~shreg[0];  //start bit reached correct pos
assign shift = Q1 & ~Q0;
assign data = fifo[outptr];
assign rdy = ~(inptr == outptr);

always @ (posedge clk) begin
  Q0 <= PS2C; Q1 <= Q0;
  shreg <= (~rst | endbit) ? 11'h7FF :
    shift ? {PS2D, shreg[10:1]} : shreg;
  outptr <= ~rst ? 0 : rdy & done ? outptr+1 : outptr;
  inptr <= ~rst ? 0 : endbit ? inptr+1 : inptr;
  if (endbit) fifo[inptr] <= shreg[8:1];
end
endmodule
```

17.2.2 The Mouse

Subsequently we present two Mouse interfaces. The first (MouseP) is based on the PS/2 Standard and caters for most commercially available mice. The second (MouseX) is included here for historical reasons. It was used by the computer Lilith in 1979, and used the same Mouse as its ancestor Alto (at PARC, 1975). It is distinguished by a very simple hardware without its own microprocessor, which is currently contained in most mice. This goes at a cost of a 9-wire cable. But today, microprocessors are cheaper than cables. We include this interface here, because it allows for a simple explanation of the principle of pointing devices.

The first interface uses the PS/2 Standard, that is, a 2-wire cable (not counting ground and power). It complies with the commercial standard of pointing devices. Details are shown on module MouseP.v.

```
module MouseP (input rst, clk,
   inout PS2C, PS2D,
   output [27:0] out);
endmodule
```

The second interface described here is not based on any standard, but it features the same interface to the software environment. Its principles are very simple and easily explained, and it refrains from the use of a mouse-internal processor. The price for this simplicity is a cable with 7 wires (plus 2 for power and ground), namely 3 for 3 buttons, and 2 for each direction, x (left/right) and y (up/down).

Let us first explain how signals indicating movements are derived. The key reason for the solution's simplicity is that these signals are directly mirrored by the position of a cursor on the display. The human user simply moves the Mouse until the cursor has reached the desired position (for example, at a displayed object). Thereby, the human eye and hand are included in the feedback loop providing the desired precision. This represents a very clever symbiosis between man and computer.

An actual movement is recognized by a simple light sensor (we will restrict our observation to a single coordinate x). The movement is transmitted to a wheel consisting of a transparent disc with intransparent spokes. A light beam shines through the disk and is received by the light sensor. Each time a spoke passes, the light is blocked. Any change in the sensor output signals a movement (see Figure 17.5). Unfortunately, this scheme does not allow to recognize the direction of the movement (left or right). A second light and sensor solve this problem. The distance between the two lights is half the distance of adjacent spokes.

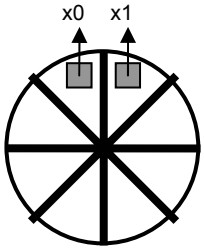


Figure 17.5 Wheel with spokes and sensors

The signal pair x0, x1 originating from a movement (with constant speed) to the left or to the right is shown in Figure 17.6.

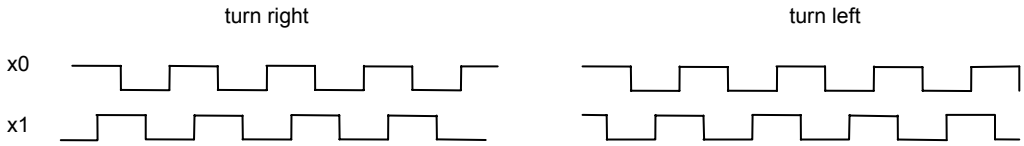


Figure 17.6 Signals resulting from movements

The logic equations for movements to the left and right (or up and down) are derived from this signal pair. For each signal a register records the state. Therefore it can be determined whether a move to the left, or to the right, or no move had occurred. The sampling frequency is irrelevant, as long as it is high enough. Let *x01* be *x00* delayed by one clock cycle, and *x11* be *x10* delayed by one cycle.

x00	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
x01	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
x10	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
x11	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
right	0	1	0	0	0	0	0	1	1	0	0	0	0	0	1	0
left	0	0	1	0	1	0	0	0	0	0	0	1	0	1	0	0

Every active *right* signal causes the 10-bit *x* counter to be incremented, and every *left* to be decremented.

An identical circuit is used for the up/down direction, with its wheel set perpendicular to the first wheel. Finally, the output is packed into a single word. 3 bits are taken by the keys, and 10 by each of the two counters.

```
module MouseX(
  input clk,
  input [6:0] in,
  output [27:0] out);

  reg x00, x01, x10, x11, y00, y01, y10, y11;
  reg ML, MM, MR;  // keys
  reg [9:0] x, y;  // counters

  wire xup, xdn, yup, ydn;

  assign xup = ~x00&~x01&~x10&x11 | ~x00&x01&x10&x11 | x00&~x01&~x10&~x11 | x00&x01&x10&~x11;
  assign yup = ~y00&~y01&~y10&y11 | ~y00&y01&y10&y11 | y00&~y01&~y10&~y11 | y00&y01&y10&~y11;
  assign xdn = ~x00&~x01&x10&~x11 | ~x00&x01&~x10&~x11 | x00&~x01&x10&x11 | x00&x01&~x10&x11;
  assign ydn = ~y00&~y01&y10&~y11 | ~y00&y01&~y10&~y11 | y00&~y01&y10&y11 | y00&y01&~y10&y11;
  assign out = {1'b0, ML, MM, MR, 2'b0, y, 2'b0, x};

  always @ (posedge clk) begin
    x00 <= in[3]; x01 <= x00; x10 <= in[2]; x11 <= x10;
    y00 <= in[1]; y01 <= y00; y10 <= in[0]; y11 <= y10;
    MR <= ~in[4]; MM <= ~in[5]; ML <= ~in[6];
    x <= xup ? x+1 : xdn ? x-1 : x;
    y <= yup ? y+1 : ydn ? y-1 : y;
  end
endmodule
```

17.2.3. The SPI interface for the SD-card (disk) and the Net

SPI (Standard Peripheral Interface) is similar to PS/2, and also synchronous. However, there may be many participants. They are configured in a loop as shown in Figure 17.7, and the clock is provided by a master, namely the RISC. SPI requires 3 wires (apart from ground).

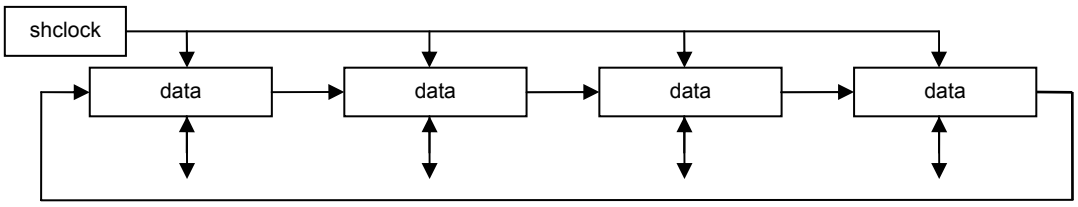


Figure 17.7 SPI-configuration as a ring

Here, however, no use is made of SPI's ring topology. Instead, One master interface is serving both the disk and the net. The connection is determined in module RISC5Top. The packet (and thus the shift register) is 32 bits long

Transmission frequency is 0.4 MHz at startup (as required by the SD-card), and then is raised to 8.33 MHz.. Details are shown in the respective program listing.

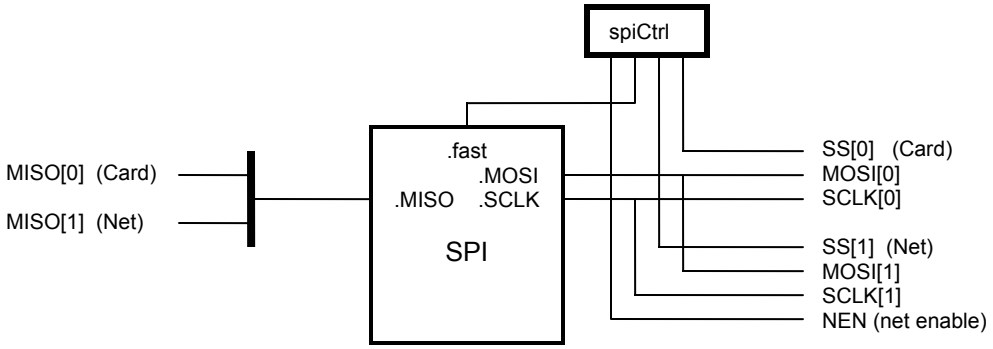


Figure 17.8 Connections between SPI, SD-card, and Net (see RISCTop5.v)

```
// Motorola Serial Peripheral Interface (SPI) PDR 23.3.12 / 16.10.13
// transmitter / receiver of words (fast, clk/3) or bytes (slow, clk/64)
// e.g 8.33MHz or ~400KHz respectively at 25MHz (slow needed for SD-card init)
// note: bytes are always MSbit first; but if fast, words are LSByte first

module SPI(
  input clk, rst,
  input start, fast,
  input [31:0] dataTx,
  output [31:0] dataRx,
  output reg rdy,
  input MISO,
  output MOSI, SCLK);
endmodule
```

The SPI specifications postulate that bytes are sent with the most significant bit first. This results in a somewhat twisted scheme for shifting bits (see Fig. 17.9).

```
shreg <= {shreg[30:24], MISO, shreg[22:16], shreg[31], shreg[14:8],
     shreg[23], shreg[6:0], shreg[15]}
```

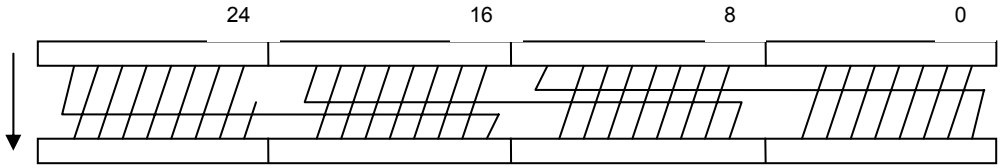


Figure 17.9 Shifting with MSB first

17.2.4. The display controller

A controller for a raster scan display feeds data from memory to the display. The data area in memory is called *frame buffer*. It contains a fixed number of bits for each pixel on the screen. In this case, there is exactly one bit per pixel, signalling black or white. For a 1024 x 768 pixel display area, 96 Kbyte are required.

The pixel position on the display is not determined by an address. Instead, data are received by rhe display purely sequentially, and the position is indirectly determined by two synchronization signals, *hsync* (for horizontal sync) at the end of each line, and *vsync* (for vertical sync) at the end of every frame. This scheme originates from cathode ray tube (CRT) monitors, where an electron beam is sweeping the screen. It is deflected by magnetic fields, which require some time to sweep back. The timing with retrace periods was retained for LCD displays as a legacy.

The heart of the controller consists of a data *buffer* (32 bits) fed from memory and shifted out bit by bit to the display, and of two counters *hcnt* and *vcnt*, representing the horizontal and vertical coordinates.The memory word address is derived from *hcnt* and *vcnt*:

vidadr = (hcnt DIV 32) + (vcnt * 32) + org

Every line consists of 1024 pixels (32 words). The challenge is to find a design with as few registers and comparators as possible. There are two signals for suppressing video data: *hblank, vblan*k. They are needed for turning the light off durich retrace.

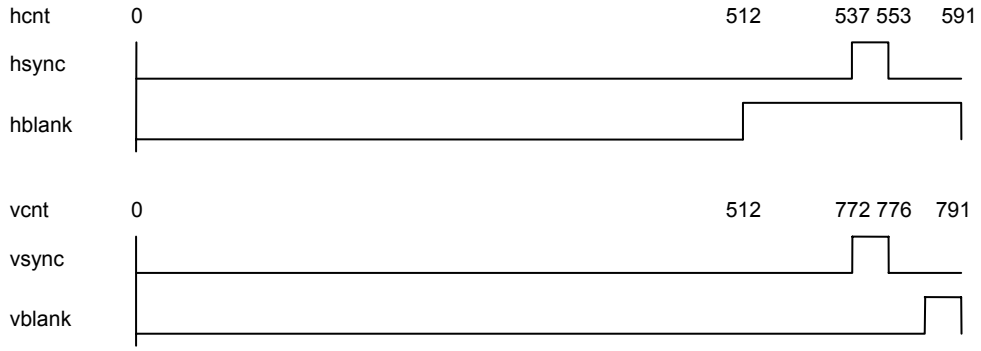


Figure 17.10 Synchronization and blanking signals

Let us generalize this scheme to displays of *w* pixels per line and *h* lines per frame. Also, let *w'* be the number of pixels per line including those of the retrace time, and *h'* be the number of lines including the vertical retrace. Also, let the number of displayed frames per second be *n*. Then the pixel frequency is

$f = w' \times h' \times n$.

This will in all probability be different from the system clock's frequency. Therefore the need arises for a diffenernt pixel clock. It is generated by the FPGA's built-in *digital clock manager* (dcm). It multiplies and divides the system clock by selectable factors. Note that the refresh rate may vary within certain bounds for all brands of monitors. Therefore, a simple factor may be chosen for division and multiplication. Examples:

(1024 x 768)	1182 x 791 x 60 = 56'097'720	rounded up to	60 MHz
(1280 x 1024)	1536 x 1280 x 60 = 117'964'800	rounded up to	125 MHz

The pixel buffer is fed from the video buffer driven by the system clock, and it is shifted and read by the pixel clock. This makes a double-buffering necessary, as shown in Figure 17.11. Also the counters are driven by this *pixel clock*. The numbers for *hcnt* and *vcnt* shown are, of course, device-specific (see Figure 17.10).

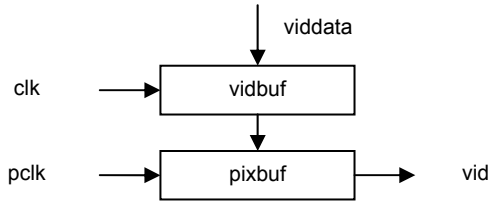


Figure 17.11 Buffering the video output

```
module VID(
   input clk, clk25, inv,
   input [31:0] viddata,
   output reg req,  // read request
   output hsync, vsync,  // to display
```

```
    output [17:0] vidadr,
    output [2:0] RGB);
  localparam Org = 18'b1101_1111_1111_0000_00;  // DFF00
  reg [9:0] hcnt, vcnt;
  reg [4:0] hword;  // from hcnt, but latched in the clk domain
  reg [31:0] vidbuf, pixbuf;
  reg hblank;
  endmodule
```

Both the display controller and the processor access memory directly. It therefore becomes necessary to arbitrate in the case where both require access simultaneously, that is, to decide which has priority. The decision is simple, because the display controller is time-critical and must not be delayed. The processor, on the other hand, can easily be delayed by the already present stalling scheme. The signal (wire) *dspreq* stalls the processor (stallX) and decides whether the memory address (SRadr) should be taken from the processor (adr) or the display controller (vidadr). The following multiplexer is placed in module RISCTop:

```
assign SRadr = dspreq ? vidadr : adr[19:2];
```

17.2.5. The RS-232 interface

RS-232 is an old standard for serial data transmission (see also Sect. 9.4). We chose to describe it here in detail because of its frequent use and inherent simplicity. RS-232 uses 2 wires (apart from ground), one for input (RxD) and one for output (TxD) as shown in Figure 17.12. Data are transmitted in packets of a fixed length, here of length 8, i.e. byte-wise. Since there is no clock wire, bytes are transmitted *asynchronously*. Their beginning is marked by a start-bit, and at the end a stop-bit is appended. Hence, a packet is 10 bits long (see Figure 17.13). Within a packet, transmission is synchronous, i.e. with a fixed clock rate, on which transmitter and receiver agree. The packet length is short enough to admit slight deviations. The standard defines several packet lengths and many clock rates. Here we use a rate of 19200 or 115200 bit/s.

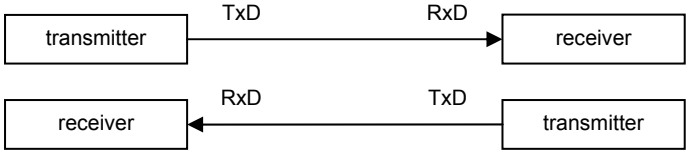


Figure 17.12 RS-232 configuration

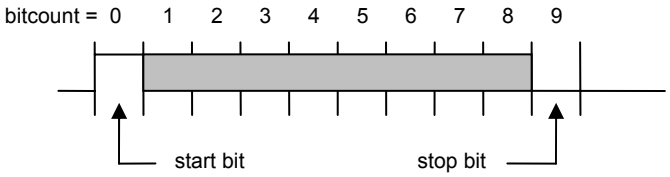


Figure 17.13 RS-232 packet format

The input signal *start* triggers the state machine by setting register *run*. The transmitter has 2 counters and a shift register. Counter *tick* runs from 0 to 1302, yielding a frequency of 25'000 / 1302 = 19.2 KHz, the transmission rate for bits. The signal *endtick* advances counter *bitcnt*, running from 0 to 9 (the number of bits in a packet). Signal *endbit* resets *ru*n and the counter to 0. Signal *rdy* indicates whether or not a next byte can be loaded and sent.

```
module RS232T(
    input clk, rst,  // system clock, 25 MHz
    input start, // request to accept and send a byte
    input [7:0] data,
```

```
    output rdy,   // status
    output TxD);  // serial data

wire endtick, endbit;
reg run;
reg [11:0] tick;
reg [3:0] bitcnt;
reg [8:0] shreg;

assign endtick = tick == 1302;
assign endbit = bitcnt == 9;
assign rdy = ~run;
assign TxD = shreg[0];

always @ (posedge clk) begin
  run <= (~rst | endtick & endbit) ? 0 : start ? 1 : run;
  tick <= (run & ~endtick) ? tick + 1 : 0;
  bitcnt <= (endtick & ~endbit) ? bitcnt + 1 :
    (endtick & endbit) ? 0 : bitcnt;
  shreg <= (~rst) ? 1 : start ? {data, 1'b0} :
    endtick ? {1'b1, shreg[8:1]} : shreg;
end
endmodule
```

The receiver is structured very similarly with 2 counters and a shift register. The state machine is triggered by an incoming start bit at RxD. The state *rdy* is set when the last data bit has been received, and it is reset by the *done* signal, generated when reading a byte. The line RxD is sampled in the middle of the bit period rather than at the end, namely when *midtick* = *endtick/2*.

```
module RS232R(
    input clk, rst,
    input done,   // "byte has been read"
    input RxD,
    output rdy,
    output [7:0] data);

wire endtick, midtick;
reg run, stat;
reg [11:0] tick;
reg [3:0] bitcnt;
reg [7:0] shreg;

assign endtick = tick == 1302;
assign midtick = tick == 651;
assign endbit = bitcnt == 8;
assign data = shreg;
assign rdy = stat;

always @ (posedge clk) begin
  run <= (~RxD) | (~rst | endtick & endbit) & run;
  tick <= (run & ~endtick) ? tick + 1 : 0;
  bitcnt <= (endtick & ~endbit) ? bitcnt + 1 :
    (endtick & endbit) ? 0 : bitcnt;
  shreg <= midtick ? {RxD, shreg[7:1]} : shreg;
  stat <= (endtick & endbit) ? 1 : (~rst | done) ? 0 : stat;
end
endmodule
```